Red Hat Fuse 7.3

Apache Camel Component Reference

Configuration reference for Camel components

Last Updated: 2021-06-15

Red Hat Fuse 7.3 Apache Camel Component Reference

Configuration reference for Camel components

Legal Notice

Abstract

Apache Camel has over 100 components and each component is highly configurable. This guide describes the settings for each of the components.

Table of Contents

CHAPTER 1. COMPONENTS OVERVIEW

This chapter provides a summary of all the components available for Apache Camel.

1.1. CONTAINER TYPES

Red Hat Fuse provides a variety of container types, into which you can deploy your Camel applications:

- Spring Boot
- Apache Karaf
- JBoss Enterprise Application Platform (JBoss EAP)

In addition, a Camel application can run as *containerless*: that is, where a Camel application runs directly in the JVM, without any special container.

In some cases, Fuse might support a Camel component in one container, but not in the others. There are various reasons for this, but in some cases a component is not suitable for all container types. For example, the **camel-ejb** component is designed spcifically for Java EE (that is, JBoss EAP), and cannot be supported in the other container types.

1.2. SUPPORTED COMPONENTS

Note the following key:

Symbol	Description
✔	Supported
▮	Unsupported or not yet supported
Deprecated	Likely to be removed in a future release

Table 1.1, “Apache Camel Component Support Matrix” provides comprehensive details about which Camel components are supported in which containers.

Table 1.1. Apache Camel Component Support Matrix

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
activemq-camel	Endpoint	✔	✔	✔	✔
camel-ahc	Endpoint	✔	✔	✔	✔
camel-ahc-ws	Endpoint	✔	✔	✔	✔
camel-ahc-wss	Endpoint	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-amqp	Endpoint	✔	✔	✔	✔
camel-apns	Endpoint	✔	✔	✔	✔
camel-asn1	Data Format	✔	✔	✔	✔
camel-asterisk	Endpoint	✔	✔	✔	✔
camel-atmos	Endpoint	✔	✔	▮	▮
camel-atmosphere-websocket	Endpoint	✔	✔	✔	✔
camel-atom	Endpoint	✔	✔	✔	✔
camel-atomix	Endpoint	✔	✔	✔	✔
camel-avro	Endpoint	✔	✔	✔	✔
camel-avro	Data Format	✔	✔	✔	✔
camel-aws	Endpoint	✔	✔	✔	✔
camel-azure	Endpoint	✔	✔	✔	✔
camel-bam	Endpoint	*Deprecated*	✔	✔	▮
camel-barcode	Data Format	✔	✔	✔	✔
camel-base64	Data Format	✔	✔	✔	✔
camel-bean	Endpoint	✔	✔	✔	✔
camel-bean	Language	✔	✔	✔	✔
camel-bean-validator	Endpoint	✔	✔	✔	✔
camel-beanio	Data Format	✔	✔	✔	✔
camel-beanstalk	Endpoint	✔	✔	✔	▮
camel-binding	Endpoint	*Deprecated*	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-bindy	Endpoint	✔	✔	✔	✔
camel-bindy	Data Format	✔	✔	✔	✔
camel-blueprint	Endpoint	✔	▮	✔	▮
camel-bonita	Endpoint	✔	▮	▮	▮
camel-boon	Data Format	✔	✔	✔	✔
camel-box	Endpoint	✔	✔	✔	✔
camel-braintree	Endpoint	✔	✔	✔	✔
camel-browse	Endpoint	✔	✔	✔	✔
camel-cache	Endpoint	*Deprecated*	✔	✔	▮
camel-caffeine	Endpoint	✔	✔	✔	✔
camel-castor	Data Format	*Deprecated*	✔	✔	✔
camel-cdi	Endpoint	✔	▮	*Deprecated*	✔
camel-chronicle-engine	Endpoint	✔	✔	✔	✔
camel-chunk	Endpoint	✔	✔	✔	✔
camel-class	Endpoint	✔	✔	✔	✔
camel-cm-sms	Endpoint	✔	✔	✔	✔
camel-cmis	Endpoint	✔	✔	✔	✔
camel-coap	Endpoint	✔	✔	✔	✔
camel-cometd	Endpoint	✔	✔	✔	✔
camel-constant	Language	✔	✔	✔	✔
camel-context	Endpoint	*Deprecated*	▮	▮	▮

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-consul	Endpoint	✔	✔	✔	✔
camel-controlbus	Endpoint	✔	✔	✔	✔
camel-couchbase	Endpoint	✔	✔	✔	✔
camel-couchdb	Endpoint	✔	✔	✔	✔
camel-cql	Endpoint	✔	✔	✔	✔
camel-crypto	Endpoint	✔	✔	✔	✔
camel-crypto	Data Format	✔	✔	✔	✔
camel-crypto-cms	Endpoint	✔	✔	✔	✔
camel-csv	Data Format	✔	✔	✔	✔
camel-cxf	Endpoint	✔	✔	✔	✔
camel-cxf-transport	Endpoint	✔	✔	✔	✔
camel-dataformat	Endpoint	✔	✔	✔	✔
camel-dataset	Endpoint	✔	✔	✔	✔
camel-digitalocean	Endpoint	✔	✔	✔	✔
camel-direct	Endpoint	✔	✔	✔	✔
camel-direct-vm	Endpoint	✔	✔	✔	✔
camel-disruptor	Endpoint	✔	✔	✔	✔
camel-dns	Endpoint	✔	✔	✔	✔
camel-docker	Endpoint	✔	✔	✔	✔
camel-dozer	Endpoint	✔	✔	✔	✔
camel-drill	Endpoint	✔	✔	✔	▮

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-dropbox	Endpoint	✔	✔	✔	✔
camel-eclipse		*Deprecated*	▮	▮	▮
camel-ehcache	Endpoint	✔	✔	✔	✔
camel-ejb	Endpoint	✔	▮	▮	✔
camel-el	Language	*Deprecated*	▮	▮	▮
camel-elasticsearch	Endpoint	✔	✔	✔	✔
camel-elasticsearch5	Endpoint	✔	✔	✔	✔
camel-elasticsearch-rest	Endpoint	✔	✔	✔	▮
camel-elsql	Endpoint	✔	✔	✔	✔
camel-etcd	Endpoint	✔	✔	✔	✔
camel-eventadmin	Endpoint	✔	▮	✔	▮
camel-exchangeProperty	Language	✔	✔	✔	✔
camel-exec	Endpoint	✔	✔	✔	✔
camel-facebook	Endpoint	✔	✔	✔	✔
camel-fhir	Data Format	✔	✔	✔	✔
camel-file	Endpoint	✔	✔	✔	✔
camel-file	Language	✔	✔	✔	✔
camel-flatpack	Endpoint	✔	✔	✔	✔
camel-flatpack	Data Format	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-flink	Endpoint	✔	✔	▮	✔
camel-fop	Endpoint	✔	✔	✔	✔
camel-freemarker	Endpoint	✔	✔	✔	✔
camel-ftp	Endpoint	✔	✔	✔	✔
camel-gae	Endpoint	*Deprecated*	▮	▮	▮
camel-ganglia	Endpoint	✔	✔	✔	✔
camel-geocoder	Endpoint	✔	✔	✔	✔
camel-git	Endpoint	✔	✔	✔	✔
camel-github	Endpoint	✔	✔	✔	✔
camel-google-bigquery	Endpoint	✔	✔	▮	✔
camel-google-calendar	Endpoint	✔	✔	✔	✔
camel-google-drive	Endpoint	✔	✔	✔	✔
camel-google-mail	Endpoint	✔	✔	✔	✔
camel-google-pubsub	Endpoint	✔	✔	✔	✔
camel-grape	Endpoint	✔	✔	✔	▮
camel-groovy	Language	✔	✔	✔	✔
camel-groovy-dsl		*Deprecated*	▮	▮	▮
camel-grpc	Endpoint	✔	✔	✔	✔
camel-guava-eventbus	Endpoint	✔	✔	✔	✔
camel-guice	Endpoint	*Deprecated*	✔	▮	▮

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-gzip	Data Format	✔	✔	✔	✔
camel-hawtdb	Endpoint	*Deprecated*	✔	✔	▮
camel-hazelcast	Endpoint	✔	✔	✔	✔
camel-hbase	Endpoint	✔	✔	▮	▮
camel-hdfs	Endpoint	*Deprecated*	✔	▮	▮
camel-hdfs2	Endpoint	✔	✔	✔	✔
camel-header	Language	✔	✔	✔	✔
camel-headersmap		✔	✔	✔	✔
camel-hessian	Data Format	*Deprecated*	✔	✔	✔
camel-hipchat	Endpoint	✔	✔	✔	✔
camel-hl7	Data Format	✔	✔	✔	✔
camel-http	Endpoint	*Deprecated*	✔	✔	▮
camel-http4	Endpoint	✔	✔	✔	✔
camel-hystrix	Endpoint	✔	✔	✔	✔
camel-ibatis	Endpoint	*Deprecated*	▮	▮	▮
camel-ical	Data Format	✔	✔	✔	✔
camel-iec60870	Endpoint	✔	✔	✔	✔
camel-ignite	Endpoint	▮	▮	▮	▮
camel-imap	Endpoint	✔	✔	✔	✔
camel-infinispan	Endpoint	✔	✔	✔	✔
camel-influxdb	Endpoint	✔	✔	✔	✔
camel-irc	Endpoint	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-ironmq	Endpoint	▮	▮	▮	▮
camel-jacksonxml	Data Format	✔	✔	✔	✔
camel-jasypt	Endpoint	✔	✔	✔	✔
camel-javaspace	Endpoint	*Deprecated*	▮	▮	▮
camel-jaxb	Data Format	✔	✔	✔	✔
camel-jbpm	Endpoint	▮	▮	▮	▮
camel-jcache	Endpoint	✔	✔	✔	✔
camel-jcifs	Endpoint	✔	▮	✔	▮
camel-jclouds	Endpoint	✔	▮	✔	✔
camel-jcr	Endpoint	✔	✔	✔	✔
camel-jdbc	Endpoint	✔	✔	✔	✔
camel-jetty	Endpoint	*Deprecated*	*Deprecated*	*Deprecated*	▮
camel-jetty8	Endpoint	▮	▮	▮	▮
camel-jetty9	Endpoint	✔	✔	✔	▮
camel-jgroups	Endpoint	✔	✔	✔	✔
camel-jibx	Data Format	✔	✔	✔	✔
camel-jing	Endpoint	✔	✔	✔	✔
camel-jira	Endpoint	✔	▮	▮	▮
camel-jms	Endpoint	✔	✔	✔	✔
camel-jmx	Endpoint	✔	✔	✔	✔
camel-jolt	Endpoint	✔	✔	✔	✔
camel-josql	Endpoint	*Deprecated*	✔	✔	▮

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-jpa	Endpoint	✔	✔	✔	✔
camel-jsch	Endpoint	✔	✔	✔	✔
camel-json-fastjson	Data Format	✔	✔	✔	✔
camel-json-gson	Data Format	✔	✔	✔	✔
camel-json-jackson	Data Format	✔	✔	✔	✔
camel-json-johnzon	Data Format	✔	✔	✔	✔
camel-json-validator	Endpoint	✔	✔	✔	▮
camel-json-xstream	Data Format	✔	✔	✔	✔
camel-jsonpath	Language	✔	✔	✔	✔
camel-jt400	Endpoint	✔	✔	✔	✔
camel-juel	Endpoint	*Deprecated*	✔	✔	▮
camel-jxpath	Language	*Deprecated*	▮	▮	▮
camel-kafka	Endpoint	✔	✔	✔	✔
camel-kestrel	Endpoint	*Deprecated*	✔	✔	▮
camel-krati	Endpoint	*Deprecated*	✔	✔	▮
camel-kubernetes	Endpoint	✔	✔	✔	✔
camel-kura		✔	▮	✔	▮
camel-ldap	Endpoint	✔	✔	✔	✔
camel-ldif	Endpoint	✔	✔	✔	▮
camel-leveldb	Endpoint	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-linkedin	Endpoint	✔	✔	✔	✔
camel-log	Endpoint	✔	✔	✔	✔
camel-lpr	Endpoint	✔	✔	✔	✔
camel-lra		▮	▮	▮	✔
camel-lucene	Endpoint	✔	✔	✔	✔
camel-lumberjack	Endpoint	✔	✔	✔	✔
camel-lzf	Data Format	✔	✔	✔	✔
camel-master		✔	✔	✔	▮
camel-mail	Endpoint	✔	✔	✔	✔
camel-metrics	Endpoint	✔	✔	✔	✔
camel-milo	Endpoint	✔	✔	✔	✔
camel-mime-multipart	Data Format	✔	✔	✔	✔
camel-mina	Endpoint	*Deprecated*	▮	✔	▮
camel-mina2	Endpoint	✔	✔	✔	✔
camel-mllp	Endpoint	✔	✔	✔	✔
camel-mock	Endpoint	✔	✔	✔	✔
camel-mongodb	Endpoint	✔	✔	✔	✔
camel-mongodb-gridfs	Endpoint	✔	✔	✔	✔
camel-mongodb3	Endpoint	✔	✔	✔	✔
camel-mqtt	Endpoint	✔	✔	✔	✔
camel-msv	Endpoint	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-mustache	Endpoint	✔	✔	✔	✔
camel-mvel	Endpoint	✔	✔	✔	✔
camel-mvel	Language	✔	✔	✔	✔
camel-mybatis	Endpoint	✔	✔	✔	✔
camel-nagios	Endpoint	✔	✔	✔	▮
camel-nats	Endpoint	✔	✔	✔	✔
camel-netty	Endpoint	*Deprecated*	✔	✔	▮
camel-netty-http	Endpoint	*Deprecated*	✔	✔	▮
camel-netty4	Endpoint	✔	✔	✔	✔
camel-netty4-http	Endpoint	✔	✔	✔	▮
camel-ognl	Language	✔	✔	✔	✔
camel-olingo2	Endpoint	✔	✔	✔	✔
camel-olingo4	Endpoint	✔	✔	✔	✔
camel-openshift	Endpoint	*Deprecated*	✔	✔	▮
camel-openstack	Endpoint	✔	✔	✔	✔
camel-opentracing		✔	✔	✔	✔
camel-optaplanner	Endpoint	✔	✔	✔	✔
camel-paho	Endpoint	✔	✔	✔	✔
camel-paxlogging	Endpoint	✔	▮	✔	▮

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-pdf	Endpoint	✔	✔	✔	✔
camel-pgevent	Endpoint	✔	✔	✔	✔
camel-pgp	Data Format	✔	✔	✔	✔
camel-php	Language	*Deprecated*	▮	▮	✔
camel-pop3	Endpoint	✔	✔	✔	✔
camel-printer	Endpoint	✔	✔	✔	✔
camel-properties	Endpoint	✔	✔	✔	✔
camel-protobuf	Data Format	✔	✔	✔	✔
camel-pubnub	Endpoint	✔	✔	✔	✔
camel-python	Language	*Deprecated*	▮	▮	▮
camel-quartz	Endpoint	*Deprecated*	✔	✔	▮
camel-quartz2	Endpoint	✔	✔	✔	✔
camel-quickfix	Endpoint	✔	✔	✔	✔
camel-rabbitmq	Endpoint	✔	✔	✔	✔
camel-reactive-streams	Endpoint	✔	✔	✔	✔
camel-reactor		✔	✔	✔	✔
camel-ref	Endpoint	✔	✔	✔	✔
camel-ref	Language	✔	✔	✔	✔
camel-rest	Endpoint	✔	✔	✔	✔
camel-rest-api	Endpoint	✔	✔	✔	✔
camel-rest-swagger	Endpoint	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-restlet	Endpoint	✔	✔	✔	▮
camel-ribbon		✔	✔	▮	✔
camel-rmi	Endpoint	✔	✔	✔	✔
camel-routebox	Endpoint	*Deprecated*	✔	✔	▮
camel-rss	Endpoint	✔	✔	✔	✔
camel-rss	Data Format	✔	✔	✔	✔
camel-ruby	Language	*Deprecated*	▮	▮	▮
camel-rx	Endpoint	*Deprecated*	✔	✔	▮
camel-saga	Endpoint	▮	▮	▮	▮
camel-salesforce	Endpoint	✔	✔	✔	✔
camel-sap	Endpoint	✔	✔	✔	✔
camel-sap-netweaver	Endpoint	✔	✔	✔	✔
camel-saxon	Endpoint	✔	✔	✔	✔
camel-scala	Endpoint	*Deprecated*	✔	✔	▮
camel-scheduler	Endpoint	✔	✔	✔	✔
camel-schematron	Endpoint	✔	✔	✔	✔
camel-scp	Endpoint	✔	✔	✔	✔
camel-scr	Endpoint	*Deprecated*	✔	*Deprecated*	▮
camel-script	Endpoint	*Deprecated*	*Deprecated*	*Deprecated*	*Deprecated*
camel-seda	Endpoint	✔	✔	✔	✔
camel-serialization	Data Format	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-servicenow	Endpoint	✔	✔	✔	✔
camel-servlet	Endpoint	✔	✔	✔	✔
camel-servletlistener	Endpoint	*Deprecated*	✔	✔	▮
camel-sftp	Endpoint	✔	✔	✔	✔
camel-shiro	Endpoint	✔	✔	✔	✔
camel-simple	Language	✔	✔	✔	✔
camel-sip	Endpoint	✔	✔	✔	✔
camel-sjms	Endpoint	✔	✔	✔	✔
camel-sjms2	Endpoint	✔	✔	✔	✔
camel-slack	Endpoint	✔	✔	✔	✔
camel-smpp	Endpoint	✔	✔	✔	✔
camel-snakeyaml	Endpoint	✔	✔	✔	✔
camel-snmp	Endpoint	✔	✔	✔	✔
camel-soapjaxb	Data Format	✔	✔	✔	✔
camel-solr	Endpoint	✔	✔	✔	✔
camel-spark	Endpoint	✔	✔	▮	▮
camel-spark-rest	Endpoint	✔	✔	▮	▮
camel-spel	Language	✔	✔	▮	✔
camel-splunk	Endpoint	✔	✔	✔	✔
camel-spring	Endpoint	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-spring-batch	Endpoint	✔	✔	✔	✔
camel-spring-boot	Endpoint	✔	✔	▮	▮
camel-spring-cloud		✔	✔	▮	▮
camel-spring-cloud-netflix		✔	✔	▮	▮
camel-spring-event	Endpoint	✔	✔	▮	✔
camel-spring-integration	Endpoint	✔	▮	▮	✔
camel-spring-javaconfig	Endpoint	✔	✔	▮	✔
camel-spring-ldap	Endpoint	✔	✔	✔	✔
camel-spring-redis	Endpoint	✔	✔	✔	✔
camel-spring-security	Endpoint	✔	✔	▮	✔
camel-spring-ws	Endpoint	✔	✔	✔	✔
camel-sql	Endpoint	✔	✔	✔	✔
camel-sql-stored	Endpoint	✔	✔	✔	✔
camel-ssh	Endpoint	✔	✔	✔	✔
camel-stax	Endpoint	✔	✔	✔	✔
camel-stomp	Endpoint	✔	✔	✔	✔
camel-stream	Endpoint	✔	✔	✔	✔
camel-string	Data Format	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-string-template	Endpoint	✔	✔	✔	✔
camel-stub	Endpoint	✔	✔	✔	✔
camel-swagger	Endpoint	*Deprecated*	✔	*Deprecated*	▮
camel-swagger-java	Endpoint	✔	✔	✔	✔
camel-syslog	Data Format	✔	✔	✔	✔
camel-tagsoup	Endpoint	✔	✔	✔	✔
camel-tarfile	Data Format	✔	✔	✔	✔
camel-telegram	Endpoint	✔	✔	✔	✔
camel-thrift	Endpoint	✔	✔	✔	✔
camel-thrift	Data Format	✔	✔	✔	✔
camel-tika	Endpoint	✔	✔	✔	✔
camel-timer	Endpoint	✔	✔	✔	✔
camel-tokenize	Language	✔	✔	✔	✔
camel-twilio	Endpoint	✔	✔	✔	✔
camel-twitter	Endpoint	✔	✔	✔	✔
camel-undertow	Endpoint	✔	✔	✔	✔
camel-univocity-csv	Data Format	✔	✔	✔	✔
camel-univocity-fixed	Data Format	✔	✔	✔	✔
camel-univocity-tsv	Data Format	✔	✔	✔	✔
camel-urlrewrite	Endpoint	*Deprecated*	✔	✔	▮

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-validator	Endpoint	✔	✔	✔	✔
camel-velocity	Endpoint	✔	✔	✔	✔
camel-vertx	Endpoint	✔	✔	✔	✔
camel-vm	Endpoint	✔	✔	✔	✔
camel-weather	Endpoint	✔	✔	✔	✔
camel-websocket	Endpoint	✔	✔	✔	▮
camel-wordpress	Endpoint	✔	✔	✔	✔
camel-xchange	Endpoint	✔	✔	✔	▮
camel-xmlbeans	Data Format	*Deprecated*	*Deprecated*	*Deprecated*	✔
camel-xmljson	Data Format	*Deprecated*	*Deprecated*	*Deprecated*	*Deprecated*
camel-xmlrpc	Endpoint	▮	▮	▮	▮
camel-xmlrpc	Data Format	▮	▮	▮	▮
camel-xmlsecurity	Endpoint	✔	✔	✔	✔
camel-xmpp	Endpoint	✔	✔	✔	✔
camel-xpath	Language	✔	✔	✔	✔
camel-xquery	Endpoint	✔	✔	✔	✔
camel-xquery	Language	✔	✔	✔	✔
camel-xslt	Endpoint	✔	✔	✔	✔
camel-xstream	Data Format	✔	✔	✔	✔
camel-xtokenize	Language	✔	✔	✔	✔
camel-yaml-snakeyaml	Data Format	✔	✔	✔	✔

Component	Type	Containerless	Spring Boot	Karaf	JBoss EAP
camel-yammer	Endpoint	✔	✔	✔	✔
camel-yql	Endpoint	▮	▮	▮	▮
camel-zendesk	Endpoint	✔	✔	▮	✔
camel-zip	Data Format	✔	✔	✔	✔
camel-zipfile	Data Format	✔	✔	✔	✔
camel-zipkin	Endpoint	✔	✔	✔	✔
camel-zookeeper	Endpoint	✔	✔	✔	✔
camel-zookeeper-master	Endpoint	✔	✔	✔	✔

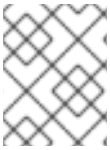
CHAPTER 2. ACTIVEMQ

ACTIVEMQ COMPONENT

The ActiveMQ component allows messages to be sent to a JMS Queue or Topic; or messages to be consumed from a JMS Queue or Topic using Apache ActiveMQ.

This component is based on the Chapter 170, *JMS Component* and uses Spring's JMS support for declarative transactions, using Spring's **JmsTemplate** for sending and a **MessageListenerContainer** for consuming. All the options from the Chapter 170, *JMS Component* component also apply for this component.

To use this component, make sure you have the **activemq.jar** or **activemq-core.jar** on your classpath along with any Apache Camel dependencies such as **camel-core.jar**, **camel-spring.jar** and **camel-jms.jar**.



TRANSACTED AND CACHING

See section **Transactions and Cache Levels**below on JMS page if you are using transactions with JMS as it can impact performance.

URI FORMAT

```
activemq:[queue:|topic:]destinationName
```

Where **destinationName** is an ActiveMQ queue or topic name. By default, the **destinationName** is interpreted as a queue name. For example, to connect to the queue, **FOO.BAR**, use:

```
activemq:FOO.BAR
```

You can include the optional **queue:** prefix, if you prefer:

```
activemq:queue:FOO.BAR
```

To connect to a topic, you must include the **topic:** prefix. For example, to connect to the topic, **Stocks.Prices**, use:

```
activemq:topic:Stocks.Prices
```

OPTIONS

See Options on the Chapter 170, *JMS Component* component as all these options also apply for this component.

CAMEL ON EAP DEPLOYMENT

This component is supported by the Camel on EAP (Wildfly Camel) framework, which offers a simplified deployment model on the Red Hat JBoss Enterprise Application Platform (JBoss EAP) container.

You can configure the ActiveMQ Camel component to work either with an embedded broker or an external broker. To embed a broker in the JBoss EAP container, configure the ActiveMQ Resource

Adapter in the EAP container configuration file – for details, see ActiveMQ Resource Adapter Configuration.

CONFIGURING THE CONNECTION FACTORY

The following test case shows how to add an ActiveMQComponent to the CamelContext using the **activeMQComponent()** method while specifying the brokerURL used to connect to ActiveMQ.

```
camelContext.addComponent("activemq", activeMQComponent("vm://localhost?
broker.persistent=false"));
```

CONFIGURING THE CONNECTION FACTORY USING SPRING XML

You can configure the ActiveMQ broker URL on the ActiveMQComponent as follows

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  </camelContext>

  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://somehost:61616"/>
  </bean>

</beans>
```

USING CONNECTION POOLING

When sending to an ActiveMQ broker using Camel it's recommended to use a pooled connection factory to handle efficient pooling of JMS connections, sessions and producers. This is documented in the page ActiveMQ Spring Support.

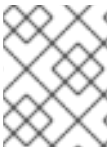
You can grab Jencks AMQ pool with Maven:

```
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-pool</artifactId>
    <version>5.3.2</version>
  </dependency>
```

And then setup the **activemq** component as follows:

```
  <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616" />
  </bean>
```

```
    <bean id="pooledConnectionFactory"
class="org.apache.activemq.pool.PooledConnectionFactory" init-method="start" destroy-
method="stop">
        <property name="maxConnections" value="8" />
        <property name="connectionFactory" ref="jmsConnectionFactory" />
    </bean>

    <bean id="jmsConfig" class="org.apache.camel.component.jms.JmsConfiguration">
        <property name="connectionFactory" ref="pooledConnectionFactory"/>
        <property name="concurrentConsumers" value="10"/>
    </bean>

    <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
        <property name="configuration" ref="jmsConfig"/>
    </bean>
```



NOTE

Notice the **init** and **destroy** methods on the pooled connection factory. This is important to ensure the connection pool is properly started and shutdown.

The **PooledConnectionFactory** will then create a connection pool with up to 8 connections in use at the same time. Each connection can be shared by many sessions. There is an option named **maxActive** you can use to configure the maximum number of sessions per connection; the default value is **500**. From **ActiveMQ 5.7** onwards the option has been renamed to better reflect its purpose, being named as **maxActiveSessionPerConnection**. Notice the **concurrentConsumers** is set to a higher value than **maxConnections** is. This is okay, as each consumer is using a session, and as a session can share the same connection, we are in the safe. In this example we can have 8 * 500 = 4000 active sessions at the same time.

INVOKING MESSAGELISTENER POJOS IN A ROUTE

The ActiveMQ component also provides a helper Type Converter from a JMS MessageListener to a Processor. This means that the Chapter 41, *Bean Component* component is capable of invoking any JMS MessageListener bean directly inside any route.

So for example you can create a MessageListener in JMS as follows:

```
public class MyListener implements MessageListener {
    public void onMessage(Message jmsMessage) {
        // ...
    }
}
```

Then use it in your route as follows

```
from("file://foo/bar").
  bean(MyListener.class);
```

That is, you can reuse any of the Apache Camel components and easily integrate them into your JMS **MessageListener** POJO\!

USING ACTIVEMQ DESTINATION OPTIONS

Available as of ActiveMQ 5.6

You can configure the Destination Options in the endpoint uri, using the "destination." prefix. For example to mark a consumer as exclusive, and set its prefetch size to 50, you can do as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://src/test/data?noop=true"/>
    <to uri="activemq:queue:foo"/>
  </route>
  <route>
    <!-- use consumer.exclusive ActiveMQ destination option, notice we have to prefix with destination.
-->
    <from uri="activemq:foo?
destination.consumer.exclusive=true&amp;destination.consumer.prefetchSize=50"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

CONSUMING ADVISORY MESSAGES

ActiveMQ can generate Advisory messages which are put in topics that you can consume. Such messages can help you send alerts in case you detect slow consumers or to build statistics (number of messages/produced per day, etc.) The following Spring DSL example shows you how to read messages from a topic.

```
<route>
 <from uri="activemq:topic:ActiveMQ.Advisory.Connection?mapJmsMessage=false" />
 <convertBodyTo type="java.lang.String"/>
 <transform>
     <simple>${in.body}&#13;</simple>
 </transform>
 <to uri="file://data/activemq/?fileExist=Append&ileName=advisoryConnection-
${date:now:yyyyMMdd}.txt" />
</route>
```

If you consume a message on a queue, you should see the following files under data/activemq folder :

advisoryConnection-20100312.txt advisoryProducer-20100312.txt

and containing string:

```
    ActiveMQMessage {commandId = 0, responseRequired = false, messageId = ID:dell-charles-
3258-1268399815140
    -1:0:0:0:221, originalDestination = null, originalTransactionId = null, producerId = ID:dell-charles-
    3258-1268399815140-1:0:0:0, destination = topic://ActiveMQ.Advisory.Connection, transactionId
= null,
    expiration = 0, timestamp = 0, arrival = 0, brokerInTime = 1268403383468, brokerOutTime =
1268403383468,
    correlationId = null, replyTo = null, persistent = false, type = Advisory, priority = 0, groupID = null,
    groupSequence = 0, targetConsumerId = null, compressed = false, userID = null, content = null,
    marshalledProperties = org.apache.activemq.util.ByteSequence@17e2705, dataStructure =
ConnectionInfo
    {commandId = 1, responseRequired = true, connectionId = ID:dell-charles-3258-1268399815140-
```

```
2:50,
    clientId = ID:dell-charles-3258-1268399815140-14:0, userName = , password = *****,
    brokerPath = null, brokerMasterConnector = false, manageable = true, clientMaster = true},
    redeliveryCounter = 0, size = 0, properties = {originBrokerName=master, originBrokerId=ID:dell-
charles-
    3258-1268399815140-0:0, originBrokerURL=vm://master}, readOnlyProperties = true,
readOnlyBody = true,
    droppable = false}
```

GETTING COMPONENT JAR

You need this dependency:

- **activemq-camel**

ActiveMQ is an extension of the Chapter 170, *JMS Component* component released with the ActiveMQ project.

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.6.0</version>
</dependency>
```

CHAPTER 3. AHC COMPONENT

Available as of Camel version 2.8

The **ahc:** component provides HTTP based endpoints for consuming external HTTP resources (as a client to call external servers using HTTP).
The component uses the Async Http Client library.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ahc</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.1. URI FORMAT

```
ahc:http://hostname[:port][/resourceUri][?options]
ahc:https://hostname[:port][/resourceUri][?options]
```

Will by default use port 80 for HTTP and 443 for HTTPS.

You can append query options to the URI in the following format, **?option=value&option=value&…**

3.2. AHCENDPOINT OPTIONS

The AHC endpoint is configured using URI syntax:

```
ahc:httpUri
```

with the following path and query parameters:

3.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
httpUri	**Required** The URI to use such as http://hostname:port/path		URI

3.2.2. Query Parameters (13 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
bridgeEndpoint (producer)	If the option is true, then the Exchange.HTTP_URI header is ignored, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the AhcProducer send all the fault response back.	false	boolean
bufferSize (producer)	The initial in-memory buffer size used when transferring data between Camel and AHC Client.	4096	int
connectionClose (producer)	Define if the Connection Close header has to be added to HTTP Request. This parameter is false by default	false	boolean
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
headerFilterStrategy (producer)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
throwExceptionOnFailure (producer)	Option to disable throwing the AhcOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
transferException (producer)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type (for example using Jetty or Servlet Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the AhcOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
binding (advanced)	To use a custom AhcBinding which allows to control how to bind between AHC and Camel.		AhcBinding
clientConfig (advanced)	To configure the AsyncHttpClient to use a custom com.ning.http.client.AsyncHttpClientConfig instance.		AsyncHttpClientConfig
clientConfigOptions (advanced)	To configure the AsyncHttpClientConfig using the key/values from the Map.		Map

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
clientConfigRealmOptions (security)	To configure the AsyncHttpClientConfig Realm using the key/values from the Map.		Map
sslContextParameters (security)	Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry. This reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility. Note that configuring this option will override any SSL/TLS configuration options provided through the clientConfig option at the endpoint or component level.		SSLContextParameters

3.3. AHCCOMPONENT OPTIONS

The AHC component supports 8 options which are listed below.

Name	Description	Default	Type
client (advanced)	To use a custom AsyncHttpClient		AsyncHttpClient
binding (advanced)	To use a custom AhcBinding which allows to control how to bind between AHC and Camel.		AhcBinding
clientConfig (advanced)	To configure the AsyncHttpClient to use a custom com.ning.http.client.AsyncHttpClientConfig instance.		AsyncHttpClientConfig
sslContextParameters (security)	Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry. Note that configuring this option will override any SSL/TLS configuration options provided through the clientConfig option at the endpoint or component level.		SSLContextParameters

Name	Description	Default	Type
allowJavaSerialized Object (advanced)	Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

Notice that setting any of the options on the **AhcComponent** will propagate those options to **AhcEndpoints** being created. However the **AhcEndpoint** can also configure/override a custom option. Options set on endpoints will always take precedence over options from the **AhcComponent**.

3.4. MESSAGE HEADERS

Name	Type	Description
Exchange.HTTP_URI	**String**	URI to call. Will override existing URI set directly on the endpoint.
Exchange.HTTP_PATH	**String**	Request URI's path, the header will be used to build the request URI with the HTTP_URI. If the path is start with "/", http producer will try to find the relative path based on the Exchange.HTTP_BASE_URI header or the **exchange.getFromEndpoint().getEndpointUri();**
Exchange.HTTP_QUERY	**String**	**Camel 2.11 onwards:** URI parameters. Will override existing URI parameters set directly on the endpoint.

Name	Type	Description
Exchange.HTTP_RESPONSE_CODE	**int**	The HTTP response code from the external server. Is 200 for OK.
Exchange.HTTP_CHARACTER_ENCODING	**String**	Character encoding.
Exchange.CONTENT_TYPE	**String**	The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as **text/html**.
Exchange.CONTENT_ENCODING	**String**	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as **gzip**.

3.5. MESSAGE BODY

Camel will store the HTTP response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so headers are preserved during routing. Additionally Camel will add the HTTP response headers as well to the OUT message headers.

3.6. RESPONSE CODE

Camel will handle according to the HTTP response code:

- Response code is in the range 100..299, Camel regards it as a success response.
- Response code is in the range 300..399, Camel regards it as a redirection response and will throw a **AhcOperationFailedException** with the information.
- Response code is 400+, Camel regards it as an external server failure and will throw a **AhcOperationFailedException** with the information.
  throwExceptionOnFailure

The option, **throwExceptionOnFailure**, can be set to **false** to prevent the **AhcOperationFailedException** from being thrown for failed response codes. This allows you to get any response from the remote server.

3.7. AHCOPERATIONFAILEDEXCEPTION

This exception contains the following information:

- The HTTP status code
- The HTTP status line (text of the status code)
- Redirect location, if server returned a redirect
- Response body as a **java.lang.String**, if server provided a body as response

3.8. CALLING USING GET OR POST

The following algorithm is used to determine if either **GET** or **POST** HTTP method should be used:
1. Use method provided in header.
2. **GET** if query string is provided in header.
3. **GET** if endpoint is configured with a query string.
4. **POST** if there is data to send (body is not null).
5. **GET** otherwise.

3.9. CONFIGURING URI TO CALL

You can set the HTTP producer's URI directly form the endpoint URI. In the route below, Camel will call out to the external server, **oldhost**, using HTTP.

```
from("direct:start")
        .to("ahc:http://oldhost");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="ahc:http://oldhost"/>
  </route>
</camelContext>
```

You can override the HTTP endpoint URI by adding a header with the key, **Exchange.HTTP_URI**, on the message.

```
from("direct:start")
    .setHeader(Exchange.HTTP_URI, constant("http://newhost"))
    .to("ahc:http://oldhost");
```

3.10. CONFIGURING URI PARAMETERS

The **ahc** producer supports URI parameters to be sent to the HTTP server. The URI parameters can either be set directly on the endpoint URI or as a header with the key **Exchange.HTTP_QUERY** on the message.

```
from("direct:start")
        .to("ahc:http://oldhost?order=123&detail=short");
```

Or options provided in a header:

```
from("direct:start")
            .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
        .to("ahc:http://oldhost");
```

3.11. HOW TO SET THE HTTP METHOD TO THE HTTP PRODUCER

The HTTP component provides a way to set the HTTP request method by setting the message header. Here is an example;

```
from("direct:start")
            .setHeader(Exchange.HTTP_METHOD, constant("POST"))
        .to("ahc:http://www.google.com")
            .to("mock:results");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <setHeader headerName="CamelHttpMethod">
        <constant>POST</constant>
    </setHeader>
    <to uri="ahc:http://www.google.com"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

3.12. CONFIGURING CHARSET

If you are using **POST** to send data you can configure the **charset** using the **Exchange** property:

```
exchange.setProperty(Exchange.CHARSET_NAME, "iso-8859-1");
```

3.12.1. URI Parameters from the endpoint URI

In this sample we have the complete URI endpoint that is just what you would have typed in a web browser. Multiple URI parameters can of course be set using the **&** character as separator, just as you would in the web browser. Camel does no tricks here.

```
// we query for Camel at the Google page
template.sendBody("ahc:http://www.google.com/search?q=Camel", null);
```

3.12.2. URI Parameters from the Message

```
Map headers = new HashMap();
headers.put(Exchange.HTTP_QUERY, "q=Camel&lr=lang_en");
// we query for Camel and English language at Google
template.sendBody("ahc:http://www.google.com/search", null, headers);
```

In the header value above notice that it should **not** be prefixed with **?** and you can separate parameters as usual with the **&** char.

3.12.3. Getting the Response Code

You can get the HTTP response code from the AHC component by getting the value from the Out message header with **Exchange.HTTP_RESPONSE_CODE**.

```
Exchange exchange = template.send("ahc:http://www.google.com/search", new Processor() {
        public void process(Exchange exchange) throws Exception {
            exchange.getIn().setHeader(Exchange.HTTP_QUERY, constant("hl=en&q=activemq"));
        }
});
Message out = exchange.getOut();
int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

3.13. CONFIGURING ASYNCHTTPCLIENT

The **AsyncHttpClient** client uses a **AsyncHttpClientConfig** to configure the client. See the documentation at
Async Http Client for more details.

In Camel **2.8**, configuration is limited to using the builder pattern provided by **AsyncHttpClientConfig.Builder**. In Camel 2.8, the **AsyncHttpClientConfig** doesn't support getters/setters so its not easy to create/configure using a Spring bean style (eg the <bean> tag in the XML file).

The example below shows how to use a builder to create the **AsyncHttpClientConfig** which we configure on the **AhcComponent**.

In Camel **2.9**, the AHC component uses Async HTTP library 1.6.4. This newer version provides added support for plain bean style configuration. The **AsyncHttpClientConfigBean** class provides getters and setters for the configuration options available in **AsyncHttpClientConfig**. An instance of **AsyncHttpClientConfigBean** may be passed directly to the AHC component or referenced in an endpoint URI using the **clientConfig** URI parameter.

Also available in Camel **2.9** is the ability to set configuration options directly in the URI. URI parameters starting with "clientConfig." can be used to set the various configurable properties of **AsyncHttpClientConfig**. The properties specified in the endpoint URI are merged with those specified in the configuration referenced by the "clientConfig" URI parameter with those being set using the "clientConfig." parameter taking priority. The **AsyncHttpClientConfig** instance referenced is always copied for each endpoint such that settings on one endpoint will remain independent of settings on any previously created endpoints. The example below shows how to configure the AHC component using the "clientConfig." type URI parameters.

```
from("direct:start")
    .to("ahc:http://localhost:8080/foo?
clientConfig.maxRequestRetry=3&clientConfig.followRedirects=true")
```

3.14. SSL SUPPORT (HTTPS)

Using the JSSE Configuration Utility

As of Camel 2.9, the AHC component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the AHC component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

AhcComponent component = context.getComponent("ahc", AhcComponent.class);
component.setSslContextParameters(scp));
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
     id="sslContextParameters">
    <camel:keyManagers
       keyPassword="keyPassword">
      <camel:keyStore
         resource="/users/home/server/keystore.jks"
         password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="ahc:https://localhost/foo?sslContextParameters=#sslContextParameters"/>
...
```

3.15. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

- Jetty
- HTTP
- HTTP4

CHAPTER 4. AHC WEBSOCKET COMPONENT

Available as of Camel version 2.14

The **ahc-ws** component provides Websocket based endpoints for a client communicating with external servers over Websocket (as a client opening a websocket connection to an external server).
The component uses the AHC component that in turn uses the Async Http Client library.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ahc-ws</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

4.1. URI FORMAT

```
ahc-ws://hostname[:port][/resourceUri][?options]
ahc-wss://hostname[:port][/resourceUri][?options]
```

Will by default use port 80 for ahc-ws and 443 for ahc-wss.

4.2. AHC-WS OPTIONS

As the AHC-WS component is based on the AHC component, you can use the various configuration options of the AHC component.

The AHC Websocket component supports 8 options which are listed below.

Name	Description	Default	Type
client (advanced)	To use a custom AsyncHttpClient		AsyncHttpClient
binding (advanced)	To use a custom AhcBinding which allows to control how to bind between AHC and Camel.		AhcBinding
clientConfig (advanced)	To configure the AsyncHttpClient to use a custom com.ning.http.client.AsyncHttpClientConfig instance.		AsyncHttpClientConfig
sslContextParameters (security)	Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry. Note that configuring this option will override any SSL/TLS configuration options provided through the clientConfig option at the endpoint or component level.		SSLContextParameters

Name	Description	Default	Type
allowJavaSerialized Object (advanced)	Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AHC Websocket endpoint is configured using URI syntax:

```
ahc-ws:httpUri
```

with the following path and query parameters:

4.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
httpUri	**Required** The URI to use such as http://hostname:port/path		URI

4.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
bridgeEndpoint (common)	If the option is true, then the Exchange.HTTP_URI header is ignored, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the AhcProducer send all the fault response back.	false	boolean

Name	Description	Default	Type
bufferSize (common)	The initial in-memory buffer size used when transferring data between Camel and AHC Client.	4096	int
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
throwExceptionOnFailure (common)	Option to disable throwing the AhcOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
transferException (common)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type (for example using Jetty or Servlet Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the AhcOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendMessageOnError (consumer)	Whether to send an message if the web-socket listener received an error.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
connectionClose (producer)	Define if the Connection Close header has to be added to HTTP Request. This parameter is false by default	false	boolean
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
useStreaming (producer)	To enable streaming to send data as multiple text fragments.	false	boolean
binding (advanced)	To use a custom AhcBinding which allows to control how to bind between AHC and Camel.		AhcBinding
clientConfig (advanced)	To configure the AsyncHttpClient to use a custom com.ning.http.client.AsyncHttpClientConfig instance.		AsyncHttpClientConfig
clientConfigOptions (advanced)	To configure the AsyncHttpClientConfig using the key/values from the Map.		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
clientConfigRealmOptions (security)	To configure the AsyncHttpClientConfig Realm using the key/values from the Map.		Map
sslContextParameters (security)	Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry. This reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility. Note that configuring this option will override any SSL/TLS configuration options provided through the clientConfig option at the endpoint or component level.		SSLContextParameters

4.3. WRITING AND READING DATA OVER WEBSOCKET

An ahc-ws endpoint can either write data to the socket or read from the socket, depending on whether the endpoint is configured as the producer or the consumer, respectively.

4.4. CONFIGURING URI TO WRITE OR READ DATA

In the route below, Camel will write to the specified websocket connection.

```
from("direct:start")
        .to("ahc-ws://targethost");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="ahc-ws://targethost"/>
  </route>
</camelContext>
```

In the route below, Camel will read from the specified websocket connection.

```
from("ahc-ws://targethost")
        .to("direct:next");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="ahc-ws://targethost"/>
    <to uri="direct:next"/>
  </route>
</camelContext>
```

4.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AHC
- Atmosphere-Websocket

CHAPTER 5. AMQP COMPONENT

Available as of Camel version 1.2

The **amqp:** component supports the AMQP 1.0 protocol using the JMS Client API of the Qpid project. In case you want to use AMQP 0.9 (in particular RabbitMQ) you might also be interested in the Camel RabbitMQ component. Please keep in mind that prior to the Camel 2.17.0 AMQP component supported AMQP 0.9 and above, however since Camel 2.17.0 it supports only AMQP 1.0.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-amqp</artifactId>
  <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>
```

5.1. URI FORMAT

```
amqp:[queue:|topic:]destinationName[?options]
```

5.2. AMQP OPTIONS

You can specify all of the various configuration options of the JMS component after the destination name.

The AMQP component supports 80 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use a shared JMS configuration		JmsConfiguration
acceptMessagesWhileStopping (consumer)	Specifies whether the consumer accept messages while it is stopping. You may consider enabling this option, if you start and stop JMS routes at runtime, while there are still messages enqueued on the queue. If this option is false, and you stop the JMS route, then messages may be rejected, and the JMS broker would have to attempt redeliveries, which yet again may be rejected, and eventually the message may be moved at a dead letter queue on the JMS broker. To avoid this its recommended to enable this option.	false	boolean

Name	Description	Default	Type
allowReplyManagerQuick Stop (consumer)	Whether the DefaultMessageListenerContainer used in the reply managers for request-reply messaging allow the DefaultMessageListenerContainer.runningAllowed flag to quick stop in case JmsConfigurationisAcceptMessagesWhileStopping is enabled, and org.apache.camel.CamelContext is currently being stopped. This quick stop ability is enabled by default in the regular JMS consumers but to enable for reply managers you must enable this flag.	false	boolean
acknowledgementMode (consumer)	The JMS acknowledgement mode defined as an Integer. Allows you to set vendor-specific extensions to the acknowledgment mode.For the regular modes, it is preferable to use the acknowledgementModeName instead.		int
eagerLoadingOfProperties (consumer)	Enables eager loading of JMS properties as soon as a message is loaded which generally is inefficient as the JMS properties may not be required but sometimes can catch early any issues with the underlying JMS provider and the use of JMS properties	false	boolean
acknowledgementModeName (consumer)	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE	AUTO_ACKNOWLEDGE	String
autoStartup (consumer)	Specifies whether the consumer container should auto-startup.	true	boolean
cacheLevel (consumer)	Sets the cache level by ID for the underlying JMS resources. See cacheLevelName option for more details.		int
cacheLevelName (consumer)	Sets the cache level by name for the underlying JMS resources. Possible values are: CACHE_AUTO, CACHE_CONNECTION, CACHE_CONSUMER, CACHE_NONE, and CACHE_SESSION. The default setting is CACHE_AUTO. See the Spring documentation and Transactions Cache Levels for more information.	CACHE_AUTO	String

Name	Description	Default	Type
replyToCacheLevelName (producer)	Sets the cache level by name for the reply consumer when doing request/reply over JMS. This option only applies when using fixed reply queues (not temporary). Camel will by default use: CACHE_CONSUMER for exclusive or shared w/ replyToSelectorName. And CACHE_SESSION for shared without replyToSelectorName. Some JMS brokers such as IBM WebSphere may require to set the replyToCacheLevelName=CACHE_NONE to work. Note: If using temporary queues then CACHE_NONE is not allowed, and you must use a higher value such as CACHE_CONSUMER or CACHE_SESSION.		String
clientId (common)	Sets the JMS client ID to use. Note that this value, if specified, must be unique and can only be used by a single JMS connection instance. It is typically only required for durable topic subscriptions. If using Apache ActiveMQ you may prefer to use Virtual Topics instead.		String
concurrentConsumers (consumer)	Specifies the default number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.	1	int
replyToConcurrent Consumers (producer)	Specifies the default number of concurrent consumers when doing request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.	1	int
connectionFactory (common)	The connection factory to be use. A connection factory must be configured either on the component or endpoint.		ConnectionFactory
username (security)	Username to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String
password (security)	Password to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String

Name	Description	Default	Type
deliveryPersistent (producer)	Specifies whether persistent delivery is used by default.	true	boolean
deliveryMode (producer)	Specifies the delivery mode to be used. Possibles values are those defined by javax.jms.DeliveryMode. NON_PERSISTENT = 1 and PERSISTENT = 2.		Integer
durableSubscriptionName (common)	The durable subscriber name for specifying durable topic subscriptions. The clientId option must be configured as well.		String
exceptionListener (advanced)	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.		ExceptionListener
errorHandler (advanced)	Specifies a org.springframework.util.ErrorHandler to be invoked in case of any uncaught exceptions thrown while processing a Message. By default these exceptions will be logged at the WARN level, if no errorHandler has been configured. You can configure logging level and whether stack traces should be logged using errorHandlerLoggingLevel and errorHandlerLogStackTrace options. This makes it much easier to configure, than having to code a custom errorHandler.		ErrorHandler
errorHandlerLoggingLevel (logging)	Allows to configure the default errorHandler logging level for logging uncaught exceptions.	WARN	LoggingLevel
errorHandlerLogStackTrace (logging)	Allows to control whether stacktraces should be logged or not, by the default errorHandler.	true	boolean
explicitQosEnabled (producer)	Set if the deliveryMode, priority or timeToLive qualities of service should be used when sending messages. This option is based on Spring's JmsTemplate. The deliveryMode, priority and timeToLive options are applied to the current endpoint. This contrasts with the preserveMessageQos option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.	false	boolean
exposeListenerSession (consumer)	Specifies whether the listener session should be exposed when consuming messages.	false	boolean

Name	Description	Default	Type
idleTaskExecutionLimit (advanced)	Specifies the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the maxConcurrentConsumers setting). There is additional doc available from Spring.	1	int
idleConsumerLimit (advanced)	Specify the limit for the number of consumers that are allowed to be idle at any given time.	1	int
maxConcurrentConsumers (consumer)	Specifies the maximum number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToMaxConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.		int
replyToMaxConcurrent Consumers (producer)	Specifies the maximum number of concurrent consumers when using request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.		int
replyOnTimeoutToMax ConcurrentConsumers (producer)	Specifies the maximum number of concurrent consumers for continue routing when timeout occurred when using request/reply over JMS.	1	int
maxMessagesPerTask (advanced)	The number of messages per task. -1 is unlimited. If you use a range for concurrent consumers (eg min max), then this option can be used to set a value to eg 100 to control how fast the consumers will shrink when less work is required.	-1	int
messageConverter (advanced)	To use a custom Spring org.springframework.jms.support.converter.MessageConverter so you can be in control how to map to/from a javax.jms.Message.		MessageConverter
mapJmsMessage (advanced)	Specifies whether Camel should auto map the received JMS message to a suited payload type, such as javax.jms.TextMessage to a String etc.	true	boolean

Name	Description	Default	Type
messageIdEnabled (advanced)	When sending, specifies whether message IDs should be added. This is just an hint to the JMS broker.If the JMS provider accepts this hint, these messages must have the message ID set to null; if the provider ignores the hint, the message ID must be set to its normal unique value	true	boolean
messageTimestampEnabled (advanced)	Specifies whether timestamps should be enabled by default on sending messages. This is just an hint to the JMS broker.If the JMS provider accepts this hint, these messages must have the timestamp set to zero; if the provider ignores the hint the timestamp must be set to its normal value	true	boolean
alwaysCopyMessage (producer)	If true, Camel will always make a JMS message copy of the message when it is passed to the producer for sending. Copying the message is needed in some situations, such as when a replyToDestinationSelectorName is set (incidentally, Camel will set the alwaysCopyMessage option to true, if a replyToDestinationSelectorName is set)	false	boolean
useMessageIDAsCorrelationID (advanced)	Specifies whether JMSMessageID should always be used as JMSCorrelationID for InOut messages.	false	boolean
priority (producer)	Values greater than 1 specify the message priority when sending (where 0 is the lowest priority and 9 is the highest). The explicitQosEnabled option must also be enabled in order for this option to have any effect.	4	int
pubSubNoLocal (advanced)	Specifies whether to inhibit the delivery of messages published by its own connection.	false	boolean
receiveTimeout (advanced)	The timeout for receiving messages (in milliseconds).	1000	long
recoveryInterval (advanced)	Specifies the interval between recovery attempts, i.e. when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.	5000	long
taskExecutor (consumer)	Allows you to specify a custom task executor for consuming messages.		TaskExecutor
timeToLive (producer)	When sending messages, specifies the time-to-live of the message (in milliseconds).	-1	long

Name	Description	Default	Type
transacted (transaction)	Specifies whether to use transacted mode	false	boolean
lazyCreateTransaction Manager (transaction)	If true, Camel will create a JmsTransactionManager, if there is no transactionManager injected when option transacted=true.	true	boolean
transactionManager (transaction)	The Spring transaction manager to use.		PlatformTransaction Manager
transactionName (transaction)	The name of the transaction to use.		String
transactionTimeout (transaction)	The timeout value of the transaction (in seconds), if using transacted mode.	-1	int
testConnectionOn Startup (common)	Specifies whether to test the connection on startup. This ensures that when Camel starts that all the JMS consumers have a valid connection to the JMS broker. If a connection cannot be granted then Camel throws an exception on startup. This ensures that Camel is not started with failed connections. The JMS producers is tested as well.	false	boolean
asyncStartListener (advanced)	Whether to startup the JmsConsumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean
asyncStopListener (advanced)	Whether to stop the JmsConsumer message listener asynchronously, when stopping a route.	false	boolean
forceSendOriginal Message (producer)	When using mapJmsMessage=false Camel will create a new JMS message to send to a new JMS destination if you touch the headers (get or set) during the route. Set this option to true to force Camel to send the original JMS message that was received.	false	boolean

Name	Description	Default	Type
requestTimeout (producer)	The timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds). The default is 20 seconds. You can include the header CamelJmsRequestTimeout to override this endpoint configured timeout value, and thus have per message individual timeout values. See also the requestTimeoutCheckerInterval option.	20000	long
requestTimeoutChecker Interval (advanced)	Configures how often Camel should check for timed out Exchanges when doing request/reply over JMS. By default Camel checks once per second. But if you must react faster when a timeout occurs, then you can lower this interval, to check more frequently. The timeout is determined by the option requestTimeout.	1000	long
transferExchange (advanced)	You can transfer the exchange over the wire instead of just the body and headers. The following fields are transferred: In body, Out body, Fault body, In headers, Out headers, Fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level. You must enable this option on both the producer and consumer side, so Camel knows the payloads is an Exchange and not a regular payload.	false	boolean
transferException (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed on the consumer side, then the caused Exception will be send back in response as a javax.jms.ObjectMessage. If the client is Camel, the returned Exception is rethrown. This allows you to use Camel JMS as a bridge in your routing - for example, using persistent queues to enable robust routing. Notice that if you also have transferExchange enabled, this option takes precedence. The caught exception is required to be serializable. The original Exception on the consumer side can be wrapped in an outer exception such as org.apache.camel.RuntimeCamelException when returned to the producer.	false	boolean

Name	Description	Default	Type
transferFault (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed with a SOAP fault (not exception) on the consumer side, then the fault flag on MessageisFault() will be send back in the response as a JMS header with the key org.apache.camel.component.jms.JmsConstantsJMS_TRANSFER_FAULTJMS_TRANSFER_FAULT. If the client is Camel, the returned fault flag will be set on the link org.apache.camel.MessagesetFault(boolean). You may want to enable this when using Camel components that support faults such as SOAP based such as cxf or spring-ws.	false	boolean
jmsOperations (advanced)	Allows you to use your own implementation of the org.springframework.jms.core.JmsOperations interface. Camel uses JmsTemplate as default. Can be used for testing purpose, but not used much as stated in the spring API docs.		JmsOperations
destinationResolver (advanced)	A pluggable org.springframework.jms.support.destination.DestinationResolver that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).		DestinationResolver
replyToType (producer)	Allows for explicitly specifying which kind of strategy to use for replyTo queues when doing request/reply over JMS. Possible values are: Temporary, Shared, or Exclusive. By default Camel will use temporary queues. However if replyTo has been configured, then Shared is used by default. This option allows you to use exclusive queues instead of shared ones. See Camel JMS documentation for more details, and especially the notes about the implications if running in a clustered environment, and the fact that Shared reply queues has lower performance than its alternatives Temporary and Exclusive.		ReplyToType

Name	Description	Default	Type
preserveMessageQos (producer)	Set to true, if you want to send message using the QoS settings specified on the message, instead of the QoS settings on the JMS endpoint. The following three headers are considered JMSPriority, JMSDeliveryMode, and JMSExpiration. You can provide all or only some of them. If not provided, Camel will fall back to use the values from the endpoint instead. So, when using this option, the headers override the values from the endpoint. The explicitQosEnabled option, by contrast, will only use options set on the endpoint, and not values from the message header.	false	boolean
asyncConsumer (consumer)	Whether the JmsConsumer processes the Exchange asynchronously. If enabled then the JmsConsumer may pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). This means that messages may be processed not 100% strictly in order. If disabled (as default) then the Exchange is fully processed before the JmsConsumer will pickup the next message from the JMS queue. Note if transacted has been enabled, then asyncConsumer=true does not run asynchronously, as transaction must be executed synchronously (Camel 3.0 may support async transactions).	false	boolean
allowNullBody (producer)	Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.	true	boolean
includeSentJMSMessageID (producer)	Only applicable when sending to JMS destination using InOnly (eg fire and forget). Enabling this option will enrich the Camel Exchange with the actual JMSMessageID that was used by the JMS client when the message was sent to the JMS destination.	false	boolean
includeAllJMSXProperties (advanced)	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.	false	boolean

Name	Description	Default	Type
defaultTaskExecutor Type (consumer)	Specifies what default TaskExecutor type to use in the DefaultMessageListenerContainer, for both consumer endpoints and the ReplyTo consumer of producer endpoints. Possible values: SimpleAsync (uses Spring's SimpleAsyncTaskExecutor) or ThreadPool (uses Spring's ThreadPoolTaskExecutor with optimal values - cached threadpool-like). If not set, it defaults to the previous behaviour, which uses a cached thread pool for consumer endpoints and SimpleAsync for reply consumers. The use of ThreadPool is recommended to reduce thread trash in elastic configurations with dynamically increasing and decreasing concurrent consumers.		DefaultTaskExecutor Type
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		JmsKeyFormatStrategy
allowAdditionalHeaders (producer)	This option is used to allow additional headers which may have values that are invalid according to JMS specification. For example some message systems such as WMQ do this with header names using prefix JMS_IBM_MQMD_ containing values with byte array or other invalid types. You can specify multiple header names separated by comma, and use as suffix for wildcard matching.		String
queueBrowseStrategy (advanced)	To use a custom QueueBrowseStrategy when browsing queues		QueueBrowseStrategy
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
waitForProvisionCorrelationToBeUpdated Counter (advanced)	Number of times to wait for provisional correlation id to be updated to the actual correlation id when doing request/reply over JMS and when the option useMessageIDAsCorrelationID is enabled.	50	int

Name	Description	Default	Type
waitForProvisionCorrelationToBeUpdatedThreadSleepingTime (advanced)	Interval in millis to sleep each time while waiting for provisional correlation id to be updated.	100	long
correlationProperty (producer)	Use this JMS property to correlate messages in InOut exchange pattern (request-reply) instead of JMSCorrelationID property. This allows you to exchange messages with systems that do not correlate messages using JMSCorrelationID JMS property. If used JMSCorrelationID will not be used or set by Camel. The value of here named property will be generated if not supplied in the header of the message under the same name.		String
subscriptionDurable (consumer)	Set whether to make the subscription durable. The durable subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a durable subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well.	false	boolean
subscriptionShared (consumer)	Set whether to make the subscription shared. The shared subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a shared subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Note that shared subscriptions may also be durable, so this flag can (and often will) be combined with subscriptionDurable as well. Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well. Requires a JMS 2.0 compatible message broker.	false	boolean

Name	Description	Default	Type
subscriptionName (consumer)	Set the name of a subscription to create. To be applied in case of a topic (pub-sub domain) with a shared or durable subscription. The subscription name needs to be unique within this client's JMS client id. Default is the class name of the specified message listener. Note: Only 1 concurrent consumer (which is the default of this message listener container) is allowed for each subscription, except for a shared subscription (which requires JMS 2.0).		String
streamMessageType Enabled (producer)	Sets whether StreamMessage type is enabled or not. Message payloads of streaming kind such as files, InputStream, etc will either by sent as BytesMessage or StreamMessage. This option controls which kind will be used. By default BytesMessage is used which enforces the entire message payload to be read into memory. By enabling this option the message payload is read into memory in chunks and each chunk is then written to the StreamMessage until no more data.	false	boolean
formatDateHeadersTo Iso8601 (producer)	Sets whether date headers should be formatted according to the ISO 8601 standard.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AMQP endpoint is configured using URI syntax:

```
amqp:destinationType:destinationName
```

with the following path and query parameters:

5.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
destinationType	The kind of destination to use	queue	String

Name	Description	Default	Type
destinationName	**Required** Name of the queue or topic to use as destination		String

5.2.2. Query Parameters (91 parameters):

Name	Description	Default	Type
clientId (common)	Sets the JMS client ID to use. Note that this value, if specified, must be unique and can only be used by a single JMS connection instance. It is typically only required for durable topic subscriptions. If using Apache ActiveMQ you may prefer to use Virtual Topics instead.		String
connectionFactory (common)	The connection factory to be use. A connection factory must be configured either on the component or endpoint.		ConnectionFactory
disableReplyTo (common)	Specifies whether Camel ignores the JMSReplyTo header in messages. If true, Camel does not send a reply back to the destination specified in the JMSReplyTo header. You can use this option if you want Camel to consume from a route and you do not want Camel to automatically send back a reply message because another component in your code handles the reply message. You can also use this option if you want to use Camel as a proxy between different message brokers and you want to route message from one system to another.	false	boolean
durableSubscriptionName (common)	The durable subscriber name for specifying durable topic subscriptions. The clientId option must be configured as well.		String
jmsMessageType (common)	Allows you to force the use of a specific javax.jms.Message implementation for sending JMS messages. Possible values are: Bytes, Map, Object, Stream, Text. By default, Camel would determine which JMS message type to use from the In body type. This option allows you to specify it.		JmsMessageType

Name	Description	Default	Type
testConnectionOnStartup (common)	Specifies whether to test the connection on startup. This ensures that when Camel starts that all the JMS consumers have a valid connection to the JMS broker. If a connection cannot be granted then Camel throws an exception on startup. This ensures that Camel is not started with failed connections. The JMS producers is tested as well.	false	boolean
acknowledgementModeName (consumer)	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE	AUTO_ACKNOWLEDGE	String
asyncConsumer (consumer)	Whether the JmsConsumer processes the Exchange asynchronously. If enabled then the JmsConsumer may pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). This means that messages may be processed not 100% strictly in order. If disabled (as default) then the Exchange is fully processed before the JmsConsumer will pickup the next message from the JMS queue. Note if transacted has been enabled, then asyncConsumer=true does not run asynchronously, as transaction must be executed synchronously (Camel 3.0 may support async transactions).	false	boolean
autoStartup (consumer)	Specifies whether the consumer container should auto-startup.	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
cacheLevel (consumer)	Sets the cache level by ID for the underlying JMS resources. See cacheLevelName option for more details.		int

Name	Description	Default	Type
cacheLevelName (consumer)	Sets the cache level by name for the underlying JMS resources. Possible values are: CACHE_AUTO, CACHE_CONNECTION, CACHE_CONSUMER, CACHE_NONE, and CACHE_SESSION. The default setting is CACHE_AUTO. See the Spring documentation and Transactions Cache Levels for more information.	CACHE_AUTO	String
concurrentConsumers (consumer)	Specifies the default number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.	1	int
maxConcurrentConsumers (consumer)	Specifies the maximum number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToMaxConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.		int
replyTo (consumer)	Provides an explicit ReplyTo destination, which overrides any incoming value of Message.getJMSReplyTo().		String
replyToDeliveryPersistent (consumer)	Specifies whether to use persistent delivery by default for replies.	true	boolean
selector (consumer)	Sets the JMS selector to use		String

Name	Description	Default	Type
subscriptionDurable (consumer)	Set whether to make the subscription durable. The durable subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a durable subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well.	false	boolean
subscriptionName (consumer)	Set the name of a subscription to create. To be applied in case of a topic (pub-sub domain) with a shared or durable subscription. The subscription name needs to be unique within this client's JMS client id. Default is the class name of the specified message listener. Note: Only 1 concurrent consumer (which is the default of this message listener container) is allowed for each subscription, except for a shared subscription (which requires JMS 2.0).		String
subscriptionShared (consumer)	Set whether to make the subscription shared. The shared subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a shared subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Note that shared subscriptions may also be durable, so this flag can (and often will) be combined with subscriptionDurable as well. Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well. Requires a JMS 2.0 compatible message broker.	false	boolean
acceptMessagesWhileStopping (consumer)	Specifies whether the consumer accept messages while it is stopping. You may consider enabling this option, if you start and stop JMS routes at runtime, while there are still messages enqueued on the queue. If this option is false, and you stop the JMS route, then messages may be rejected, and the JMS broker would have to attempt redeliveries, which yet again may be rejected, and eventually the message may be moved at a dead letter queue on the JMS broker. To avoid this its recommended to enable this option.	false	boolean

Name	Description	Default	Type
allowReplyManagerQuickStop (consumer)	Whether the DefaultMessageListenerContainer used in the reply managers for request-reply messaging allow the DefaultMessageListenerContainer.runningAllowed flag to quick stop in case JmsConfigurationisAcceptMessagesWhileStopping is enabled, and org.apache.camel.CamelContext is currently being stopped. This quick stop ability is enabled by default in the regular JMS consumers but to enable for reply managers you must enable this flag.	false	boolean
consumerType (consumer)	The consumer type to use, which can be one of: Simple, Default, or Custom. The consumer type determines which Spring JMS listener to use. Default will use org.springframework.jms.listener.DefaultMessageListenerContainer, Simple will use org.springframework.jms.listener.SimpleMessageListenerContainer. When Custom is specified, the MessageListenerContainerFactory defined by the messageListenerContainerFactory option will determine what org.springframework.jms.listener.AbstractMessageListenerContainer to use.	Default	ConsumerType
defaultTaskExecutorType (consumer)	Specifies what default TaskExecutor type to use in the DefaultMessageListenerContainer, for both consumer endpoints and the ReplyTo consumer of producer endpoints. Possible values: SimpleAsync (uses Spring's SimpleAsyncTaskExecutor) or ThreadPool (uses Spring's ThreadPoolTaskExecutor with optimal values - cached threadpool-like). If not set, it defaults to the previous behaviour, which uses a cached thread pool for consumer endpoints and SimpleAsync for reply consumers. The use of ThreadPool is recommended to reduce thread trash in elastic configurations with dynamically increasing and decreasing concurrent consumers.		DefaultTaskExecutor Type
eagerLoadingOfProperties (consumer)	Enables eager loading of JMS properties and payload as soon as a message is loaded which generally is inefficient as the JMS properties may not be required but sometimes can catch early any issues with the underlying JMS provider and the use of JMS properties	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
exposeListenerSession (consumer)	Specifies whether the listener session should be exposed when consuming messages.	false	boolean
replyToSameDestinationAllowed (consumer)	Whether a JMS consumer is allowed to send a reply message to the same destination that the consumer is using to consume from. This prevents an endless loop by consuming and sending back the same message to itself.	false	boolean
taskExecutor (consumer)	Allows you to specify a custom task executor for consuming messages.		TaskExecutor
deliveryMode (producer)	Specifies the delivery mode to be used. Possibles values are those defined by javax.jms.DeliveryMode. NON_PERSISTENT = 1 and PERSISTENT = 2.		Integer
deliveryPersistent (producer)	Specifies whether persistent delivery is used by default.	true	boolean
explicitQosEnabled (producer)	Set if the deliveryMode, priority or timeToLive qualities of service should be used when sending messages. This option is based on Spring's JmsTemplate. The deliveryMode, priority and timeToLive options are applied to the current endpoint. This contrasts with the preserveMessageQos option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.	false	Boolean
formatDateHeadersToIso8601 (producer)	Sets whether JMS date properties should be formatted according to the ISO 8601 standard.	false	boolean

Name	Description	Default	Type
preserveMessageQos (producer)	Set to true, if you want to send message using the QoS settings specified on the message, instead of the QoS settings on the JMS endpoint. The following three headers are considered JMSPriority, JMSDeliveryMode, and JMSExpiration. You can provide all or only some of them. If not provided, Camel will fall back to use the values from the endpoint instead. So, when using this option, the headers override the values from the endpoint. The explicitQosEnabled option, by contrast, will only use options set on the endpoint, and not values from the message header.	false	boolean
priority (producer)	Values greater than 1 specify the message priority when sending (where 0 is the lowest priority and 9 is the highest). The explicitQosEnabled option must also be enabled in order for this option to have any effect.	4	int
replyToConcurrentConsumers (producer)	Specifies the default number of concurrent consumers when doing request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.	1	int
replyToMaxConcurrent Consumers (producer)	Specifies the maximum number of concurrent consumers when using request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.		int
replyToOnTimeoutMax ConcurrentConsumers (producer)	Specifies the maximum number of concurrent consumers for continue routing when timeout occurred when using request/reply over JMS.	1	int
replyToOverride (producer)	Provides an explicit ReplyTo destination in the JMS message, which overrides the setting of replyTo. It is useful if you want to forward the message to a remote Queue and receive the reply message from the ReplyTo destination.		String

Name	Description	Default	Type
replyToType (producer)	Allows for explicitly specifying which kind of strategy to use for replyTo queues when doing request/reply over JMS. Possible values are: Temporary, Shared, or Exclusive. By default Camel will use temporary queues. However if replyTo has been configured, then Shared is used by default. This option allows you to use exclusive queues instead of shared ones. See Camel JMS documentation for more details, and especially the notes about the implications if running in a clustered environment, and the fact that Shared reply queues has lower performance than its alternatives Temporary and Exclusive.		ReplyToType
requestTimeout (producer)	The timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds). The default is 20 seconds. You can include the header CamelJmsRequestTimeout to override this endpoint configured timeout value, and thus have per message individual timeout values. See also the requestTimeoutCheckerInterval option.	20000	long
timeToLive (producer)	When sending messages, specifies the time-to-live of the message (in milliseconds).	-1	long
allowAdditionalHeaders (producer)	This option is used to allow additional headers which may have values that are invalid according to JMS specification. For example some message systems such as WMQ do this with header names using prefix JMS_IBM_MQMD_ containing values with byte array or other invalid types. You can specify multiple header names separated by comma, and use as suffix for wildcard matching.		String
allowNullBody (producer)	Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.	true	boolean
alwaysCopyMessage (producer)	If true, Camel will always make a JMS message copy of the message when it is passed to the producer for sending. Copying the message is needed in some situations, such as when a replyToDestinationSelectorName is set (incidentally, Camel will set the alwaysCopyMessage option to true, if a replyToDestinationSelectorName is set)	false	boolean

Name	Description	Default	Type
correlationProperty (producer)	When using InOut exchange pattern use this JMS property instead of JMSCorrelationID JMS property to correlate messages. If set messages will be correlated solely on the value of this property JMSCorrelationID property will be ignored and not set by Camel.		String
disableTimeToLive (producer)	Use this option to force disabling time to live. For example when you do request/reply over JMS, then Camel will by default use the requestTimeout value as time to live on the message being sent. The problem is that the sender and receiver systems have to have their clocks synchronized, so they are in sync. This is not always so easy to archive. So you can use disableTimeToLive=true to not set a time to live value on the sent message. Then the message will not expire on the receiver system. See below in section About time to live for more details.	false	boolean
forceSendOriginalMessage (producer)	When using mapJmsMessage=false Camel will create a new JMS message to send to a new JMS destination if you touch the headers (get or set) during the route. Set this option to true to force Camel to send the original JMS message that was received.	false	boolean
includeSentJMSMessageID (producer)	Only applicable when sending to JMS destination using InOnly (eg fire and forget). Enabling this option will enrich the Camel Exchange with the actual JMSMessageID that was used by the JMS client when the message was sent to the JMS destination.	false	boolean
replyToCacheLevelName (producer)	Sets the cache level by name for the reply consumer when doing request/reply over JMS. This option only applies when using fixed reply queues (not temporary). Camel will by default use: CACHE_CONSUMER for exclusive or shared w/ replyToSelectorName. And CACHE_SESSION for shared without replyToSelectorName. Some JMS brokers such as IBM WebSphere may require to set the replyToCacheLevelName=CACHE_NONE to work. Note: If using temporary queues then CACHE_NONE is not allowed, and you must use a higher value such as CACHE_CONSUMER or CACHE_SESSION.		String
replyToDestinationSelectorName (producer)	Sets the JMS Selector using the fixed name to be used so you can filter out your own replies from the others when using a shared queue (that is, if you are not using a temporary reply queue).		String

Name	Description	Default	Type
streamMessageTypeEnabled (producer)	Sets whether StreamMessage type is enabled or not. Message payloads of streaming kind such as files, InputStream, etc will either by sent as BytesMessage or StreamMessage. This option controls which kind will be used. By default BytesMessage is used which enforces the entire message payload to be read into memory. By enabling this option the message payload is read into memory in chunks and each chunk is then written to the StreamMessage until no more data.	false	boolean
allowSerializedHeaders (advanced)	Controls whether or not to include serialized headers. Applies only when transferExchange is true. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
asyncStartListener (advanced)	Whether to startup the JmsConsumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean
asyncStopListener (advanced)	Whether to stop the JmsConsumer message listener asynchronously, when stopping a route.	false	boolean
destinationResolver (advanced)	A pluggable org.springframework.jms.support.destination.DestinationResolver that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).		DestinationResolver

Name	Description	Default	Type
errorHandler (advanced)	Specifies a org.springframework.util.ErrorHandler to be invoked in case of any uncaught exceptions thrown while processing a Message. By default these exceptions will be logged at the WARN level, if no errorHandler has been configured. You can configure logging level and whether stack traces should be logged using errorHandlerLoggingLevel and errorHandlerLogStackTrace options. This makes it much easier to configure, than having to code a custom errorHandler.		ErrorHandler
exceptionListener (advanced)	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.		ExceptionListener
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
idleConsumerLimit (advanced)	Specify the limit for the number of consumers that are allowed to be idle at any given time.	1	int
idleTaskExecutionLimit (advanced)	Specifies the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the maxConcurrentConsumers setting). There is additional doc available from Spring.	1	int
includeAllJMSXProperties (advanced)	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.	false	boolean
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		String

Name	Description	Default	Type
mapJmsMessage (advanced)	Specifies whether Camel should auto map the received JMS message to a suited payload type, such as javax.jms.TextMessage to a String etc.	true	boolean
maxMessagesPerTask (advanced)	The number of messages per task. -1 is unlimited. If you use a range for concurrent consumers (eg min max), then this option can be used to set a value to eg 100 to control how fast the consumers will shrink when less work is required.	-1	int
messageConverter (advanced)	To use a custom Spring org.springframework.jms.support.converter.MessageConverter so you can be in control how to map to/from a javax.jms.Message.		MessageConverter
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
messageIdEnabled (advanced)	When sending, specifies whether message IDs should be added. This is just an hint to the JMS broker.If the JMS provider accepts this hint, these messages must have the message ID set to null; if the provider ignores the hint, the message ID must be set to its normal unique value	true	boolean
messageListenerContainerFactory (advanced)	Registry ID of the MessageListenerContainerFactory used to determine what org.springframework.jms.listener.AbstractMessageListenerContainer to use to consume messages. Setting this will automatically set consumerType to Custom.		MessageListenerContainerFactory
messageTimestampEnabled (advanced)	Specifies whether timestamps should be enabled by default on sending messages. This is just an hint to the JMS broker.If the JMS provider accepts this hint, these messages must have the timestamp set to zero; if the provider ignores the hint the timestamp must be set to its normal value	true	boolean
pubSubNoLocal (advanced)	Specifies whether to inhibit the delivery of messages published by its own connection.	false	boolean
receiveTimeout (advanced)	The timeout for receiving messages (in milliseconds).	1000	long

Name	Description	Default	Type
recoveryInterval (advanced)	Specifies the interval between recovery attempts, i.e. when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.	5000	long
requestTimeoutChecker Interval (advanced)	Configures how often Camel should check for timed out Exchanges when doing request/reply over JMS. By default Camel checks once per second. But if you must react faster when a timeout occurs, then you can lower this interval, to check more frequently. The timeout is determined by the option requestTimeout.	1000	long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
transferException (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed on the consumer side, then the caused Exception will be send back in response as a javax.jms.ObjectMessage. If the client is Camel, the returned Exception is rethrown. This allows you to use Camel JMS as a bridge in your routing - for example, using persistent queues to enable robust routing. Notice that if you also have transferExchange enabled, this option takes precedence. The caught exception is required to be serializable. The original Exception on the consumer side can be wrapped in an outer exception such as org.apache.camel.RuntimeCamelException when returned to the producer.	false	boolean
transferExchange (advanced)	You can transfer the exchange over the wire instead of just the body and headers. The following fields are transferred: In body, Out body, Fault body, In headers, Out headers, Fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level. You must enable this option on both the producer and consumer side, so Camel knows the payloads is an Exchange and not a regular payload.	false	boolean

Name	Description	Default	Type
transferFault (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed with a SOAP fault (not exception) on the consumer side, then the fault flag on MessageisFault() will be send back in the response as a JMS header with the key org.apache.camel.component.jms.JmsConstantsJMS_TRANSFER_FAULTJMS_TRANSFER_FAULT. If the client is Camel, the returned fault flag will be set on the link org.apache.camel.MessagesetFault(boolean). You may want to enable this when using Camel components that support faults such as SOAP based such as cxf or spring-ws.	false	boolean
useMessageIDAsCorrelation ID (advanced)	Specifies whether JMSMessageID should always be used as JMSCorrelationID for InOut messages.	false	boolean
waitForProvisionCorrelationToBeUpdatedCounter (advanced)	Number of times to wait for provisional correlation id to be updated to the actual correlation id when doing request/reply over JMS and when the option useMessageIDAsCorrelationID is enabled.	50	int
waitForProvisionCorrelationToBeUpdatedThreadSleeping Time (advanced)	Interval in millis to sleep each time while waiting for provisional correlation id to be updated.	100	long
errorHandlerLoggingLevel (logging)	Allows to configure the default errorHandler logging level for logging uncaught exceptions.	WARN	LoggingLevel
errorHandlerLogStackTrace (logging)	Allows to control whether stacktraces should be logged or not, by the default errorHandler.	true	boolean
password (security)	Password to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String
username (security)	Username to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String
transacted (transaction)	Specifies whether to use transacted mode	false	boolean

Name	Description	Default	Type
lazyCreateTransaction Manager (transaction)	If true, Camel will create a JmsTransactionManager, if there is no transactionManager injected when option transacted=true.	true	boolean
transactionManager (transaction)	The Spring transaction manager to use.		PlatformTransaction Manager
transactionName (transaction)	The name of the transaction to use.		String
transactionTimeout (transaction)	The timeout value of the transaction (in seconds), if using transacted mode.	-1	int

5.3. USAGE

As AMQP component is inherited from JMS component, the usage of the former is almost identical to the latter:

Using AMQP component

```
// Consuming from AMQP queue
from("amqp:queue:incoming").
  to(...);

// Sending message to the AMQP topic
from(...).
  to("amqp:topic:notify");
```

5.4. CONFIGURING AMQP COMPONENT

Starting from the Camel 2.16.1 you can also use the **AMQPComponent#amqp10Component(String connectionURI)** factory method to return the AMQP 1.0 component with the pre-configured topic prefix:

Creating AMQP 1.0 component

```
 AMQPComponent amqp =
AMQPComponent.amqp10Component("amqp://guest:guest@localhost:5672");
```

Keep in mind that starting from the Camel 2.17 the **AMQPComponent#amqp10Component(String connectionURI)** factory method has been deprecated on the behalf of the **AMQPComponent#amqpComponent(String connectionURI)**:

Creating AMQP 1.0 component

```
AMQPComponent amqp = AMQPComponent.amqpComponent("amqp://localhost:5672");

AMQPComponent authorizedAmqp = AMQPComponent.amqpComponent("amqp://localhost:5672",
```

```
"user", "password");
```

Starting from Camel 2.17, in order to automatically configure the AMQP component, you can also add an instance of **org.apache.camel.component.amqp.AMQPConnectionDetails** to the registry. For example for Spring Boot you just have to define bean:

AMQP connection details auto-configuration

```
@Bean
AMQPConnectionDetails amqpConnection() {
  return new AMQPConnectionDetails("amqp://localhost:5672");
}

@Bean
AMQPConnectionDetails securedAmqpConnection() {
  return new AMQPConnectionDetails("amqp://lcoalhost:5672", "username", "password");
}
```

Likewise, you can also use CDI producer methods when using Camel-CDI

AMQP connection details auto-configuration for CDI

```
@Produces
AMQPConnectionDetails amqpConnection() {
  return new AMQPConnectionDetails("amqp://localhost:5672");
}
```

You can also rely on the Camel properties to read the AMQP connection details. Factory method **AMQPConnectionDetails.discoverAMQP()** attempts to read Camel properties in a Kubernetes-like convention, just as demonstrated on the snippet below:

AMQP connection details auto-configuration

```
export AMQP_SERVICE_HOST = "mybroker.com"
export AMQP_SERVICE_PORT = "6666"
export AMQP_SERVICE_USERNAME = "username"
export AMQP_SERVICE_PASSWORD = "password"

...

@Bean
AMQPConnectionDetails amqpConnection() {
  return AMQPConnectionDetails.discoverAMQP();
}
```

Enabling AMQP specific options

If you, for example, need to enable **amqp.traceFrames** you can do that by appending the option to your URI, like the following example:

```
AMQPComponent amqp = AMQPComponent.amqpComponent("amqp://localhost:5672?
amqp.traceFrames=true");
```

For reference take a look at the QPID JMS client configuration

5.5. USING TOPICS

To have using topics working with **camel-amqp** you need to configure the component to use **topic://** as topic prefix, as shown below:

```
<bean id="amqp" class="org.apache.camel.component.amqp.AmqpComponent">
  <property name="connectionFactory">
    <bean class="org.apache.qpid.jms.JmsConnectionFactory" factory-method="createFromURL">
      <property name="remoteURI" value="amqp://localhost:5672" />
      <property name="topicPrefix" value="topic://" />  <!-- only necessary when connecting to
ActiveMQ over AMQP 1.0 -->
    </bean>
  </property>
 </bean>
```

Keep in mind that both **AMQPComponent#amqpComponent()** methods and **AMQPConnectionDetails** pre-configure the component with the topic prefix, so you don't have to configure it explicitly.

5.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 6. APNS COMPONENT

Available as of Camel version 2.8

The **apns** component is used for sending notifications to iOS devices. The apns components use javapns library.
The component supports sending notifications to Apple Push Notification Servers (APNS) and consuming feedback from the servers.

The consumer is configured with 3600 seconds for polling by default because it is a best practice to consume feedback stream from Apple Push Notification Servers only from time to time. For example: every 1 hour to avoid flooding the servers.

The feedback stream gives informations about inactive devices. You only need to get this informations every some hours if your mobile application is not a heavily used one.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-apns</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

6.1. URI FORMAT

To send notifications:

```
apns:notify[?options]
```

To consume feedback:

```
apns:consumer[?options]
```

6.2. OPTIONS

The APNS component supports 2 options which are listed below.

Name	Description	Default	Type
apnsService (common)	**Required** The ApnsService to use. The org.apache.camel.component.apns.factory.ApnsServiceFactory can be used to build a ApnsService		ApnsService
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The APNS endpoint is configured using URI syntax:

```
apns:name
```

with the following path and query parameters:

6.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	Name of the endpoint		String

6.2.2. Query Parameters (20 parameters):

Name	Description	Default	Type
tokens (common)	Configure this property in case you want to statically declare tokens related to devices you want to notify. Tokens are separated by comma.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service

Name	Description	Default	Type
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

You can append query options to the URI in the following format, **?option=value&option=value&…**

6.2.3. Component

The **ApnsComponent** must be configured with a **com.notnoop.apns.ApnsService**. The service can be created and configured using the **org.apache.camel.component.apns.factory.ApnsServiceFactory**. See further below for an example. And as well in the test source code.

6.2.3.1. SSL Setting

In order to use secure connection, an instance of **org.apache.camel.util.jsse.SSLContextParameters** should be injected to **org.apache.camel.component.apns.factory.ApnsServiceFactory** which is used to configure the component. See the test resources for an example. ssl example

6.3. EXCHANGE DATA FORMAT

When Camel will fetch feedback data corresponding to inactive devices, it will retrieve a List of InactiveDevice objects. Each InactiveDevice object of the retrieved list will be setted as the In body, and then processed by the consumer endpoint.

6.4. MESSAGE HEADERS

Camel Apns uses these headers.

Property	Default	Description
CamelApnsTokens		Empty by default.
CamelApnsMessageType	**STRING, PAYLOAD, APNS_NOTIFICATION**	In case you choose PAYLOAD for the message type, then the message will be considered as a APNS payload and sent as is. In case you choose STRING, message will be converted as a APNS payload. From **Camel 2.16** onwards APNS_NOTIFICATION is used for sending message body as com.notnoop.apns.ApnsNotification types.

6.5. APNSSERVICEFACTORY BUILDER CALLBACK

ApnsServiceFactory comes with the empty callback method that could be used to configure (or even replace) the default **ApnsServiceBuilder** instance. The signature of the method could look as follows:

```
protected ApnsServiceBuilder configureServiceBuilder(ApnsServiceBuilder serviceBuilder);
```

And could be used like as follows:

```
ApnsServiceFactory proxiedApnsServiceFactory = new ApnsServiceFactory(){

  @Override
  protected ApnsServiceBuilder configureServiceBuilder(ApnsServiceBuilder serviceBuilder) {
    return serviceBuilder.withSocksProxy("my.proxy.com", 6666);
  }

};
```

6.6. SAMPLES

6.6.1. Camel Xml route

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

    <!-- Replace by desired values -->
    <bean id="apnsServiceFactory"
class="org.apache.camel.component.apns.factory.ApnsServiceFactory">
```

```
        <!-- Optional configuration of feedback host and port -->
        <!-- <property name="feedbackHost" value="localhost" /> -->
        <!-- <property name="feedbackPort" value="7843" /> -->

        <!-- Optional configuration of gateway host and port -->
        <!-- <property name="gatewayHost" value="localhost" /> -->
        <!-- <property name="gatewayPort" value="7654" /> -->

        <!-- Declaration of certificate used -->
            <!-- from Camel 2.11 onwards you can use prefix: classpath:, file: to refer to load the
certificate from classpath or file. Default it classpath -->
        <property name="certificatePath" value="certificate.p12" />
        <property name="certificatePassword" value="MyCertPassword" />

        <!-- Optional connection strategy - By Default: No need to configure -->
        <!-- Possible options: NON_BLOCKING, QUEUE, POOL or Nothing -->
        <!-- <property name="connectionStrategy" value="POOL" /> -->
        <!-- Optional pool size -->
        <!-- <property name="poolSize" value="15" /> -->

        <!-- Optional connection strategy - By Default: No need to configure -->
        <!-- Possible options: EVERY_HALF_HOUR, EVERY_NOTIFICATION or Nothing (Corresponds
to NEVER javapns option) -->
        <!-- <property name="reconnectionPolicy" value="EVERY_HALF_HOUR" /> -->
    </bean>

    <bean id="apnsService" factory-bean="apnsServiceFactory" factory-method="getApnsService" />

    <!-- Replace this declaration by wanted configuration -->
    <bean id="apns" class="org.apache.camel.component.apns.ApnsComponent">
        <property name="apnsService" ref="apnsService" />
    </bean>

    <camelContext id="camel-apns-test" xmlns="http://camel.apache.org/schema/spring">
            <route id="apns-test">
                <from uri="apns:consumer?initialDelay=10&amp;delay=3600&amp;timeUnit=SECONDS"
/>
                <to uri="log:org.apache.camel.component.apns?showAll=true&amp;multiline=true" />
                <to uri="mock:result" />
            </route>
    </camelContext>

</beans>
```

6.6.2. Camel Java route

Create camel context and declare apns component programmatically

```
    protected CamelContext createCamelContext() throws Exception {
        CamelContext camelContext = super.createCamelContext();

        ApnsServiceFactory apnsServiceFactory = new ApnsServiceFactory();
        apnsServiceFactory.setCertificatePath("classpath:/certificate.p12");
        apnsServiceFactory.setCertificatePassword("MyCertPassword");
```

```
        ApnsService apnsService = apnsServiceFactory.getApnsService(camelContext);

        ApnsComponent apnsComponent = new ApnsComponent(apnsService);
        camelContext.addComponent("apns", apnsComponent);

        return camelContext;
    }
```

[[APNS-ApnsProducer-iOStargetdevicedynamicallyconfiguredviaheader:"CamelApnsTokens"]]
ApnsProducer - iOS target device dynamically configured via header: **"CamelApnsTokens"**

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("direct:test")
                    .setHeader(ApnsConstants.HEADER_TOKENS, constant(IOS_DEVICE_TOKEN))
                    .to("apns:notify");
                }
        }
    }
```

ApnsProducer - iOS target device statically configured via uri

```
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("direct:test").
                to("apns:notify?tokens=" + IOS_DEVICE_TOKEN);
            }
        };
    }
```

ApnsConsumer

```
from("apns:consumer?initialDelay=10&delay=3600&timeUnit=SECONDS")
    .to("log:com.apache.camel.component.apns?showAll=true&multiline=true")
    .to("mock:result");
```

6.7. SEE ALSO

- Component
- Endpoint * Blog about using APNS (in french)

CHAPTER 7. ASN.1 FILE DATAFORMAT

Available as of Camel version 2.20

The ASN.1 Data Format Data Format [Intoduction to ASN.1](https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx) is a Camel Frameworks's data format implementation based on Bouncy Castle's bcprov-jdk15on library and jASN.1's java compiler for the formal notation used for describing data transmitted by telecommunications protocols, regardless of language implementation and physical representation of these data, whatever the application, whether complex or very simple. Messages can be unmarshalled (conversion to simple Java POJO(s)) to plain Java objects. By the help of Camel's routing engine and data transformations you can then play with POJO(s) and apply customised formatting and call other Camel Component's to convert and send messages to upstream systems.

7.1. ASN.1 DATA FORMAT OPTIONS

The ASN.1 File dataformat supports 3 options which are listed below.

Name	Default	Java Type	Description
usingIterator	**false**	**Boolean**	If the asn1 file has more then one entry, the setting this option to true, allows to work with the splitter EIP, to split the data using an iterator in a streaming mode.
clazzName		**String**	Name of class to use when unmarshalling
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

7.2. UNMARSHAL

There are 3 different ways to unmarshal ASN.1 structured messages. (Usually binary files)

In this first example we unmarshal BER file payload to OutputStream and send it to mock endpoint.

```
from("direct:unmarshal").unmarshal(asn1).to("mock:unmarshal");
```

In the second example we unmarshal BER file payload to byte array using Split EIP. The reason for applying Split EIP is that usually each BER file or (ASN.1 structured file) contains multiple records to process and Split EIP helps us to get each record in a file as byte arrays which is actually ASN1Primitive's instance (by the use of Bouncy Castle's ASN.1 support in bcprov-jdk15on library) Byte arrays then may be converted to ASN1Primitive by the help of public static method in (ASN1Primitive.fromByteArray) In such example, note that you need to set **usingIterator=true**

```
from("direct:unmarshal").unmarshal(asn1).split(body(Iterator.class)).streaming().to("mock:unmarshal"
);
```

In the last example we unmarshal BER file payload to plain old Java Objects using Split EIP. The reason for applying Split EIP is already mentioned in the previous example. Please note and keep in mind that reason. In such example we also need to set the fully qualified name of the class or <YourObject>.class reference through data format. The important thing to note here is that your object should have been generated by jasn1 compiler which is a nice tool to generate java object representations of your ASN.1 structure. For the reference usage of jasn1 compiler see [JASN.1 Project Page] (https://www.openmuc.org/asn1/) and please also see how the compiler is invoked with the help of maven's exec plugin. For example, in this data format's unit tests an example ASN.1 structure(TestSMSBerCdr.asn1) is added in **src/test/resources/asn1_structure**. jasn1 compiler is invoked and java object's representations are generated in **${basedir}/target/generated/src/test/java** The nice thing about this example, you will get POJO instance at the mock endpoint or at whatever your endpoint is.

```
from("direct:unmarshaldsl")
        .unmarshal()
        .asn1("org.apache.camel.dataformat.asn1.model.testsmscbercdr.SmsCdr")
        .split(body(Iterator.class)).streaming()
.to("mock:unmarshaldsl");
```

7.3. DEPENDENCIES

To use ASN.1 data format in your camel routes you need to add a dependency on **camel-asn1** which implements this data format.

If you use Maven you can just add the following to your **pom.xml**, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-asn1</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 8. ASTERISK COMPONENT

Available as of Camel version 2.18

The **asterisk:** component allows you to work easily with an Asterisk PBX Server http://www.asterisk.org/ using asterisk-java

This component help to interface with Asterisk Manager Interface

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-asterisk</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

8.1. URI FORMAT

```
asterisk:name[?options]
```

8.2. OPTIONS

The Asterisk component has no options.

The Asterisk endpoint is configured using URI syntax:

```
asterisk:name
```

with the following path and query parameters:

8.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Logical name		String

8.2.2. Query Parameters (8 parameters):

Name	Description	Default	Type
hostname (common)	**Required** The hostname of the asterisk server		String
password (common)	**Required** Login password		String

Name	Description	Default	Type
username (common)	**Required** Login username		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
action (producer)	What action to perform such as getting queue status, sip peers or extension state.		AsteriskAction
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

8.3. ACTION

Supported actions are:

- QUEUE_STATUS, Queue Status
- SIP_PEERS, List SIP Peers
- EXTENSION_STATE, Check Extension Status

CHAPTER 9. ATMOS COMPONENT

Available as of Camel version 2.15

Camel-Atmos is an Apache Camel component that allows you to work with ViPR object data services using the Atmos Client.

```
from("atmos:foo/get?remotePath=/path").to("mock:test");
```

9.1. OPTIONS

The Atmos component supports 5 options which are listed below.

Name	Description	Default	Type
fullTokenId (security)	The token id to pass to the Atmos client		String
secretKey (security)	The secret key to pass to the Atmos client		String
uri (advanced)	The URI of the server for the Atmos client to connect to		String
sslValidation (security)	Whether the Atmos client should perform SSL validation	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atmos endpoint is configured using URI syntax:

```
atmos:name/operation
```

with the following path and query parameters:

9.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
name	Atmos name		String
operation	**Required** Operation to perform		AtmosOperation

9.1.2. Query Parameters (12 parameters):

Name	Description	Default	Type
enableSslValidation (common)	Atmos SSL validation	false	boolean
fullTokenId (common)	Atmos client fullTokenId		String
localPath (common)	Local path to put files		String
newRemotePath (common)	New path on Atmos when moving files		String
query (common)	Search query on Atmos		String
remotePath (common)	Where to put files on Atmos		String
secretKey (common)	Atmos shared secret		String
uri (common)	Atomos server uri		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

9.2. DEPENDENCIES

To use Atmos in your camel routes you need to add the dependency on **camel-atmos** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atmos</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

9.3. INTEGRATIONS

When you look at atmos integrations, there is one type of consumer, GetConsumer, which is a type of ScheduledPollConsumer.

- **Get**

Whereas there are 4 types of producers which are

- **Get**
- **Del**
- **Move**
- **Put**

9.4. EXAMPLES

These example are taken from tests:

```
from("atmos:foo/get?remotePath=/path").to("mock:test");
```

Here, this is a consumer example. **remotePath** represents the path from where the data will be read and passes the camel exchange to regarding producer Underneath, this component uses atmos client API for this and every other operations.

```
from("direct:start")
.to("atmos://get?remotePath=/dummy/dummy.txt")
.to("mock:result");
```

Here, this a producer sample. **remotePath** represents the path where the operations occur on ViPR object data service. In producers, operations(**Get**,**Del**, **Move**,**Put**) run on ViPR object data services and results are set on headers of camel exchange.

Regarding the operations, the following headers are set on camel exhange

```
DOWNLOADED_FILE, DOWNLOADED_FILES, UPLOADED_FILE, UPLOADED_FILES,
FOUND_FILES, DELETED_PATH, MOVED_PATH;
```

9.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 10. ATMOSPHERE WEBSOCKET COMPONENT

Available as of Camel version 2.14

The **atmosphere-websocket:** component provides Websocket based endpoints for a servlet communicating with external clients over Websocket (as a servlet accepting websocket connections from external clients).
The component uses the SERVLET component and uses the Atmosphere library to support the Websocket transport in various Servlet containers (e..g., Jetty, Tomcat, ...).

Unlike the Websocket component that starts the embedded Jetty server, this component uses the servlet provider of the container.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atmosphere-websocket</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

10.1. ATMOSPHERE-WEBSOCKET OPTIONS

The Atmosphere Websocket component supports 8 options which are listed below.

Name	Description	Default	Type
servletName (common)	Default name of servlet to use. The default name is CamelServlet.		String
httpRegistry (common)	To use a custom org.apache.camel.component.servlet.HttpRegistry.		HttpRegistry
attachmentMultipart Binding (common)	Whether to automatic bind multipart/form-data as attachments on the Camel Exchange. The options attachmentMultipartBinding=true and disableStreamCache=false cannot work together. Remove disableStreamCache to use AttachmentMultipartBinding. This is turn off by default as this may require servlet specific configuration to enable this when using Servlet's.	false	boolean
httpBinding (advanced)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
httpConfiguration (advanced)	To use the shared HttpConfiguration as base configuration.		HttpConfiguration

Name	Description	Default	Type
allowJavaSerialized Object (advanced)	Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atmosphere Websocket endpoint is configured using URI syntax:

```
atmosphere-websocket:servicePath
```

with the following path and query parameters:

10.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
servicePath	**Required** Name of websocket endpoint		String

10.1.2. Query Parameters (37 parameters):

Name	Description	Default	Type
chunked (common)	If this option is false the Servlet will disable the HTTP streaming and set the content-length header on the response	true	boolean

Name	Description	Default	Type
disableStreamCache (common)	Determines whether or not the raw input stream from Servlet is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Servlet input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times. If you use Servlet to bridge/proxy an endpoint then consider enabling this option to improve performance, in case you do not need to read the message payload multiple times. The http/http4 producer will by default cache the response body stream. If setting this option to true, then the producers will not cache the response body stream but use the response stream as-is as the message body.	false	boolean
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
sendToAll (common)	Whether to send to all (broadcast) or send to a single receiver.	false	boolean
transferException (common)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
useStreaming (common)	To enable streaming to send data as multiple text fragments.	false	boolean
httpBinding (common)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
async (consumer)	Configure the consumer to work in async mode	false	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
httpMethodRestrict (consumer)	Used to only allow consuming if the HttpMethod matches, such as GET/POST/PUT etc. Multiple methods can be specified separated by comma.		String
matchOnUriPrefix (consumer)	Whether or not the consumer should try to find a target consumer by matching the URI prefix if no exact match is found.	false	boolean
responseBufferSize (consumer)	To use a custom buffer size on the javax.servlet.ServletResponse.		Integer
servletName (consumer)	Name of the servlet to use	Camel Servlet	String
attachmentMultipartBinding (consumer)	Whether to automatic bind multipart/form-data as attachments on the Camel Exchange. The options attachmentMultipartBinding=true and disableStreamCache=false cannot work together. Remove disableStreamCache to use AttachmentMultipartBinding. This is turn off by default as this may require servlet specific configuration to enable this when using Servlet's.	false	boolean
eagerCheckContentAvailable (consumer)	Whether to eager check whether the HTTP requests has content if the content-length header is 0 or not present. This can be turned on in case HTTP clients do not send streamed data.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
optionsEnabled (consumer)	Specifies whether to enable HTTP OPTIONS for this Servlet consumer. By default OPTIONS is turned off.	false	boolean
traceEnabled (consumer)	Specifies whether to enable HTTP TRACE for this Servlet consumer. By default TRACE is turned off.	false	boolean
bridgeEndpoint (producer)	If the option is true, HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the option throwExceptionOnFailure to be false to let the HttpProducer send all the fault response back.	false	boolean
connectionClose (producer)	Specifies whether a Connection Close header must be added to HTTP Request. By default connectionClose is false.	false	boolean
copyHeaders (producer)	If this option is true then IN exchange headers will be copied to OUT exchange headers according to copy strategy. Setting this to false, allows to only include the headers from the HTTP response (not propagating IN headers).	true	boolean
httpMethod (producer)	Configure the HTTP method to use. The HttpMethod header cannot override this option if set.		HttpMethods
ignoreResponseBody (producer)	If this option is true, The http producer won't read response body and cache the input stream	false	boolean
preserveHostHeader (producer)	If the option is true, HttpProducer will set the Host header to the value contained in the current exchange Host header, useful in reverse proxy applications where you want the Host header received by the downstream server to reflect the URL called by the upstream client, this allows applications which use the Host header to generate accurate URL's for a proxied service	false	boolean
throwExceptionOnFailure (producer)	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler

Name	Description	Default	Type
okStatusCodeRange (producer)	The status codes which are considered a success response. The values are inclusive. Multiple ranges can be defined, separated by comma, e.g. 200-204,209,301-304. Each range must be a single number or from-to with the dash included.	200-299	String
urlRewrite (producer)	**Deprecated** Refers to a custom org.apache.camel.component.http.UrlRewrite which allows you to rewrite urls when you bridge/proxy endpoints. See more details at http://camel.apache.org/urlrewrite.html		UrlRewrite
mapHttpMessageBody (advanced)	If this option is true then IN exchange Body of the exchange will be mapped to HTTP body. Setting this to false will avoid the HTTP mapping.	true	boolean
mapHttpMessageFormUrlEncodedBody (advanced)	If this option is true then IN exchange Form Encoded body of the exchange will be mapped to HTTP. Setting this to false will avoid the HTTP Form Encoded body mapping.	true	boolean
mapHttpMessageHeaders (advanced)	If this option is true then IN exchange Headers of the exchange will be mapped to HTTP headers. Setting this to false will avoid the HTTP Headers mapping.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
proxyAuthScheme (proxy)	Proxy authentication scheme to use		String
proxyHost (proxy)	Proxy hostname to use		String
proxyPort (proxy)	Proxy port to use		int
authHost (security)	Authentication host to use with NTML		String

10.2. URI FORMAT

```
atmosphere-websocket:///relative path[?options]
```

10.3. READING AND WRITING DATA OVER WEBSOCKET

An atmopshere-websocket endpoint can either write data to the socket or read from the socket, depending on whether the endpoint is configured as the producer or the consumer, respectively.

10.4. CONFIGURING URI TO READ OR WRITE DATA

In the route below, Camel will read from the specified websocket connection.

```
from("atmosphere-websocket:///servicepath")
        .to("direct:next");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="atmosphere-websocket:///servicepath"/>
    <to uri="direct:next"/>
  </route>
</camelContext>
```

In the route below, Camel will read from the specified websocket connection.

```
from("direct:next")
        .to("atmosphere-websocket:///servicepath");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:next"/>
    <to uri="atmosphere-websocket:///servicepath"/>
  </route>
</camelContext>
```

10.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- SERVLET
- AHC-WS * Websocket

CHAPTER 11. ATOM COMPONENT

Available as of Camel version 1.2

The **atom:** component is used for polling Atom feeds.

Camel will poll the feed every 60 seconds by default.
Note: The component currently only supports polling (consuming) feeds.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atom</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

11.1. URI FORMAT

```
atom://atomUri[?options]
```

Where **atomUri** is the URI to the Atom feed to poll.

11.2. OPTIONS

The Atom component has no options.

The Atom endpoint is configured using URI syntax:

```
atom:feedUri
```

with the following path and query parameters:

11.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
feedUri	**Required** The URI to the feed to poll.		String

11.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
feedHeader (consumer)	Sets whether to add the feed object as a header	true	boolean
filter (consumer)	Sets whether to use filtering or not of the entries.	true	boolean
lastUpdate (consumer)	Sets the timestamp to be used for filtering entries from the atom feeds. This options is only in conjunction with the splitEntries.		Date
password (consumer)	Sets the password to be used for basic authentication when polling from a HTTP feed		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
sortEntries (consumer)	Sets whether to sort entries by published date. Only works when splitEntries = true.	false	boolean
splitEntries (consumer)	Sets whether or not entries should be sent individually or whether the entire feed should be sent as a single message	true	boolean
throttleEntries (consumer)	Sets whether all entries identified in a single feed poll should be delivered immediately. If true, only one entry is processed per consumer.delay. Only applicable when splitEntries = true.	true	boolean
username (consumer)	Sets the username to be used for basic authentication when polling from a HTTP feed		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel

Name	Description	Default	Type
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

You can append query options to the URI in the following format, **?option=value&option=value&…**

11.3. EXCHANGE DATA FORMAT

Camel will set the In body on the returned **Exchange** with the entries. Depending on the **splitEntries** flag Camel will either return one **Entry** or a **List<Entry>**.

Option	Value	Behavior
splitEntries	**true**	Only a single entry from the currently being processed feed is set: **exchange.in.body(Entry)**
splitEntries	**false**	The entire list of entries from the feed is set: **exchange.in.body(List<Entry>)**

Camel can set the **Feed** object on the In header (see **feedHeader** option to disable this):

11.4. MESSAGE HEADERS

Camel atom uses these headers.

Header	Description
Camel Atom Feed	When consuming the **org.apache.abdera.model.Feed** object is set to this header.

11.5. SAMPLES

In this sample we poll James Strachan's blog.

```
from("atom://http://macstrac.blogspot.com/feeds/posts/default").to("seda:feeds");
```

In this sample we want to filter only good blogs we like to a SEDA queue. The sample also shows how to setup Camel standalone, not running in any Container or using Spring.

11.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- RSS

CHAPTER 12. ATOMIX MAP COMPONENT

Available as of Camel version 2.20

The camel atomix-map component allows you to work with Atomix's Distributed Map collection.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atomix</artifactId>
  <version>${camel-version}</version>
</dependency>
```

12.1. URI FORMAT

```
atomix-map:mapName
```

12.2. OPTIONS

The Atomix Map component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (common)	The shared component configuration		AtomixMapConfiguration
atomix (common)	The shared AtomixClient instance		AtomixClient
nodes (common)	The nodes the AtomixClient should connect to		List
configurationUri (common)	The path to the AtomixClient configuration		String
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atomix Map endpoint is configured using URI syntax:

```
atomix-map:resourceName
```

with the following path and query parameters:

12.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceName	**Required** The distributed resource name		String

12.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
atomix (common)	The Atomix instance to use		Atomix
configurationUri (common)	The Atomix configuration uri.		String
defaultAction (common)	The default action.	PUT	Action
key (common)	The key to use if none is set in the header or to listen for events for a specific key.		Object
nodes (common)	The address of the nodes composing the cluster.		String
resultHeader (common)	The header that wil carry the result.		String
transport (common)	Sets the Atomix transport.	io.atomix.catalyst.transport.netty.NettyTransport	Transport
ttl (common)	The resource ttl.		long
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
defaultResourceConfig (advanced)	The cluster wide default resource configuration.		Properties
defaultResourceOptions (advanced)	The local default resource options.		Properties
ephemeral (advanced)	Sets if the local member should join groups as PersistentMember or not. If set to ephemeral the local member will receive an auto generated ID thus the local one is ignored.	false	boolean
readConsistency (advanced)	The read consistency level.		ReadConsistency
resourceConfigs (advanced)	Cluster wide resources configuration.		Map
resourceOptions (advanced)	Local resources configurations		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

12.3. HEADERS

Name	Type	Values	Description

Name	Type	Values	Description
CamelAtomixResourceAction	AtomixMap.Action	• PUT • PUT_IF_ABSENT • GET • CLEAR • SIZE • CONTAINS_KEY • CONTAINS_VALUE • IS_EMPTY • ENTRY_SET • REMOVE • REPLACE • VALUES	The action to perform
CamelAtomixResourceKey	Object	-	The key to operate on
CamelAtomixResourceValue	Object	-	The value, if missing In Body is used
CamelAtomixResourceOldValue	Object	-	The old value
CamelAtomixResourceTTL	String / long	-	The entry TTL

Name	Type	Values	Description
Camel Atomix ResourceReadConsistency	ReadConsistency	• ATOMIC • ATOMIC_LEASE • SEQUENTIAL • LOCAL	The read consistency level

12.4. CONFIGURING THE COMPONENT TO CONNECT TO AN ATOMIX CLUSTER

The nodes of the Atomix cluster you want to join can be se at Endpoint or component level (recommended), below some examples:

- **Endpoint:**

```
<beans xmlns="...">
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <from uri="direct:start"/>
            <to uri="atomix-map:myMap?nodes=node-1.atomix.cluster:8700,node-
2.atomix.cluster:8700"/>
        </route>
    </camelContext>
</beans>
```

- **Component:**

```
<beans xmlns="...">
    <bean id="atomix-map"
class="org.apache.camel.component.atomix.client.map.AtomixMapComponent">
        <property name="nodes" value="nodes=node-1.atomix.cluster:8700,node-
2.atomix.cluster:8700"/>
    </bean>

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <from uri="direct:start"/>
            <to uri="atomix-map:myMap"/>
        </route>
    </camelContext>
</beans>
```

12.5. USAGE EXAMPLES:

- **PUT an element with TTL of 1 second:**

```
FluentProducerTemplate.on(context)
    .withHeader(AtomixClientConstants.RESOURCE_ACTION, AtomixMap.Action.PUT)
    .withHeader(AtomixClientConstants.RESOURCE_KEY, key)
```

```
    .withHeader(AtomixClientConstants.RESOURCE_TTL, "1s")
    .withBody(val)
    .to("direct:start")
    .send();
```

CHAPTER 13. ATOMIX MESSAGING COMPONENT

Available as of Camel version 2.20

The camel atomix-messaging component allows you to work with Atomix's Group Messaging.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atomix</artifactId>
  <version>${camel-version}</version>
</dependency>
```

13.1. URI FORMAT

```
atomix-messaging:group
```

The Atomix Messaging component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (common)	The shared component configuration		AtomixMessagingConfiguration
atomix (common)	The shared AtomixClient instance		AtomixClient
nodes (common)	The nodes the AtomixClient should connect to		List
configurationUri (common)	The path to the AtomixClient configuration		String
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atomix Messaging endpoint is configured using URI syntax:

```
atomix-messaging:resourceName
```

with the following path and query parameters:

13.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceName	**Required** The distributed resource name		String

13.1.2. Query Parameters (19 parameters):

Name	Description	Default	Type
atomix (common)	The Atomix instance to use		Atomix
broadcastType (common)	The broadcast type.	ALL	BroadcastType
channelName (common)	The messaging channel name		String
configurationUri (common)	The Atomix configuration uri.		String
defaultAction (common)	The default action.	DIRECT	Action
memberName (common)	The Atomix Group member name		String
nodes (common)	The address of the nodes composing the cluster.		String
resultHeader (common)	The header that wil carry the result.		String
transport (common)	Sets the Atomix transport.	io.atomix.catalyst.transport.netty.NettyTransport	Transport
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
defaultResourceConfig (advanced)	The cluster wide default resource configuration.		Properties
defaultResourceOptions (advanced)	The local default resource options.		Properties
ephemeral (advanced)	Sets if the local member should join groups as PersistentMember or not. If set to ephemeral the local member will receive an auto generated ID thus the local one is ignored.	false	boolean
readConsistency (advanced)	The read consistency level.		ReadConsistency
resourceConfigs (advanced)	Cluster wide resources configuration.		Map
resourceOptions (advanced)	Local resources configurations		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 14. ATOMIX MULTIMAP COMPONENT

Available as of Camel version 2.20

The camel atomix-multimap component allows you to work with Atomix's Distributed MultiMap collection.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atomix</artifactId>
  <version>${camel-version}</version>
</dependency>
```

14.1. URI FORMAT

```
atomix-multimap:multiMapName
```

The Atomix MultiMap component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (consumer)	The shared component configuration		AtomixMultiMapConfiguration
atomix (consumer)	The shared AtomixClient instance		AtomixClient
nodes (consumer)	The nodes the AtomixClient should connect to		List
configurationUri (consumer)	The path to the AtomixClient configuration		String
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atomix MultiMap endpoint is configured using URI syntax:

```
atomix-multimap:resourceName
```

with the following path and query parameters:

14.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceName	**Required** The distributed resource name		String

14.1.2. Query Parameters (18 parameters):

Name	Description	Default	Type
atomix (consumer)	The Atomix instance to use		Atomix
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
configurationUri (consumer)	The Atomix configuration uri.		String
defaultAction (consumer)	The default action.	PUT	Action
key (consumer)	The key to use if none is set in the header or to listen for events for a specific key.		Object
nodes (consumer)	The address of the nodes composing the cluster.		String
resultHeader (consumer)	The header that wil carry the result.		String
transport (consumer)	Sets the Atomix transport.	io.atomix.catalyst.transport.netty.NettyTransport	Transport
ttl (consumer)	The resource ttl.		long

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
defaultResourceConfig (advanced)	The cluster wide default resource configuration.		Properties
defaultResourceOptions (advanced)	The local default resource options.		Properties
ephemeral (advanced)	Sets if the local member should join groups as PersistentMember or not. If set to ephemeral the local member will receive an auto generated ID thus the local one is ignored.	false	boolean
readConsistency (advanced)	The read consistency level.		ReadConsistency
resourceConfigs (advanced)	Cluster wide resources configuration.		Map
resourceOptions (advanced)	Local resources configurations		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 15. ATOMIX QUEUE COMPONENT

Available as of Camel version 2.20

The camel atomix-queue component allows you to work with Atomix's Distributed Queue collection.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atomix</artifactId>
  <version>${camel-version}</version>
</dependency>
```

15.1. URI FORMAT

```
atomix-queue:queueName
```

The Atomix Queue component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (common)	The shared component configuration		AtomixQueue Configuration
atomix (common)	The shared AtomixClient instance		AtomixClient
nodes (common)	The nodes the AtomixClient should connect to		List
configurationUri (common)	The path to the AtomixClient configuration		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atomix Queue endpoint is configured using URI syntax:

```
atomix-queue:resourceName
```

with the following path and query parameters:

15.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceName	**Required** The distributed resource name		String

15.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
atomix (common)	The Atomix instance to use		Atomix
configurationUri (common)	The Atomix configuration uri.		String
defaultAction (common)	The default action.	ADD	Action
nodes (common)	The address of the nodes composing the cluster.		String
resultHeader (common)	The header that wil carry the result.		String
transport (common)	Sets the Atomix transport.	io.atomix.catalyst.transport.netty.NettyTransport	Transport
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
defaultResourceConfig (advanced)	The cluster wide default resource configuration.		Properties
defaultResourceOptions (advanced)	The local default resource options.		Properties
ephemeral (advanced)	Sets if the local member should join groups as PersistentMember or not. If set to ephemeral the local member will receive an auto generated ID thus the local one is ignored.	false	boolean
readConsistency (advanced)	The read consistency level.		ReadConsistency
resourceConfigs (advanced)	Cluster wide resources configuration.		Map
resourceOptions (advanced)	Local resources configurations		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 16. ATOMIX SET COMPONENT

Available as of Camel version 2.20

The camel atomix-set component allows you to work with Atomix's Distributed Set collection.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atomix</artifactId>
  <version>${camel-version}</version>
</dependency>
```

16.1. URI FORMAT

```
atomix-set:setName
```

The Atomix Set component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (common)	The shared component configuration		AtomixSetConfiguration
atomix (common)	The shared AtomixClient instance		AtomixClient
nodes (common)	The nodes the AtomixClient should connect to		List
configurationUri (common)	The path to the AtomixClient configuration		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atomix Set endpoint is configured using URI syntax:

```
atomix-set:resourceName
```

with the following path and query parameters:

16.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceName	**Required** The distributed resource name		String

16.1.2. Query Parameters (17 parameters):

Name	Description	Default	Type
atomix (common)	The Atomix instance to use		Atomix
configurationUri (common)	The Atomix configuration uri.		String
defaultAction (common)	The default action.	ADD	Action
nodes (common)	The address of the nodes composing the cluster.		String
resultHeader (common)	The header that wil carry the result.		String
transport (common)	Sets the Atomix transport.	io.atomix.catalyst.transport.netty.NettyTransport	Transport
ttl (common)	The resource ttl.		long
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
defaultResourceConfig (advanced)	The cluster wide default resource configuration.		Properties
defaultResourceOptions (advanced)	The local default resource options.		Properties
ephemeral (advanced)	Sets if the local member should join groups as PersistentMember or not. If set to ephemeral the local member will receive an auto generated ID thus the local one is ignored.	false	boolean
readConsistency (advanced)	The read consistency level.		ReadConsistency
resourceConfigs (advanced)	Cluster wide resources configuration.		Map
resourceOptions (advanced)	Local resources configurations		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 17. ATOMIX VALUE COMPONENT

Available as of Camel version 2.20

The camel atomix-value component allows you to work with Atomix's Distributed Value.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-atomix</artifactId>
  <version>${camel-version}</version>
</dependency>
```

17.1. URI FORMAT

```
atomix-value:valueName
```

The Atomix Value component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (common)	The shared component configuration		AtomixValue Configuration
atomix (common)	The shared AtomixClient instance		AtomixClient
nodes (common)	The nodes the AtomixClient should connect to		List
configurationUri (common)	The path to the AtomixClient configuration		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Atomix Value endpoint is configured using URI syntax:

```
atomix-value:resourceName
```

with the following path and query parameters:

17.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceName	**Required** The distributed resource name		String

17.1.2. Query Parameters (17 parameters):

Name	Description	Default	Type
atomix (common)	The Atomix instance to use		Atomix
configurationUri (common)	The Atomix configuration uri.		String
defaultAction (common)	The default action.	SET	Action
nodes (common)	The address of the nodes composing the cluster.		String
resultHeader (common)	The header that wil carry the result.		String
transport (common)	Sets the Atomix transport.	io.atomix.catalyst.transport.netty.NettyTransport	Transport
ttl (common)	The resource ttl.		long
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
defaultResourceConfig (advanced)	The cluster wide default resource configuration.		Properties
defaultResourceOptions (advanced)	The local default resource options.		Properties
ephemeral (advanced)	Sets if the local member should join groups as PersistentMember or not. If set to ephemeral the local member will receive an auto generated ID thus the local one is ignored.	false	boolean
readConsistency (advanced)	The read consistency level.		ReadConsistency
resourceConfigs (advanced)	Cluster wide resources configuration.		Map
resourceOptions (advanced)	Local resources configurations		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 18. AVRO COMPONENT

Available as of Camel version 2.10

This component provides a dataformat for avro, which allows serialization and deserialization of messages using Apache Avro's binary dataformat. Moreover, it provides support for Apache Avro's rpc, by providing producers and consumers endpoint for using avro over netty or http.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-avro</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

18.1. APACHE AVRO OVERVIEW

Avro allows you to define message types and a protocol using a json like format and then generate java code for the specified types and messages. An example of how a schema looks like is below.

```
{"namespace": "org.apache.camel.avro.generated",
 "protocol": "KeyValueProtocol",

 "types": [
    {"name": "Key", "type": "record",
     "fields": [
         {"name": "key",    "type": "string"}
     ]
    },
    {"name": "Value", "type": "record",
     "fields": [
         {"name": "value",    "type": "string"}
     ]
    }
 ],

 "messages": {
    "put": {
       "request": [{"name": "key", "type": "Key"}, {"name": "value", "type": "Value"} ],
       "response": "null"
    },
    "get": {
       "request": [{"name": "key", "type": "Key"}],
       "response": "Value"
    }
 }
}
```

You can easily generate classes from a schema, using maven, ant etc. More details can be found at the Apache Avro documentation.

However, it doesn't enforce a schema first approach and you can create schema for your existing

classes. **Since 2.12** you can use existing protocol interfaces to make RCP calls. You should use interface for the protocol itself and POJO beans or primitive/String classes for parameter and result types. Here is an example of the class that corresponds to schema above:

```
package org.apache.camel.avro.reflection;

public interface KeyValueProtocol {
    void put(String key, Value value);
    Value get(String key);
}

class Value {
    private String value;
    public String getValue() { return value; }
    public void setValue(String value) { this.value = value; }
}
```

Note: Existing classes can be used only for RPC (see below), not in data format.

18.2. USING THE AVRO DATA FORMAT

Using the avro data format is as easy as specifying that the class that you want to marshal or unmarshal in your route.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:in"/>
        <marshal>
            <avro instanceClass="org.apache.camel.dataformat.avro.Message"/>
        </marshal>
        <to uri="log:out"/>
    </route>
</camelContext>
```

An alternative can be to specify the dataformat inside the context and reference it from your route.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <avro id="avro" instanceClass="org.apache.camel.dataformat.avro.Message"/>
    </dataFormats>
    <route>
        <from uri="direct:in"/>
        <marshal ref="avro"/>
        <to uri="log:out"/>
    </route>
</camelContext>
```

In the same manner you can umarshal using the avro data format.

18.3. USING AVRO RPC IN CAMEL

As mentioned above Avro also provides RPC support over multiple transports such as http and netty. Camel provides consumers and producers for these two transports.

```
avro:[transport]:[host]:[port][?options]
```

The supported transport values are currently http or netty.

Since 2.12 you can specify message name right in the URI:

```
avro:[transport]:[host]:[port][/messageName][?options]
```

For consumers this allows you to have multiple routes attached to the same socket. Dispatching to correct route will be done by the avro component automatically. Route with no messageName specified (if any) will be used as default.

When using camel producers for avro ipc, the "in" message body needs to contain the parameters of the operation specified in the avro protocol. The response will be added in the body of the "out" message.

In a similar manner when using camel avro consumers for avro ipc, the requests parameters will be placed inside the "in" message body of the created exchange and once the exchange is processed the body of the "out" message will be send as a response.

Note: By default consumer parameters are wrapped into array. If you've got only one parameter, **since 2.12** you can use **singleParameter** URI option to receive it direcly in the "in" message body without array wrapping.

18.4. AVRO RPC URI OPTIONS

The Avro component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use a shared AvroConfiguration to configure options once		AvroConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Avro endpoint is configured using URI syntax:

```
avro:transport:host:port/messageName
```

with the following path and query parameters:

18.4.1. Path Parameters (4 parameters):

Name	Description	Default	Type
transport	**Required** Transport to use		AvroTransport

Name	Description	Default	Type
port	**Required** Port number to use		int
host	**Required** Hostname to use		String
messageName	The name of the message to send.		String

18.4.2. Query Parameters (10 parameters):

Name	Description	Default	Type
protocol (common)	Avro protocol to use		Protocol
protocolClassName (common)	Avro protocol to use defined by the FQN class name		String
protocolLocation (common)	Avro protocol location		String
reflectionProtocol (common)	If protocol object provided is reflection protocol. Should be used only with protocol parameter because for protocolClassName protocol type will be auto detected	false	boolean
singleParameter (common)	If true, consumer parameter won't be wrapped into array. Will fail if protocol specifies more then 1 parameter for the message	false	boolean
uriAuthority (common)	Authority to use (username and password)		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

18.5. AVRO RPC HEADERS

Name	Description
CamelAvroMessageName	The name of the message to send. In consumer overrides message name from URI (if any)

18.6. EXAMPLES

An example of using camel avro producers via http:

```
        <route>
            <from uri="direct:start"/>
            <to uri="avro:http:localhost:{{avroport}}?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol"/>
            <to uri="log:avro"/>
        </route>
```

In the example above you need to fill **CamelAvroMessageName** header. **Since 2.12** you can use following syntax to call constant messages:

```
        <route>
            <from uri="direct:start"/>
            <to uri="avro:http:localhost:{{avroport}}/put?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol"/>
            <to uri="log:avro"/>
        </route>
```

An example of consuming messages using camel avro consumers via netty:

```
        <route>
            <from uri="avro:netty:localhost:{{avroport}}?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol"/>
            <choice>
```

```
            <when>
              <el>${in.headers.CamelAvroMessageName == 'put'}</el>
              <process ref="putProcessor"/>
            </when>
            <when>
              <el>${in.headers.CamelAvroMessageName == 'get'}</el>
              <process ref="getProcessor"/>
            </when>
          </choice>
        </route>
```

Since 2.12 you can set up two distinct routes to perform the same task:

```
        <route>
            <from uri="avro:netty:localhost:{{avroport}}/put?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol">
            <process ref="putProcessor"/>
        </route>
        <route>
            <from uri="avro:netty:localhost:{{avroport}}/get?
protocolClassName=org.apache.camel.avro.generated.KeyValueProtocol&singleParameter=true"/>
            <process ref="getProcessor"/>
        </route>
```

In the example above, get takes only one parameter, so **singleParameter** is used and **getProcessor** will receive Value class directly in body, while **putProcessor** will receive an array of size 2 with String key and Value value filled as array contents.

CHAPTER 19. AVRO DATAFORMAT

Available as of Camel version 2.14

This component provides a dataformat for avro, which allows serialization and deserialization of messages using Apache Avro's binary dataformat. Moreover, it provides support for Apache Avro's rpc, by providing producers and consumers endpoint for using avro over netty or http.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-avro</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

19.1. APACHE AVRO OVERVIEW

Avro allows you to define message types and a protocol using a json like format and then generate java code for the specified types and messages. An example of how a schema looks like is below.

```
{"namespace": "org.apache.camel.avro.generated",
 "protocol": "KeyValueProtocol",

 "types": [
    {"name": "Key", "type": "record",
     "fields": [
         {"name": "key",    "type": "string"}
     ]
    },
    {"name": "Value", "type": "record",
     "fields": [
         {"name": "value",    "type": "string"}
     ]
    }
 ],

 "messages": {
    "put": {
       "request": [{"name": "key", "type": "Key"}, {"name": "value", "type": "Value"} ],
       "response": "null"
    },
    "get": {
       "request": [{"name": "key", "type": "Key"}],
       "response": "Value"
    }
 }
}
```

You can easily generate classes from a schema, using maven, ant etc. More details can be found at the Apache Avro documentation.

However, it doesn't enforce a schema first approach and you can create schema for your existing

classes. **Since 2.12** you can use existing protocol interfaces to make RCP calls. You should use interface for the protocol itself and POJO beans or primitive/String classes for parameter and result types. Here is an example of the class that corresponds to schema above:

```
package org.apache.camel.avro.reflection;

public interface KeyValueProtocol {
    void put(String key, Value value);
    Value get(String key);
}

class Value {
    private String value;
    public String getValue() { return value; }
    public void setValue(String value) { this.value = value; }
}
```

Note: Existing classes can be used only for RPC (see below), not in data format.

19.2. USING THE AVRO DATA FORMAT

Using the avro data format is as easy as specifying that the class that you want to marshal or unmarshal in your route.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:in"/>
        <marshal>
            <avro instanceClass="org.apache.camel.dataformat.avro.Message"/>
        </marshal>
        <to uri="log:out"/>
    </route>
</camelContext>
```

An alternative can be to specify the dataformat inside the context and reference it from your route.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <avro id="avro" instanceClass="org.apache.camel.dataformat.avro.Message"/>
    </dataFormats>
    <route>
        <from uri="direct:in"/>
        <marshal ref="avro"/>
        <to uri="log:out"/>
    </route>
</camelContext>
```

In the same manner you can umarshal using the avro data format.

19.3. AVRO DATAFORMAT OPTIONS

The Avro dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
instanceClassName		**String**	Class name to use for marshal and unmarshalling
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

CHAPTER 20. AWS CLOUDWATCH COMPONENT

Available as of Camel version 2.11

The CW component allows messages to be sent to an Amazon CloudWatch metrics. The implementation of the Amazon API is provided by the AWS SDK.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon CloudWatch. More information are available at Amazon CloudWatch.

20.1. URI FORMAT

```
aws-cw://namespace[?options]
```

The metrics will be created if they don't already exists.
You can append query options to the URI in the following format, **?options=value&option2=value&...**

20.2. URI OPTIONS

The AWS CloudWatch component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS CW default configuration		CwConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	The region in which CW client needs to work		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS CloudWatch endpoint is configured using URI syntax:

```
aws-cw:namespace
```

with the following path and query parameters:

20.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
namespace	**Required** The metric namespace		String

20.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
amazonCwClient (producer)	To use the AmazonCloudWatch as the client		AmazonCloudWatch
name (producer)	The metric name		String
proxyHost (producer)	To define a proxy host when instantiating the CW client		String
proxyPort (producer)	To define a proxy port when instantiating the CW client		Integer
region (producer)	The region in which CW client needs to work		String
timestamp (producer)	The metric timestamp		Date
unit (producer)	The metric unit		String
value (producer)	The metric value		Double
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required CW component options

You have to provide the amazonCwClient in the Registry or your accessKey and secretKey to access the Amazon's CloudWatch.

20.3. USAGE

20.3.1. Message headers evaluated by the CW producer

Header	Type	Description
CamelAwsCwMetricName	**String**	The Amazon CW metric name.
CamelAwsCwMetricValue	**Double**	The Amazon CW metric value.
CamelAwsCwMetricUnit	**String**	The Amazon CW metric unit.
CamelAwsCwMetricNamespace	**String**	The Amazon CW metric namespace.
CamelAwsCwMetricTimestamp	**Date**	The Amazon CW metric timestamp.
CamelAwsCwMetricDimensionName	**String**	**Camel 2.12:** The Amazon CW metric dimension name.
CamelAwsCwMetricDimensionValue	**String**	**Camel 2.12:** The Amazon CW metric dimension value.

Header	Type	Description
CamelAwsCwMetricDimensions	**Map<String, String>**	**Camel 2.12:** A map of dimension names and dimension values.

20.3.2. Advanced AmazonCloudWatch configuration

If you need more control over the **AmazonCloudWatch** instance configuration you can create your own instance and refer to it from the URI:

```
from("direct:start")
.to("aws-cw://namepsace?amazonCwClient=#client");
```

The **#client** refers to a **AmazonCloudWatch** in the Registry.

For example if your Camel Application is running behind a firewall:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonCloudWatch client = new AmazonCloudWatchClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

20.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.10 or higher).

20.5. SEE ALSO

- Configuring Camel
- Component

- Endpoint
- Getting Started
- AWS Component

CHAPTER 21. AWS DYNAMODB COMPONENT

Available as of Camel version 2.10

The DynamoDB component supports storing and retrieving data from/to Amazon's DynamoDB service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon DynamoDB. More information are available at Amazon DynamoDB.

21.1. URI FORMAT

```
aws-ddb://domainName[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

21.2. URI OPTIONS

The AWS DynamoDB component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS DDB default configuration		DdbConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	The region in which DDB client needs to work		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS DynamoDB endpoint is configured using URI syntax:

```
aws-ddb:tableName
```

with the following path and query parameters:

21.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
tableName	**Required** The name of the table currently worked with.		String

21.2.2. Query Parameters (13 parameters):

Name	Description	Default	Type
amazonDDBClient (producer)	To use the AmazonDynamoDB as the client		AmazonDynamoDB
consistentRead (producer)	Determines whether or not strong consistency should be enforced when data is read.	false	boolean
keyAttributeName (producer)	Attribute name when creating table		String
keyAttributeType (producer)	Attribute type when creating table		String
operation (producer)	What operation to perform	PutItem	DdbOperations
proxyHost (producer)	To define a proxy host when instantiating the DDB client		String
proxyPort (producer)	To define a proxy port when instantiating the DDB client		Integer
readCapacity (producer)	The provisioned throughput to reserve for reading resources from your table		Long
region (producer)	The region in which DDB client needs to work		String
writeCapacity (producer)	The provisioned throughput to reserved for writing resources to your table		Long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accessKey (security)	Amazon AWS Access Key		String

Name	Description	Default	Type
secretKey (security)	Amazon AWS Secret Key		String

Required DDB component options

You have to provide the amazonDDBClient in the Registry or your accessKey and secretKey to access the Amazon's DynamoDB.

21.3. USAGE

21.3.1. Message headers evaluated by the DDB producer

Header	Type	Description
CamelAwsDdbBatchItems	**Map<String, KeysAndAttributes>**	A map of the table name and corresponding items to get by primary key.
CamelAwsDdbTableName	**String**	Table Name for this operation.
CamelAwsDdbKey	**Key**	The primary key that uniquely identifies each item in a table. **From Camel 2.16.0 the type of this header is Map<String, AttributeValue> and not Key**
CamelAwsDdbReturnValues	**String**	Use this parameter if you want to get the attribute name-value pairs before or after they are modified(NONE, ALL_OLD, UPDATED_OLD, ALL_NEW, UPDATED_NEW).
CamelAwsDdbUpdateCondition	**Map<String, ExpectedAttributeValue>**	Designates an attribute for a conditional modification.

Header	Type	Description
CamelAwsDdbAttributeNames	**Collection<String>**	If attribute names are not specified then all attributes will be returned.
CamelAwsDdbConsistentRead	**Boolean**	If set to true, then a consistent read is issued, otherwise eventually consistent is used.
CamelAwsDdbIndexName	**String**	If set will be used as Secondary Index for Query operation.
CamelAwsDdbItem	**Map<String, AttributeValue>**	A map of the attributes for the item, and must include the primary key values that define the item.
CamelAwsDdbExactCount	**Boolean**	If set to true, Amazon DynamoDB returns a total number of items that match the query parameters, instead of a list of the matching items and their attributes. **From Camel 2.16.0 this header doesn't exist anymore.**
CamelAwsDdbKeyConditions	**Map<String, Condition>**	**From Camel 2.16.0**. This header specify the selection criteria for the query, and merge together the two old headers **CamelAwsDdbHashKeyValue** and **CamelAwsDdbScanRangeKeyCondition**
CamelAwsDdbStartKey	**Key**	Primary key of the item from which to continue an earlier query.
CamelAwsDdbHashKeyValue	**AttributeValue**	Value of the hash component of the composite primary key. **From Camel 2.16.0 this header doesn't exist anymore.**

Header	Type	Description
CamelAwsDdbLimit	**Integer**	The maximum number of items to return.
CamelAwsDdbScanRangeKeyCondition	**Condition**	A container for the attribute values and comparison operators to use for the query.**From Camel 2.16.0 this header doesn't exist anymore.**
CamelAwsDdbScanIndexForward	**Boolean**	Specifies forward or backward traversal of the index.
CamelAwsDdbScanFilter	**Map<String, Condition>**	Evaluates the scan results and returns only the desired values.
CamelAwsDdbUpdateValues	**Map<String, AttributeValueUpdate>**	Map of attribute name to the new value and action for the update.

21.3.2. Message headers set during BatchGetItems operation

Header	Type	Description
CamelAwsDdbBatchResponse	**Map<String,BatchResponse>**	Table names and the respective item attributes from the tables.

Header	Type	Description
CamelAwsDdbUnprocessedKeys	**Map<String,KeysAndAttributes>**	Contains a map of tables and their respective keys that were not processed with the current response.

21.3.3. Message headers set during DeleteItem operation

Header	Type	Description
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	The list of attributes returned by the operation.

21.3.4. Message headers set during DeleteTable operation

Header	Type	Description
CamelAwsDdbProvisionedThroughput		
ProvisionedThroughputDescription		The value of the ProvisionedThroughput property for this table
CamelAwsDdbCreationDate	**Date**	Creation DateTime of this table.

Header	Type	Description
CamelAwsDdbTableItemCount	**Long**	Item count for this table.
CamelAwsDdbKeySchema	**KeySchema**	The KeySchema that identifies the primary key for this table. **From Camel 2.16.0 the type of this header is List<KeySchemaElement> and not KeySchema**
CamelAwsDdbTableName	**String**	The table name.
CamelAwsDdbTableSize	**Long**	The table size in bytes.
CamelAwsDdbTableStatus	**String**	The status of the table: CREATING, UPDATING, DELETING, ACTIVE

21.3.5. Message headers set during DescribeTable operation

Header	Type	Description
CamelAwsDdbProvisionedThroughput	{{ProvisionedThroughputDescription}}	The value of the ProvisionedThroughput property for this table
CamelAwsDdbCreationDate	**Date**	Creation DateTime of this table.

Header	Type	Description
CamelAwsDdbTableItemCount	**Long**	Item count for this table.
CamelAwsDdbKeySchema	{{KeySchema}}	The KeySchema that identifies the primary key for this table. **From Camel 2.16.0 the type of this header is List<KeySchemaElement> and not KeySchema**
CamelAwsDdbTableName	**String**	The table name.
CamelAwsDdbTableSize	**Long**	The table size in bytes.
CamelAwsDdbTableStatus	**String**	The status of the table: CREATING, UPDATING, DELETING, ACTIVE
CamelAwsDdbReadCapacity	**Long**	ReadCapacityUnits property of this table.
CamelAwsDdbWriteCapacity	**Long**	WriteCapacityUnits property of this table.

21.3.6. Message headers set during GetItem operation

Header	Type	Description
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	The list of attributes returned by the operation.

21.3.7. Message headers set during PutItem operation

Header	Type	Description
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	The list of attributes returned by the operation.

21.3.8. Message headers set during Query operation

Header	Type	Description
CamelAwsDdbItems	**List<java.util.Map<String,AttributeValue>>**	The list of attributes returned by the operation.
CamelAwsDdbLastEvaluatedKey	**Key**	Primary key of the item where the query operation stopped, inclusive of the previous result set.
CamelAwsDdbConsumedCapacity	**Double**	The number of Capacity Units of the provisioned throughput of the table consumed during the operation.

Header	Type	Description
CamelAwsDdbCount	**Integer**	Number of items in the response.

21.3.9. Message headers set during Scan operation

Header	Type	Description
CamelAwsDdbItems	**List<java.util.Map<String,AttributeValue>>**	The list of attributes returned by the operation.
CamelAwsDdbLastEvaluatedKey	**Key**	Primary key of the item where the query operation stopped, inclusive of the previous result set.
CamelAwsDdbConsumedCapacity	**Double**	The number of Capacity Units of the provisioned throughput of the table consumed during the operation.
CamelAwsDdbCount	**Integer**	Number of items in the response.
CamelAwsDdbScannedCount	**Integer**	Number of items in the complete scan before any filters are applied.

21.3.10. Message headers set during UpdateItem operation

Header	Type	Description
CamelAwsDdbAttributes	**Map<String, AttributeValue>**	The list of attributes returned by the operation.

21.3.11. Advanced AmazonDynamoDB configuration

If you need more control over the **AmazonDynamoDB** instance configuration you can create your own instance and refer to it from the URI:

```
from("direct:start")
.to("aws-ddb://domainName?amazonDDBClient=#client");
```

The **#client** refers to a **AmazonDynamoDB** in the Registry.

For example if your Camel Application is running behind a firewall:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonDynamoDB client = new AmazonDynamoDBClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

21.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.10 or higher).

21.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint

- Getting Started
- AWS Component

CHAPTER 22. AWS DYNAMODB STREAMS COMPONENT

Available as of Camel version 2.17

The DynamoDB Stream component supports receiving messages from Amazon DynamoDB Stream service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon DynamoDB Streams. More information are available at AWS DynamoDB

22.1. URI FORMAT

```
aws-ddbstream://table-name[?options]
```

The stream needs to be created prior to it being used.
You can append query options to the URI in the following format, ?options=value&option2=value&...

22.2. URI OPTIONS

The AWS DynamoDB Streams component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS DDB stream default configuration		DdbStreamConfiguration
accessKey (consumer)	Amazon AWS Access Key		String
secretKey (consumer)	Amazon AWS Secret Key		String
region (consumer)	Amazon AWS Region		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS DynamoDB Streams endpoint is configured using URI syntax:

```
aws-ddbstream:tableName
```

with the following path and query parameters:

22.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
tableName	**Required** Name of the dynamodb table		String

22.2.2. Query Parameters (28 parameters):

Name	Description	Default	Type
amazonDynamoDbStreams Client (consumer)	Amazon DynamoDB client to use for all requests for this endpoint		AmazonDynamoDBStreams
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
iteratorType (consumer)	Defines where in the DynaboDB stream to start getting records. Note that using TRIM_HORIZON can cause a significant delay before the stream has caught up to real-time. if AT,AFTER_SEQUENCE_NUMBER are used, then a sequenceNumberProvider MUST be supplied.	LATEST	ShardIteratorType
maxResultsPerRequest (consumer)	Maximum number of records that will be fetched in each poll		int
proxyHost (consumer)	To define a proxy host when instantiating the DDBStreams client		String
proxyPort (consumer)	To define a proxy port when instantiating the DDBStreams client		Integer
region (consumer)	The region in which DDBStreams client needs to work		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean

Name	Description	Default	Type
sequenceNumberProvider (consumer)	Provider for the sequence number when using one of the two ShardIteratorType.AT,AFTER_SEQUENCE_NUMBER iterator types. Can be a registry reference or a literal sequence number.		SequenceNumberProvider
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long

Name	Description	Default	Type
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required DynampDBStream component options

You have to provide the amazonDynamoDbStreamsClient in the Registry with proxies and relevant credentials configured.

22.3. SEQUENCE NUMBERS

You can provide a literal string as the sequence number or provide a bean in the registry. An example of using the bean would be to save your current position in the change feed and restore it on Camel startup.

It is an error to provide a sequence number that is greater than the largest sequence number in the describe-streams result, as this will lead to the AWS call returning an HTTP 400.

22.4. BATCH CONSUMER

This component implements the Batch Consumer.

This allows you for instance to know how many messages exists in this batch and for instance let the Aggregator aggregate this number of messages.

22.5. USAGE

22.5.1. AmazonDynamoDBStreamsClient configuration

You will need to create an instance of AmazonDynamoDBStreamsClient and bind it to the registry

```
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

Region region = Region.getRegion(Regions.fromName(region));
region.createClient(AmazonDynamoDBStreamsClient.class, null, clientConfiguration);
// the 'null' here is the AWSCredentialsProvider which defaults to an instance of
DefaultAWSCredentialsProviderChain

registry.bind("kinesisClient", client);
```

22.5.2. Providing AWS Credentials

It is recommended that the credentials are obtained by using the DefaultAWSCredentialsProviderChain that is the default when creating a new ClientConfiguration instance, however, a different AWSCredentialsProvider can be specified when calling createClient(...).

22.6. COPING WITH DOWNTIME

22.6.1. AWS DynamoDB Streams outage of less than 24 hours

The consumer will resume from the last seen sequence number (as implemented for CAMEL-9515), so you should receive a flood of events in quick succession, as long as the outage did not also include DynamoDB itself.

22.6.2. AWS DynamoDB Streams outage of more than 24 hours

Given that AWS only retain 24 hours worth of changes, you will have missed change events no matter what mitigations are in place.

22.7. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.7 or higher).

22.8. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component
  +

CHAPTER 23. AWS EC2 COMPONENT

Available as of Camel version 2.16

The EC2 component supports create, run, start, stop and terminate AWS EC2 instances.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon EC2. More information are available at Amazon EC2.

23.1. URI FORMAT

```
aws-ec2://label[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

23.2. URI OPTIONS

The AWS EC2 component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS EC2 default configuration		EC2Configuration
region (producer)	The region in which EC2 client needs to work		String
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS EC2 endpoint is configured using URI syntax:

```
aws-ec2:label
```

with the following path and query parameters:

23.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
label	**Required** Logical name		String

23.2.2. Query Parameters (8 parameters):

Name	Description	Default	Type
accessKey (producer)	Amazon AWS Access Key		String
amazonEc2Client (producer)	To use a existing configured AmazonEC2Client as client		AmazonEC2Client
operation (producer)	**Required** The operation to perform. It can be createAndRunInstances, startInstances, stopInstances, terminateInstances, describeInstances, describeInstancesStatus, rebootInstances, monitorInstances, unmonitorInstances, createTags or deleteTags		EC2Operations
proxyHost (producer)	To define a proxy host when instantiating the EC2 client		String
proxyPort (producer)	To define a proxy port when instantiating the EC2 client		Integer
region (producer)	The region in which EC2 client needs to work		String
secretKey (producer)	Amazon AWS Secret Key		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Required EC2 component options

You have to provide the amazonEc2Client in the Registry or your accessKey and secretKey to access the Amazon EC2 service.

23.3. USAGE

23.3.1. Message headers evaluated by the EC2 producer

Header	Type	Description
CamelAwsEC2ImageId	**String**	An image ID of the AWS marketplace
CamelAwsEC2InstanceType	com.amazonaws.services.ec2.model.InstanceType	The instance type we want to create and run
CamelAwsEC2Operation	**String**	The operation we want to perform
CamelAwsEC2InstanceMinCount	**Int**	The mininum number of instances we want to run.
CamelAwsEC2InstanceMaxCount	**Int**	The maximum number of instances we want to run.
CamelAwsEC2InstanceMonitoring	Boolean	Define if we want the running instances to be monitored
CamelAwsEC2InstanceEbsOptimized	**Boolean**	Define if the creating instance is optimized for EBS I/O.

Header	Type	Description
CamelAwsEC2InstanceSecurityGroups	Collection	The security groups to associate to the instances
CamelAwsEC2InstancesIds	**Collection**	A collection of instances IDS to execute start, stop, describe and terminate operations on.
CamelAwsEC2InstancesTags	**Collection**	A collection of tags to add or remove from EC2 resources

Dependencies

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.16 or higher).

23.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 24. AWS KINESIS COMPONENT

Available as of Camel version 2.17

The Kinesis component supports receiving messages from and sending messages to Amazon Kinesis service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon Kinesis. More information are available at AWS Kinesis

24.1. URI FORMAT

```
aws-kinesis://stream-name[?options]
```

The stream needs to be created prior to it being used.
You can append query options to the URI in the following format, ?options=value&option2=value&...

24.2. URI OPTIONS

The AWS Kinesis component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS S3 default configuration		KinesisConfiguration
accessKey (common)	Amazon AWS Access Key		String
secretKey (common)	Amazon AWS Secret Key		String
region (common)	Amazon AWS Region		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS Kinesis endpoint is configured using URI syntax:

```
aws-kinesis:streamName
```

with the following path and query parameters:

24.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
streamName	**Required** Name of the stream		String

24.2.2. Query Parameters (30 parameters):

Name	Description	Default	Type
amazonKinesisClient (common)	Amazon Kinesis client to use for all requests for this endpoint		AmazonKinesis
proxyHost (common)	To define a proxy host when instantiating the DDBStreams client		String
proxyPort (common)	To define a proxy port when instantiating the DDBStreams client		Integer
region (common)	The region in which Kinesis client needs to work		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
iteratorType (consumer)	Defines where in the Kinesis stream to start getting records	TRIM_HORIZON	ShardIteratorType
maxResultsPerRequest (consumer)	Maximum number of records that will be fetched in each poll	1	int
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
sequenceNumber (consumer)	The sequence number to start polling from. Required if iteratorType is set to AFTER_SEQUENCE_NUMBER or AT_SEQUENCE_NUMBER		String

Name	Description	Default	Type
shardClosed (consumer)	Define what will be the behavior in case of shard closed. Possible value are ignore, silent and fail.In case of ignore a message will be logged and the consumer will restart from the beginning,in case of silent there will be no logging and the consumer will start from the beginning,in case of fail a ReachedClosedStateException will be raised	ignore	KinesisShardClosed StrategyEnum
shardId (consumer)	Defines which shardId in the Kinesis stream to get records from		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int

Name	Description	Default	Type
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required Kinesis component options

You have to provide the amazonKinesisClient in the Registry with proxies and relevant credentials configured.

24.3. BATCH CONSUMER

This component implements the Batch Consumer.

This allows you for instance to know how many messages exists in this batch and for instance let the Aggregator aggregate this number of messages.

24.4. USAGE

24.4.1. Message headers set by the Kinesis consumer

Header	Type	Description
CamelAwsKinesisSequenceNumber	**String**	The sequence number of the record. This is represented as a String as it size is not defined by the API. If it is to be used as a numerical type then use
CamelAwsKinesisApproximateArrivalTimestamp	**String**	The time AWS assigned as the arrival time of the record.
CamelAwsKinesisPartitionKey	**String**	Identifies which shard in the stream the data record is assigned to.

24.4.2. AmazonKinesis configuration

You will need to create an instance of AmazonKinesisClient and bind it to the registry

```
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

Region region = Region.getRegion(Regions.fromName(region));
region.createClient(AmazonKinesisClient.class, null, clientConfiguration);
// the 'null' here is the AWSCredentialsProvider which defaults to an instance of
```

```
DefaultAWSCredentialsProviderChain

registry.bind("kinesisClient", client);
```

You then have to reference the AmazonKinesisClient in the **amazonKinesisClient** URI option.

```
from("aws-kinesis://mykinesisstream?amazonKinesisClient=#kinesisClient")
  .to("log:out?showAll=true");
```

24.4.3. Providing AWS Credentials

It is recommended that the credentials are obtained by using the DefaultAWSCredentialsProviderChain that is the default when creating a new ClientConfiguration instance, however, a different AWSCredentialsProvider can be specified when calling createClient(...).

24.4.4. Message headers used by the Kinesis producer to write to Kinesis. The producer expects that the message body is a ByteBuffer.

Header	Type	Description
CamelAwsKinesisPartitionKey	**String**	The PartitionKey to pass to Kinesis to store this record.
CamelAwsKinesisSequenceNumber	**String**	Optional paramter to indicate the sequence number of this record.

24.4.5. Message headers set by the Kinesis producer on successful storage of a Record

Header	Type	Description
CamelAwsKinesisSequenceNumber	**String**	The sequence number of the record, as defined in Response Syntax
CamelAwsKinesisShardId	**String**	The shard ID of where the Record was stored

24.5. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.17 or higher).

24.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 25. AWS KINESIS FIREHOSE COMPONENT

Available as of Camel version 2.19

The Kinesis Firehose component supports sending messages to Amazon Kinesis Firehose service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon Kinesis Firehose. More information are available at AWS Kinesis Firehose

25.1. URI FORMAT

```
aws-kinesis-firehose://delivery-stream-name[?options]
```

The stream needs to be created prior to it being used.
You can append query options to the URI in the following format, ?options=value&option2=value&...

25.2. URI OPTIONS

The AWS Kinesis Firehose component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS Kinesis Firehose default configuration		KinesisFirehoseConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	Amazon AWS Region		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS Kinesis Firehose endpoint is configured using URI syntax:

```
aws-kinesis-firehose:streamName
```

with the following path and query parameters:

25.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
streamName	**Required** Name of the stream		String

25.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
amazonKinesisFirehoseClient (producer)	Amazon Kinesis Firehose client to use for all requests for this endpoint		AmazonKinesisFirehose
proxyHost (producer)	To define a proxy host when instantiating the DDBStreams client		String
proxyPort (producer)	To define a proxy port when instantiating the DDBStreams client		Integer
region (producer)	The region in which Kinesis client needs to work		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required Kinesis Firehose component options

You have to provide the amazonKinesisClient in the Registry with proxies and relevant credentials configured.

25.3. USAGE

25.3.1. Amazon Kinesis Firehose configuration

You will need to create an instance of AmazonKinesisClient and bind it to the registry

```
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

Region region = Region.getRegion(Regions.fromName(region));
region.createClient(AmazonKinesisClient.class, null, clientConfiguration);
```

```
// the 'null' here is the AWSCredentialsProvider which defaults to an instance of
DefaultAWSCredentialsProviderChain

registry.bind("kinesisFirehoseClient", client);
```

You then have to reference the AmazonKinesisFirehoseClient in the **amazonKinesisFirehoseClient** URI option.

```
from("aws-kinesis-firehose://mykinesisdeliverystream?amazonKinesisFirehoseClient=#kinesisClient")
  .to("log:out?showAll=true");
```

25.3.2. Providing AWS Credentials

It is recommended that the credentials are obtained by using the DefaultAWSCredentialsProviderChain that is the default when creating a new ClientConfiguration instance, however, a different AWSCredentialsProvider can be specified when calling createClient(...).

25.3.3. Message headers set by the Kinesis producer on successful storage of a Record

Header	Type	Description
CamelAwsKinesisFirehoseRecordId	**String**	The record ID, as defined in Response Syntax

25.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.19 or higher).

25.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint

- Getting Started
- AWS Component

CHAPTER 26. AWS KMS COMPONENT

Available as of Camel version 2.21

The KMS component supports create, run, start, stop and terminate AWS KMS instances.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon KMS. More information are available at Amazon KMS.

26.1. URI FORMAT

```
aws-kms://label[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

26.2. URI OPTIONS

The AWS KMS component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS MQ default configuration		KMSConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	The region in which MQ client needs to work		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS KMS endpoint is configured using URI syntax:

```
aws-kms:label
```

with the following path and query parameters:

26.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
label	**Required** Logical name		String

26.2.2. Query Parameters (8 parameters):

Name	Description	Default	Type
accessKey (producer)	Amazon AWS Access Key		String
kmsClient (producer)	To use a existing configured AWS KMS as client		AWSKMS
operation (producer)	**Required** The operation to perform		KMSOperations
proxyHost (producer)	To define a proxy host when instantiating the KMS client		String
proxyPort (producer)	To define a proxy port when instantiating the KMS client		Integer
region (producer)	The region in which KMS client needs to work		String
secretKey (producer)	Amazon AWS Secret Key		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Required KMS component options

You have to provide the amazonKmsClient in the Registry or your accessKey and secretKey to access the Amazon KMS service.

26.3. USAGE

26.3.1. Message headers evaluated by the MQ producer

Header	Type	Description
CamelAwsKMSLimit	**Integer**	The limit number of keys to return while performing a listKeys operation
CamelAwsKMSOperation	**String**	The operation we want to perform
CamelAwsKMSDescription	**String**	A key description to use while performing a createKey operation
CamelAwsKMSKeyId	**String**	The key Id

Dependencies

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.16 or higher).

26.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 27. AWS LAMBDA COMPONENT

Available as of Camel version 2.20

The Lambda component supports create, get, list, delete and invoke AWS Lambda functions.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon Lambda. More information are available at Amazon Lambda.

When creating a Lambda function, you need to specify a IAM role which has at least the AWSLambdaBasicExecuteRole policy attached.

Warning

Lambda is regional service. Unlike S3 bucket, Lambda function created in a given region is not available on other regions.

27.1. URI FORMAT

```
aws-lambda://functionName[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

27.2. URI OPTIONS

The AWS Lambda component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS Lambda default configuration		LambdaConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	Amazon AWS Region		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS Lambda endpoint is configured using URI syntax:

```
aws-lambda:function
```

with the following path and query parameters:

27.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
function	**Required** Name of the Lambda function.		String

27.2.2. Query Parameters (8 parameters):

Name	Description	Default	Type
operation (producer)	**Required** The operation to perform. It can be listFunctions, getFunction, createFunction, deleteFunction or invokeFunction		LambdaOperations
region (producer)	Amazon AWS Region		String
awsLambdaClient (advanced)	To use a existing configured AwsLambdaClient as client		AWSLambda
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
proxyHost (proxy)	To define a proxy host when instantiating the Lambda client		String
proxyPort (proxy)	To define a proxy port when instantiating the Lambda client		Integer
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required Lambda component options

You have to provide the awsLambdaClient in the Registry or your accessKey and secretKey to access the Amazon Lambda service.

27.3. USAGE

27.3.1. Message headers evaluated by the Lambda producer

Operation	Header	Type	Description	Required
All	**CamelAwsLambdaOperation**	**String**	The operation we want to perform. Override operation passed as query parameter	Yes
createFunction	**CamelAwsLambdaS3Bucket**	**String**	Amazon S3 bucket name where the .zip file containing your deployment package is stored. This bucket must reside in the same AWS region where you are creating the Lambda function.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaS3Key**	**String**	The Amazon S3 object (the deployment package) key name you want to upload.	No
createFunction	**CamelAwsLambdaS3ObjectVersion**	String	The Amazon S3 object (the deployment package) version you want to upload.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaZipFile**	**String**	The local path of the zip file (the deployment package). Content of zip file can also be put in Message body.	No
createFunction	**CamelAwsLambdaRole**	**String**	The Amazon Resource Name (ARN) of the IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources.	Yes

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaRuntime**	String	The runtime environment for the Lambda function you are uploading. (nodejs, nodejs4.3, nodejs6.10, java8, python2.7, python3.6, dotnetcore1.0, odejs4.3-edge)	Yes
createFunction	**CamelAwsLambdaHandler**	**String**	The function within your code that Lambda calls to begin execution. For Node.js, it is the module-name.export value in your function. For Java, it can be package.class-name::handler or package.class-name.	Yes

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaDescription**	**String**	The user-provided description.	No
createFunction	**CamelAwsLambdaTargetArn**	**String**	The parent object that contains the target ARN (Amazon Resource Name) of an Amazon SQS queue or Amazon SNS topic.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaMemorySize**	**Integer**	The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaKMSKeyArn**	**String**	The Amazon Resource Name (ARN) of the KMS key used to encrypt your function's environment variables. If not provided, AWS Lambda will use a default service key.	No
createFunction	**CamelAwsLambdaPublish**	**Boolean**	This boolean parameter can be used to request AWS Lambda to create the Lambda function and publish a version as an atomic operation.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaTimeout**	**Integer**	The function execution time at which Lambda should terminate the function. The default is 3 seconds.	No
createFunction	**CamelAwsLambdaTracingConfig**	**String**	Your function's tracing settings (Active or PassThrough).	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaEnvironmentVariables**	**Map<String, String>**	The key-value pairs that represent your environment's configuration settings.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaEnvironmentTags**	**Map<String, String>**	The list of tags (key-value pairs) assigned to the new function.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaSecurityGroupIds**	**List<String>**	If your Lambda function accesses resources in a VPC, a list of one or more security groups IDs in your VPC.	No

Operation	Header	Type	Description	Required
createFunction	**CamelAwsLambdaSubnetIds**	**List<String>**	If your Lambda function accesses resources in a VPC, a list of one or more subnet IDs in your VPC.	No

Dependencies

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version**} must be replaced by the actual version of Camel (2.16 or higher).

27.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

- AWS Component

CHAPTER 28. AWS MQ COMPONENT

Available as of Camel version 2.21

The EC2 component supports create, run, start, stop and terminate AWS MQ instances.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon MQ. More information are available at Amazon MQ.

28.1. URI FORMAT

```
aws-mq://label[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

28.2. URI OPTIONS

The AWS MQ component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS MQ default configuration		MQConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	The region in which MQ client needs to work		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS MQ endpoint is configured using URI syntax:

```
aws-mq:label
```

with the following path and query parameters:

28.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
label	**Required** Logical name		String

28.2.2. Query Parameters (8 parameters):

Name	Description	Default	Type
accessKey (producer)	Amazon AWS Access Key		String
amazonMqClient (producer)	To use a existing configured AmazonMQClient as client		AmazonMQ
operation (producer)	**Required** The operation to perform. It can be listBrokers,createBroker,deleteBroker		MQOperations
proxyHost (producer)	To define a proxy host when instantiating the MQ client		String
proxyPort (producer)	To define a proxy port when instantiating the MQ client		Integer
region (producer)	The region in which MQ client needs to work		String
secretKey (producer)	Amazon AWS Secret Key		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Required EC2 component options

You have to provide the amazonEc2Client in the Registry or your accessKey and secretKey to access the Amazon EC2 service.

28.3. USAGE

28.3.1. Message headers evaluated by the MQ producer

Header	Type	Description
CamelAwsMQMaxResults	**String**	The number of results that must be retrieved from listBrokers operation
CamelAwsMQBrokerName	**String**	The broker name
CamelAwsMQOperation	**String**	The operation we want to perform
CamelAwsMQBrokerId	**String**	The broker id
CamelAwsMQBrokerDeploymentMode	**String**	The deployment mode for the broker in the createBroker operation

Dependencies

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.16 or higher).

28.4. SEE ALSO

- Configuring Camel
- Component

- Endpoint
- Getting Started
- AWS Component

CHAPTER 29. AWS S3 STORAGE SERVICE COMPONENT

Available as of Camel version 2.8

The S3 component supports storing and retrieving objetcs from/to Amazon's S3 service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon S3. More information are available at Amazon S3.

29.1. URI FORMAT

```
aws-s3://bucketNameOrArn[?options]
```

The bucket will be created if it don't already exists.
You can append query options to the URI in the following format, ?options=value&option2=value&...

For example in order to read file **hello.txt** from bucket **helloBucket**, use the following snippet:

```
from("aws-s3:helloBucket?accessKey=yourAccessKey&secretKey=yourSecretKey&prefix=hello.txt")
  .to("file:/var/downloaded");
```

29.2. URI OPTIONS

The AWS S3 Storage Service component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS S3 default configuration		S3Configuration
accessKey (common)	Amazon AWS Access Key		String
secretKey (common)	Amazon AWS Secret Key		String
region (common)	The region where the bucket is located. This option is used in the com.amazonaws.services.s3.model.CreateBucketRequest.		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS S3 Storage Service endpoint is configured using URI syntax:

```
aws-s3:bucketNameOrArn
```

with the following path and query parameters:

29.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
bucketNameOrArn	**Required** Bucket name or ARN		String

29.2.2. Query Parameters (50 parameters):

Name	Description	Default	Type
amazonS3Client (common)	Reference to a com.amazonaws.services.sqs.AmazonS3 in the link:registry.htmlRegistry.		AmazonS3
pathStyleAccess (common)	Whether or not the S3 client should use path style access	false	boolean
policy (common)	The policy for this queue to set in the com.amazonaws.services.s3.AmazonS3setBucketPolicy() method.		String
proxyHost (common)	To define a proxy host when instantiating the SQS client		String
proxyPort (common)	Specify a proxy port to be used inside the client definition.		Integer
region (common)	The region in which S3 client needs to work		String
useIAMCredentials (common)	Set whether the S3 client should expect to load credentials on an EC2 instance or to expect static credentials to be passed in.	false	boolean
encryptionMaterials (common)	The encryption materials to use in case of Symmetric/Asymmetric client usage		EncryptionMaterials
useEncryption (common)	Define if encryption must be used or not	false	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
deleteAfterRead (consumer)	Delete objects from S3 after they have been retrieved. The delete is only performed if the Exchange is committed. If a rollback occurs, the object is not deleted. If this option is false, then the same objects will be retrieve over and over again on the polls. Therefore you need to use the Idempotent Consumer EIP in the route to filter out duplicates. You can filter using the link S3ConstantsBUCKET_NAME and link S3ConstantsKEY headers, or only the link S3ConstantsKEY header.	true	boolean
fileName (consumer)	To get the object from the bucket with the given file name		String
includeBody (consumer)	If it is true, the exchange body will be set to a stream to the contents of the file. If false, the headers will be set with the S3 object metadata, but the body will be null. This option is strongly related to autocloseBody option. In case of setting includeBody to true and autocloseBody to false, it will be up to the caller to close the S3Object stream. Setting autocloseBody to true, will close the S3Object stream automatically.	true	boolean
maxConnections (consumer)	Set the maxConnections parameter in the S3 client configuration	60	int
maxMessagesPerPoll (consumer)	Gets the maximum number of messages as a limit to poll at each polling. Is default unlimited, but use 0 or negative number to disable it as unlimited.	10	int
prefix (consumer)	The prefix which is used in the com.amazonaws.services.s3.model.ListObjectsRequest to only consume objects we are interested in.		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean

Name	Description	Default	Type
autocloseBody (consumer)	If this option is true and includeBody is true, then the S3Object.close() method will be called on exchange completion. This option is strongly related to includeBody option. In case of setting includeBody to true and autocloseBody to false, it will be up to the caller to close the S3Object stream. Setting autocloseBody to true, will close the S3Object stream automatically.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
deleteAfterWrite (producer)	Delete file object after the S3 file has been uploaded	false	boolean
multiPartUpload (producer)	If it is true, camel will upload the file with multi part format, the part size is decided by the option of partSize	false	boolean
operation (producer)	The operation to do in case the user don't want to do only an upload		S3Operations
partSize (producer)	Setup the partSize which is used in multi part upload, the default size is 25M.	26214400	long
serverSideEncryption (producer)	Sets the server-side encryption algorithm when encrypting the object using AWS-managed keys. For example use AES256.		String
storageClass (producer)	The storage class to set in the com.amazonaws.services.s3.model.PutObjectRequest request.		String
awsKMSKeyId (producer)	Define the id of KMS key to use in case KMS is enabled		String

Name	Description	Default	Type
useAwsKMS (producer)	Define if KMS must be used or not	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accelerateModeEnabled (advanced)	Define if Accelerate Mode enabled is true or false	false	boolean
chunkedEncodingDisabled (advanced)	Define if disabled Chunked Encoding is true or false	false	boolean
dualstackEnabled (advanced)	Define if Dualstack enabled is true or false	false	boolean
forceGlobalBucketAccess Enabled (advanced)	Define if Force Global Bucket Access enabled is true or false	false	boolean
payloadSigningEnabled (advanced)	Define if Payload Signing enabled is true or false	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean

Name	Description	Default	Type
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required S3 component options

You have to provide the amazonS3Client in the Registry or your accessKey and secretKey to access the Amazon's S3.

29.3. BATCH CONSUMER

This component implements the Batch Consumer.

This allows you for instance to know how many messages exists in this batch and for instance let the Aggregator aggregate this number of messages.

29.4. USAGE

29.4.1. Message headers evaluated by the S3 producer

Header	Type	Description
CamelAwsS3BucketName	**String**	The bucket Name which this object will be stored or which will be used for the current operation
CamelAwsS3BucketDestinationName	**String**	**Camel 2.18**: The bucket Destination Name which will be used for the current operation
CamelAwsS3ContentLength	**Long**	The content length of this object.
CamelAwsS3ContentType	**String**	The content type of this object.
CamelAwsS3ContentControl	**String**	**Camel 2.8.2:** The content control of this object.
CamelAwsS3ContentDisposition	**String**	**Camel 2.8.2:** The content disposition of this object.
CamelAwsS3ContentEncoding	**String**	**Camel 2.8.2:** The content encoding of this object.

Header	Type	Description
CamelAwsS3ContentMD5	**String**	**Camel 2.8.2:** The md5 checksum of this object.
CamelAwsS3DestinationKey	**String**	**Camel 2.18**:The Destination key which will be used for the current operation
CamelAwsS3Key	**String**	The key under which this object will be stored or which will be used for the current operation
CamelAwsS3LastModified	**java.util.Date**	**Camel 2.8.2:** The last modified timestamp of this object.
CamelAwsS3Operation	**String**	**Camel 2.18**: The operation to perform. Permitted values are copyObject, listBuckets, deleteBucket, downloadLink
CamelAwsS3StorageClass	**String**	**Camel 2.8.4:** The storage class of this object.
CamelAwsS3CannedAcl	**String**	**Camel 2.11.0:** The canned acl that will be applied to the object. see **com.amazonaws.services.s3.model.CannedAccessControlList** for allowed values.
CamelAwsS3Acl	**com.amazonaws.services.s3.model.AccessControlList**	**Camel 2.11.0:** a well constructed Amazon S3 Access Control List object. see **com.amazonaws.services.s3.model.AccessControlList** for more details

Header	Type	Description
CamelAwsS3Headers	**Map<String,String>**	**Camel 2.15.0**: support to get or set custom objectMetadata headers.
CamelAwsS3ServerSideEncryption	String	**Camel 2.16:** Sets the server-side encryption algorithm when encrypting the object using AWS-managed keys. For example use AES256.
CamelAwsS3VersionId	**String**	The version Id of the object to be stored or returned from the current operation

29.4.2. Message headers set by the S3 producer

Header	Type	Description
CamelAwsS3ETag	**String**	The ETag value for the newly uploaded object.
CamelAwsS3VersionId	**String**	The **optional** version ID of the newly uploaded object.
CamelAwsS3DownloadLinkExpiration	**String**	The expiration (millis) of URL download link. The link will be stored into **CamelAwsS3DownloadLink** response header.

29.4.3. Message headers set by the S3 consumer

Header	Type	Description
CamelAwsS3Key	**String**	The key under which this object is stored.

Header	Type	Description
CamelAwsS3BucketName	**String**	The name of the bucket in which this object is contained.
CamelAwsS3ETag	**String**	The hex encoded 128-bit MD5 digest of the associated object according to RFC 1864. This data is used as an integrity check to verify that the data received by the caller is the same data that was sent by Amazon S3.
CamelAwsS3LastModified	**Date**	The value of the Last-Modified header, indicating the date and time at which Amazon S3 last recorded a modification to the associated object.
CamelAwsS3VersionId	**String**	The version ID of the associated Amazon S3 object if available. Version IDs are only assigned to objects when an object is uploaded to an Amazon S3 bucket that has object versioning enabled.
CamelAwsS3ContentType	**String**	The Content-Type HTTP header, which indicates the type of content stored in the associated object. The value of this header is a standard MIME type.
CamelAwsS3ContentMD5	**String**	The base64 encoded 128-bit MD5 digest of the associated object (content - not including headers) according to RFC 1864. This data is used as a message integrity check to verify that the data received by Amazon S3 is the same data that the caller sent.
CamelAwsS3ContentLength	**Long**	The Content-Length HTTP header indicating the size of the associated object in bytes.
CamelAwsS3ContentEncoding	**String**	The **optional** Content-Encoding HTTP header specifying what content encodings have been applied to the object and what decoding mechanisms must be applied in order to obtain the media-type referenced by the Content-Type field.

Header	Type	Description
CamelAwsS3ContentDisposition	**String**	The **optional** Content-Disposition HTTP header, which specifies presentational information such as the recommended filename for the object to be saved as.
CamelAwsS3ContentControl	**String**	The **optional** Cache-Control HTTP header which allows the user to specify caching behavior along the HTTP request/reply chain.
CamelAwsS3ServerSideEncryption	String	**Camel 2.16:** The server-side encryption algorithm when encrypting the object using AWS-managed keys.

29.4.4. Advanced AmazonS3 configuration

If your Camel Application is running behind a firewall or if you need to have more control over the **AmazonS3** instance configuration, you can create your own instance:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");

ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonS3 client = new AmazonS3Client(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

and refer to it in your Camel aws-s3 component configuration:

```
from("aws-s3://MyBucket?amazonS3Client=#client&delay=5000&maxMessagesPerPoll=5")
.to("mock:result");
```

29.4.5. Use KMS with the S3 component

To use AWS KMS to encrypt/decrypt data by using AWS infrastructure you can use the options introduced in 2.21.x like in the following example

```
from("file:tmp/test?fileName=test.txt")
    .setHeader(S3Constants.KEY, constant("testFile"))
    .to("aws-s3://mybucket?amazonS3Client=#client&useAwsKMS=true&awsKMSKeyId=3f0637ad-
```

```
296a-3dfe-a796-e60654fb128c");
```

In this way you'll ask to S3, to use the KMS key 3f0637ad-296a-3dfe-a796-e60654fb128c, to encrypt the file test.txt. When you'll ask to download this file, the decryption will be done directly before the download.

29.4.6. Use "useIAMCredentials" with the s3 component

To use AWS IAM credentials, you must first verify that the EC2 in which you are launching the Camel application on has an IAM role associated with it containing the appropriate policies attached to run effectively. Keep in mind that this feature should only be set to "true" on remote instances. To clarify even further, you must still use static credentials locally since IAM is an AWS specific component, but AWS environments should now be easier to manage. After this is implemented and understood, you can set the query parameter "useIAMCredentials" to "true" for AWS environments! To effectively toggle this on and off based on local and remote environments, you can consider enabling this query parameter with system environment variables. For example, your code could set the "useIAMCredentials" query parameter to "true", when the system environment variable called "isRemote" is set to true (there are many other ways to do this and this should act as a simple example). Although it doesn't take away the need for static credentials completely, using IAM credentials on AWS environments takes away the need to refresh on remote environments and adds a major security boost (IAM credentials are refreshed automatically every 6 hours and update when their policies are updated). This is the AWS recommended way to manage credentials and therefore should be used as often as possible.

29.5. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.8 or higher).

29.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 30. AWS SIMPLEDB COMPONENT

Available as of Camel version 2.9

The sdb component supports storing and retrieving data from/to Amazon's SDB service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon SDB. More information are available at Amazon SDB.

30.1. URI FORMAT

```
aws-sdb://domainName[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

30.2. URI OPTIONS

The AWS SimpleDB component has no options.

The AWS SimpleDB endpoint is configured using URI syntax:

```
aws-sdb:domainName
```

with the following path and query parameters:

30.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
domainName	**Required** The name of the domain currently worked with.		String

30.2.2. Query Parameters (10 parameters):

Name	Description	Default	Type
accessKey (producer)	Amazon AWS Access Key		String
amazonSDBClient (producer)	To use the AmazonSimpleDB as the client		AmazonSimpleDB
consistentRead (producer)	Determines whether or not strong consistency should be enforced when data is read.	false	boolean

Name	Description	Default	Type
maxNumberOfDomains (producer)	The maximum number of domain names you want returned. The range is 1 to 100.		Integer
operation (producer)	Operation to perform	PutAttributes	SdbOperations
proxyHost (producer)	To define a proxy host when instantiating the SDB client		String
proxyPort (producer)	To define a proxy port when instantiating the SDB client		Integer
region (producer)	The region in which SDB client needs to work		String
secretKey (producer)	Amazon AWS Secret Key		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Required SDB component options

You have to provide the amazonSDBClient in the Registry or your accessKey and secretKey to access the Amazon's SDB.

30.3. USAGE

30.3.1. Message headers evaluated by the SDB producer

Header	Type	Description
CamelAwsSdbAttributes	**Collection<Attribute>**	List of attributes to be acted upon.
CamelAwsSdbAttributeNames	**Collection<String>**	The names of the attributes to be retrieved.

Header	Type	Description
CamelAwsSdbConsistentRead	**Boolean**	Determines whether or not strong consistency should be enforced when data is read.
CamelAwsSdbDeletableItems	**Collection<DeletableItem>**	A list of items on which to perform the delete operation in a batch.
CamelAwsSdbDomainName	**String**	The name of the domain currently worked with.
CamelAwsSdbItemName	**String**	The unique key for this item
CamelAwsSdbMaxNumberOfDomains	**Integer**	The maximum number of domain names you want returned. The range is 1 * to 100.
CamelAwsSdbNextToken	**String**	A string specifying where to start the next list of domain/item names.
CamelAwsSdbOperation	**String**	To override the operation from the URI options.

Header	Type	Description
CamelAwsSdbReplaceableAttributes	**Collection<ReplaceableAttribute>**	List of attributes to put in an Item.
CamelAwsSdbReplaceableItems	**Collection<ReplaceableItem>**	A list of items to put in a Domain.
CamelAwsSdbSelectExpression	**String**	The expression used to query the domain.
CamelAwsSdbUpdateCondition	**UpdateCondition**	The update condition which, if specified, determines whether the specified attributes will be updated/deleted or not.

30.3.2. Message headers set during DomainMetadata operation

Header	Type	Description
CamelAwsSdbTimestamp	**Integer**	The data and time when metadata was calculated, in Epoch (UNIX) seconds.
CamelAwsSdbItemCount	**Integer**	The number of all items in the domain.

Header	Type	Description
CamelAwsSdbAttributeNameCount	**Integer**	The number of unique attribute names in the domain.
CamelAwsSdbAttributeValueCount	**Integer**	The number of all attribute name/value pairs in the domain.
CamelAwsSdbAttributeNameSize	**Long**	The total size of all unique attribute names in the domain, in bytes.
CamelAwsSdbAttributeValueSize	**Long**	The total size of all attribute values in the domain, in bytes.
CamelAwsSdbItemNameSize	**Long**	The total size of all item names in the domain, in bytes.

30.3.3. Message headers set during GetAttributes operation

Header	Type	Description
CamelAwsSdbAttributes	**List<Attribute>**	The list of attributes returned by the operation.

30.3.4. Message headers set during ListDomains operation

Header	Type	Description
CamelAwsSdbDomainNames	**List<String>**	A list of domain names that match the expression.
CamelAwsSdbNextToken	**String**	An opaque token indicating that there are more domains than the specified MaxNumberOfDomains still available.

30.3.5. Message headers set during Select operation

Header	Type	Description
CamelAwsSdbItems	**List<Item>**	A list of items that match the select expression.
CamelAwsSdbNextToken	**String**	An opaque token indicating that more items than MaxNumberOfItems were matched, the response size exceeded 1 megabyte, or the execution time exceeded 5 seconds.

30.3.6. Advanced AmazonSimpleDB configuration

If you need more control over the **AmazonSimpleDB** instance configuration you can create your own instance and refer to it from the URI:

```
from("direct:start")
.to("aws-sdb://domainName?amazonSDBClient=#client");
```

The **#client** refers to a **AmazonSimpleDB** in the Registry.

For example if your Camel Application is running behind a firewall:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonSimpleDB client = new AmazonSimpleDBClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

30.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.8.4 or higher).

30.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 31. AWS SIMPLE EMAIL SERVICE COMPONENT

Available as of Camel version 2.9

The ses component supports sending emails with Amazon's SES service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon SES. More information are available at Amazon SES.

31.1. URI FORMAT

```
aws-ses://from[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

31.2. URI OPTIONS

The AWS Simple Email Service component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS SES default configuration		SesConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	The region in which SES client needs to work		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS Simple Email Service endpoint is configured using URI syntax:

```
aws-ses:from
```

with the following path and query parameters:

31.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
from	**Required** The sender's email address.		String

31.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
amazonSESClient (producer)	To use the AmazonSimpleEmailService as the client		AmazonSimpleEmail Service
proxyHost (producer)	To define a proxy host when instantiating the SES client		String
proxyPort (producer)	To define a proxy port when instantiating the SES client		Integer
region (producer)	The region in which SES client needs to work		String
replyToAddresses (producer)	List of reply-to email address(es) for the message, override it using 'CamelAwsSesReplyToAddresses' header.		List
returnPath (producer)	The email address to which bounce notifications are to be forwarded, override it using 'CamelAwsSesReturnPath' header.		String
subject (producer)	The subject which is used if the message header 'CamelAwsSesSubject' is not present.		String
to (producer)	List of destination email address. Can be overriden with 'CamelAwsSesTo' header.		List
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required SES component options

You have to provide the amazonSESClient in the Registry or your accessKey and secretKey to access the Amazon's SES.

31.3. USAGE

31.3.1. Message headers evaluated by the SES producer

Header	Type	Description
CamelAwsSesFrom	**String**	The sender's email address.
CamelAwsSesTo	**List<String>**	The destination(s) for this email.
CamelAwsSesSubject	**String**	The subject of the message.
CamelAwsSesReplyToAddresses	**List<String>**	The reply-to email address(es) for the message.
CamelAwsSesReturnPath	**String**	The email address to which bounce notifications are to be forwarded.
CamelAwsSesHtmlEmail	**Boolean**	**Since Camel 2.12.3** The flag to show if email content is HTML.

31.3.2. Message headers set by the SES producer

Header	Type	Description
CamelAwsSesMessageId	**String**	The Amazon SES message ID.

31.3.3. Advanced AmazonSimpleEmailService configuration

If you need more control over the **AmazonSimpleEmailService** instance configuration you can create your own instance and refer to it from the URI:

```
from("direct:start")
.to("aws-ses://example@example.com?amazonSESClient=#client");
```

The **#client** refers to a **AmazonSimpleEmailService** in the Registry.

For example if your Camel Application is running behind a firewall:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);
AmazonSimpleEmailService client = new AmazonSimpleEmailServiceClient(awsCredentials,
clientConfiguration);

registry.bind("client", client);
```

31.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.8.4 or higher).

31.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 32. AWS SIMPLE NOTIFICATION SYSTEM COMPONENT

Available as of Camel version 2.8

The SNS component allows messages to be sent to an Amazon Simple Notification Topic. The implementation of the Amazon API is provided by the AWS SDK.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon SNS. More information are available at Amazon SNS.

32.1. URI FORMAT

```
aws-sns://topicNameOrArn[?options]
```

The topic will be created if they don't already exists.
You can append query options to the URI in the following format, **?options=value&option2=value&...**

32.2. URI OPTIONS

The AWS Simple Notification System component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS SNS default configuration		SnsConfiguration
accessKey (producer)	Amazon AWS Access Key		String
secretKey (producer)	Amazon AWS Secret Key		String
region (producer)	The region in which SNS client needs to work		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS Simple Notification System endpoint is configured using URI syntax:

```
aws-sns:topicNameOrArn
```

with the following path and query parameters:

32.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
topicNameOrArn	**Required** Topic name or ARN		String

32.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
amazonSNSClient (producer)	To use the AmazonSNS as the client		AmazonSNS
headerFilterStrategy (producer)	To use a custom HeaderFilterStrategy to map headers to/from Camel.		HeaderFilterStrategy
messageStructure (producer)	The message structure to use such as json		String
policy (producer)	The policy for this queue		String
proxyHost (producer)	To define a proxy host when instantiating the SNS client		String
proxyPort (producer)	To define a proxy port when instantiating the SNS client		Integer
region (producer)	The region in which SNS client needs to work		String
subject (producer)	The subject which is used if the message header 'CamelAwsSnsSubject' is not present.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required SNS component options

You have to provide the amazonSNSClient in the Registry or your accessKey and secretKey to access the Amazon's SNS.

32.3. USAGE

32.3.1. Message headers evaluated by the SNS producer

Header	Type	Description
CamelAwsSnsSubject	**String**	The Amazon SNS message subject. If not set, the subject from the **SnsConfiguration** is used.

32.3.2. Message headers set by the SNS producer

Header	Type	Description
CamelAwsSnsMessageId	**String**	The Amazon SNS message ID.

32.3.3. Advanced AmazonSNS configuration

If you need more control over the **AmazonSNS** instance configuration you can create your own instance and refer to it from the URI:

```
from("direct:start")
.to("aws-sns://MyTopic?amazonSNSClient=#client");
```

The **#client** refers to a **AmazonSNS** in the Registry.

For example if your Camel Application is running behind a firewall:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);
AmazonSNS client = new AmazonSNSClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

32.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
  <groupId>org.apache.camel</groupId>
```

```
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.8 or higher).

32.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 33. AWS SIMPLE QUEUE SERVICE COMPONENT

Available as of Camel version 2.6

The sqs component supports sending and receiving messages to Amazon's SQS service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon SQS. More information are available at Amazon SQS.

33.1. URI FORMAT

```
aws-sqs://queueNameOrArn[?options]
```

The queue will be created if they don't already exists.
You can append query options to the URI in the following format, ?options=value&option2=value&...

33.2. URI OPTIONS

The AWS Simple Queue Service component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS SQS default configuration		SqsConfiguration
accessKey (common)	Amazon AWS Access Key		String
secretKey (common)	Amazon AWS Secret Key		String
region (common)	Specify the queue region which could be used with queueOwnerAWSAccountId to build the service URL.		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS Simple Queue Service endpoint is configured using URI syntax:

```
aws-sqs:queueNameOrArn
```

with the following path and query parameters:

33.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
queueNameOrArn	**Required** Queue name or ARN		String

33.2.2. Query Parameters (46 parameters):

Name	Description	Default	Type
amazonAWSHost (common)	The hostname of the Amazon AWS cloud.	amazonaws.com	String
amazonSQSClient (common)	To use the AmazonSQS as client		AmazonSQS
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to map headers to/from Camel.		HeaderFilterStrategy
queueOwnerAWSAccountId (common)	Specify the queue owner aws account id when you need to connect the queue with different account owner.		String
region (common)	Specify the queue region which could be used with queueOwnerAWSAccountId to build the service URL.		String
attributeNames (consumer)	A list of attribute names to receive when consuming. Multiple names can be separated by comma.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
concurrentConsumers (consumer)	Allows you to use multiple threads to poll the sqs queue to increase throughput	1	int
defaultVisibilityTimeout (consumer)	The default visibility timeout (in seconds)		Integer
deleteAfterRead (consumer)	Delete message from SQS after it has been read	true	boolean

Name	Description	Default	Type
deleteIfFiltered (consumer)	Whether or not to send the DeleteMessage to the SQS queue if an exchange fails to get through a filter. If 'false' and exchange does not make it through a Camel filter upstream in the route, then don't send DeleteMessage.	true	boolean
extendMessageVisibility (consumer)	If enabled then a scheduled background task will keep extending the message visibility on SQS. This is needed if it takes a long time to process the message. If set to true defaultVisibilityTimeout must be set. See details at Amazon docs.	false	boolean
maxMessagesPerPoll (consumer)	Gets the maximum number of messages as a limit to poll at each polling. Is default unlimited, but use 0 or negative number to disable it as unlimited.		int
messageAttributeNames (consumer)	A list of message attribute names to receive when consuming. Multiple names can be separated by comma.		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
visibilityTimeout (consumer)	The duration (in seconds) that the received messages are hidden from subsequent retrieve requests after being retrieved by a ReceiveMessage request to set in the com.amazonaws.services.sqs.model.SetQueueAttributesRequest. This only make sense if its different from defaultVisibilityTimeout. It changes the queue visibility timeout attribute permanently.		Integer
waitTimeSeconds (consumer)	Duration in seconds (0 to 20) that the ReceiveMessage action call will wait until a message is in the queue to include in the response.		Integer
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
delaySeconds (producer)	Delay sending messages for a number of seconds.		Integer
messageDeduplicationId Strategy (producer)	Only for FIFO queues. Strategy for setting the messageDeduplicationId on the message. Can be one of the following options: useExchangeId, useContentBasedDeduplication. For the useContentBasedDeduplication option, no messageDeduplicationId will be set on the message.	useExchangeId	MessageDeduplicationId Strategy
messageGroupId Strategy (producer)	Only for FIFO queues. Strategy for setting the messageGroupId on the message. Can be one of the following options: useConstant, useExchangeId, usePropertyValue. For the usePropertyValue option, the value of property CamelAwsMessageGroupId will be used.		MessageGroupIdStrategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long

Name	Description	Default	Type
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
proxyHost (proxy)	To define a proxy host when instantiating the SQS client		String
proxyPort (proxy)	To define a proxy port when instantiating the SQS client		Integer
maximumMessageSize (queue)	The maximumMessageSize (in bytes) an SQS message can contain for this queue.		Integer
messageRetentionPeriod (queue)	The messageRetentionPeriod (in seconds) a message will be retained by SQS for this queue.		Integer

Name	Description	Default	Type
policy (queue)	The policy for this queue		String
receiveMessageWaitTime Seconds (queue)	If you do not specify WaitTimeSeconds in the request, the queue attribute ReceiveMessageWaitTimeSeconds is used to determine how long to wait.		Integer
redrivePolicy (queue)	Specify the policy that send message to DeadLetter queue. See detail at Amazon docs.		String
accessKey (security)	Amazon AWS Access Key		String
secretKey (security)	Amazon AWS Secret Key		String

Required SQS component options

You have to provide the amazonSQSClient in the Registry or your accessKey and secretKey to access the Amazon's SQS.

33.3. BATCH CONSUMER

This component implements the Batch Consumer.

This allows you for instance to know how many messages exists in this batch and for instance let the Aggregator aggregate this number of messages.

33.4. USAGE

33.4.1. Message headers set by the SQS producer

Header	Type	Description
CamelAwsSqsMD5OfBody	**String**	The MD5 checksum of the Amazon SQS message.
CamelAwsSqsMessageId	**String**	The Amazon SQS message ID.

Header	Type	Description
CamelAwsSqsDelaySeconds	**Integer**	Since **Camel 2.11**, the delay seconds that the Amazon SQS message can be see by others.

33.4.2. Message headers set by the SQS consumer

Header	Type	Description
CamelAwsSqsMD5OfBody	**String**	The MD5 checksum of the Amazon SQS message.
CamelAwsSqsMessageId	**String**	The Amazon SQS message ID.
CamelAwsSqsReceiptHandle	**String**	The Amazon SQS message receipt handle.
CamelAwsSqsAttributes	**Map<String, String>**	The Amazon SQS message attributes.

33.4.3. Advanced AmazonSQS configuration

If your Camel Application is running behind a firewall or if you need to have more control over the AmazonSQS instance configuration, you can create your own instance:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");

ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);
```

```
AmazonSQS client = new AmazonSQSClient(awsCredentials, clientConfiguration);

registry.bind("client", client);
```

and refer to it in your Camel aws-sqs component configuration:

```
from("aws-sqs://MyQueue?amazonSQSClient=#client&delay=5000&maxMessagesPerPoll=5")
.to("mock:result");
```

33.5. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.6 or higher).

33.6. JMS-STYLE SELECTORS

SQS does not allow selectors, but you can effectively achieve this by using the Camel Filter EIP and setting an appropriate **visibilityTimeout**. When SQS dispatches a message, it will wait up to the visibility timeout before it will try to dispatch the message to a different consumer unless a DeleteMessage is received. By default, Camel will always send the DeleteMessage at the end of the route, unless the route ended in failure. To achieve appropriate filtering and not send the DeleteMessage even on successful completion of the route, use a Filter:

```
from("aws-sqs://MyQueue?
amazonSQSClient=#client&defaultVisibilityTimeout=5000&deleteIfFiltered=false")
.filter("${header.login} == true")
.to("mock:result");
```

In the above code, if an exchange doesn't have an appropriate header, it will not make it through the filter AND also not be deleted from the SQS queue. After 5000 miliseconds, the message will become visible to other consumers.

33.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- AWS Component

CHAPTER 34. AWS SIMPLE WORKFLOW COMPONENT

Available as of Camel version 2.13

The Simple Workflow component supports managing workflows from Amazon's Simple Workflow service.

Prerequisites

You must have a valid Amazon Web Services developer account, and be signed up to use Amazon Simple Workflow. More information are available at Amazon Simple Workflow.

34.1. URI FORMAT

```
aws-swf://<workflow|activity>[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

34.2. URI OPTIONS

The AWS Simple Workflow component supports 5 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	The AWS SWF default configuration		SWFConfiguration
accessKey (common)	Amazon AWS Access Key.		String
secretKey (common)	Amazon AWS Secret Key.		String
region (common)	Amazon AWS Region.		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The AWS Simple Workflow endpoint is configured using URI syntax:

```
aws-swf:type
```

with the following path and query parameters:

34.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
type	**Required** Activity or workflow		String

34.2.2. Query Parameters (30 parameters):

Name	Description	Default	Type
amazonSWClient (common)	To use the given AmazonSimpleWorkflowClient as client		AmazonSimpleWorkflow Client
dataConverter (common)	An instance of com.amazonaws.services.simpleworkflow.flow.DataConverter to use for serializing/deserializing the data.		DataConverter
domainName (common)	The workflow domain to use.		String
eventName (common)	The workflow or activity event name to use.		String
region (common)	Amazon AWS Region.		String
version (common)	The workflow or activity event version to use.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
clientConfiguration Parameters (advanced)	To configure the ClientConfiguration using the key/values from the Map.		Map

Name	Description	Default	Type
startWorkflowOptions Parameters (advanced)	To configure the StartWorkflowOptions using the key/values from the Map.		Map
sWClientParameters (advanced)	To configure the AmazonSimpleWorkflowClient using the key/values from the Map.		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
activityList (activity)	The list name to consume activities from.		String
activitySchedulingOptions (activity)	Activity scheduling options		ActivitySchedulingOptions
activityThreadPoolSize (activity)	Maximum number of threads in work pool for activity.	100	int
activityTypeExecution Options (activity)	Activity execution options		ActivityTypeExecution Options
activityTypeRegistration Options (activity)	Activity registration options		ActivityTypeRegistrationOptions
childPolicy (workflow)	The policy to use on child workflows when terminating a workflow.		String
executionStartToClose Timeout (workflow)	Set the execution start to close timeout.	3600	String
operation (workflow)	Workflow operation	START	String
signalName (workflow)	The name of the signal to send to the workflow.		String
stateResultType (workflow)	The type of the result when a workflow state is queried.		String

Name	Description	Default	Type
taskStartToCloseTimeout (workflow)	Set the task start to close timeout.	600	String
terminationDetails (workflow)	Details for terminating a workflow.		String
terminationReason (workflow)	The reason for terminating a workflow.		String
workflowList (workflow)	The list name to consume workflows from.		String
workflowTypeRegistration Options (workflow)	Workflow registration options		WorkflowTypeRegistrationOptions
accessKey (security)	Amazon AWS Access Key.		String
secretKey (security)	Amazon AWS Secret Key.		String

Required SWF component options

You have to provide the amazonSWClient in the Registry or your accessKey and secretKey to access the Amazon's Simple Workflow Service.

34.3. USAGE

34.3.1. Message headers evaluated by the SWF Workflow Producer

A workflow producer allows interacting with a workflow. It can start a new workflow execution, query its state, send signals to a running workflow, or terminate and cancel it.

Header	Type	Description
CamelSWFOperation	**String**	The operation to perform on the workflow. Supported operations are: SIGNAL, CANCEL, TERMINATE, GET_STATE, START, DESCRIBE, GET_HISTORY.
CamelSWFWorkflowId	**String**	A workflow ID to use.

Header	Type	Description
CamelAwsDdbKeyCamelSWFRunId	**String**	A worfklow run ID to use.
CamelSWFStateResultType	**String**	The type of the result when a workflow state is queried.
CamelSWFEventName	**String**	The workflow or activity event name to use.
CamelSWFVersion	**String**	The workflow or activity event version to use.
CamelSWFReason	**String**	The reason for terminating a workflow.
CamelSWFDetails	**String**	Details for terminating a workflow.
CamelSWFChildPolicy	**String**	The policy to use on child workflows when terminating a workflow.

34.3.2. Message headers set by the SWF Workflow Producer

Header	Type	Description
CamelSWFWorkflowId	**String**	The worfklow ID used or newly generated.

Header	Type	Description
CamelAwsDdbKey CamelSWFRunId	**String**	The worfklow run ID used or generated.

34.3.3. Message headers set by the SWF Workflow Consumer

A workflow consumer represents the workflow logic. When it is started, it will start polling workflow decision tasks and process them. In addition to processing decision tasks, a workflow consumer route, will also receive signals (send from a workflow producer) or state queries. The primary purpose of a workflow consumer is to schedule activity tasks for execution using activity producers. Actually activity tasks can be scheduled only from a thread started by a workflow consumer.

Header	Type	Description
CamelSWFAction	**String**	Indicates what type is the current event: CamelSWFActionExecute, CamelSWFSignalReceivedAction or CamelSWFGetStateAction.
CamelSWFWorkflowReplaying	**boolean**	Indicates whether the current decision task is a replay or not.
CamelSWFWorkflowStartTime	**long**	The time of the start event for this decision task.

34.3.4. Message headers set by the SWF Activity Producer

An activity producer allows scheduling activity tasks. An activity producer can be used only from a thread started by a workflow consumer ie, it can process synchronous exchanges started by a workflow consumer.

Header	Type	Description
CamelSWFEventName	**String**	The activity name to schedule.

Header	Type	Description
CamelSWFVersion	**String**	The activity version to schedule.

34.3.5. Message headers set by the SWF Activity Consumer

Header	Type	Description
CamelSWFTaskToken	**String**	The task token that is required to report task completion for manually completed tasks.

34.3.6. Advanced amazonSWClient configuration

If you need more control over the AmazonSimpleWorkflowClient instance configuration you can create your own instance and refer to it from the URI:

The **#client** refers to a AmazonSimpleWorkflowClient in the Registry.

For example if your Camel Application is running behind a firewall:

```
AWSCredentials awsCredentials = new BasicAWSCredentials("myAccessKey", "mySecretKey");
ClientConfiguration clientConfiguration = new ClientConfiguration();
clientConfiguration.setProxyHost("http://myProxyHost");
clientConfiguration.setProxyPort(8080);

AmazonSimpleWorkflowClient client = new AmazonSimpleWorkflowClient(awsCredentials,
clientConfiguration);

registry.bind("client", client);
```

34.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-aws</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.13 or higher).

34.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

AWS Component

CHAPTER 35. AWS XRAY COMPONENT

Available as of Camel 2.21

The camel-aws-xray component is used for tracing and timing incoming and outgoing Camel messages using AWS XRay.

Events (subsegments) are captured for incoming and outgoing messages being sent to/from Camel.

35.1. DEPENDENCY

In order to include AWS XRay support into Camel, the archive containing the Camel related AWS XRay related classes need to be added to the project. In addition to that, AWS XRay libraries also need to be available.

To include both, AWS XRay and Camel, dependencies use the following Maven imports:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-xray-recorder-sdk-bom</artifactId>
      <version>1.3.1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-aws-xray</artifactId>
    </dependency>

    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-xray-recorder-sdk-core</artifactId>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-xray-recorder-sdk-aws-sdk</artifactId>
    </dependency>
<dependencies>
```

35.2. CONFIGURATION

The configuration properties for the AWS XRay tracer are:

Option	Default	Description
addExcludePatterns		Sets exclude pattern(s) that will disable tracing for Camel messages that matches the pattern. The content is a Set<String> where the key is a pattern matching routeId's. The pattern uses the rules from Intercept.
setTracingStrategy	NoopTracingStrategy	Allows a custom Camel **InterceptStrategy** to be provided in order to track invoked processor definitions like **BeanDefinition** or **ProcessDefinition**. **TraceAnnotatedTracingStrategy** will track any classes invoked via **.bean(...)** or **.process(...)** that contain a **@XRayTrace** annotation at class level.

There is currently only one way an AWS XRay tracer can be configured to provide distributed tracing for a Camel application:

35.2.1. Explicit

Include the **camel-aws-xray** component in your POM, along with any specific dependencies associated with the AWS XRay Tracer.

To explicitly configure AWS XRay support, instantiate the **XRayTracer** and initialize the camel context. You can optionally specify a **Tracer**, or alternatively it can be implicitly discovered using the **Registry** or **ServiceLoader**.

```
XRayTracer xrayTracer = new XRayTracer();
// By default it uses a NoopTracingStrategy, but you can override it with a specific InterceptStrategy implementation.
xrayTracer.setTracingStrategy(...);
// And then initialize the context
xrayTracer.init(camelContext);
```

To use XRayTracer in XML, all you need to do is to define the AWS XRay tracer bean. Camel will automatically discover and use it.

```
<bean id="tracingStrategy" class="..."/>
<bean id="aws-xray-tracer" class="org.apache.camel.component.aws.xray.XRayTracer" />
  <property name="tracer" ref="tracingStrategy"/>
</bean>
```

In case of the default **NoopTracingStrategy** only the creation and deletion of exchanges is tracked but not the invocation of certain beans or EIP patterns.

35.2.2. Tracking of comprehensive route execution

In order to track the execution of an exchange among multiple routes, on exchange creation a unique trace ID is generated and stored in the headers if no corresponding value was yet available. This trace ID is copied over to new exchanges in order to keep a consistent view of the processed exchange.

As AWS XRay traces work on a thread-local basis the current sub/segment should be copied over to the new thread and set as explained in in the AWS XRay documentation . The Camel AWS XRay component therefore provides an additional header field that the component will use in order to set the passed AWS XRay **Entity** to the new thread and thus keep the tracked data to the route rather than exposing a new segment which seems uncorrelated with any of the executed routes.

The component will use the following constants found in the headers of the exchange:

Header	Description
Camel-AWS-XRay-Trace-ID	Contains a reference to the AWS XRay **TraceID** object to provide a comprehensive view of the invoked routes
Camel-AWS-XRay-Trace-Entity	Contains a reference to the actual AWS XRay **Segment** or **Subsegment** which is copied over to the new thread. This header should be set in case a new thread is spawned and the performed tasks should be exposed as part of the executed route instead of creating a new unrelated segment.

Note that the AWS XRay **Entity** (i.e., **Segment** and **Subsegment**) are not serializable and therefore should not get passed to other JVM processes.

35.3. EXAMPLE

You can find an example demonstrating the way to configure AWS XRay tracing within the tests accompanying this project.

CHAPTER 36. CAMEL COMPONENTS FOR WINDOWS AZURE SERVICES

The Camel Components for Windows Azure Services provide connectivity to Azure services from Camel.

Azure Service	Camel Component	Camel Version	Component Description
Storage Blob Service	Azure-Blob	2.9.0	Supports storing and retrieving of blobs
Storage Queue Service	Azure-Queue	2.9.0	Supports storing and retrieving of messages in the queues

CHAPTER 37. AZURE STORAGE BLOB SERVICE COMPONENT

Available as of Camel version 2.19

The Azure Blob component supports storing and retrieving the blobs to/from Azure Storage Blob service.

Prerequisites

You must have a valid Windows Azure Storage account. More information is available at Azure Documentation Portal.

37.1. URI FORMAT

```
azure-blob://accountName/containerName[/blobName][?options]
```

In most cases a blobName is required and the blob will be created if it does not already exist.
You can append query options to the URI in the following format, ?options=value&option2=value&...

For example in order to download a blob content from the public block blob **blockBlob** located on the **container1** in the **camelazure** storage account, use the following snippet:

```
from("azure-blob:/camelazure/container1/blockBlob").
to("file://blobdirectory");
```

37.2. URI OPTIONS

The Azure Storage Blob Service component has no options.

The Azure Storage Blob Service endpoint is configured using URI syntax:

```
azure-blob:containerOrBlobUri
```

with the following path and query parameters:

37.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
containerOrBlobUri	**Required** Container or Blob compact Uri		String

37.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
azureBlobClient (common)	The blob service client		CloudBlob
blobOffset (common)	Set the blob offset for the upload or download operations, default is 0	0	Long
blobType (common)	Set a blob type, 'blockblob' is default	blockblob	BlobType
closeStreamAfterRead (common)	Close the stream after read or keep it open, default is true	true	boolean
credentials (common)	Set the storage credentials, required in most cases		StorageCredentials
dataLength (common)	Set the data length for the download or page blob upload operations		Long
fileDir (common)	Set the file directory where the downloaded blobs will be saved to		String
publicForRead (common)	Storage resources can be public for reading their content, if this property is enabled then the credentials do not have to be set	false	boolean
streamReadSize (common)	Set the minimum read size in bytes when reading the blob content		int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
blobMetadata (producer)	Set the blob meta-data		Map
blobPrefix (producer)	Set a prefix which can be used for listing the blobs		String
closeStreamAfterWrite (producer)	Close the stream after write or keep it open, default is true	true	boolean
operation (producer)	Blob service operation hint to the producer	listBlobs	BlobServiceOperations
streamWriteSize (producer)	Set the size of the buffer for writing block and page blocks		int
useFlatListing (producer)	Specify if the flat or hierarchical blob listing should be used	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Required Azure Storage Blob Service component options

You have to provide the containerOrBlob name and the credentials if the private blob needs to be accessed.

37.3. USAGE

37.3.1. Message headers evaluated by the Azure Storage Blob Service producer

Header	Type	Description

37.3.2. Message headers set by the Azure Storage Blob Service producer

Header	Type	Description
CamelFileName	**String**	The file name for the downloaded blob content.

37.3.3. Message headers set by the Azure Storage Blob Service producer consumer

Header	Type	Description
CamelFileName	**String**	The file name for the downloaded blob content.

37.3.4. Azure Blob Service operations

Operations common to all block types

Operation	Description
getBlob	Get the content of the blob. You can restrict the output of this operation to a blob range.
deleteBlob	Delete the blob.
listBlobs	List the blobs.

Block blob operations

Operation	Description
updateBlockBlob	Put block blob content that either creates a new block blob or overwrites the existing block blob content.
uploadBlobBlocks	Upload block blob content, by first generating a sequence of blob blocks and then committing them to a blob. If you enable the message **CommitBlockListLater** property, you can execute the commit later with the **commitBlobBlockList** operation. You can later update individual block blobs.
commitBlobBlockList	Commit a sequence of blob blocks to the block list that you previously uploaded to the blob service (by using the **updateBlockBlob** operation with the message **CommitBlockListLater** property enabled).
getBlobBlockList	Get the block blob list.

Append blob operations

Operation	Description
createAppendBlob	Create an append block. By default, if the block already exists then it is not reset. Note that you can alternately create an append blob by enabling the message **AppendBlobCreated** property and using the **updateAppendBlob** operation.

Operation	Description
updateAppendBlob	Append the new content to the blob. This operation also creates the blob if it does not already exist and if you enabled a message **AppendBlobCreated** property.

Page Block operations

Operation	Description
createPageBlob	Create a page block. By default, if the block already exists then it is not reset. Note that you can also create a page blob (and set its contents) by enabling a message **PageBlobCreated** property and by using the **updatePageBlob** operation.
updatePageBlob	Create a page block (unless you enable a message **PageBlobCreated** property and the identically named block already exists) and set the content of this blob.
resizePageBlob	Resize the page blob.
clearPageBlob	Clear the page blob.
getPageBlobRanges	Get the page blob page ranges.

37.3.5. Azure Blob Client configuration

If your Camel Application is running behind a firewall or if you need to have more control over the Azure Blob Client configuration, you can create your own instance:

```
StorageCredentials credentials = new StorageCredentialsAccountAndKey("camelazure", "thekey");

CloudBlob client = new CloudBlob("camelazure", credentials);

registry.bind("azureBlobClient", client);
```

and refer to it in your Camel azure-blob component configuration:

```
from("azure-blob:/camelazure/container1/blockBlob?azureBlobClient=#client")
.to("mock:result");
```

37.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-azure</artifactId>
```

```
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.19.0 or higher).

37.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Azure Component

CHAPTER 38. AZURE STORAGE QUEUE SERVICE COMPONENT

Available as of Camel version 2.19

The Azure Queue component supports storing and retrieving the messages to/from Azure Storage Queue service.

Prerequisites

You must have a valid Windows Azure Storage account. More information is available at Azure Documentation Portal.

38.1. URI FORMAT

```
azure-queue://accountName/queueName[?options]
```

The queue will be created if it does not already exist.
You can append query options to the URI in the following format, ?options=value&option2=value&...

For example in order to get a message content from the queue **messageQueue** in the **camelazure** storage account and, use the following snippet:

```
from("azure-queue:/camelazure/messageQueue").
to("file://queuedirectory");
```

38.2. URI OPTIONS

The Azure Storage Queue Service component has no options.

The Azure Storage Queue Service endpoint is configured using URI syntax:

```
azure-queue:containerAndQueueUri
```

with the following path and query parameters:

38.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
containerAndQueueUri	**Required** Container Queue compact Uri		String

38.2.2. Query Parameters (10 parameters):

Name	Description	Default	Type
azureQueueClient (common)	The queue service client		CloudQueue
credentials (common)	Set the storage credentials, required in most cases		StorageCredentials
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
messageTimeToLive (producer)	Message Time To Live in seconds		int
messageVisibilityDelay (producer)	Message Visibility Delay in seconds		int
operation (producer)	Queue service operation hint to the producer	listQueues	QueueServiceOperations
queuePrefix (producer)	Set a prefix which can be used for listing the queues		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Required Azure Storage Queue Service component options

You have to provide the containerAndQueue URI and the credentials.

38.3. USAGE

38.3.1. Message headers evaluated by the Azure Storage Queue Service producer

Header	Type	Description

38.3.2. Message headers set by the Azure Storage Queue Service producer

Header	Type	Description

38.3.3. Message headers set by the Azure Storage Queue Service producer consumer

Header	Type	Description

38.3.4. Azure Queue Service operations

Operation	Description
listQueues	List the queues.
createQueue	Create the queue.
deleteQueue	Delete the queue.
addMessage	Add a message to the queue.
retrieveMessage	Retrieve a message from the queue.
peekMessage	View the message inside the queue, for example, to determine whether the message arrived at the correct queue.
updateMessage	Update the message in the queue.
deleteMessage	Delete the message in the queue.

38.3.5. Azure Queue Client configuration

If your Camel Application is running behind a firewall or if you need to have more control over the Azure Queue Client configuration, you can create your own instance:

```
StorageCredentials credentials = new StorageCredentialsAccountAndKey("camelazure", "thekey");

CloudQueue client = new CloudQueue("camelazure", credentials);

registry.bind("azureQueueClient", client);
```

and refer to it in your Camel azure-queue component configuration:

```
from("azure-queue:/camelazure/messageQueue?azureQueueClient=#client")
.to("mock:result");
```

38.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-azure</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.19.0 or higher).

38.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Azure Component

CHAPTER 39. BARCODE DATAFORMAT

Available as of Camel version 2.14

The barcode data format is based on the zxing library. The goal of this component is to create a barcode image from a String (marshal) and a String from a barcode image (unmarshal). You're free to use all features that zxing offers.

39.1. DEPENDENCIES

To use the barcode data format in your camel routes you need to add the a dependency on **camel-barcode** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-barcode</artifactId>
  <version>x.x.x</version>
</dependency>
```

39.2. BARCODE OPTIONS

The Barcode dataformat supports 5 options which are listed below.

Name	Default	Java Type	Description
width		**Integer**	Width of the barcode
height		**Integer**	Height of the barcode
imageType		**String**	Image type of the barcode such as png
barcodeFormat		**String**	Barcode format such as QR-Code
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

39.3. USING THE JAVA DSL

First you have to initialize the barcode data fomat class. You can use the default constructor, or one of parameterized (see JavaDoc). The default values are:

Parameter	Default Value
imageType (BarcodeImageType)	PNG
width	100 px
height	100 px
encoding	UTF-8
barcodeFormat (BarcodeFormat)	QR-Code

```
// QR-Code default
DataFormat code = new BarcodeDataFormat();
```

If you want to use zxing hints, you can use the 'addToHintMap' method of your BarcodeDataFormat instance:

```
code.addToHintMap(DecodeHintType.TRY_HARDER, Boolean.true);
```

For possible hints, please consult the xzing documentation.

39.3.1. Marshalling

```
from("direct://code")
  .marshal(code)
  .to("file://barcode_out");
```

You can call the route from a test class with:

```
template.sendBody("direct://code", "This is a testmessage!");
```

You should find inside the 'barcode_out' folder this image:

39.3.2. Unmarshalling

The unmarshaller is generic. For unmarshalling you can use any BarcodeDataFormat instance. If you've two instances, one for (generating) QR-Code and one for PDF417, it doesn't matter which one will be used.

```
from("file://barcode_in?noop=true")
  .unmarshal(code) // for unmarshalling, the instance doesn't matter
  .to("mock:out");
```

If you'll paste the QR-Code image above into the 'barcode_in' folder, you should find 'This is a testmessage!' inside the mock. You can find the barcode data format as header variable:

Name	Type	Description
BarcodeFormat	String	Value of com.google.zxing.BarcodeFormat.

CHAPTER 40. BASE64 DATAFORMAT

Available as of Camel version 2.11

The Base64 data format is used for base64 encoding and decoding.

40.1. OPTIONS

The Base64 dataformat supports 4 options which are listed below.

Name	Default	Java Type	Description
lineLength	**76**	**Integer**	To specific a maximum line length for the encoded data. By default 76 is used.
lineSeparator		**String**	The line separators to use. Uses new line characters (CRLF) by default.
urlSafe	**false**	**Boolean**	Instead of emitting '' and '/' we emit '-' and '_' respectively. urlSafe is only applied to encode operations. Decoding seamlessly handles both modes. Is by default false.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

In Spring DSL, you configure the data format using this tag:

```
<camelContext>
   <dataFormats>
      <!-- for a newline character (\n), use the HTML entity notation coupled with the ASCII code. -->
      <base64 lineSeparator="&#10;" id="base64withNewLine" />
      <base64 lineLength="64" id="base64withLineLength64" />
   </dataFormats>
   ...
</camelContext>
```

Then you can use it later by its reference:

```
<route>
    <from uri="direct:startEncode" />
    <marshal ref="base64withLineLength64" />
    <to uri="mock:result" />
</route>
```

Most of the time, you won't need to declare the data format if you use the default options. In that case, you can declare the data format inline as shown below.

40.2. MARSHAL

In this example we marshal the file content to base64 object.

```
from("file://data.bin")
    .marshal().base64()
    .to("jms://myqueue");
```

In Spring DSL:

```
<from uri="file://data.bin">
<marshal>
    <base64/>
</marshal>
<to uri="jms://myqueue"/>
```

40.3. UNMARSHAL

In this example we unmarshal the payload from the JMS queue to a byte[] object, before its processed by the newOrder processor.

```
from("jms://queue/order")
    .unmarshal().base64()
    .process("newOrder");
```

In Spring DSL:

```
<from uri="jms://queue/order">
<marshal>
    <base64/>
</marshal>
<to uri="bean:newOrder"/>
```

40.4. DEPENDENCIES

To use Base64 in your Camel routes you need to add a dependency on **camel-base64** which implements this data format.

If you use Maven you can just add the following to your pom.xml:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-base64</artifactId>
  <version>x.x.x</version>  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 41. BEAN COMPONENT

Available as of Camel version 1.0

The **bean:** component binds beans to Camel message exchanges.

41.1. URI FORMAT

```
bean:beanName[?options]
```

Where **beanID** can be any string which is used to look up the bean in the Registry

41.2. OPTIONS

The Bean component has no options.

The Bean endpoint is configured using URI syntax:

```
bean:beanName
```

with the following path and query parameters:

41.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
beanName	**Required** Sets the name of the bean to invoke		String

41.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
method (producer)	Sets the name of the method to invoke on the bean		String
cache (advanced)	If enabled, Camel will cache the result of the first Registry look-up. Cache can be enabled if the bean in the Registry is defined as a singleton scope.	false	boolean
multiParameterArray (advanced)	**Deprecated** How to treat the parameters which are passed from the message body; if it is true, the message body should be an array of parameters. Note: This option is used internally by Camel, and is not intended for end users to use. Deprecation note: This option is used internally by Camel, and is not intended for end users to use.	false	boolean

Name	Description	Default	Type
parameters (advanced)	Used for configuring additional properties on the bean		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

You can append query options to the URI in the following format, **?option=value&option=value&…**

41.3. USING

The object instance that is used to consume messages must be explicitly registered with the Registry. For example, if you are using Spring you must define the bean in the Spring configuration, **spring.xml**; or if you don't use Spring, by registering the bean in JNDI.

Error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20

Once an endpoint has been registered, you can build Camel routes that use it to process exchanges.

A **bean:** endpoint cannot be defined as the input to the route; i.e. you cannot consume from it, you can only route from some inbound message Endpoint to the bean endpoint as output. So consider using a **direct:** or **queue:** endpoint as the input.

You can use the **createProxy()** methods on ProxyHelper to create a proxy that will generate BeanExchanges and send them to any endpoint:

And the same route using Spring DSL:

```
<route>
  <from uri="direct:hello">
  <to uri="bean:bye"/>
</route>
```

41.4. BEAN AS ENDPOINT

Camel also supports invoking Bean as an Endpoint. In the route below:

What happens is that when the exchange is routed to the **myBean** Camel will use the Bean Binding to invoke the bean.
The source for the bean is just a plain POJO:

Camel will use Bean Binding to invoke the **sayHello** method, by converting the Exchange's In body to the **String** type and storing the output of the method on the Exchange Out body.

41.5. JAVA DSL BEAN SYNTAX

Java DSL comes with syntactic sugar for the Bean component. Instead of specifying the bean explicitly as the endpoint (i.e. **to("bean:beanName")**) you can use the following syntax:

```
// Send message to the bean endpoint
// and invoke method resolved using Bean Binding.
from("direct:start").beanRef("beanName");

// Send message to the bean endpoint
// and invoke given method.
from("direct:start").beanRef("beanName", "methodName");
```

Instead of passing name of the reference to the bean (so that Camel will lookup for it in the registry), you can specify the bean itself:

```
// Send message to the given bean instance.
from("direct:start").bean(new ExampleBean());

// Explicit selection of bean method to be invoked.
from("direct:start").bean(new ExampleBean(), "methodName");

// Camel will create the instance of bean and cache it for you.
from("direct:start").bean(ExampleBean.class);
```

41.6. BEAN BINDING

How bean methods to be invoked are chosen (if they are not specified explicitly through the **method** parameter) and how parameter values are constructed from the Message are all defined by the Bean Binding mechanism which is used throughout all of the various Bean Integration mechanisms in Camel.

41.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Class component
- Bean Binding
- Bean Integration

CHAPTER 42. BEANIO DATAFORMAT

Available as of Camel version 2.10

The BeanIO Data Format uses BeanIO to handle flat payloads (such as XML, CSV, delimited, or fixed length formats).

BeanIO is configured using a mappings XML file where you define the mapping from the flat format to Objects (POJOs). This mapping file is mandatory to use.

42.1. OPTIONS

The BeanIO dataformat supports 9 options which are listed below.

Name	Default	Java Type	Description
mapping		**String**	The BeanIO mapping file. Is by default loaded from the classpath. You can prefix with file:, http:, or classpath: to denote from where to load the mapping file.
streamName		**String**	The name of the stream to use.
ignoreUnidentifiedRecords	**false**	**Boolean**	Whether to ignore unidentified records.
ignoreUnexpectedRecords	**false**	**Boolean**	Whether to ignore unexpected records.
ignoreInvalidRecords	**false**	**Boolean**	Whether to ignore invalid records.
encoding		**String**	The charset to use. Is by default the JVM platform default charset.
beanReaderErrorHandlerType		**String**	To use a custom org.apache.camel.dataformat.beanio.BeanIOErrorHandler as error handler while parsing. Configure the fully qualified class name of the error handler. Notice the options ignoreUnidentifiedRecords, ignoreUnexpectedRecords, and ignoreInvalidRecords may not be in use when you use a custom error handler.
unmarshalSingleObject	**false**	**Boolean**	This options controls whether to unmarshal as a list of objects or as a single object only. The former is the default mode, and the latter is only intended in special use-cases where beanio maps the Camel message to a single POJO bean.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

42.2. USAGE

An example of a mapping file is here.

42.2.1. Using Java DSL

To use the **BeanIODataFormat** you need to configure the data format with the mapping file, as well the name of the stream.
In Java DSL this can be done as shown below. The streamName is "employeeFile".

Then we have two routes. The first route is for transforming CSV data into a List<Employee> Java objects. Which we then split, so the mock endpoint
receives a message for each row.

The 2nd route is for the reverse operation, to transform a List<Employee> into a stream of CSV data.

The CSV data could for example be as below:

42.2.2. Using XML DSL

To use the BeanIO data format in XML, you need to configure it using the <beanio> XML tag as shown below. The routes is similar to the example above.

42.3. DEPENDENCIES

To use BeanIO in your Camel routes you need to add a dependency on **camel-beanio** which implements this data format.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-beanio</artifactId>
  <version>2.10.0</version>
</dependency>
```

CHAPTER 43. BEANSTALK COMPONENT

Available as of Camel version 2.15

camel-beanstalk project provides a Camel component for job retrieval and post-processing of Beanstalk jobs.

You can find the detailed explanation of Beanstalk job lifecycle at Beanstalk protocol.

43.1. DEPENDENCIES

Maven users need to add the following dependency to their **pom.xml**

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-beanstalk</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.15.0 or higher).

43.2. URI FORMAT

```
beanstalk://[host[:port]][/tube][?options]
```

You may omit either **port** or both **host** and **port**: for the Beanstalk defaults to be used ("localhost" and 11300). If you omit **tube**, Beanstalk component will use the tube with name "default".

When listening, you may probably want to watch for jobs from several tubes. Just separate them with plus sign, e.g.

```
beanstalk://localhost:11300/tube1+tube2
```

Tube name will be URL decoded, so if your tube names include special characters like + or ?, you need to URL-encode them appropriately, or use the RAW syntax, see more details here.

By the way, you cannot specify several tubes when you are writing jobs into Beanstalk.

43.3. BEANSTALK OPTIONS

The Beanstalk component supports 2 options which are listed below.

Name	Description	Default	Type
connectionSettings Factory (common)	Custom ConnectionSettingsFactory. Specify which ConnectionSettingsFactory to use to make connections to Beanstalkd. Especially useful for unit testing without beanstalkd daemon (you can mock ConnectionSettings)		ConnectionSettings Factory

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Beanstalk endpoint is configured using URI syntax:

```
beanstalk:connectionSettings
```

with the following path and query parameters:

43.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
connectionSettings	Connection settings host:port/tube		String

43.3.2. Query Parameters (26 parameters):

Name	Description	Default	Type
command (common)	put means to put the job into Beanstalk. Job body is specified in the Camel message body. Job ID will be returned in beanstalk.jobId message header. delete, release, touch or bury expect Job ID in the message header beanstalk.jobId. Result of the operation is returned in beanstalk.result message header kick expects the number of jobs to kick in the message body and returns the number of jobs actually kicked out in the message header beanstalk.result.		BeanstalkCommand
jobDelay (common)	Job delay in seconds.	0	int
jobPriority (common)	Job priority. (0 is the highest, see Beanstalk protocol)	1000	long
jobTimeToRun (common)	Job time to run in seconds. (when 0, the beanstalkd daemon raises it to 1 automatically, see Beanstalk protocol)	60	int

Name	Description	Default	Type
awaitJob (consumer)	Whether to wait for job to complete before ack the job from beanstalk	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
onFailure (consumer)	Command to use when processing failed.		BeanstalkCommand
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
useBlockIO (consumer)	Whether to use blockIO.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit

Name	Description	Default	Type
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

Producer behavior is affected by the **command** parameter which tells what to do with the job, it can be

The consumer may delete the job immediately after reserving it or wait until Camel routes process it. While the first scenario is more like a "message queue", the second is similar to "job queue". This behavior is controlled by **consumer.awaitJob** parameter, which equals **true** by default (following Beanstalkd nature).

When synchronous, the consumer calls **delete** on successful job completion and calls **bury** on failure. You can choose which command to execute in the case of failure by specifying **consumer.onFailure** parameter in the URI. It can take values of **bury**, **delete** or **release**.

There is a boolean parameter **consumer.useBlockIO** which corresponds to the same parameter in JavaBeanstalkClient library. By default it is **true**.

Be careful when specifying **release**, as the failed job will immediately become available in the same tube and your consumer will try to acquire it again. You can **release** and specify *jobDelay* though.

The beanstalk consumer is a Scheduled Polling Consumer which means there is more options you can configure, such as how frequent the consumer should poll. For more details see Polling Consumer.

43.4. CONSUMER HEADERS

The consumer stores a number of job headers in the Exchange message:

Property	Type	Description
beanstalk.jobId	long	Job ID
beanstalk.tube	string	the name of the tube that contains this job
beanstalk.state	string	"ready" or "delayed" or "reserved" or "buried" (must be "reserved")
beanstalk.priority	long	the priority value set
beanstalk.age	int	the time in seconds since the put command that created this job

Property	Type	Description
beanstalk.time-left	int	the number of seconds left until the server puts this job into the ready queue
beanstalk.timeouts	int	the number of times this job has timed out during a reservation
beanstalk.releases	int	the number of times a client has released this job from a reservation
beanstalk.buries	int	the number of times this job has been buried
beanstalk.kicks	int	the number of times this job has been kicked

43.5. EXAMPLES

This Camel component lets you both request the jobs for processing and supply them to Beanstalkd daemon. Our simple demo routes may look like

```
from("beanstalk:testTube").
  log("Processing job #${property.beanstalk.jobId} with body ${in.body}").
  process(new Processor() {
    @Override
    public void process(Exchange exchange) {
      // try to make integer value out of body
      exchange.getIn().setBody( Integer.valueOf(exchange.getIn().getBody(classOf[String])) );
    }
  }).
  log("Parsed job #${property.beanstalk.jobId} to body ${in.body}");
```

```
from("timer:dig?period=30seconds").
  setBody(constant(10)).log("Kick ${in.body} buried/delayed tasks").
  to("beanstalk:testTube?command=kick");
```

In the first route we are listening for new jobs in tube “testTube”. When they are arriving, we are trying to parse integer value from the message body. If done successful, we log it and this successful exchange completion makes Camel component to *delete* this job from Beanstalk automatically. Contrary, when we cannot parse the job data, the exchange failed and the Camel component *buries* it by default, so that it can be processed later or probably we are going to inspect failed jobs manually.

So the second route periodically requests Beanstalk to *kick* 10 jobs out of buried and/or delayed state to the normal queue.

43.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 44. BEAN VALIDATOR COMPONENT

Available as of Camel version 2.3

The Validator component performs bean validation of the message body using the Java Bean Validation API (JSR 303). Camel uses the reference implementation, which is Hibernate Validator.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-bean-validator</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

44.1. URI FORMAT

```
bean-validator:label[?options]
```

or

```
bean-validator://label[?options]
```

Where **label** is an arbitrary text value describing the endpoint.
You can append query options to the URI in the following format, ?option=value&option=value&...

44.2. URI OPTIONS

The Bean Validator component has no options.

The Bean Validator endpoint is configured using URI syntax:

```
bean-validator:label
```

with the following path and query parameters:

44.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
label	**Required** Where label is an arbitrary text value describing the endpoint		String

44.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
constraintValidatorFactory (producer)	To use a custom ConstraintValidatorFactory		ConstraintValidator Factory
group (producer)	To use a custom validation group	javax.validation.groups.Default	String
messageInterpolator (producer)	To use a custom MessageInterpolator		MessageInterpolator
traversableResolver (producer)	To use a custom TraversableResolver		TraversableResolver
validationProviderResolver (producer)	To use a a custom ValidationProviderResolver		ValidationProviderResolver
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

44.3. OSGI DEPLOYMENT

To use Hibernate Validator in the OSGi environment use dedicated **ValidationProviderResolver** implementation, just as **org.apache.camel.component.bean.validator.HibernateValidationProviderResolver**. The snippet below demonstrates this approach. Keep in mind that you can use **HibernateValidationProviderResolver** starting from the Camel 2.13.0.

Using HibernateValidationProviderResolver

```
from("direct:test").
  to("bean-validator://ValidationProviderResolverTest?
validationProviderResolver=#myValidationProviderResolver");

...

<bean id="myValidationProviderResolver"
class="org.apache.camel.component.bean.validator.HibernateValidationProviderResolver"/>
```

If no custom **ValidationProviderResolver** is defined and the validator component has been deployed into the OSGi environment, the **HibernateValidationProviderResolver** will be automatically used.

44.4. EXAMPLE

Assumed we have a java bean with the following annotations

Car.java

```
public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 5, max = 14, groups = OptionalChecks.class)
    private String licensePlate;

    // getter and setter
}
```

and an interface definition for our custom validation group

OptionalChecks.java

```
public interface OptionalChecks {
}
```

with the following Camel route, only the @**NotNull** constraints on the attributes manufacturer and licensePlate will be validated (Camel uses the default group **javax.validation.groups.Default**).

```
from("direct:start")
.to("bean-validator://x")
.to("mock:end")
```

If you want to check the constraints from the group **OptionalChecks**, you have to define the route like this

```
from("direct:start")
.to("bean-validator://x?group=OptionalChecks")
.to("mock:end")
```

If you want to check the constraints from both groups, you have to define a new interface first

AllChecks.java

```
@GroupSequence({Default.class, OptionalChecks.class})
public interface AllChecks {
}
```

and then your route definition should looks like this

```
from("direct:start")
.to("bean-validator://x?group=AllChecks")
.to("mock:end")
```

And if you have to provide your own message interpolator, traversable resolver and constraint validator factory, you have to write a route like this

```
<bean id="myMessageInterpolator" class="my.ConstraintValidatorFactory" />
<bean id="myTraversableResolver" class="my.TraversableResolver" />
<bean id="myConstraintValidatorFactory" class="my.ConstraintValidatorFactory" />

from("direct:start")
.to("bean-validator://x?group=AllChecks&messageInterpolator=#myMessageInterpolator
&traversableResolver=#myTraversableResolver&constraintValidatorFactory=#myConstraintValidatorFac
tory")
.to("mock:end")
```

It's also possible to describe your constraints as XML and not as Java annotations. In this case, you have to provide the file **META-INF/validation.xml** which could looks like this

validation.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
    xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">
    <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
    <message-
interpolator>org.hibernate.validator.engine.ResourceBundleMessageInterpolator</message-
interpolator>
    <traversable-
resolver>org.hibernate.validator.engine.resolver.DefaultTraversableResolver</traversable-resolver>
    <constraint-validator-
factory>org.hibernate.validator.engine.ConstraintValidatorFactoryImpl</constraint-validator-factory>

    <constraint-mapping>/constraints-car.xml</constraint-mapping>
</validation-config>
```

and the **constraints-car.xml** file

constraints-car.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<constraint-mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping validation-mapping-1.0.xsd"
    xmlns="http://jboss.org/xml/ns/javax/validation/mapping">
    <default-package>org.apache.camel.component.bean.validator</default-package>

    <bean class="CarWithoutAnnotations" ignore-annotations="true">
        <field name="manufacturer">
            <constraint annotation="javax.validation.constraints.NotNull" />
        </field>

        <field name="licensePlate">
            <constraint annotation="javax.validation.constraints.NotNull" />

            <constraint annotation="javax.validation.constraints.Size">
                <groups>
                    <value>org.apache.camel.component.bean.validator.OptionalChecks</value>
                </groups>
                <element name="min">5</element>
```

```
            <element name="max">14</element>
        </constraint>
    </field>
  </bean>
</constraint-mappings>
```

Here is the XML syntax for the example route definition where **OrderedChecks** can be https://github.com/apache/camel/blob/master/components/camel-bean-validator/src/test/java/org/apache/camel/component/bean/validator/OrderedChecks.java

Note that the body should include an instance of a class to validate.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:start"/>
            <to uri="bean-validator://x?
group=org.apache.camel.component.bean.validator.OrderedChecks"/>
        </route>
    </camelContext>
</beans>
```

44.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 45. BINDING COMPONENT (DEPRECATED)

Available as of Camel version 2.11

In Camel terms a *binding* is a way of wrapping an Endpoint in a contract; such as a Data Format, a Content Enricher or validation step. Bindings are completely optional and you can choose to use them on any camel endpoint.

Bindings are inspired by the work of SwitchYard project adding service contracts to various technologies like Camel and many others. But rather than the SwitchYard approach of wrapping Camel in SCA, *Camel Bindings* provide a way of wrapping Camel endpoints with contracts inside the Camel framework itself; so you can use them easily inside any Camel route.

45.1. OPTIONS

The Binding component has no options.

The Binding endpoint is configured using URI syntax:

```
binding:bindingName:delegateUri
```

with the following path and query parameters:

45.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
bindingName	**Required** Name of the binding to lookup in the Camel registry.		String
delegateUri	**Required** Uri of the delegate endpoint.		String

45.1.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

45.2. USING BINDINGS

A Binding is currently a bean which defines the contract (though we'll hopefully add bindings to the Camel DSL).

There are a few approaches to defining a bound endpoint (i.e. an endpoint bound with a Binding).

45.3. USING THE BINDING URI

You can prefix any endpoint URI with **binding:nameOfBinding:** where *nameOfBinding* is the name of the Binding bean in your registry.

```
from("binding:jaxb:activemq:myQueue").to("binding:jaxb:activemq:anotherQueue")
```

Here we are using the "jaxb" binding which may, for example, use the JAXB Data Format to marshal and unmarshal messages.

45.4. USING A BINDINGCOMPONENT

There is a Component called BindingComponent which can be configured in your Registry by dependency injection which allows the creation of endpoints which are already bound to some binding.

For example if you registered a new component called "jsonmq" in your registry using code like this

```
JacksonDataFormat format = new JacksonDataFormat(MyBean.class);
context.bind("jsonmq", new BindingComponent(new DataFormatBinding(format), "activemq:foo."));
```

Then you could use the endpoint as if it were any other endpoint.

```
from("jsonmq:myQueue").to("jsonmq:anotherQueue")
```

which would be using the queueus "foo.myQueue" and "foo.anotherQueue" and would use the given Jackson Data Format to marshal on and off the queue.

45.5. WHEN TO USE BINDINGS

If you only use an endpoint once in a single route; a binding may actually be more complex and more work than just using the 'raw' endpoint directly and using explicit marshalling and validation in the camel route as normal.

However bindings can help when you are composing many routes together; or using a single route as a 'template' that is configured input and output endpoints; bindings then provide a nice way to wrap up a contract and endpoint together.

Another good use case for bindings is when you are using many endpoints which use the same binding; rather than always having to mention a specific data format or validation rule, you can just use the BindingComponent to wrap the endpoints in the binding of your choice.

So bindings are a composition tool really; only use them when they make sense - the extra complexity may not be worth it unless you have lots of routes or endpoints.

CHAPTER 46. BINDY DATAFORMAT

Available as of Camel version 2.0

The goal of this component is to allow the parsing/binding of non-structured data (or to be more precise non-XML data)
to/from Java Beans that have binding mappings defined with annotations. Using Bindy, you can bind data from sources such as :

- CSV records,
- Fixed-length records,
- FIX messages,
- or almost any other non-structured data

to one or many Plain Old Java Object (POJO). Bindy converts the data according to the type of the java property. POJOs can be linked together with one-to-many relationships available in some cases. Moreover, for data type like Date, Double, Float, Integer, Short, Long and BigDecimal, you can provide the pattern to apply during the formatting of the property.

For the BigDecimal numbers, you can also define the precision and the decimal or grouping separators.

Type	Format Type	Pattern example	Link
Date	DateFormat	**dd-MM-yyyy**	http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html
Decimal*	DecimalFormat	**..##**	http://java.sun.com/j2se/1.5.0/docs/api/java/text/DecimalFormat.html

Decimal* = Double, Integer, Float, Short, Long

> *Format supported*

This first release only support comma separated values fields and key value pair fields (e.g. : FIX messages).

To work with camel-bindy, you must first define your model in a package (e.g. com.acme.model) and for each model class (e.g. Order, Client, Instrument, ...) add the required annotations (described hereafter) to the Class or field.

> *Multiple models*

If you use multiple models, each model has to be placed in it's own package to prevent unpredictable results.

From **Camel 2.16** onwards this is no longer the case, as you can safely have multiple models in the same package, as you configure bindy using class names instead of package names now.

46.1. OPTIONS

The Bindy dataformat supports 5 options which are listed below.

Name	Default	Java Type	Description
type		**Bindy Type**	Whether to use csv, fixed or key value pairs mode. The default value is either Csv or KeyValue depending on chosen dataformat.
classType		**String**	Name of model class to use.
locale		**String**	To configure a default locale to use, such as us for united states. To use the JVM platform default locale then use the name default
unwrapSingleInstance	**true**	**Boolean**	When unmarshalling should a single instance be unwrapped and returned instead of wrapped in a java.util.List.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

46.2. ANNOTATIONS

The annotations created allow to map different concept of your model to the POJO like :

- Type of record (csv, key value pair (e.g. FIX message), fixed length ...),
- Link (to link object in another object),
- DataField and their properties (int, type, ...),
- KeyValuePairField (for key = value format like we have in FIX financial messages),
- Section (to identify header, body and footer section),
- OneToMany,
- BindyConverter (since 2.18.0),
- FormatFactories (since 2.18.0)

This section will describe them :

46.3. 1. CSVRECORD

The CsvRecord annotation is used to identified the root class of the model. It represents a record = a line of a CSV file and can be linked to several children model classes.

Annotation name	Record type	Level
CsvRecord	csv	Class

Parameter name	type	Info
separator	string	mandatory - can be ',' or ';' or 'anything'. This value is interpreted as a regular expression. If you want to use a sign which has a special meaning in regular expressions, e.g. the '|' sign, than you have to mask it, like '|'
skipFirstLine	boolean	optional - default value = false - allow to skip the first line of the CSV file
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value. WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
generateHeaderColumns	boolean	optional - default value = false - uses to generate the header columns of the CSV generates
autospanLine	boolean	**Camel 2.13/2.12.2:** optional - default value = false - if enabled then the last column is auto spanned to end of line, for example if its a comment, etc this allows the line to contain all characters, also the delimiter char.
isOrdered	boolean	optional - default value = false - allow to change the order of the fields when CSV is generated
quote	String	**Camel 2.8.3/2.9:** option - allow to specify a quote character of the fields when CSV is generated. This annotation is associated to the root class of the model and must be declared one time.
quoting	boolean	*Camel 2.11:*optional - default value = false - Indicate if the values (and headers) must be quoted when marshaling when CSV is generated.
endWithLineBreak	boolean	**Camel 2.21:** optional - default value = true - Indicate if the CSV generated file should end with a line break.

case 1 : separator = ','

The separator used to segregate the fields in the CSV record is ',' :

```
10, J, Pauline, M, XD12345678, Fortis Dynamic 15/15, 2500,
USD,08-01-2009
```

```
@CsvRecord( separator = "," )
public Class Order {

}
```

case 2 : separator = ';'

Compare to the previous case, the separator here is ';' instead of ',' :

10; J; Pauline; M; XD12345678; Fortis Dynamic 15/15; 2500; USD; 08-01-2009

```
@CsvRecord( separator = ";" )
public Class Order {

}
```

case 3 : separator = '|'

Compare to the previous case, the separator here is '|' instead of ';' :

```
10| J| Pauline| M| XD12345678| Fortis Dynamic 15/15| 2500| USD|
08-01-2009
```

```
@CsvRecord( separator = "\\|" )
public Class Order {

}
```

case 4 : separator = '\",\"'

Applies for Camel 2.8.2 or older

When the field to be parsed of the CSV record contains ',' or ';' which is also used as separator, we whould find another strategy
to tell camel bindy how to handle this case. To define the field containing the data with a comma, you will use simple or double quotes
as delimiter (e.g : '10', 'Street 10, NY', 'USA' or "10", "Street 10, NY", "USA").
Remark : In this case, the first and last character of the line which are a simple or double quotes will removed by bindy

```
"10","J","Pauline"," M","XD12345678","Fortis Dynamic 15,15"
2500","USD","08-01-2009"
```

```
@CsvRecord( separator = "\",\"" )
public Class Order {
```

```
}
```

From **Camel 2.8.3/2.9 or never** bindy will automatic detect if the record is enclosed with either single or double quotes and automatic remove those quotes when unmarshalling from CSV to Object. Therefore do **not** include the quotes in the separator, but simple do as below:

```
"10","J","Pauline"," M","XD12345678","Fortis Dynamic 15,15"
2500","USD","08-01-2009"
```

```
@CsvRecord( separator = "," )
public Class Order {

}
```

Notice that if you want to marshal from Object to CSV and use quotes, then you need to specify which quote character to use, using the **quote** attribute on the @CsvRecord as shown below:

```
@CsvRecord( separator = ",", quote = "\"" )
public Class Order {

}
```

case 5 : separator & skipfirstline

The feature is interesting when the client wants to have in the first line of the file, the name of the data fields :

order id, client id, first name, last name, isin code, instrument name, quantity, currency, date

To inform bindy that this first line must be skipped during the parsing process, then we use the attribute :

```
@CsvRecord(separator = ",", skipFirstLine = true)
public Class Order {

}
```

case 6 : generateHeaderColumns

To add at the first line of the CSV generated, the attribute generateHeaderColumns must be set to true in the annotation like this :

```
@CsvRecord( generateHeaderColumns = true )
public Class Order {

}
```

As a result, Bindy during the unmarshaling process will generate CSV like this :

order id, client id, first name, last name, isin code, instrument name, quantity, currency, date

```
10, J, Pauline, M, XD12345678, Fortis Dynamic 15/15, 2500, USD,08-01-2009
```

case 7 : carriage return

If the platform where camel-bindy will run is not Windows but Macintosh or Unix, than you can change the crlf property like this. Three values are available : WINDOWS, UNIX or MAC

```
@CsvRecord(separator = ",", crlf="MAC")
public Class Order {

}
```

Additionally, if for some reason you need to add a different line ending character, you can opt to specify it using the crlf parameter. In the following example, we can end the line with a comma followed by the newline character:

```
@CsvRecord(separator = ",", crlf=",\n")
public Class Order {

}
```

case 8 : isOrdered

Sometimes, the order to follow during the creation of the CSV record from the model is different from the order used during the parsing. Then, in this case, we can use the attribute isOrdered = true to indicate this in combination with attribute 'position' of the DataField annotation.

```
@CsvRecord(isOrdered = true)
public Class Order {

  @DataField(pos = 1, position = 11)
  private int orderNr;

  @DataField(pos = 2, position = 10)
  private String clientNr;

}
```

Remark : pos is used to parse the file, stream while positions is used to generate the CSV

46.4. 2. LINK

The link annotation will allow to link objects together.

Annotation name	Record type	Level
Link	all	Class & Property

Parameter name	type	Info
linkType	LinkType	optional - by default the value is LinkType.oneToOne - so you are not obliged to mention it

Only one-to-one relation is allowed.

e.g : If the model Class Client is linked to the Order class, then use annotation Link in the Order class like this :

Property Link

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @Link
    private Client client;
}
```

AND for the class Client :

Class Link

```
@Link
public class Client {

}
```

46.5. 3. DATAFIELD

The DataField annotation defines the property of the field. Each datafield is identified by its position in the record, a type (string, int, date, ...) and optionally of a pattern

Annotation name	Record type	Level
DataField	all	Property

Parameter name	type	Info
pos	int	mandatory - The **input** position of the field. digit number starting from 1 to ... - See the position parameter.
pattern	string	optional - default value = "" - will be used to format Decimal, Date,
length	int	optional - represents the length of the field for fixed length format
precision	int	optional - represents the precision to be used when the Decimal number will be formatted/parsed
pattern	string	optional - default value = "" - is used by the Java formatter (SimpleDateFormat by example) to format/validate data. If using pattern, then setting locale on bindy data format is recommended. Either set to a known locale such as "us" or use "default" to use platform default locale. Notice that "default" requires Camel 2.14/2.13.3/2.12.5.
position	int	optional - must be used when the position of the field in the CSV generated (output message) must be different compare to input position (pos). See the pos parameter.
required	boolean	optional - default value = "false"
trim	boolean	optional - default value = "false"
defaultValue	string	**Camel 2.10:** optional - default value = "" - defines the field's default value when the respective CSV field is empty/not available
impliedDecimalSeparator	boolean	**Camel 2.11:** optional - default value = "false" - Indicates if there is a decimal point implied at a specified location
lengthPos	int	**Camel 2.11**: optional - can be used to identify a data field in a fixed-length record that defines the fixed length for this field
align	string	optional - default value = "R" - Align the text to the right or left within a fixed-length field. Use values 'R' or 'L'
delimiter	string	**Camel 2.11:** optional - can be used to demarcate the end of a variable-length field within a fixed-length record

case 1 : pos

This parameter/attribute represents the position of the field in the csv record

Position

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 5)
    private String isinCode;

}
```

As you can see in this example the position starts at '1' but continues at '5' in the class Order. The numbers from '2' to '4' are defined in the class Client (see here after).

Position continues in another model class

```
public class Client {

    @DataField(pos = 2)
    private String clientNr;

    @DataField(pos = 3)
    private String firstName;

    @DataField(pos = 4)
    private String lastName;
}
```

case 2 : pattern

The pattern allows to enrich or validates the format of your data

Pattern

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 5)
    private String isinCode;

    @DataField(name = "Name", pos = 6)
    private String instrumentName;

    @DataField(pos = 7, precision = 2)
    private BigDecimal amount;

    @DataField(pos = 8)
    private String currency;

    // pattern used during parsing or when the date is created
```

```
    @DataField(pos = 9, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

case 3 : precision

The precision is helpful when you want to define the decimal part of your number

Precision

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @Link
    private Client client;

    @DataField(pos = 5)
    private String isinCode;

    @DataField(name = "Name", pos = 6)
    private String instrumentName;

    @DataField(pos = 7, precision = 2)
    private BigDecimal amount;

    @DataField(pos = 8)
    private String currency;

    @DataField(pos = 9, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

case 4 : Position is different in output

The position attribute will inform bindy how to place the field in the CSV record generated. By default, the position used corresponds to the position defined with the attribute 'pos'. If the position is different (that means that we have an asymetric processus comparing marshaling from unmarshaling) than we can use 'position' to indicate this.

Here is an example

Position is different in output

```
@CsvRecord(separator = ",", isOrdered = true)
public class Order {

    // Positions of the fields start from 1 and not from 0

    @DataField(pos = 1, position = 11)
    private int orderNr;

    @DataField(pos = 2, position = 10)
    private String clientNr;
```

```
    @DataField(pos = 3, position = 9)
    private String firstName;

    @DataField(pos = 4, position = 8)
    private String lastName;

    @DataField(pos = 5, position = 7)
    private String instrumentCode;

    @DataField(pos = 6, position = 6)
    private String instrumentNumber;
}
```

This attribute of the annotation @DataField must be used in combination with attribute isOrdered = true of the annotation @CsvRecord

case 5 : required

If a field is mandatory, simply use the attribute 'required' setted to true

Required

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 2, required = true)
    private String clientNr;

    @DataField(pos = 3, required = true)
    private String firstName;

    @DataField(pos = 4, required = true)
    private String lastName;
}
```

If this field is not present in the record, than an error will be raised by the parser with the following information :

Some fields are missing (optional or mandatory), line :

case 6 : trim

If a field has leading and/or trailing spaces which should be removed before they are processed, simply use the attribute 'trim' setted to true

Trim

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1, trim = true)
    private int orderNr;
```

```
    @DataField(pos = 2, trim = true)
    private Integer clientNr;

    @DataField(pos = 3, required = true)
    private String firstName;

    @DataField(pos = 4)
    private String lastName;
}
```

case 7 : defaultValue

If a field is not defined then uses the value indicated by the defaultValue attribute

Default value

```
@CsvRecord(separator = ",")
public class Order {

    @DataField(pos = 1)
    private int orderNr;

    @DataField(pos = 2)
    private Integer clientNr;

    @DataField(pos = 3, required = true)
    private String firstName;

    @DataField(pos = 4, defaultValue = "Barin")
    private String lastName;
}
```

This attribute is only applicable to optional fields.

46.6. 4. FIXEDLENGTHRECORD

The FixedLengthRecord annotation is used to identified the root class of the model. It represents a record = a line of a file/message containing data fixed length formatted and can be linked to several children model classes. This format is a bit particular beause data of a field can be aligned to the right or to the left.
When the size of the data does not fill completely the length of the field, we can then add 'padd' characters.

Annotation name	Record type	Level
FixedLengthRecord	fixed	Class

Parameter name	type	Info
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value. WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s). This option is used only during marshalling, whereas unmarshalling uses system default JDK provided line delimiter unless eol is customized
eol	string	optional - default="" which is empty string. Character to be used to process considering end of line after each record while unmarshalling (optional - default = "" which help default JDK provided line delimiter to be used unless any other line delimiter provided). This option is used only during unmarshalling, where marshalling uses system default provided line delimiter as "WINDOWS" unless any other value is provided
paddingChar	char	mandatory - default value = ' '
length	int	mandatory = size of the fixed length record
hasHeader	boolean	**Camel 2.11** - optional - Indicates that the record(s) of this type may be preceded by a single header record at the beginning of the file / stream
hasFooter	boolean	**Camel 2.11** - optional - Indicates that the record(s) of this type may be followed by a single footer record at the end of the file / stream
skipHeader	boolean	**Camel 2.11** - optional - Configures the data format to skip marshalling / unmarshalling of the header record. Configure this parameter on the primary record (e.g., not the header or footer).
skipFooter	boolean	**Camel 2.11** - optional - Configures the data format to skip marshalling / unmarshalling of the footer record Configure this parameter on the primary record (e.g., not the header or footer)..
isHeader	boolean	**Camel 2.11** - optional - Identifies this FixedLengthRecord as a header record
isFooter	boolean	**Camel 2.11** - optional - Identifies this FixedLengthRecords as a footer record
ignoreTrailingChars	boolean	**Camel 2.11.1** - optional - Indicates that characters beyond the last mapped filed can be ignored when unmarshalling / parsing. This annotation is associated to the root class of the model and must be declared one time.

The hasHeader/hasFooter parameters are mutually exclusive with isHeader/isFooter. A record may not be both a header/footer and a primary fixed-length record.

case 1 : Simple fixed length record

This simple example shows how to design the model to parse/format a fixed message

```
10A9PaulineMISINXD12345678BUYShare2500.45USD01-08-2009
```

Fixed-simple

```
@FixedLengthRecord(length=54, paddingChar=' ')
public static class Order {

    @DataField(pos = 1, length=2)
    private int orderNr;

    @DataField(pos = 3, length=2)
    private String clientNr;

    @DataField(pos = 5, length=7)
    private String firstName;

    @DataField(pos = 12, length=1, align="L")
    private String lastName;

    @DataField(pos = 13, length=4)
    private String instrumentCode;

    @DataField(pos = 17, length=10)
    private String instrumentNumber;

    @DataField(pos = 27, length=3)
    private String orderType;

    @DataField(pos = 30, length=5)
    private String instrumentType;

    @DataField(pos = 35, precision = 2, length=7)
    private BigDecimal amount;

    @DataField(pos = 42, length=3)
    private String currency;

    @DataField(pos = 45, length=10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

case 2 : Fixed length record with alignment and padding

This more elaborated example show how to define the alignment for a field and how to assign a padding character which is ' ' here''

```
10A9 PaulineM ISINXD12345678BUYShare2500.45USD01-08-2009
```

Fixed-padding-align

```
@FixedLengthRecord(length=60, paddingChar=' ')
public static class Order {

    @DataField(pos = 1, length=2)
    private int orderNr;
```

```
    @DataField(pos = 3, length=2)
    private String clientNr;

    @DataField(pos = 5, length=9)
    private String firstName;

    @DataField(pos = 14, length=5, align="L")   // align text to the LEFT zone of the block
    private String lastName;

    @DataField(pos = 19, length=4)
    private String instrumentCode;

    @DataField(pos = 23, length=10)
    private String instrumentNumber;

    @DataField(pos = 33, length=3)
    private String orderType;

    @DataField(pos = 36, length=5)
    private String instrumentType;

    @DataField(pos = 41, precision = 2, length=7)
    private BigDecimal amount;

    @DataField(pos = 48, length=3)
    private String currency;

    @DataField(pos = 51, length=10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

case 3 : Field padding

Sometimes, the default padding defined for record cannnot be applied to the field as we have a number format where we would like to padd with '0' instead of ' '. In this case, you can use in the model the attribute paddingField to set this value.

```
10A9 PaulineM ISINXD12345678BUYShare000002500.45USD01-08-2009
```

Fixed-padding-field

```
@FixedLengthRecord(length = 65, paddingChar = ' ')
public static class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 3, length = 2)
    private String clientNr;

    @DataField(pos = 5, length = 9)
    private String firstName;

    @DataField(pos = 14, length = 5, align = "L")
```

```
    private String lastName;

    @DataField(pos = 19, length = 4)
    private String instrumentCode;

    @DataField(pos = 23, length = 10)
    private String instrumentNumber;

    @DataField(pos = 33, length = 3)
    private String orderType;

    @DataField(pos = 36, length = 5)
    private String instrumentType;

    @DataField(pos = 41, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 53, length = 3)
    private String currency;

    @DataField(pos = 56, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

case 4: Fixed length record with delimiter

Fixed-length records sometimes have delimited content within the record. The firstName and lastName fields are delimited with the '^' character in the following example:

```
10A9Pauline^M^ISINXD12345678BUYShare000002500.45USD01-08-2009
```

Fixed-delimited

```
@FixedLengthRecord()
public static class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, delimiter = "^")
    private String firstName;

    @DataField(pos = 4, delimiter = "^")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 10)
    private String instrumentNumber;

    @DataField(pos = 7, length = 3)
```

```
    private String orderType;

    @DataField(pos = 8, length = 5)
    private String instrumentType;

    @DataField(pos = 9, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 10, length = 3)
    private String currency;

    @DataField(pos = 11, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

As of **Camel 2.11** the 'pos' value(s) in a fixed-length record may optionally be defined using ordinal, sequential values instead of precise column numbers.

case 5 : Fixed length record with record-defined field length

Occasionally a fixed-length record may contain a field that define the expected length of another field within the same record. In the following example the length of the instrumentNumber field value is defined by the value of instrumentNumberLen field in the record.

```
10A9Pauline^M^ISIN10XD12345678BUYShare000002500.45USD01-08-2009
```

Fixed-delimited

```
@FixedLengthRecord()
public static class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, delimiter = "^")
    private String firstName;

    @DataField(pos = 4, delimiter = "^")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 2, align = "R", paddingChar = '0')
    private int instrumentNumberLen;

    @DataField(pos = 7, lengthPos=6)
    private String instrumentNumber;

    @DataField(pos = 8, length = 3)
    private String orderType;
```

```
    @DataField(pos = 9, length = 5)
    private String instrumentType;

    @DataField(pos = 10, precision = 2, length = 12, paddingChar = '0')
    private BigDecimal amount;

    @DataField(pos = 11, length = 3)
    private String currency;

    @DataField(pos = 12, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}
```

case 6 : Fixed length record with header and footer

Bindy will discover fixed-length header and footer records that are configured as part of the model – provided that the annotated classes exist either in the same package as the primary @FixedLengthRecord class, or within one of the configured scan packages. The following text illustrates two fixed-length records that are bracketed by a header record and footer record.

```
101-08-2009
10A9 PaulineM ISINXD12345678BUYShare000002500.45USD01-08-2009
10A9 RichN ISINXD12345678BUYShare000002700.45USD01-08-2009
9000000002
```

Fixed-header-and-footer-main-class

```
@FixedLengthRecord(hasHeader = true, hasFooter = true)
public class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 2, length = 2)
    private String clientNr;

    @DataField(pos = 3, length = 9)
    private String firstName;

    @DataField(pos = 4, length = 5, align = "L")
    private String lastName;

    @DataField(pos = 5, length = 4)
    private String instrumentCode;

    @DataField(pos = 6, length = 10)
    private String instrumentNumber;

    @DataField(pos = 7, length = 3)
    private String orderType;

    @DataField(pos = 8, length = 5)
    private String instrumentType;

    @DataField(pos = 9, precision = 2, length = 12, paddingChar = '0')
```

```
    private BigDecimal amount;

    @DataField(pos = 10, length = 3)
    private String currency;

    @DataField(pos = 11, length = 10, pattern = "dd-MM-yyyy")
    private Date orderDate;
}

@FixedLengthRecord(isHeader = true)
public  class OrderHeader {
    @DataField(pos = 1, length = 1)
    private int recordType = 1;

    @DataField(pos = 2, length = 10, pattern = "dd-MM-yyyy")
    private Date recordDate;
}

@FixedLengthRecord(isFooter = true)
public class OrderFooter {

    @DataField(pos = 1, length = 1)
    private int recordType = 9;

    @DataField(pos = 2, length = 9, align = "R", paddingChar = '0')
    private int numberOfRecordsInTheFile;
}
```

case 7 : Skipping content when parsing a fixed length record. (Camel 2.11.1)

It is common to integrate with systems that provide fixed-length records containing more information than needed for the target use case. It is useful in this situation to skip the declaration and parsing of those fields that we do not need. To accomodate this, Bindy will skip forward to the next mapped field within a record if the 'pos' value of the next declared field is beyond the cursor position of the last parsed field. Using absolute 'pos' locations for the fields of interest (instead of ordinal values) causes Bindy to skip content between two fields.

Similarly, it is possible that none of the content beyond some field is of interest. In this case, you can tell Bindy to skip parsing of everything beyond the last mapped field by setting the **ignoreTrailingChars** property on the @FixedLengthRecord declaration.

```
@FixedLengthRecord(ignoreTrailingChars = true)
public static class Order {

    @DataField(pos = 1, length = 2)
    private int orderNr;

    @DataField(pos = 3, length = 2)
    private String clientNr;

    // any characters that appear beyond the last mapped field will be ignored

}
```

46.7. 5. MESSAGE

The Message annotation is used to identified the class of your model who will contain key value pairs fields. This kind of format is used mainly in Financial Exchange Protocol Messages (FIX). Nevertheless, this annotation can be used for any other format where data are identified by keys. The key pair values are separated each other by a separator which can be a special character like a tab delimitor (unicode representation : \u0009) or a start of heading (unicode representation : \u0001)

> *"FIX information"*

More information about FIX can be found on this web site : http://www.fixprotocol.org/. To work with FIX messages, the model must contain a Header and Trailer classes linked to the root message class which could be a Order class. This is not mandatory but will be very helpful when you will use camel-bindy in combination with camel-fix which is a Fix gateway based on quickFix project http://www.quickfixj.org/.

Annotation name	Record type	Level
Message	key value pair	Class

Parameter name	type	Info
pairSeparator	string	mandatory - can be '=' or ';' or 'anything'
keyValuePairSeparair	string	mandatory - can be '\u0001', '\u0009', '#' or 'anything'
crlf	string	optional - possible values = WINDOWS,UNIX,MAC, or custom; default value = WINDOWS - allow to define the carriage return character to use. If you specify a value other than the three listed before, the value you enter (custom) will be used as the CRLF character(s)
type	string	optional - define the type of message (e.g. FIX, EMX, ...)
version	string	optional - version of the message (e.g. 4.1)
isOrdered	boolean	optional - default value = false - allow to change the order of the fields when FIX message is generated. This annotation is associated to the message class of the model and must be declared one time.

case 1 : separator = 'u0001'

The separator used to segregate the key value pair fields in a FIX message is the ASCII '01' character or in unicode format '\u0001'. This character must be escaped a second time to avoid a java runtime error. Here is an example :

```
8=FIX.4.1 9=20 34=1 35=0 49=INVMGR 56=BRKR 1=BE.CHM.001 11=CHM0001-01
22=4 ...
```

and how to use the annotation

FIX - message

```
@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type="FIX", version="4.1")
public class Order {

}
```

Look at test cases

The ASCII character like tab, ... cannot be displayed in WIKI page. So, have a look to the test case of camel-bindy to see exactly how the FIX message looks like (src\test\data\fix\fix.txt) and the Order, Trailer, Header classes (src\test\java\org\apache\camel\dataformat\bindy\model\fix\simple\Order.java)

46.8. 6. KEYVALUEPAIRFIELD

The KeyValuePairField annotation defines the property of a key value pair field. Each KeyValuePairField is identified by a tag (= key) and its value associated, a type (string, int, date, ...), optionaly a pattern and if the field is required

Annotation name	Record type	Level
KeyValuePairField	Key Value Pair - FIX	Property

Parameter name	type	Info
tag	int	mandatory - digit number identifying the field in the message - must be unique
pattern	string	optional - default value = "" - will be used to format Decimal, Date, ...
precision	int	optional - digit number - represents the precision to be used when the Decimal number will be formatted/parsed

Parameter name	type	Info
position	int	optional - must be used when the position of the key/tag in the FIX message must be different
required	boolean	optional - default value = "false"
impliedDecimalSeparator	boolean	**Camel 2.11:** optional - default value = "false" - Indicates if there is a decimal point implied at a specified location

case 1 : tag

This parameter represents the key of the field in the message

FIX message - Tag

```
@Message(keyValuePairSeparator = "=", pairSeparator = "\u0001", type="FIX", version="4.1")
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1) // Client reference
    private String Account;

    @KeyValuePairField(tag = 11) // Order reference
    private String ClOrdId;

    @KeyValuePairField(tag = 22) // Fund ID type (Sedol, ISIN, ...)
    private String IDSource;

    @KeyValuePairField(tag = 48) // Fund code
    private String SecurityId;

    @KeyValuePairField(tag = 54) // Movement type ( 1 = Buy, 2 = sell)
    private String Side;

    @KeyValuePairField(tag = 58) // Free text
    private String Text;
}
```

case 2 : Different position in output

If the tags/keys that we will put in the FIX message must be sorted according to a predefine order, then use the attribute 'position' of the annotation @KeyValuePairField

FIX message - Tag - sort

```
@Message(keyValuePairSeparator = "=", pairSeparator = "\\u0001", type = "FIX", version = "4.1",
isOrdered = true)
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1, position = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11, position = 3) // Order reference
    private String clOrdId;
}
```

46.9. 7. SECTION

In FIX message of fixed length records, it is common to have different sections in the representation of the information : header, body and section. The purpose of the annotation @Section is to inform bindy about which class of the model represents the header (= section 1), body (= section 2) and footer (= section 3)

Only one attribute/parameter exists for this annotation.

Annotation name	Record type	Level
Section	FIX	Class

Parameter name	type	Info
number	int	digit number identifying the section position

case 1 : Section

Definition of the header section

FIX message - Section - Header

```
@Section(number = 1)
public class Header {

    @KeyValuePairField(tag = 8, position = 1) // Message Header
    private String beginString;
```

```
    @KeyValuePairField(tag = 9, position = 2) // Checksum
    private int bodyLength;
}
```

Definition of the body section

FIX message - Section - Body

```
@Section(number = 2)
@Message(keyValuePairSeparator = "=", pairSeparator = "\\u0001", type = "FIX", version = "4.1",
isOrdered = true)
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1, position = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11, position = 3) // Order reference
    private String clOrdId;
```

Definition of the footer section

FIX message - Section - Footer

```
@Section(number = 3)
public class Trailer {

    @KeyValuePairField(tag = 10, position = 1)
    // CheckSum
    private int checkSum;

    public int getCheckSum() {
        return checkSum;
    }
```

46.10. 8. ONETOMANY

The purpose of the annotation @OneToMany is to allow to work with a **List<?>** field defined a POJO class or from a record containing repetitive groups.

```
*Restrictions OneToMany*
```

Be careful, the one to many of bindy does not allow to handle repetitions defined on several levels of the hierarchy

The relation OneToMany ONLY WORKS in the following cases :

- Reading a FIX message containing repetitive groups (= group of tags/keys)
- Generating a CSV with repetitive data

Annotation name	Record type	Level
OneToMany	all	property

Parameter name	type	Info
mappedTo	string	optional - string - class name associated to the type of the List<Type of the Class>

case 1 : Generating CSV with repetitive data

Here is the CSV output that we want :

```
Claus,Ibsen,Camel in Action 1,2010,35
Claus,Ibsen,Camel in Action 2,2012,35
Claus,Ibsen,Camel in Action 3,2013,35
Claus,Ibsen,Camel in Action 4,2014,35
```

Remark : the repetitive data concern the title of the book and its publication date while first, last name and age are common

and the classes used to modeling this. The Author class contains a List of Book.

Generate CSV with repetitive data

```
@CsvRecord(separator=",")
public class Author {

    @DataField(pos = 1)
    private String firstName;

    @DataField(pos = 2)
    private String lastName;

    @OneToMany
    private List<Book> books;

    @DataField(pos = 5)
    private String Age;
}

public class Book {

    @DataField(pos = 3)
    private String title;
```

```
    @DataField(pos = 4)
    private String year;
}
```

Very simple isn't it !!!

case 2 : Reading FIX message containing group of tags/keys

Here is the message that we would like to process in our model :

```
8=FIX 4.19=2034=135=049=INVMGR56=BRKR
1=BE.CHM.00111=CHM0001-0158=this is a camel - bindy test
22=448=BE000124567854=1
22=548=BE000987654354=2
22=648=BE000999999954=3
10=220
```

tags 22, 48 and 54 are repeated

and the code

Reading FIX message containing group of tags/keys

```
public class Order {

    @Link Header header;

    @Link Trailer trailer;

    @KeyValuePairField(tag = 1) // Client reference
    private String account;

    @KeyValuePairField(tag = 11) // Order reference
    private String clOrdId;

    @KeyValuePairField(tag = 58) // Free text
    private String text;

    @OneToMany(mappedTo =
"org.apache.camel.dataformat.bindy.model.fix.complex.onetomany.Security")
    List<Security> securities;
}

public class Security {

    @KeyValuePairField(tag = 22) // Fund ID type (Sedol, ISIN, ...)
    private String idSource;

    @KeyValuePairField(tag = 48) // Fund code
    private String securityCode;

    @KeyValuePairField(tag = 54) // Movement type ( 1 = Buy, 2 = sell)
    private String side;
}
```

46.11. 9. BINDYCONVERTER

The purpose of the annotation @BindyConverter is define a converter to be used on field level. The provided class must implement the Format interface.

```
@FixedLengthRecord(length = 10, paddingChar = ' ')
public static class DataModel {
    @DataField(pos =  1, length = 10, trim = true)
    @BindyConverter(CustomConverter.class)
    public String field1;
}

public static class CustomConverter implements Format<String> {
    @Override
    public String format(String object) throws Exception {
        return (new StringBuilder(object)).reverse().toString();
    }

    @Override
    public String parse(String string) throws Exception {
        return (new StringBuilder(string)).reverse().toString();
    }
}
```

46.12. 10. FORMATFACTORIES

The purpose of the annotation @FormatFactories is to define a set of converters at record-level. The provided classes must implement the FormatFactoryInterface interface.

```
@CsvRecord(separator = ",")
@FormatFactories({OrderNumberFormatFactory.class})
public static class Order {

    @DataField(pos = 1)
    private OrderNumber orderNr;

    @DataField(pos = 2)
    private String firstName;
}

public static class OrderNumber {
    private int orderNr;

    public static OrderNumber ofString(String orderNumber) {
        OrderNumber result = new OrderNumber();
        result.orderNr = Integer.valueOf(orderNumber);
        return result;
    }
}

public static class OrderNumberFormatFactory extends AbstractFormatFactory {

    {
        supportedClasses.add(OrderNumber.class);
```

```
    }

    @Override
    public Format<?> build(FormattingOptions formattingOptions) {
        return new Format<OrderNumber>() {
            @Override
            public String format(OrderNumber object) throws Exception {
                return String.valueOf(object.orderNr);
            }

            @Override
            public OrderNumber parse(String string) throws Exception {
                return OrderNumber.ofString(string);
            }
        };
    }
}
```

46.13. SUPPORTED DATATYPES

The DefaultFormatFactory makes formatting of the following datatype available by returning an instance of the interface FormatFactoryInterface based on the provided FormattingOptions:

- BigDecimal
- BigInteger
- Boolean
- Byte
- Character
- Date
- Double
- Enums
- Float
- Integer
- LocalDate (java 8, since 2.18.0)
- LocalDateTime (java 8, since 2.18.0)
- LocalTime (java 8, since 2.18.0)
- Long
- Short
- String

The DefaultFormatFactory can be overridden by providing an instance of FactoryRegistry in the registry in use (e.g. spring or JNDI).

46.14. USING THE JAVA DSL

The next step consists in instantiating the DataFormat *bindy* class associated with this record type and providing Java package name(s) as parameter.

For example the following uses the class **BindyCsvDataFormat** (who correspond to the class associated with the CSV record type) which is configured with **com.acme.model** package name to initialize the model objects configured in this package.

```
// Camel 2.15 or older (configure by package name)
DataFormat bindy = new BindyCsvDataFormat("com.acme.model");


// Camel 2.16 onwards (configure by class name)
DataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);
```

46.14.1. Setting locale

Bindy supports configuring the locale on the dataformat, such as

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

bindy.setLocale("us");
```

Or to use the platform default locale then use "default" as the locale name. Notice this requires Camel 2.14/2.13.3/2.12.5.

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

bindy.setLocale("default");
```

for older releases you can set it using Java code as shown

```
// Camel 2.15 or older (configure by package name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat("com.acme.model");
// Camel 2.16 onwards (configure by class name)
BindyCsvDataFormat bindy = new BindyCsvDataFormat(com.acme.model.MyModel.class);

bindy.setLocale(Locale.getDefault().getISO3Country());
```

46.14.2. Unmarshaling

```
from("file://inbox")
  .unmarshal(bindy)
  .to("direct:handleOrders");
```

Alternatively, you can use a named reference to a data format which can then be defined in your Registry e.g. your Spring XML file:

```
from("file://inbox")
  .unmarshal("myBindyDataFormat")
  .to("direct:handleOrders");
```

The Camel route will pick-up files in the inbox directory, unmarshall CSV records into a collection of model objects and send the collection
to the route referenced by 'handleOrders'.

The collection returned is a **List of Map** objects. Each Map within the list contains the model objects that were marshalled out of each line of the CSV. The reason behind this is that *each line can correspond to more than one object*. This can be confusing when you simply expect one object to be returned per line.

Each object can be retrieve using its class name.

```
List<Map<String, Object>> unmarshaledModels = (List<Map<String, Object>>)
exchange.getIn().getBody();

int modelCount = 0;
for (Map<String, Object> model : unmarshaledModels) {
  for (String className : model.keySet()) {
     Object obj = model.get(className);
     LOG.info("Count : " + modelCount + ", " + obj.toString());
  }
 modelCount++;
}

LOG.info("Total CSV records received by the csv bean : " + modelCount);
```

Assuming that you want to extract a single Order object from this map for processing in a route, you could use a combination of a Splitter and a Processor as per the following:

```
from("file://inbox")
    .unmarshal(bindy)
    .split(body())
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                Message in = exchange.getIn();
                Map<String, Object> modelMap = (Map<String, Object>) in.getBody();
                in.setBody(modelMap.get(Order.class.getCanonicalName()));
            }
        })
        .to("direct:handleSingleOrder")
    .end();
```

Take care of the fact that Bindy uses CHARSET_NAME property or the CHARSET_NAME header as define in the Exchange interface to do a characterset conversion of the inputstream received for unmarshalling. In some producers (e.g. file-endpoint) you can define a characterset. The characterset

conversion can already been done by this producer. Sometimes you need to remove this property or header from the exchange before sending it to the unmarshal. If you don't remove it the conversion might be done twice which might lead to unwanted results.

```
from("file://inbox?charset=Cp922")
  .removeProperty(Exchange.CHARSET_NAME)
  .unmarshal("myBindyDataFormat")
  .to("direct:handleOrders");
```

46.14.3. Marshaling

To generate CSV records from a collection of model objects, you create the following route :

```
from("direct:handleOrders")
   .marshal(bindy)
   .to("file://outbox")
```

46.15. USING SPRING XML

This is really easy to use Spring as your favorite DSL language to declare the routes to be used for camel-bindy. The following example shows two routes where the first will pick-up records from files, unmarshal the content and bind it to their model. The result is then send to a pojo (doing nothing special) and place them into a queue.

The second route will extract the pojos from the queue and marshal the content to generate a file containing the csv record. The example above is for using Camel 2.16 onwards.

spring dsl

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://camel.apache.org/schema/spring
      http://camel.apache.org/schema/spring/camel-spring.xsd">

   <!-- Queuing engine - ActiveMq - work locally in mode virtual memory -->
   <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
      <property name="brokerURL" value="vm://localhost:61616"/>
   </bean>

   <camelContext xmlns="http://camel.apache.org/schema/spring">
      <dataFormats>
        <bindy id="bindyDataformat" type="Csv" classType="org.apache.camel.bindy.model.Order"/>
      </dataFormats>

      <route>
         <from uri="file://src/data/csv/?noop=true" />
         <unmarshal ref="bindyDataformat" />
         <to uri="bean:csv" />
         <to uri="activemq:queue:in" />
```

```
        </route>

        <route>
            <from uri="activemq:queue:in" />
            <marshal ref="bindyDataformat" />
            <to uri="file://src/data/csv/out/" />
        </route>
    </camelContext>
</beans>
```

NOTE

Please verify that your model classes implements serializable otherwise the queue manager will raise an error

46.16. DEPENDENCIES

To use Bindy in your camel routes you need to add the a dependency on **camel-bindy** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-bindy</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 47. USING OSGI BLUEPRINT WITH CAMEL

A custom XML namespace for Blueprint has been created to let you leverage the nice XML dialect. Given Blueprint custom namespaces are not standardized yet, this namespace can only be used on the Apache Aries Blueprint implementation, which is the one used by Apache Karaf.

47.1. OVERVIEW

The XML schema is mostly the same as the one for Spring, so all the xml snippets throughout the documentation referring to Spring XML also apply to Blueprint routes.

Here is a very simple route definition using blueprint:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="timer:test" />
      <to uri="log:test" />
    </route>
  </camelContext>

</blueprint>
```

There are a few limitations at this point about the supported xml elements (compared to the Spring xml syntax):

- beanPostProcessor are specific to Spring and aren't allowed

However, using blueprint when you deploy your applications in an OSGi enviroment has several advantages:

- when upgrading to a new camel version, you don't have to change the namespace, as the correct version will be selected based on the camel packages that are imported by your bundle
- no startup ordering issue with respect to the custom namespaces and your bundles
- you can use Blueprint property placeholders

47.2. USING CAMEL-BLUEPRINT

To leverage camel-blueprint in OSGi, you only need the Aries Blueprint bundle and the camel-blueprint bundle, in addition to camel-core and its dependencies.

If you use Karaf, you can use the feature named camel-blueprint which will install all the required bundles.

CHAPTER 48. BONITA COMPONENT

Available as of Camel version 2.19

Used for communicating with a remote Bonita BPM process engine.

48.1. URI FORMAT

```
bonita://[operation]?[options]
```

Where **operation** is the specific action to perform on Bonita.

48.2. GENERAL OPTIONS

The Bonita component has no options.

The Bonita endpoint is configured using URI syntax:

```
bonita:operation
```

with the following path and query parameters:

48.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
operation	**Required** Operation to use		BonitaOperation

48.2.2. Query Parameters (9 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
hostname (consumer)	Hostname where Bonita engine runs	localhost	String
port (consumer)	Port of the server hosting Bonita engine	8080	String

Name	Description	Default	Type
processName (consumer)	Name of the process involved in the operation		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
password (security)	Password to authenticate to Bonita engine.		String
username (security)	Username to authenticate to Bonita engine.		String

48.3. BODY CONTENT

For the startCase operation, the input variables are retrieved from the body message. This one has to contains a Map<String,Serializable>.

48.4. EXAMPLES

The following example start a new case in Bonita:

```
from("direct:start").to("bonita:startCase?
hostname=localhost&amp;port=8080&amp;processName=TestProcess&amp;username=install&amp;pa
ssword=install")
```

48.5. DEPENDENCIES

To use Bonita in your Camel routes you need to add a dependency on **camel-bonita**, which implements the component.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest and greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-bonita</artifactId>
```

```
  <version>x.x.x</version>
</dependency>
```

CHAPTER 49. BOON DATAFORMAT

Available as of Camel version 2.16

Boon is a Data Format which uses the Boon JSON marshalling library to unmarshal an JSON payload into Java objects or to marshal Java objects into an JSON payload. Boon aims to be a simple and fast parser than other common parsers currently used.

49.1. OPTIONS

The Boon dataformat supports 3 options which are listed below.

Name	Default	Java Type	Description
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
useList	**false**	**Boolean**	To unarmshal to a List of Map or a List of Pojo.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

49.2. USING THE JAVA DSL

```
DataFormat boonDataFormat = new BoonDataFormat("com.acme.model.Person");

from("activemq:My.Queue")
  .unmarshal(boonDataFormat)
  .to("mqseries:Another.Queue");
```

49.3. USING BLUEPRINT XML

```
<bean id="boonDataFormat" class="org.apache.camel.component.boon.BoonDataFormat">
  <argument value="com.acme.model.Person"/>
</bean>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/blueprint">
  <route>
    <from uri="activemq:My.Queue"/>
    <unmarshal ref="boonDataFormat"/>
    <to uri="mqseries:Another.Queue"/>
  </route>
</camelContext>
```

49.4. DEPENDENCIES

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-boon</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 50. BOX COMPONENT

Available as of Camel version 2.14

The Box component provides access to all of the Box.com APIs accessible using https://github.com/box/box-java-sdk. It allows producing messages to upload and download files, create, edit, and manage folders, etc. It also supports APIs that allow polling for updates to user accounts and even changes to enterprise accounts, etc.

Box.com requires the use of OAuth2.0 for all client application authentication. In order to use camel-box with your account, you'll need to create a new application within Box.com at https://developer.box.com. The Box application's client id and secret will allow access to Box APIs which require a current user. A user access token is generated and managed by the API for an end user.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-box</artifactId>
    <version>${camel-version}</version>
</dependency>
```

50.1. CONNECTION AUTHENTICATION TYPES

The Box component supports three different types of authenticated connections.

50.1.1. Standard Authentication

Standard Authentication uses the **OAuth 2.0 three-legged authentication process** to authenticate its connections with Box.com. This type of authentication enables Box **managed users** and **external users** to access, edit, and save their Box content through the Box component.

50.1.2. App Enterprise Authentication

App Enterprise Authentication uses the **OAuth 2.0 with JSON Web Tokens (JWT)** to authenticate its connections as a **Service Account** for a **Box Application**. This type of authentication enables a service account to access, edit, and save the Box content of its **Box Application** through the Box component.

50.1.3. App User Authentication

App User Authentication uses the **OAuth 2.0 with JSON Web Tokens (JWT)** to authenticate its connections as an **App User** for a **Box Application**. This type of authentication enables app users to access, edit, and save their Box content in its **Box Application** through the Box component.

50.2. BOX OPTIONS

The Box component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		BoxConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Box endpoint is configured using URI syntax:

```
box:apiName/methodName
```

with the following path and query parameters:

50.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		BoxApiName
methodName	**Required** What sub operation to use for the selected operation		String

50.2.2. Query Parameters (20 parameters):

Name	Description	Default	Type
clientId (common)	Box application client ID		String
enterpriseId (common)	The enterprise ID to use for an App Enterprise.		String
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
userId (common)	The user ID to use for an App User.		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
httpParams (advanced)	Custom HTTP params for settings like proxy host		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accessTokenCache (security)	Custom Access Token Cache for storing and retrieving access tokens.		IAccessTokenCache
clientSecret (security)	Box application client secret		String
encryptionAlgorithm (security)	The type of encryption algorithm for JWT. Supported Algorithms: RSA_SHA_256 RSA_SHA_384 RSA_SHA_512	RSA_SHA_256	EncryptionAlgorithm
maxCacheEntries (security)	The maximum number of access tokens in cache.	100	int
authenticationType (authentication)	The type of authentication for connection. Types of Authentication: STANDARD_AUTHENTICATION - OAuth 2.0 (3-legged) SERVER_AUTHENTICATION - OAuth 2.0 with JSON Web Tokens	APP_USER_AUTHENTICATION	String
privateKeyFile (security)	The private key for generating the JWT signature.		String

Name	Description	Default	Type
privateKeyPassword (security)	The password for the private key.		String
publicKeyId (security)	The ID for public key for validating the JWT signature.		String
sslContextParameters (security)	To configure security using SSLContextParameters.		SSLContextParameters
userName (security)	Box user name, MUST be provided		String
userPassword (security)	Box user password, MUST be provided if authSecureStorage is not set, or returns null on first call		String

50.3. URI FORMAT

```
box:apiName/methodName
```

apiName can be one of:

- collaborations
- comments
- event-logs
- files
- folders
- groups
- events
- search
- tasks
- users

50.4. PRODUCER ENDPOINTS:

Producer endpoints can use endpoint prefixes followed by endpoint names and associated options described next. A shorthand alias can be used for some endpoints. The endpoint URI MUST contain a prefix.

Endpoint options that are not mandatory are denoted by **[]**. When there are no mandatory options for an endpoint, one of the set of **[]** options MUST be provided. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelBox.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelBox.option** header.

If a value is not provided for the option **defaultRequest** either in the endpoint URI or in a message header, it will be assumed to be **null**. Note that the **null** value will only be used if other options do not satisfy matching endpoints.

In case of Box API errors the endpoint will throw a RuntimeCamelException with a **com.box.sdk.BoxAPIException** derived exception cause.

50.4.1. Endpoint Prefix *collaborations*

For more information on Box collaborations see https://developer.box.com/reference#collaboration-object. The following endpoints can be invoked with the prefix **collaborations** as follows:

```
box:collaborations/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
addFolderCollaboration	add	folderId, collaborator, role	com.box.sdk.BoxCollaboration
addFolderCollaborationByEmail	addByEmail	folderId, email, role	com.box.sdk.BoxCollaboration
deleteCollaboration	delete	collaborationId	
getFolderCollaborations	collaborations	folderId	java.util.Collection

Endpoint	Shorthand Alias	Options	Result Body Type
getPendingCollaborations	pendingCollaborations		java.util.Collection
getCollaborationInfo	info	collaborationId	com.box.sdk.BoxCollaboration.Info
updateCollaborationInfo	updateInfo	collaborationId, info	com.box.sdk.BoxCollaboration

URI Options for *collaborations*

Name	Type
collaborationId	String
collaborator	com.box.sdk.BoxCollaborator
role	com.box.sdk.BoxCollaboration.Role
folderId	String
email	String
info	com.box.sdk.BoxCollaboration.Info

50.4.2. Endpoint Prefix *comments*

For more information on Box comments see https://developer.box.com/reference#comment-object. The following endpoints can be invoked with the prefix **comments** as follows:

```
box:comments/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
addFileComment	add	fileId, message	com.box.sdk.BoxFile
changeCommentMessage	updateMessage	commentId, message	com.box.sdk.BoxComment
deleteComment	delete	commentId	
getCommentInfo	info	commentId	com.box.sdk.BoxComment.Info
getFileComments	comments	fileId	java.util.List
replyToComment	reply	commentId, message	com.box.sdk.BoxComment

URI Options for *collaborations*

Name	Type
commentId	String
fileId	String
message	String

50.4.3. Endpoint Prefix *events-logs*

For more information on Box event logs see https://developer.box.com/reference#events. The following endpoints can be invoked with the prefix **events** as follows:

```
box:event-logs/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
getEnterpriseEvents	events	position, after, before, [types]	java.util.List

URI Options for *event-logs*

Name	Type
position	String
after	Date
before	Date
types	com.box.sdk.BoxEvent.Types[]

50.4.4. Endpoint Prefix *files*

For more information on Box files see https://developer.box.com/reference#file-object. The following endpoints can be invoked with the prefix **files** as follows.

```
box:files/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
uploadFile	upload	parentFolderId, content, fileName, [created], [modified], [size], [listener]	com.box.sdk.BoxFile

Endpoint	Shorthand Alias	Options	Result Body Type
downloadFile	download	fileId, output, [rangeStart], [rangeEnd], [listener]	java.io.OutputStream
copyFile	copy	fileId, destinationFolderId, [newName]	com.box.sdk.BoxFile
moveFile	move	fileId, destinationFolderId, [newName]	com.box.sdk.BoxFile
renameFile	rename	fileId, newFileName	com.box.sdk.BoxFile
createFileSharedLink	link	fileId, access, [unshareDate], [permissions]	com.box.sdk.BoxSharedLink
deleteFile	delete	fileId	
uploadNewFileVersion	uploadVersion	fileId, fileContent, [modified], [fileSize], [listener]	com.box.boxsdk.BoxFile

Endpoint	Shorthand Alias	Options	Result Body Type
promoteFileVersion	promoteVersion	fileId, version	com.box.sdk.BoxFileVersion
getFileVersions	versions	fileId	java.util.Collection
downloadPreviousFileVersions	downloadVersion	fileId, version, output, [listener]	java.io.OutputStream
deleteFileVersion	deleteVersion	fileId, version	
getFileInfo	info	fileId, fields	com.box.sdk.BoxFile.Info
updateFileInfo	updateInfo	fileId, info	com.box.sdk.BoxFile
createFileMetadata	createMetadata	fileId, metadata, [typeName]	com.box.sdk.Metadata
getFileMetadata	metadata	fileId, [typeName]	com.box.sdk.Metadata
updateFileMetadata	updateMetadata	fileId, metadata	com.box.sdk.Metadata
deleteFileMetadata	deleteMetadata	fileId	
getDownloadUrl	url	fileId	java.net.URL

Endpoint	Shorthand Alias	Options	Result Body Type
getPreviewLink	preview	fileId	java.net.URL
getFileThumbnail	thumbnail	fileId, fileType, minWidth, minHeight, maxWidth, maxHeight	byte[]

URI Options for *files*

Name	Type
parentFolderId	String
content	java.io.InputStream
fileName	String
created	Date
modified	Date
size	Long
listener	com.box.sdk.ProgressListener
output	java.io.OutputStream
rangeStart	Long

Name	Type
rangeEnd	Long
outputStreams	java.io.OutputStream[]
destinationFolderId	String
newName	String
fields	String[]
info	com.box.sdk.BoxFile.Info
fileSize	Long
version	Integer
access	com.box.sdk.BoxSharedLink.Access
unshareDate	Date
permissions	com.box.sdk.BoxSharedLink.Permissions
fileType	com.box.sdk.BoxFile.ThumbnailFileType
minWidth	Integer
minHeight	Integer
maxWidth	Integer
maxHeight	Integer

Name	Type
metadata	com.box.sdk.Metadata
typeName	String

50.4.5. Endpoint Prefix *folders*

For more information on Box folders see https://developer.box.com/reference#folder-object. The following endpoints can be invoked with the prefix **folders** as follows.

```
box:folders/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
getRootFolder	root		com.box.sdk.BoxFolder
createFolder	create	parentFolderId, folderName	com.box.sdk.BoxFolder
createFolder	create	parentFolderId, path	com.box.sdk.BoxFolder
copyFolder	copy	folderId, destinationfolderId, [newName]	com.box.sdk.BoxFolder
moveFolder	move	folderId, destinationFolderId, newName	com.box.sdk.BoxFolder

Endpoint	Shorthand Alias	Options	Result Body Type
renameFolder	rename	folderId, newFolderName	com.box.sdk.BoxFolder
createFolderSharedLink	link	folderId, access, [unsharedDate], [permissions]	java.util.List
deleteFolder	delete	folderId	
getFolder	folder	path	com.box.sdk.BoxFolder
getFolderInfo	info	folderId, fields	com.box.sdk.BoxFolder.Info
getFolderItems	items	folderId, offset, limit, fields	com.box.sdk.BoxFolder
updateFolderInfo	updateInfo	folderId, info	com.box.sdk.BoxFolder

URI Options for *folders*

Name	Type
path	String[]
folderId	String
offset	Long

Name	Type
limit	Long
fields	String[]
parentFolderId	String
folderName	String
destinationFolderId	String
newName	String
newFolderName	String
info	String
access	com.box.sdk.BoxSharedLink.Access
unshareDate	Date
permissions	com.box.sdk.BoxSharedLink.Permissions

50.4.6. Endpoint Prefix *groups*

For more information on Box groups see https://developer.box.com/reference#group-object. The following endpoints can be invoked with the prefix **groups** as follows:

```
box:groups/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type

Endpoint	Shorthand Alias	Options	Result Body Type
createGroup	create	name, [provenance, externalSyncIdentifier, description, invitabilityLevel, memberViewabilityLevel]	com.box.sdk.BoxGroup
addGroupMembership	createMembership	groupId, userId, role	com.box.sdk.BoxGroupMembership
deleteGroup	delete	groupId	
getAllGroups	groups		java.util.Collection
getGroupInfo	info	groupId	com.box.sdk.BoxGroup.Info
updateGroupInfo	updateInfo	groupId, groupInfo	com.box.sdk.BoxGroup
addGroupMembership	addMembership	groupId, userId, role	com.box.sdk.BoxGroupMembership
deleteGroupMembership	deleteMembership	groupMembershipId	

Endpoint	Shorthand Alias	Options	Result Body Type
getGroupMemberships	memberships	groupId	java.uti.Collection
getGroupMembershipInfo	membershipInfo	groupMemebershipId	com.box.sdk.BoxGroup.Info
updateGroupMembershipInfo	updateMembershipInfo	groupMemebershipId, info	com.box.sdk.BoxGroupMembership

URI Options for *groups*

Name	Type
name	String
groupId	String
userId	String
role	com.box.sdk.BoxGroupMembership.Role
groupMembershipId	String
info	com.box.sdk.BoxGroupMembership.Info

50.4.7. Endpoint Prefix *search*

For more information on Box search API see https://developer.box.com/reference#searching-for-content. The following endpoints can be invoked with the prefix **search** as follows:

```
box:search/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
searchFolder	search	folderId, query	java.util.Collection

URI Options for *search*

Name	Type
folderId	String
query	String

50.4.8. Endpoint Prefix *tasks*

For information on Box tasks see https://developer.box.com/reference#task-object-1. The following endpoints can be invoked with the prefix **tasks** as follows:

```
box:tasks/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
addFileTask	add	fileId, action, dueAt, [message]	com.box.sdk.BoxUser
deleteTask	delete	taskId	
getFileTasks	tasks	fileId	java.util.List
getTaskInfo	info	taskId	com.box.sdk.BoxTask.Info
updateTaskInfo	updateInfo	taskId, info	com.box.sdk.BoxTask

Endpoint	Shorthand Alias	Options	Result Body Type
addAssignmentToTask	addAssignment	taskId, assignTo	com.box.sdk.BoxTask
deleteTaskAssignment	deleteAssignment	taskAssignmentId	
getTaskAssignments	assignments	taskId	java.util.List
getTaskAssignmentInfo	assignmentInfo	taskAssignmentId	com.box.sdk.BoxTaskAssignment.Info

URI Options for *tasks*

Name	Type
fileId	String
action	com.box.sdk.BoxTask.Action
dueAt	Date
message	String
taskId	String
info	com.box.sdk.BoxTask.Info
assignTo	com.box.sdk.BoxUser
taskAssignmentId	String

50.4.9. Endpoint Prefix *users*

For information on Box users see https://developer.box.com/reference#user-object. The following endpoints can be invoked with the prefix **users** as follows:

```
box:users/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
getCurrentUser	currentUser		com.box.sdk.BoxUser
getAllEnterpriseOrExternalUsers	users	filterTerm, [fields]	com.box.sdk.BoxUser
createAppUser	create	name, [params]	com.box.sdk.BoxUser
createEnterpriseUser	create	login, name, [params]	com.box.sdk.BoxUser
deleteUser	delete	userId, notifyUser, force	
getUserEmailAlias	emailAlias	userId	com.box.sdk.BoxUser
deleteUserEmailAlias	deleteEmailAlias	userId, emailAliasId	java.util.List
getUserInfo	info	userId	com.box.sdk.BoxUser.Info
updateUserInfo	updateInfo	userId, info	com.box.sdk.BoxUser

Endpoint	Shorthand Alias	Options	Result Body Type
moveFolderToUser	-	userId, sourceUserId	com.box.sdk.BoxFolder.Info

URI Options for *users*

Name	Type
defaultRequest	com.box.restclientv2.requestsbase.BoxDefaultRequestObject
emailAliasRequest	com.box.boxjavalibv2.requests.requestobjects.BoxEmailAliasRequestObject
emailId	String
filterTerm	String
folderId	String
simpleUserRequest	com.box.boxjavalibv2.requests.requestobjects.BoxSimpleUserRequestObject
userDeleteRequest	com.box.boxjavalibv2.requests.requestobjects.BoxUserDeleteRequestObject
userId	String
userRequest	com.box.boxjavalibv2.requests.requestobjects.BoxUserRequestObject
userUpdateLoginRequest	com.box.boxjavalibv2.requests.requestobjects.BoxUserUpdateLoginRequestObject

50.5. CONSUMER ENDPOINTS:

For more information on Box events see https://developer.box.com/reference#events. Consumer endpoints can only use the endpoint prefix **events** as shown in the example next.

```
box:events/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
events		[startingPosition]	com.box.sdk.BoxEvent

URI Options for *events*

Name	Type
startingPosition	Long

50.6. MESSAGE HEADER

Any of the options can be provided in a message header for producer endpoints with **CamelBox.** prefix.

50.7. MESSAGE BODY

All result message bodies utilize objects provided by the Box Java SDK. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint parameter.

50.8. SAMPLES

The following route uploads new files to the user's root folder:

```
from("file:...")
    .to("box://files/upload/inBody=fileUploadRequest");
```

The following route polls user's account for updates:

```
from("box://events/listen?startingPosition=-1")
    .to("bean:blah");
```

The following route uses a producer with dynamic header options. The **fileId** property has the Box file id and the **output** property has the output stream of the file contents, so they are assigned to the **CamelBox.fileId** header and **CamelBox.output** header respectively as follows:

```
from("direct:foo")
    .setHeader("CamelBox.fileId", header("fileId"))
    .setHeader("CamelBox.output", header("output"))
```

```
    .to("box://files/download")
    .to("file://...");
```

CHAPTER 51. BRAINTREE COMPONENT

Available as of Camel version 2.17

The Braintree component provides access to Braintree Payments trough through theirs Java SDK.

All client applications need API credential in order to process payments. In order to use camel-braintree with your account, you'll need to create a new Sandbox or Production account.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-braintree</artifactId>
  <version>${camel-version}</version>
</dependency>
```

51.1. BRAINTREE OPTIONS

The Braintree component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		BraintreeConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Braintree endpoint is configured using URI syntax:

```
braintree:apiName/methodName
```

with the following path and query parameters:

51.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		BraintreeApiName
methodName	What sub operation to use for the selected operation		String

51.1.2. Query Parameters (14 parameters):

Name	Description	Default	Type
environment (common)	The environment Either SANDBOX or PRODUCTION		String
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
merchantId (common)	The merchant id provided by Braintree.		String
privateKey (common)	The private key provided by Braintree.		String
publicKey (common)	The public key provided by Braintree.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
accessToken (advanced)	The access token granted by a merchant to another in order to process transactions on their behalf. Used in place of environment, merchant id, public key and private key fields.		String
httpReadTimeout (advanced)	Set read timeout for http calls.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
httpLogLevel (logging)	Set logging level for http calls, see java.util.logging.Level		String
proxyHost (proxy)	The proxy host		String
proxyPort (proxy)	The proxy port		Integer

51.2. URI FORMAT

```
braintree://endpoint-prefix/endpoint?[options]
```

Endpoint prefix can be one of:

- addOn
- address
- clientToken
- creditCardverification
- customer
- discount
- merchantAccount
- paymentmethod
- paymentmethodNonce
- plan
- settlementBatchSummary
- subscription
- transaction
- webhookNotification

51.3. BRAINTREECOMPONENT

The Braintree Component can be configured with the options below. These options can be provided using the component's bean property **configuration** of type **org.apache.camel.component.braintree.BraintreeConfiguration**.

Option	Type	Description
environment	String	Value that specifies where requests should be directed – sandbox or production
merchantId	String	A unique identifier for your gateway account, which is different than your merchant account ID
publicKey	String	User-specific public identifier
privateKey	String	User-specific secure identifier that should not be shared – even with us!
accessToken	String	Token granted to a merchant using Braintree Auth allowing them to process transactions on another's behalf. Used in place of the environment, merchantId, publicKey and privateKey options.

All the options above are provided by Braintree Payments

51.4. PRODUCER ENDPOINTS:

Producer endpoints can use endpoint prefixes followed by endpoint names and associated options described next. A shorthand alias can be used for some endpoints. The endpoint URI MUST contain a prefix.

Endpoint options that are not mandatory are denoted by []. When there are no mandatory options for an endpoint, one of the set of [] options MUST be provided. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelBraintree.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelBraintree.option** header.

For more information on the endpoints and options see Braintree references at https://developers.braintreepayments.com/reference/overview

51.4.1. Endpoint prefix *addOn*

The following endpoints can be invoked with the prefix **addOn** as follows:

```
braintree://addOn/endpoint
```

Endpoint	Shorthand Alias	Options	Result Body Type
all			List<com.braintreegateway.Addon>

51.4.2. Endpoint prefix *address*

The following endpoints can be invoked with the prefix **address** as follows:

```
braintree://address/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
create		customerId, request	com.braintreegateway.Result<com.braintreegateway.Address>
delete		customerId, id	com.braintreegateway.Result<com.braintreegateway.Address>
find		customerId, id	com.braintreegateway.Address
update		customerId, id, request	com.braintreegateway.Result<com.braintreegateway.Address>

URI Options for *address*

Name	Type
customerId	String
request	com.braintreegateway.AddressRequest
id	String

51.4.3. Endpoint prefix *clientToken*

The following endpoints can be invoked with the prefix **clientToken** as follows:

```
braintree://clientToken/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
generate		request	String

URI Options for *clientToken*

Name	Type
request	com.braintreegateway.ClientTokenrequest

51.4.4. Endpoint prefix *creditCardVerification*

The following endpoints can be invoked with the prefix **creditCardverification** as follows:

```
braintree://creditCardVerification/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
find		id	com.braintreegateway.CreditCardVerification
search		query	com.braintreegateway.ResourceCollection<com.braintreegateway.CreditCardVerification>

URI Options for *creditCardVerification*

Name	Type
id	String
query	com.braintreegateway.CreditCardVerificationSearchRequest

51.4.5. Endpoint prefix *customer*

The following endpoints can be invoked with the prefix **customer** as follows:

```
braintree://customer/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
all			
create		request	com.braintreegateway.Result<com.braintreegateway.Customer>
delete		id	com.braintreegateway.Result<com.braintreegateway.Customer>
find		id	com.braintreegateway.Customer
search		query	com.braintreegateway.ResourceCollection<com.braintreegateway.Customer>
update		id, request	com.braintreegateway.Result<com.braintreegateway.Customer>

URI Options for *customer*

Name	Type
id	String
request	com.braintreegateway.CustomerRequest
query	com.braintreegateway.CustomerSearchRequest

51.4.6. Endpoint prefix *discount*

The following endpoints can be invoked with the prefix **discount** as follows:

```
braintree://discount/endpoint
```

Endpoint	Shorthand Alias	Options	Result Body Type
all			List<com.braintreegateway.Discount>

+

+

51.4.7. Endpoint prefix *merchantAccount*

The following endpoints can be invoked with the prefix **merchantAccount** as follows:

braintree://*merchantAccount/endpoint?[options]*

Endpoint	Shorthand Alias	Options	Result Body Type
create		request	com.braintreegateway.Result<com.braintreegateway.MerchantAccount>
createForCurrency		currencyRequest	com.braintreegateway.Result<com.braintreegateway.MerchantAccount>
find		id	com.braintreegateway.MerchantAccount
update		id, request	com.braintreegateway.Result<com.braintreegateway.MerchantAccount>

URI Options for *merchantAccount*

Name	Type
id	String
request	com.braintreegateway.MerchantAccountRequest
currencyRequest	com.braintreegateway.MerchantAccountCreateForCurrencyRequest

51.4.8. Endpoint prefix *paymentMethod*

The following endpoints can be invoked with the prefix **paymentMethod** as follows:

```
braintree://paymentMethod/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
create		request	com.braintreegateway.Result<com.braintreegateway.PaymentMethod>
delete		token, deleteRequest	com.braintreegateway.Result<com.braintreegateway.PaymentMethod>
find		token	com.braintreegateway.PaymentMethod
update		token, request	com.braintreegateway.Result<com.braintreegateway.PaymentMethod>

URI Options for *paymentMethod*

Name	Type
token	String
request	com.braintreegateway.PaymentMethodRequest
deleteRequest	com.braintreegateway.PaymentMethodDeleteRequest

51.4.9. Endpoint prefix *paymentMethodNonce*

The following endpoints can be invoked with the prefix **paymentMethodNonce** as follows:

```
braintree://paymentMethodNonce/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
create		paymentMethodToken	com.braintreegateway.Result<com.braintreegateway.PaymentMethodNonce>

Endpoint	Shorthand Alias	Options	Result Body Type
find		paymentMethodNonce	com.braintreegateway.PaymentMethodNonce

URI Options for *paymentMethodNonce*

Name	Type
paymentMethodToken	String
paymentMethodNonce	String

51.4.10. Endpoint prefix *plan*

The following endpoints can be invoked with the prefix **plan** as follows:

```
braintree://plan/endpoint
```

Endpoint	Shorthand Alias	Options	Result Body Type
all			List<com.braintreegateway.Plan>

51.4.11. Endpoint prefix *settlementBatchSummary*

The following endpoints can be invoked with the prefix **settlementBatchSummary** as follows:

```
braintree://settlementBatchSummary/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
generate		request	com.braintreegateway.Result<com.braintreegateway.SettlementBatchSummary>

URI Options for *settlementBatchSummary*

Name	Type
settlementDate	Calendar
groupByCustomField	String

51.4.12. Endpoint prefix *subscription*

The following endpoints can be invoked with the prefix **subscription** as follows:

```
braintree://subscription/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
cancel		id	com.braintreegateway.Result<com.braintreegateway.Subscription>
create		request	com.braintreegateway.Result<com.braintreegateway.Subscription>
delete		customerId, id	com.braintreegateway.Result<com.braintreegateway.Subscription>
find		id	com.braintreegateway.Subscription
retryCharge		subscriptionId, amount	com.braintreegateway.Result<com.braintreegateway.Transaction>
search		searchRequest	com.braintreegateway.ResourceCollection<com.braintreegateway.Subscription>
update		id, request	com.braintreegateway.Result<com.braintreegateway.Subscription>

URI Options for *subscription*

Name	Type
id	String
request	com.braintreegateway.SubscriptionRequest
customerId	String
subscriptionId	String
amount	BigDecimal
searchRequest	com.braintreegateway.SubscriptionSearchRequest.

51.4.13. Endpoint prefix *transaction*

The following endpoints can be invoked with the prefix **transaction** as follows:

```
braintree://transaction/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
cancelRelease		id	com.braintreegateway.Result<com.braintreegateway.Transaction>
cloneTransaction		id, cloneRequest	com.braintreegateway.Result<com.braintreegateway.Transaction>
credit		request	com.braintreegateway.Result<com.braintreegateway.Transaction>
find		id	com.braintreegateway.Transaction
holdInEscrow		id	com.braintreegateway.Result<com.braintreegateway.Transaction>

Endpoint	Shorthand Alias	Options	Result Body Type
releaseFromEscrow		id	com.braintreegateway.Result<com.braintreegateway.Transaction>
refund		id, amount, refundRequest	com.braintreegateway.Result<com.braintreegateway.Transaction>
sale		request	com.braintreegateway.Result<com.braintreegateway.Transaction>
search		query	com.braintreegateway.ResourceCollection<com.braintreegateway.Transaction>
submitForPartialSettlement		id, amount	com.braintreegateway.Result<com.braintreegateway.Transaction>
submitForSettlement		id, amount, request	com.braintreegateway.Result<com.braintreegateway.Transaction>
voidTransaction		id	com.braintreegateway.Result<com.braintreegateway.Transaction>

URI Options for *transaction*

Name	Type
id	String
request	com.braintreegateway.TransactionCloneRequest
cloneRequest	com.braintreegateway.TransactionCloneRequest
refundRequest	com.braintreegateway.TransactionRefundRequest
amount	BigDecimal
query	com.braintreegateway.TransactionSearchRequest

51.4.14. Endpoint prefix *webhookNotification*

The following endpoints can be invoked with the prefix **webhookNotification** as follows:

```
braintree://webhookNotification/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
parse		signature, payload	com.braintreegateway.WebhookNotification
verify		challenge	String

URI Options for *webhookNotification*

Name	Type
signature	String
payload	String
challenge	String

51.5. CONSUMER ENDPOINTS

Any of the producer endpoints can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. By default Consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange. To change this behavior use the property **consumer.splitResults=true** to return a single exchange for the entire list or array.

51.6. MESSAGE HEADERS

Any URI option can be provided in a message header for producer endpoints with a **CamelBraintree.** prefix.

51.7. MESSAGE BODY

All result message bodies utilize objects provided by the Braintree Java SDK. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint parameter.

51.8. EXAMPLES

Blueprint

```
<?xml version="1.0"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
          xsi:schemaLocation="
            http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0
http://aries.apache.org/schemas/blueprint-cm/blueprint-cm-1.0.0.xsd
            http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
            http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-
blueprint.xsd">


    <cm:property-placeholder id="placeholder" persistent-id="camel.braintree">
    </cm:property-placeholder>

    <bean id="braintree" class="org.apache.camel.component.braintree.BraintreeComponent">
        <property name="configuration">
            <bean class="org.apache.camel.component.braintree.BraintreeConfiguration">
                <property name="environment" value="${environment}"/>
                <property name="merchantId" value="${merchantId}"/>
                <property name="publicKey" value="${publicKey}"/>
                <property name="privateKey" value="${privateKey}"/>
            </bean>
        </property>
    </bean>

    <camelContext trace="true" xmlns="http://camel.apache.org/schema/blueprint" id="braintree-
example-context">
        <route id="braintree-example-route">
            <from uri="direct:generateClientToken"/>
            <to uri="braintree://clientToken/generate"/>
            <to uri="stream:out"/>
        </route>
    </camelContext>

</blueprint>
```

51.9. SEE ALSO

* Configuring Camel * Component * Endpoint * Getting Started

CHAPTER 52. BROWSE COMPONENT

Available as of Camel version 1.3

The Browse component provides a simple BrowsableEndpoint which can be useful for testing, visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.

52.1. URI FORMAT

```
browse:someName[?options]
```

Where **someName** can be any string to uniquely identify the endpoint.

52.2. OPTIONS

The Browse component has no options.

The Browse endpoint is configured using URI syntax:

```
browse:name
```

with the following path and query parameters:

52.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** A name which can be any string to uniquely identify the endpoint		String

52.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

52.3. SAMPLE

In the route below, we insert a **browse:** component to be able to browse the Exchanges that are passing through:

```
from("activemq:order.in").to("browse:orderReceived").to("bean:processOrder");
```

We can now inspect the received exchanges from within the Java code:

```
private CamelContext context;

public void inspectRecievedOrders() {
    BrowsableEndpoint browse = context.getEndpoint("browse:orderReceived",
BrowsableEndpoint.class);
    List<Exchange> exchanges = browse.getExchanges();

    // then we can inspect the list of received exchanges from Java
    for (Exchange exchange : exchanges) {
        String payload = exchange.getIn().getBody();
        // do something with payload
    }
}
```

52.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 53. EHCACHE COMPONENT (DEPRECATED)

Available as of Camel version 2.1

The **cache** component enables you to perform caching operations using EHCache as the Cache Implementation. The cache itself is created on demand or if a cache of that name already exists then it is simply utilized with its original settings.

This component supports producer and event based consumer endpoints.

The Cache consumer is an event based consumer and can be used to listen and respond to specific cache activities. If you need to perform selections from a pre-existing cache, use the processors defined for the cache component.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cache</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

53.1. URI FORMAT

```
cache://cacheName[?options]
```

You can append query options to the URI in the following format, **?option=value&option=#beanRef&...**

53.2. OPTIONS

The EHCache component supports 4 options which are listed below.

Name	Description	Default	Type
cacheManagerFactory (advanced)	To use the given CacheManagerFactory for creating the CacheManager. By default the DefaultCacheManagerFactory is used.		CacheManagerFactory
configuration (common)	Sets the Cache configuration		CacheConfiguration
configurationFile (common)	Sets the location of the ehcache.xml file to load from classpath or file system. By default the file is loaded from classpath:ehcache.xml	classpath:ehcache.xml	String

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The EHCache endpoint is configured using URI syntax:

```
cache:cacheName
```

with the following path and query parameters:

53.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** Name of the cache		String

53.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
diskExpiryThreadInterval Seconds (common)	The number of seconds between runs of the disk expiry thread.		long
diskPersistent (common)	Whether the disk store persists between restarts of the application.	false	boolean
diskStorePath (common)	**Deprecated** This parameter is ignored. CacheManager sets it using setter injection.		String
eternal (common)	Sets whether elements are eternal. If eternal, timeouts are ignored and the element never expires.	false	boolean
key (common)	The default key to use. If a key is provided in the message header, then the key from the header takes precedence.		String
maxElementsInMemory (common)	The number of elements that may be stored in the defined cache in memory.	1000	int

Name	Description	Default	Type
memoryStoreEvictionPolicy (common)	Which eviction strategy to use when maximum number of elements in memory is reached. The strategy defines which elements to be removed. LRU - Lest Recently Used LFU - Lest Frequently Used FIFO - First In First Out	LFU	MemoryStoreEviction Policy
objectCache (common)	Whether to turn on allowing to store non serializable objects in the cache. If this option is enabled then overflow to disk cannot be enabled as well.	false	boolean
operation (common)	The default cache operation to use. If an operation in the message header, then the operation from the header takes precedence.		String
overflowToDisk (common)	Specifies whether cache may overflow to disk	true	boolean
timeToIdleSeconds (common)	The maximum amount of time between accesses before an element expires	300	long
timeToLiveSeconds (common)	The maximum time between creation time and when an element expires. Is used only if the element is not eternal	300	long
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
cacheLoaderRegistry (advanced)	To configure cache loader using the CacheLoaderRegistry		CacheLoaderRegistry

Name	Description	Default	Type
cacheManagerFactory (advanced)	To use a custom CacheManagerFactory for creating the CacheManager to be used by this endpoint. By default the CacheManagerFactory configured on the component is used.		CacheManagerFactory
eventListenerRegistry (advanced)	To configure event listeners using the CacheEventListenerRegistry		CacheEventListenerRegistry
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

53.3. SENDING/RECEIVING MESSAGES TO/FROM THE CACHE

53.3.1. Message Headers up to Camel 2.7

Header	Description
CACHE_OPERATION	The operation to be performed on the cache. Valid options are * GET * CHECK * ADD * UPDATE * DELETE * DELETEALL **GET** and **CHECK** requires **Camel 2.3** onwards.
CACHE_KEY	The cache key used to store the Message in the cache. The cache key is optional if the CACHE_OPERATION is DELETEALL

53.3.2. Message Headers Camel 2.8+

Header changes in Camel 2.8

The header names and supported values have changed to be prefixed with 'CamelCache' and use mixed case. This makes them easier to identify and keep separate from other headers. The CacheConstants variable names remain unchanged, just their values have been changed. Also, these headers are now removed from the exchange after the cache operation is performed.

Header	Description
CamelCacheOperation	The operation to be performed on the cache. The valid options are * CamelCacheGet * CamelCacheCheck * CamelCacheAdd * CamelCacheUpdate * CamelCacheDelete * CamelCacheDeleteAll
CamelCacheKey	The cache key used to store the Message in the cache. The cache key is optional if the CamelCacheOperation is CamelCacheDeleteAll

The **CamelCacheAdd** and **CamelCacheUpdate** operations support additional headers:

Header	Type	Description
CamelCacheTimeToLive	**Integer**	**Camel 2.11:** Time to live in seconds.
CamelCacheTimeToIdle	**Integer**	**Camel 2.11:** Time to idle in seconds.
CamelCacheEternal	**Boolean**	**Camel 2.11:** Whether the content is eternal.

53.3.3. Cache Producer

Sending data to the cache involves the ability to direct payloads in exchanges to be stored in a pre-existing or created-on-demand cache. The mechanics of doing this involve

- setting the Message Exchange Headers shown above.
- ensuring that the Message Exchange Body contains the message directed to the cache

53.3.4. Cache Consumer

Receiving data from the cache involves the ability of the CacheConsumer to listen on a pre-existing or created-on-demand Cache using an event Listener and receive automatic notifications when any cache activity take place (i.e CamelCacheGet/CamelCacheUpdate/CamelCacheDelete/CamelCacheDeleteAll). Upon such an activity taking place

- an exchange containing Message Exchange Headers and a Message Exchange Body containing the just added/updated payload is placed and sent.
- in case of a CamelCacheDeleteAll operation, the Message Exchange Header CamelCacheKey and the Message Exchange Body are not populated.

53.3.5. Cache Processors

There are a set of nice processors with the ability to perform cache lookups and selectively replace payload content at the

- body
- token
- xpath level

53.4. CACHE USAGE SAMPLES

53.4.1. Example 1: Configuring the cache

```
from("cache://MyApplicationCache" +
        "?maxElementsInMemory=1000" +
        "&memoryStoreEvictionPolicy=" +
           "MemoryStoreEvictionPolicy.LFU" +
        "&overflowToDisk=true" +
        "&eternal=true" +
        "&timeToLiveSeconds=300" +
        "&timeToIdleSeconds=true" +
        "&diskPersistent=true" +
        "&diskExpiryThreadIntervalSeconds=300")
```

53.4.2. Example 2: Adding keys to the cache

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
    from("direct:start")
    .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_ADD))
    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
    .to("cache://TestCache1")
  }
};
```

53.4.3. Example 2: Updating existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
    from("direct:start")
    .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_UPDATE))
    .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
    .to("cache://TestCache1")
  }
};
```

53.4.4. Example 3: Deleting existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
    from("direct:start")
    .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_DELETE))
    .setHeader(CacheConstants.CACHE_KEY", constant("Ralph_Waldo_Emerson"))
    .to("cache://TestCache1")
  }
};
```

53.4.5. Example 4: Deleting all existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
    from("direct:start")
    .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_DELETEALL))
    .to("cache://TestCache1");
   }
};
```

53.4.6. Example 5: Notifying any changes registering in a Cache to Processors and other Producers

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
    from("cache://TestCache1")
    .process(new Processor() {
      public void process(Exchange exchange)
           throws Exception {
         String operation = (String)
exchange.getIn().getHeader(CacheConstants.CACHE_OPERATION);
         String key = (String) exchange.getIn().getHeader(CacheConstants.CACHE_KEY);
         Object body = exchange.getIn().getBody();
         // Do something
      }
    })
  }
};
```

53.4.7. Example 6: Using Processors to selectively replace payload with cache values

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    //Message Body Replacer
    from("cache://TestCache1")
    .filter(header(CacheConstants.CACHE_KEY).isEqualTo("greeting"))
    .process(new CacheBasedMessageBodyReplacer("cache://TestCache1","farewell"))
    .to("direct:next");

   //Message Token replacer
   from("cache://TestCache1")
   .filter(header(CacheConstants.CACHE_KEY).isEqualTo("quote"))
   .process(new CacheBasedTokenReplacer("cache://TestCache1","novel","#novel#"))
   .process(new CacheBasedTokenReplacer("cache://TestCache1","author","#author#"))
   .process(new CacheBasedTokenReplacer("cache://TestCache1","number","#number#"))
   .to("direct:next");

   //Message XPath replacer
   from("cache://TestCache1").
   .filter(header(CacheConstants.CACHE_KEY).isEqualTo("XML_FRAGMENT"))
   .process(new CacheBasedXPathReplacer("cache://TestCache1","book1","/books/book1"))
   .process (new CacheBasedXPathReplacer("cache://TestCache1","book2","/books/book2"))
```

```
    .to("direct:next");
  }
};
```

53.4.8. Example 7: Getting an entry from the Cache

```
from("direct:start")
   // Prepare headers
   .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_GET))
   .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
   .to("cache://TestCache1").
   // Check if entry was not found
   .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
      // If not found, get the payload and put it to cache
      .to("cxf:bean:someHeavyweightOperation").
      .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_ADD))
      .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
      .to("cache://TestCache1")
   .end()
   .to("direct:nextPhase");
```

53.4.9. Example 8: Checking for an entry in the Cache

Note: The CHECK command tests existence of an entry in the cache but doesn't place a message in the body.

```
from("direct:start")
   // Prepare headers
   .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_CHECK))
   .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
   .to("cache://TestCache1").
   // Check if entry was not found
   .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
      // If not found, get the payload and put it to cache
      .to("cxf:bean:someHeavyweightOperation").
      .setHeader(CacheConstants.CACHE_OPERATION,
constant(CacheConstants.CACHE_OPERATION_ADD))
      .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
      .to("cache://TestCache1")
   .end();
```

53.5. MANAGEMENT OF EHCACHE

EHCache has its own statistics and management from JMX.

Here's a snippet on how to expose them via JMX in a Spring application context:

```
<bean id="ehCacheManagementService" class="net.sf.ehcache.management.ManagementService"
init-method="init" lazy-init="false">
  <constructor-arg>
```

```
    <bean class="net.sf.ehcache.CacheManager" factory-method="getInstance"/>
  </constructor-arg>
  <constructor-arg>
    <bean class="org.springframework.jmx.support.JmxUtils" factory-method="locateMBeanServer"/>
  </constructor-arg>
  <constructor-arg value="true"/>
  <constructor-arg value="true"/>
  <constructor-arg value="true"/>
  <constructor-arg value="true"/>
</bean>
```

Of course you can do the same thing in straight Java:

```
ManagementService.registerMBeans(CacheManager.getInstance(), mbeanServer, true, true, true,
true);
```

You can get cache hits, misses, in-memory hits, disk hits, size stats this way. You can also change CacheConfiguration parameters on the fly.

53.6. CACHE REPLICATION CAMEL 2.8

The Camel Cache component is able to distribute a cache across server nodes using several different replication mechanisms including: RMI, JGroups, JMS and Cache Server.

There are two different ways to make it work:

1. You can configure **ehcache.xml** manually

OR

2. You can configure these three options:

- cacheManagerFactory
- eventListenerRegistry
- cacheLoaderRegistry

Configuring Camel Cache replication using the first option is a bit of hard work as you have to configure all caches separately. So in a situation when the all names of caches are not known, using **ehcache.xml** is not a good idea.

The second option is much better when you want to use many different caches as you do not need to define options per cache. This is because replication options are set per **CacheManager** and per **CacheEndpoint**. Also it is the only way when cache names are not know at the development phase.

Note: It might be useful to read the EHCache manual to get a better understanding of the Camel Cache replication mechanism.

53.6.1. Example: JMS cache replication

JMS replication is the most powerful and secured replication method. Used together with Camel Cache replication makes it also rather simple. An example is available on a separate page.

CHAPTER 54. CAFFEINE CACHE COMPONENT

Available as of Camel version 2.20

The **caffeine-cache** component enables you to perform caching operations using The simple cache from Caffeine.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-caffeine</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

54.1. URI FORMAT

```
caffeine-cache://cacheName[?options]
```

You can append query options to the URI in the following format, **?option=value&option=#beanRef&...**

54.2. OPTIONS

The Caffeine Cache component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	Sets the global component configuration		CaffeineConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Caffeine Cache endpoint is configured using URI syntax:

```
caffeine-cache:cacheName
```

with the following path and query parameters:

54.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** the cache name		String

54.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
createCacheIfNotExist (common)	Configure if a cache need to be created if it does exist or can't be pre-configured.	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
action (producer)	To configure the default cache action. If an action is set in the message header, then the operation from the header takes precedence.		String
cache (producer)	To configure the default an already instantianted cache to be used		Cache
cacheLoader (producer)	To configure a CacheLoader in case of a LoadCache use		CacheLoader
evictionType (producer)	Set the eviction Type for this cache	SIZE_BASED	EvictionType
expireAfterAccessTime (producer)	Set the expire After Access Time in case of time based Eviction (in seconds)	300	int

Name	Description	Default	Type
expireAfterWriteTime (producer)	Set the expire After Access Write in case of time based Eviction (in seconds)	300	int
initialCapacity (producer)	Set the initial Capacity for the cache	10000	int
key (producer)	To configure the default action key. If a key is set in the message header, then the key from the header takes precedence.		Object
maximumSize (producer)	Set the maximum size for the cache	10000	int
removalListener (producer)	Set a specific removal Listener for the cache		RemovalListener
statsCounter (producer)	Set a specific Stats Counter for the cache stats		StatsCounter
statsEnabled (producer)	To enable stats on the cache	false	boolean
keyType (advanced)	The cache key type, default java.lang.Object	java.lang.Object	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
valueType (advanced)	The cache value type, default java.lang.Object	java.lang.Object	String

CHAPTER 55. CAFFEINE LOADCACHE COMPONENT

Available as of Camel version 2.20

The **caffeine-loadcache** component enables you to perform caching operations using The Load cache from Caffeine.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-caffeine</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

55.1. URI FORMAT

```
caffeine-loadcache://cacheName[?options]
```

You can append query options to the URI in the following format, **?option=value&option=#beanRef&...**

55.2. OPTIONS

The Caffeine LoadCache component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	Sets the global component configuration		CaffeineConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Caffeine LoadCache endpoint is configured using URI syntax:

```
caffeine-loadcache:cacheName
```

with the following path and query parameters:

55.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** the cache name		String

55.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
createCacheIfNotExist (common)	Configure if a cache need to be created if it does exist or can't be pre-configured.	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
action (producer)	To configure the default cache action. If an action is set in the message header, then the operation from the header takes precedence.		String
cache (producer)	To configure the default an already instantianted cache to be used		Cache
cacheLoader (producer)	To configure a CacheLoader in case of a LoadCache use		CacheLoader
evictionType (producer)	Set the eviction Type for this cache	SIZE_BASED	EvictionType
expireAfterAccessTime (producer)	Set the expire After Access Time in case of time based Eviction (in seconds)	300	int

Name	Description	Default	Type
expireAfterWriteTime (producer)	Set the expire After Access Write in case of time based Eviction (in seconds)	300	int
initialCapacity (producer)	Set the initial Capacity for the cache	10000	int
key (producer)	To configure the default action key. If a key is set in the message header, then the key from the header takes precedence.		Object
maximumSize (producer)	Set the maximum size for the cache	10000	int
removalListener (producer)	Set a specific removal Listener for the cache		RemovalListener
statsCounter (producer)	Set a specific Stats Counter for the cache stats		StatsCounter
statsEnabled (producer)	To enable stats on the cache	false	boolean
keyType (advanced)	The cache key type, default java.lang.Object	java.lang.Object	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
valueType (advanced)	The cache value type, default java.lang.Object	java.lang.Object	String

CHAPTER 56. CASTOR DATAFORMAT (DEPRECATED)

Available as of Camel version 2.1

Castor is a Data Format which uses the Castor XML library to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

As usually you can use either Java DSL or Spring XML to work with Castor Data Format.

56.1. USING THE JAVA DSL

```
from("direct:order").
  marshal().castor().
  to("activemq:queue:order");
```

For example the following uses a named DataFormat of Castor which uses default Castor data binding features.

```
CastorDataFormat castor = new CastorDataFormat ();

from("activemq:My.Queue").
  unmarshal(castor).
  to("mqseries:Another.Queue");
```

If you prefer to use a named reference to a data format which can then be defined in your Registry such as via your Spring XML file. e.g.

```
from("activemq:My.Queue").
  unmarshal("mycastorType").
  to("mqseries:Another.Queue");
```

If you want to override default mapping schema by providing a mapping file you can set it as follows.

```
CastorDataFormat castor = new CastorDataFormat ();
castor.setMappingFile("mapping.xml");
```

Also if you want to have more control on Castor Marshaller and Unmarshaller you can access them as below.

```
castor.getMarshaller();
castor.getUnmarshaller();
```

56.2. USING SPRING XML

The following example shows how to use Castor to unmarshal using Spring configuring the castor data type

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <unmarshal>
      <castor validation="true" />
```

```
    </unmarshal>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

This example shows how to configure the data type just once and reuse it on multiple routes. You have to set the <castor> element directly in <camelContext>.

```
<camelContext>
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <castor id="myCastor"/>
  </dataFormats>

  <route>
    <from uri="direct:start"/>
    <marshal ref="myCastor"/>
    <to uri="direct:marshalled"/>
  </route>
  <route>
    <from uri="direct:marshalled"/>
    <unmarshal ref="myCastor"/>
    <to uri="mock:result"/>
  </route>

</camelContext>
```

56.3. OPTIONS

The Castor dataformat supports 9 options which are listed below.

Name	Default	Java Type	Description
mappingFile		**String**	Path to a Castor mapping file to load from the classpath.
whitelistEnabled	**true**	**Boolean**	Define if Whitelist feature is enabled or not
allowedUnmarshallObjects		**String**	Define the allowed objects to be unmarshalled. You can specify the FQN class name of allowed objects, and you can use comma to separate multiple entries. It is also possible to use wildcards and regular expression which is based on the pattern defined by link org.apache.camel.util.EndpointHelpermatchPattern(String, String). Denied objects takes precedence over allowed objects.
deniedUnmarshallObjects		**String**	Define the denied objects to be unmarshalled. You can specify the FQN class name of deined objects, and you can use comma to separate multiple entries. It is also possible to use wildcards and regular expression which is based on the pattern defined by link org.apache.camel.util.EndpointHelpermatchPattern(String, String). Denied objects takes precedence over allowed objects.

Name	Default	Java Type	Description
validation	true	Boolean	Whether validation is turned on or off. Is by default true.
encoding	UTF-8	String	Encoding to use when marshalling an Object to XML. Is by default UTF-8
packages		String[]	Add additional packages to Castor XmlContext
classes		String[]	Add additional class names to Castor XmlContext
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

56.4. DEPENDENCIES

To use Castor in your camel routes you need to add the a dependency on **camel-castor** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-castor</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 57. CAMEL CDI

The Camel CDI component provides auto-configuration for Apache Camel using CDI as dependency injection framework based on *convention-over-configuration*. It auto-detects Camel routes available in the application and provides beans for common Camel primitives like **Endpoint**, **FluentProducerTemplate**, **ProducerTemplate** or **TypeConverter**. It implements standard Camel bean integration so that Camel annotations like **@Consume**, **@Produce** and **@PropertyInject** can be used seamlessly in CDI beans. Besides, it bridges Camel events (e.g. **RouteAddedEvent**, **CamelContextStartedEvent**, **ExchangeCompletedEvent**, ...) as CDI events and provides a CDI events endpoint that can be used to consume / produce CDI events from / to Camel routes.

While the Camel CDI component is available as of **Camel 2.10**, it's been rewritten in **Camel 2.17** to better fit into the CDI programming model. Hence some of the features like the Camel events to CDI events bridge and the CDI events endpoint only apply starting Camel 2.17.

More details on how to test Camel CDI applications are available in Camel CDI testing.

CAUTION

camel-cdi is deprecated in OSGi and not supported. Use OSGi Blueprint if using Camel with OSGi.

57.1. AUTO-CONFIGURED CAMEL CONTEXT

Camel CDI automatically deploys and configures a **CamelContext** bean. That **CamelContext** bean is automatically instantiated, configured and started (resp. stopped) when the CDI container initializes (resp. shuts down). It can be injected in the application, e.g.:

```
@Inject
CamelContext context;
```

That default **CamelContext** bean is qualified with the built-in **@Default** qualifier, is scoped **@ApplicationScoped** and is of type **DefaultCamelContext**.

Note that this bean can be customized programmatically and other Camel context beans can be deployed in the application as well.

57.2. AUTO-DETECTING CAMEL ROUTES

Camel CDI automatically collects all the **RoutesBuilder** beans in the application, instantiates and add them to the **CamelContext** bean instance when the CDI container initializes. For example, adding a Camel route is as simple as declaring a class, e.g.:

```
class MyRouteBean extends RouteBuilder {

    @Override
    public void configure() {
        from("jms:invoices").to("file:/invoices");
    }
}
```

Note that you can declare as many **RoutesBuilder** beans as you want. Besides, **RouteContainer** beans are also automatically collected, instantiated and added to the **CamelContext** bean instance managed by Camel CDI when the container initializes.

Available as of Camel 2.19

In some situations, it may be necessary to disable the auto-configuration of the **RouteBuilder** and **RouteContainer** beans. That can be achieved by observing for the **CdiCamelConfiguration** event, e.g.:

```
static void configuration(@Observes CdiCamelConfiguration configuration) {
    configuration.autoConfigureRoutes(false);
}
```

Similarly, it is possible to deactivate the automatic starting of the configured **CamelContext** beans, e.g.:

```
static void configuration(@Observes CdiCamelConfiguration configuration) {
    configuration.autoStartContexts(false);
}
```

57.3. AUTO-CONFIGURED CAMEL PRIMITIVES

Camel CDI provides beans for common Camel primitives that can be injected in any CDI beans, e.g.:

```
@Inject
@Uri("direct:inbound")
ProducerTemplate producerTemplate;

@Inject
@Uri("direct:inbound")
FluentProducerTemplate fluentProducerTemplate;

@Inject
MockEndpoint outbound; // URI defaults to the member name, i.e. mock:outbound

@Inject
@Uri("direct:inbound")
Endpoint endpoint;

@Inject
TypeConverter converter;
```

57.4. CAMEL CONTEXT CONFIGURATION

If you just want to change the name of the default **CamelContext** bean, you can used the **@ContextName** qualifier provided by Camel CDI, e.g.:

```
@ContextName("camel-context")
class MyRouteBean extends RouteBuilder {

    @Override
    public void configure() {
        from("jms:invoices").to("file:/invoices");
    }
}
```

Else, if more customization is needed, any **CamelContext** class can be used to declare a custom Camel context bean. Then, the **@PostConstruct** and **@PreDestroy** lifecycle callbacks can be done to do the customization, e.g.:

```
@ApplicationScoped
class CustomCamelContext extends DefaultCamelContext {

    @PostConstruct
    void customize() {
        // Set the Camel context name
        setName("custom");
        // Disable JMX
        disableJMX();
    }

    @PreDestroy
    void cleanUp() {
        // ...
    }
}
```

Producer and disposer methods can also be used as well to customize the Camel context bean, e.g.:

```
class CamelContextFactory {

    @Produces
    @ApplicationScoped
    CamelContext customize() {
        DefaultCamelContext context = new DefaultCamelContext();
        context.setName("custom");
        return context;
    }

    void cleanUp(@Disposes CamelContext context) {
        // ...
    }
}
```

Similarly, producer fields can be used, e.g.:

```
@Produces
@ApplicationScoped
CamelContext context = new CustomCamelContext();

class CustomCamelContext extends DefaultCamelContext {

    CustomCamelContext() {
        setName("custom");
    }
}
```

This pattern can be used for example to avoid having the Camel context routes started automatically when the container initializes by calling the **setAutoStartup** method, e.g.:

```
@ApplicationScoped
```

```
class ManualStartupCamelContext extends DefaultCamelContext {

    @PostConstruct
    void manual() {
        setAutoStartup(false);
    }
}
```

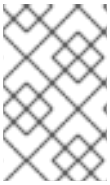
57.5. MULTIPLE CAMEL CONTEXTS

Any number of **CamelContext** beans can actually be declared in the application as documented above. In that case, the CDI qualifiers declared on these **CamelContext** beans are used to bind the Camel routes and other Camel primitives to the corresponding Camel contexts. From example, if the following beans get declared:

```
@ApplicationScoped
@ContextName("foo")
class FooCamelContext extends DefaultCamelContext {
}

@ApplicationScoped
@BarContextQualifier
class BarCamelContext extends DefaultCamelContext {
}

@ContextName("foo")
class RouteAddedToFooCamelContext extends RouteBuilder {

    @Override
    public void configure() {
        // ...
    }
}

@BarContextQualifier
class RouteAddedToBarCamelContext extends RouteBuilder {

    @Override
    public void configure() {
        // ...
    }
}

@ContextName("baz")
class RouteAddedToBazCamelContext extends RouteBuilder {

    @Override
    public void configure() {
        // ...
    }
}

@MyOtherQualifier
class RouteNotAddedToAnyCamelContext extends RouteBuilder {
```

```
    @Override
    public void configure() {
        // ...
    }
}
```

The **RoutesBuilder** beans qualified with **@ContextName** are automatically added to the corresponding **CamelContext** beans by Camel CDI. If no such **CamelContext** bean exists, it gets automatically created, as for the **RouteAddedToBazCamelContext** bean. Note this only happens for the **@ContextName** qualifier provided by Camel CDI. Hence the **RouteNotAddedToAnyCamelContext** bean qualified with the user-defined **@MyOtherQualifier** qualifier does not get added to any Camel contexts. That may be useful, for example, for Camel routes that may be required to be added later during the application execution.



NOTE

Since Camel version 2.17.0, Camel CDI is capable of managing any kind of **CamelContext** beans (e.g. **DefaultCamelContext**). In previous versions, it is only capable of managing beans of type **CdiCamelContext** so it is required to extend it.

The CDI qualifiers declared on the **CamelContext** beans are also used to bind the corresponding Camel primitives, e.g.:

```
@Inject
@ContextName("foo")
@Uri("direct:inbound")
ProducerTemplate producerTemplate;

@Inject
@ContextName("foo")
@Uri("direct:inbound")
FluentProducerTemplate fluentProducerTemplate;

@Inject
@BarContextQualifier
MockEndpoint outbound; // URI defaults to the member name, i.e. mock:outbound

@Inject
@ContextName("baz")
@Uri("direct:inbound")
Endpoint endpoint;
```

57.6. CONFIGURATION PROPERTIES

To configure the sourcing of the configuration properties used by Camel to resolve properties placeholders, you can declare a **PropertiesComponent** bean qualified with **@Named("properties")**, e.g.:

```
@Produces
@ApplicationScoped
@Named("properties")
PropertiesComponent propertiesComponent() {
    Properties properties = new Properties();
    properties.put("property", "value");
```

```
    PropertiesComponent component = new PropertiesComponent();
    component.setInitialProperties(properties);
    component.setLocation("classpath:placeholder.properties");
    return component;
}
```

If you want to use DeltaSpike configuration mechanism you can declare the following **PropertiesComponent** bean:

```
@Produces
@ApplicationScoped
@Named("properties")
PropertiesComponent properties(PropertiesParser parser) {
    PropertiesComponent component = new PropertiesComponent();
    component.setPropertiesParser(parser);
    return component;
}

// PropertiesParser bean that uses DeltaSpike to resolve properties
static class DeltaSpikeParser extends DefaultPropertiesParser {
    @Override
    public String parseProperty(String key, String value, Properties properties) {
        return ConfigResolver.getPropertyValue(key);
    }
}
```

You can see the **camel-example-cdi-properties** example for a working example of a Camel CDI application using DeltaSpike configuration mechanism.

57.7. AUTO-CONFIGURED TYPE CONVERTERS

CDI beans annotated with the **@Converter** annotation are automatically registered into the deployed Camel contexts, e.g.:

```
@Converter
public class MyTypeConverter {

    @Converter
    public Output convert(Input input) {
        //...
    }
}
```

Note that CDI injection is supported within the type converters.

57.8. CAMEL BEAN INTEGRATION

57.8.1. Camel annotations

As part of the Camel bean integration, Camel comes with a set of annotations that are seamlessly supported by Camel CDI. So you can use any of these annotations in your CDI beans, e.g.:

	Camel annotation	CDI equivalent

Configuration property	`@PropertyInject("key")`
	`String value;`
		If using DeltaSpike configuration mechanism:
		`@Inject`
		`@ConfigProperty(name = "key")`
		`String value;`
		See configuration properties for more details.

Producer template injection (default Camel context)	`@Produce(uri = "mock:outbound")`
	`ProducerTemplate producer;`
	
	`@Produce(uri = "mock:outbound")`
	`FluentProducerTemplate producer;`
		`@Inject`
		`@Uri("direct:outbound")`
		`ProducerTemplate producer;`
		
		`@Produce(uri = "direct:outbound")`
		`FluentProducerTemplate producer;`

Endpoint injection (default Camel context)	`@EndpointInject(uri = "direct:inbound")`
	`Endpoint endpoint;`
		`@Inject`
		`@Uri("direct:inbound")`
		`Endpoint endpoint;`

Endpoint injection (Camel context by name)	`@EndpointInject(uri = "direct:inbound",`
	`context = "foo")`
	`Endpoint contextEndpoint;`
		`@Inject`
		`@ContextName("foo")`
		`@Uri("direct:inbound")`
		`Endpoint contextEndpoint;`

Bean injection (by type)	`@BeanInject` `MyBean bean;`	`@Inject` `MyBean bean;`

Bean injection (by name)	`@BeanInject("foo")`
	`MyBean bean;`
		`@Inject`
		`@Named("foo")`
		`MyBean bean;`

POJO consuming	`@Consume(uri = "seda:inbound")`
	`void consume(@Body String body) {`
	`//...`
	`}`	

57.8.2. Bean component

You can refer to CDI beans, either by type or name, From the Camel DSL, e.g. with the Java Camel DSL:

```
class MyBean {
    //...
}

from("direct:inbound").bean(MyBean.class);
```

Or to lookup a CDI bean by name from the Java DSL:

```
@Named("foo")
class MyNamedBean {
    //...
}

from("direct:inbound").bean("foo");
```

57.8.3. Referring beans from Endpoint URIs

When configuring endpoints using the URI syntax you can refer to beans in the Registry using the **#** notation. If the URI parameter value starts with a **#** sign then Camel CDI will lookup for a bean of the given type by name, e.g.:

```
from("jms:queue:{{destination}}?
transacted=true&transactionManager=#jtaTransactionManager").to("...");
```

Having the following CDI bean qualified with **@Named("jtaTransactionManager")**:

```
@Produces
@Named("jtaTransactionManager")
PlatformTransactionManager createTransactionManager(TransactionManager transactionManager,
UserTransaction userTransaction) {
    JtaTransactionManager jtaTransactionManager = new JtaTransactionManager();
    jtaTransactionManager.setUserTransaction(userTransaction);
    jtaTransactionManager.setTransactionManager(transactionManager);
    jtaTransactionManager.afterPropertiesSet();
    return jtaTransactionManager;
}
```

57.9. CAMEL EVENTS TO CDI EVENTS

Available as of Camel 2.17

Camel provides a set of management events that can be subscribed to for listening to Camel context, service, route and exchange events. Camel CDI seamlessly translates these Camel events into CDI events that can be observed using CDI observer methods, e.g.:

```
void onContextStarting(@Observes CamelContextStartingEvent event) {
    // Called before the default Camel context is about to start
}
```

As of Camel 2.18, it is possible to observe events for a particular route (**RouteAddedEvent**, **RouteStartedEvent**, **RouteStoppedEvent** and **RouteRemovedEvent**) should it have an explicit defined, e.g.:

```
from("...").routeId("foo").to("...");

void onRouteStarted(@Observes @Named("foo") RouteStartedEvent event) {
   // Called after the route "foo" has started
}
```

When multiple Camel contexts exist in the CDI container, the Camel context bean qualifiers, like **@ContextName**, can be used to refine the observer method resolution to a particular Camel context as specified in observer resolution, e.g.:

```
void onRouteStarted(@Observes @ContextName("foo") RouteStartedEvent event) {
   // Called after the route 'event.getRoute()' for the Camel context 'foo' has started
}

void onContextStarted(@Observes @Manual CamelContextStartedEvent event) {
   // Called after the the Camel context qualified with '@Manual' has started
}
```

Similarly, the **@Default** qualifier can be used to observe Camel events for the *default* Camel context if multiples contexts exist, e.g.:

```
void onExchangeCompleted(@Observes @Default ExchangeCompletedEvent event) {
   // Called after the exchange 'event.getExchange()' processing has completed
}
```

In that example, if no qualifier is specified, the **@Any** qualifier is implicitly assumed, so that corresponding events for all the Camel contexts get received.

Note that the support for Camel events translation into CDI events is only activated if observer methods listening for Camel events are detected in the deployment, and that per Camel context.

57.10. CDI EVENTS ENDPOINT

Available as of Camel 2.17

The CDI event endpoint bridges the CDI events with the Camel routes so that CDI events can be seamlessly observed / consumed (resp. produced / fired) from Camel consumers (resp. by Camel producers).

The **CdiEventEndpoint<T>** bean provided by Camel CDI can be used to observe / consume CDI events whose *event type* is **T**, for example:

```
@Inject
CdiEventEndpoint<String> cdiEventEndpoint;

from(cdiEventEndpoint).log("CDI event received: ${body}");
```

This is equivalent to writing:

```
@Inject
```

```
@Uri("direct:event")
ProducerTemplate producer;

void observeCdiEvents(@Observes String event) {
    producer.sendBody(event);
}

from("direct:event").log("CDI event received: ${body}");
```

Conversely, the **CdiEventEndpoint<T>** bean can be used to produce / fire CDI events whose *event type* is **T**, for example:

```
@Inject
CdiEventEndpoint<String> cdiEventEndpoint;

from("direct:event").to(cdiEventEndpoint).log("CDI event sent: ${body}");
```

This is equivalent to writing:

```
@Inject
Event<String> event;

from("direct:event").process(new Processor() {
    @Override
    public void process(Exchange exchange) {
        event.fire(exchange.getBody(String.class));
    }
}).log("CDI event sent: ${body}");
```

Or using a Java 8 lambda expression:

```
@Inject
Event<String> event;

from("direct:event")
    .process(exchange -> event.fire(exchange.getIn().getBody(String.class)))
    .log("CDI event sent: ${body}");
```

The type variable **T** (resp. the qualifiers) of a particular **CdiEventEndpoint<T>** injection point are automatically translated into the parameterized *event type* (resp. into the *event qualifiers*) e.g.:

```
@Inject
@FooQualifier
CdiEventEndpoint<List<String>> cdiEventEndpoint;

from("direct:event").to(cdiEventEndpoint);

void observeCdiEvents(@Observes @FooQualifier List<String> event) {
    logger.info("CDI event: {}", event);
}
```

When multiple Camel contexts exist in the CDI container, the Camel context bean qualifiers, like **@ContextName**, can be used to qualify the **CdiEventEndpoint<T>** injection points, e.g.:

```
@Inject
@ContextName("foo")
CdiEventEndpoint<List<String>> cdiEventEndpoint;
// Only observes / consumes events having the @ContextName("foo") qualifier
from(cdiEventEndpoint).log("Camel context (foo) > CDI event received: ${body}");
// Produces / fires events with the @ContextName("foo") qualifier
from("...").to(cdiEventEndpoint);

void observeCdiEvents(@Observes @ContextName("foo") List<String> event) {
    logger.info("Camel context (foo) > CDI event: {}", event);
}
```

Note that the CDI event Camel endpoint dynamically adds an observer method for each unique combination of *event type* and *event qualifiers* and solely relies on the container typesafe observer resolution, which leads to an implementation as efficient as possible.

Besides, as the impedance between the *typesafe* nature of CDI and the *dynamic* nature of the Camel component model is quite high, it is not possible to create an instance of the CDI event Camel endpoint via URIs. Indeed, the URI format for the CDI event component is:

```
cdi-event://PayloadType<T1,...,Tn>[?qualifiers=QualifierType1[,...[,QualifierTypeN]...]]
```

With the authority **PayloadType** (resp. the **QualifierType**) being the URI escaped fully qualified name of the payload (resp. qualifier) raw type followed by the type parameters section delimited by angle brackets for payload parameterized type. Which leads to *unfriendly* URIs, e.g.:

```
cdi-event://org.apache.camel.cdi.example.EventPayload%3Cjava.lang.Integer%3E?
qualifiers=org.apache.camel.cdi.example.FooQualifier%2Corg.apache.camel.cdi.example.BarQualifier
```

But more fundamentally, that would prevent efficient binding between the endpoint instances and the observer methods as the CDI container doesn't have any ways of discovering the Camel context model during the deployment phase.

57.11. CAMEL XML CONFIGURATION IMPORT

Available as of Camel 2.18

While CDI favors a typesafe dependency injection mechanism, it may be useful to reuse existing Camel XML configuration files into a Camel CDI application. In other use cases, it might be handy to rely on the Camel XML DSL to configure its Camel context(s).

You can use the **@ImportResource** annotation that's provided by Camel CDI on any CDI beans and Camel CDI will automatically load the Camel XML configuration at the specified locations, e.g.:

```
@ImportResource("camel-context.xml")
class MyBean {
}
```

Camel CDI will load the resources at the specified locations from the classpath (other protocols may be added in the future).

Every **CamelContext** elements and other Camel *primitives* from the imported resources are automatically deployed as CDI beans during the container bootstrap so that they benefit from the auto-configuration provided by Camel CDI and become available for injection at runtime. If such an element

has an explicit **id** attribute set, the corresponding CDI bean is qualified with the **@Named** qualifier, e.g., given the following Camel XML configuration:

```
<camelContext id="foo">
  <endpoint id="bar" uri="seda:inbound">
    <property key="queue" value="#queue"/>
    <property key="concurrentConsumers" value="10"/>
  </endpoint>
<camelContext/>
```

The corresponding CDI beans are automatically deployed and can be injected, e.g.:

```
@Inject
@ContextName("foo")
CamelContext context;

@Inject
@Named("bar")
Endpoint endpoint;
```

Note that the **CamelContext** beans are automatically qualified with both the **@Named** and **@ContextName** qualifiers. If the imported **CamelContext** element doesn't have an **id** attribute, the corresponding bean is deployed with the built-in **@Default** qualifier.

Conversely, CDI beans deployed in the application can be referred to from the Camel XML configuration, usually using the **ref** attribute, e.g., given the following bean declared:

```
@Produces
@Named("baz")
Processor processor = exchange -> exchange.getIn().setHeader("qux", "quux");
```

A reference to that bean can be declared in the imported Camel XML configuration, e.g.:

```
<camelContext id="foo">
  <route>
    <from uri="..."/>
    <process ref="baz"/>
  </route>
<camelContext/>
```

57.12. TRANSACTION SUPPORT

Available as of Camel 2.19

Camel CDI provides support for Camel transactional client using JTA.

That support is optional hence you need to have JTA in your application classpath, e.g., by explicitly add JTA as a dependency when using Maven:

```
<dependency>
  <groupId>javax.transaction</groupId>
  <artifactId>javax.transaction-api</artifactId>
  <scope>runtime</scope>
</dependency>
```

You'll have to have your application deployed in a JTA capable container or provide a standalone JTA implementation.

CAUTION

Note that, for the time being, the transaction manager is looked up as JNDI resource with the **java:/TransactionManager** key.

More flexible strategies will be added in the future to support a wider range of deployment scenarios.

57.12.1. Transaction policies

Camel CDI provides implementation for the typically supported Camel **TransactedPolicy** as CDI beans. It is possible to have these policies looked up by name using the transacted EIP, e.g.:

```
class MyRouteBean extends RouteBuilder {

    @Override
    public void configure() {
        from("activemq:queue:foo")
            .transacted("PROPAGATION_REQUIRED")
            .bean("transformer")
            .to("jpa:my.application.entity.Bar")
            .log("${body.id} inserted");
    }
}
```

This would be equivalent to:

```
class MyRouteBean extends RouteBuilder {

    @Inject
    @Named("PROPAGATION_REQUIRED")
    Policy required;

    @Override
    public void configure() {
        from("activemq:queue:foo")
            .policy(required)
            .bean("transformer")
            .to("jpa:my.application.entity.Bar")
            .log("${body.id} inserted");
    }
}
```

The list of supported transaction policy names is:

- **PROPAGATION_NEVER**,
- **PROPAGATION_NOT_SUPPORTED**,
- **PROPAGATION_SUPPORTS**,
- **PROPAGATION_REQUIRED**,

- **PROPAGATION_REQUIRES_NEW**,
- **PROPAGATION_NESTED**,
- **PROPAGATION_MANDATORY**.

57.12.2. Transactional error handler

Camel CDI provides a transactional error handler that extends the redelivery error handler, forces a rollback whenever an exception occurs and creates a new transaction for each redelivery.

Camel CDI provides the **CdiRouteBuilder** class that exposes the **transactionErrorHandler** helper method to enable quick access to the configuration, e.g.:

```
class MyRouteBean extends CdiRouteBuilder {

    @Override
    public void configure() {
        errorHandler(transactionErrorHandler()
            .setTransactionPolicy("PROPAGATION_SUPPORTS")
            .maximumRedeliveries(5)
            .maximumRedeliveryDelay(5000)
            .collisionAvoidancePercent(10)
            .backOffMultiplier(1.5));
    }
}
```

57.13. AUTO-CONFIGURED OSGI INTEGRATION

Available as of Camel 2.17

The Camel context beans are automatically adapted by Camel CDI so that they are registered as OSGi services and the various resolvers (like **ComponentResolver** and **DataFormatResolver**) integrate with the OSGi registry. That means that the Karaf Camel commands can be used to operate the Camel contexts auto-configured by Camel CDI, e.g.:

```
karaf@root()> camel:context-list
 Context       Status            Total #       Failed #     Inflight #   Uptime
 -------       ------            -------       --------     ----------   ------
 camel-cdi     Started                 1              0              0   1 minute
```

See the **camel-example-cdi-osgi** example for a working example of the Camel CDI OSGi integration.

57.14. LAZY INJECTION / PROGRAMMATIC LOOKUP

While the CDI programmatic model favors a typesafe resolution mechanism that occurs at application initialization time, it is possible to perform dynamic / lazy injection later during the application execution using the programmatic lookup mechanism.

Camel CDI provides for convenience the annotation literals corresponding to the CDI qualifiers that you can use for standard injection of Camel primitives. These annotation literals can be used in conjunction with the **javax.enterprise.inject.Instance** interface which is the CDI entry point to perform lazy injection / programmatic lookup.

For example, you can use the provided annotation literal for the **@Uri** qualifier to lazily lookup for Camel primitives, e.g. for **ProducerTemplate** beans:

```
@Any
@Inject
Instance<ProducerTemplate> producers;

ProducerTemplate inbound = producers
    .select(Uri.Literal.of("direct:inbound"))
    .get();
```

Or for **Endpoint** beans, e.g.:

```
@Any
@Inject
Instance<Endpoint> endpoints;

MockEndpoint outbound = endpoints
    .select(MockEndpoint.class, Uri.Literal.of("mock:outbound"))
    .get();
```

Similarly, you can use the provided annotation literal for the **@ContextName** qualifier to lazily lookup for **CamelContext** beans, e.g.:

```
@Any
@Inject
Instance<CamelContext> contexts;

CamelContext context = contexts
    .select(ContextName.Literal.of("foo"))
    .get();
```

You can also refined the selection based on the Camel context type, e.g.:

```
@Any
@Inject
Instance<CamelContext> contexts;

// Refine the selection by type
Instance<DefaultCamelContext> context = contexts.select(DefaultCamelContext.class);

// Check if such a bean exists then retrieve a reference
if (!context.isUnsatisfied())
    context.get();
```

Or even iterate over a selection of Camel contexts, e.g.:

```
@Any
@Inject
Instance<CamelContext> contexts;

for (CamelContext context : contexts)
    context.setUseBreadcrumb(true);
```

57.15. MAVEN ARCHETYPE

Among the available Camel Maven archetypes, you can use the provided **camel-archetype-cdi** to generate a Camel CDI Maven project, e.g.:

```
mvn archetype:generate -DarchetypeGroupId=org.apache.camel.archetypes -
DarchetypeArtifactId=camel-archetype-cdi
```

57.16. SUPPORTED CONTAINERS

The Camel CDI component is compatible with any CDI 1.0, CDI 1.1 and CDI 1.2 compliant runtime. It's been successfully tested against the following runtimes:

Container	Version	Runtime
Weld SE	**1.1.28.Final**	CDI 1.0 / Java SE 7
OpenWebBeans	**1.2.7**	CDI 1.0 / Java SE 7
Weld SE	**2.4.2.Final**	CDI 1.2 / Java SE 7
OpenWebBeans	**1.7.2**	CDI 1.2 / Java SE 7
WildFly	**8.2.1.Final**	CDI 1.2 / Java EE 7
WildFly	**9.0.1.Final**	CDI 1.2 / Java EE 7
WildFly	**10.1.0.Final**	CDI 1.2 / Java EE 7

57.17. EXAMPLES

The following examples are available in the **examples** directory of the Camel project:

Example	Description
camel-example-cdi	Illustrates how to work with Camel using CDI to configure components, endpoints and beans
camel-example-cdi-kubernetes	Illustrates the integration between Camel, CDI and Kubernetes
camel-example-cdi-metrics	Illustrates the integration between Camel, Dropwizard Metrics and CDI
camel-example-cdi-properties	Illustrates the integration between Camel, DeltaSpike and CDI for configuration properties

Example	Description
camel-example-cdi-osgi	A CDI application using the SJMS component that can be executed inside an OSGi container using PAX CDI
camel-example-cdi-rest-servlet	Illustrates the Camel REST DSL being used in a Web application that uses CDI as dependency injection framework
camel-example-cdi-test	Demonstrates the testing features that are provided as part of the integration between Camel and CDI
camel-example-cdi-xml	Illustrates the use of Camel XML configuration files into a Camel CDI application
camel-example-swagger-cdi	An example using REST DSL and Swagger Java with CDI
camel-example-widget-gadget-cdi	The Widget and Gadget use-case from the EIP book implemented in Java with CDI dependency Injection

57.18. SEE ALSO

- Camel CDI testing
- CDI specification Web site
- CDI ecosystem
- Weld home page
- OpenWebBeans home page
- Going further with CDI and Camel (See Camel CDI section)

57.19. CAMEL CDI FOR EAR DEPLOYMENTS ON {WILDFLY-CAMEL}

Camel CDI EAR deployments on {wildfly-camel} have some differences in class and resource loading behaviour, compared to standard WAR or JAR deployments.

{wildfly} bootstraps Weld using the EAR deployment ClassLoader. {wildfly} also mandates that only a single CDI extension is created and shared by all EAR sub-deployments.

This results in the 'Auto-configured' CDI Camel Context using the EAR deployment ClassLoader to dynamically load classes and resources. By default, this ClassLoader does not have access to resources within EAR sub-deployments.

For EAR deployments, it is recommended that usage of the 'Auto-configured' CDI Camel Context is avoided and that **RouteBuilder** classes are annotated with **@ContextName**, or that a **CamelContext** is created via the **@ImportResource** annotation or through CDI producer methods and fields. This helps {wildfly-camel} to determine the correct ClassLoader to use with Camel.

CHAPTER 58. CHRONICLE ENGINE COMPONENT

Available as of Camel version 2.18

The camel chronicle-engine component let you leverage the power of OpenHFT's Chronicle-Engine

58.1. URI FORMAT

```
chronicle-engine:addresses/path[?options]
```

58.2. URI OPTIONS

The Chronicle Engine component has no options.

The Chronicle Engine endpoint is configured using URI syntax:

```
chronicle-engine:addresses/path
```

with the following path and query parameters:

58.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
addresses	**Required** Engine addresses. Multiple addresses can be separated by comma.		String
path	**Required** Engine path		String

58.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type
action (common)	The default action to perform, valid values are: - PUBLISH - PPUBLISH_AND_INDEX - PPUT - PGET_AND_PUT - PPUT_ALL - PPUT_IF_ABSENT - PGET - PGET_AND_REMOVE - PREMOVE - PIS_EMPTY - PSIZE		String
clusterName (common)	Cluster name for queue		String
filteredMapEvents (common)	A comma separated list of Map event type to filer, valid values are: INSERT, UPDATE, REMOVE.		String

Name	Description	Default	Type
persistent (common)	Enable/disable data persistence	true	boolean
subscribeMapEvents (common)	Set if consumer should subscribe to Map events, default true.	true	boolean
subscribeTopicEvents (common)	Set if consumer should subscribe to TopicEvents,default false.	false	boolean
subscribeTopologicalEvents (common)	Set if consumer should subscribe to TopologicalEvents,default false.	false	boolean
wireType (common)	The Wire type to use, default to binary wire.	BINARY	String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 59. CHUNK COMPONENT

Available as of Camel version 2.15

The **chunk:** component allows for processing a message using a Chunk template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-chunk</artifactId>
<version>x.x.x</version> <!-- use the same version as your Camel core version -->
</dependency>
```

59.1. URI FORMAT

```
chunk:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke.

You can append query options to the URI in the following format, **?option=value&option=value&...**

59.2. OPTIONS

The Chunk component has no options.

The Chunk endpoint is configured using URI syntax:

```
chunk:resourceUri
```

with the following path and query parameters:

59.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

59.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
encoding (producer)	Define the encoding of the body		String
extension (producer)	Define the file extension of the template		String
themeFolder (producer)	Define the themes folder to scan		String
themeLayer (producer)	Define the theme layer to elaborate		String
themeSubfolder (producer)	Define the themes subfolder to scan		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Chunk component will look for a specific template in *themes* folder with extensions *.chtml* or _.cxml. _If you need to specify a different folder or extensions, you will need to use the specific options listed above.

59.3. CHUNK CONTEXT

Camel will provide exchange information in the Chunk context (just a **Map**). The **Exchange** is transferred as:

key	value
exchange	The **Exchange** itself.
exchange.properties	The **Exchange** properties.
headers	The headers of the In message.
camelContext	The Camel Context.
request	The In message.

key	value
body	The In message body.
response	The Out message (only for InOut message exchange pattern).

59.4. DYNAMIC TEMPLATES

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description	Support Version
ChunkConstants.CHUNK_RESOURCE_URI	String	A URI for the template resource to use instead of the endpoint configured.	
ChunkConstants.CHUNK_TEMPLATE	String	The template to use instead of the endpoint configured.	

59.5. SAMPLES

For example you could use something like:

```
from("activemq:My.Queue").
to("chunk:template");
```

To use a Chunk template to formulate a response for a message for InOut message exchanges (where there is a **JMSReplyTo** header).

If you want to use InOnly and consume the message and send it to another destination you could use:

```
from("activemq:My.Queue").
to("chunk:template").
to("activemq:Another.Queue");
```

It's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
setHeader(ChunkConstants.CHUNK_RESOURCE_URI).constant("template").
to("chunk:dummy");
```

An example of Chunk component options use:

```
from("direct:in").
to("chunk:file_example?themeFolder=template&themeSubfolder=subfolder&extension=chunk");
```

In this example Chunk component will look for the file *file_example.chunk* in the folder *template/subfolder.*

59.6. THE EMAIL SAMPLE

In this sample we want to use Chunk templating for an order confirmation email. The email template is laid out in Chunk as:

```
Dear {$headers.lastName}, {$headers.firstName}

Thanks for the order of {$headers.item}.

Regards Camel Riders Bookstore
{$body}
```

59.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 60. CLASS COMPONENT

Available as of Camel version 2.4

The **class:** component binds beans to Camel message exchanges. It works in the same way as the Bean component but instead of looking up beans from a Registry it creates the bean based on the class name.

60.1. URI FORMAT

```
class:className[?options]
```

Where **className** is the fully qualified class name to create and use as bean.

60.2. OPTIONS

The Class component has no options.

The Class endpoint is configured using URI syntax:

```
class:beanName
```

with the following path and query parameters:

60.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
beanName	**Required** Sets the name of the bean to invoke		String

60.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
method (producer)	Sets the name of the method to invoke on the bean		String
cache (advanced)	If enabled, Camel will cache the result of the first Registry look-up. Cache can be enabled if the bean in the Registry is defined as a singleton scope.	false	boolean
multiParameterArray (advanced)	**Deprecated** How to treat the parameters which are passed from the message body; if it is true, the message body should be an array of parameters. Note: This option is used internally by Camel, and is not intended for end users to use. Deprecation note: This option is used internally by Camel, and is not intended for end users to use.	false	boolean

Name	Description	Default	Type
parameters (advanced)	Used for configuring additional properties on the bean		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

60.3. USING

You simply use the **class** component just as the Bean component but by specifying the fully qualified classname instead.
For example to use the **MyFooBean** you have to do as follows:

```
from("direct:start").to("class:org.apache.camel.component.bean.MyFooBean").to("mock:result");
```

You can also specify which method to invoke on the **MyFooBean**, for example **hello**:

```
from("direct:start").to("class:org.apache.camel.component.bean.MyFooBean?
method=hello").to("mock:result");
```

60.4. SETTING PROPERTIES ON THE CREATED INSTANCE

In the endpoint uri you can specify properties to set on the created instance, for example if it has a **setPrefix** method:

```
// Camel 2.17 onwards
from("direct:start")
    .to("class:org.apache.camel.component.bean.MyPrefixBean?bean.prefix=Bye")
    .to("mock:result");

// Camel 2.16 and older
from("direct:start")
    .to("class:org.apache.camel.component.bean.MyPrefixBean?prefix=Bye")
    .to("mock:result");
```

And you can also use the **#** syntax to refer to properties to be looked up in the Registry.

```
// Camel 2.17 onwards
from("direct:start")
    .to("class:org.apache.camel.component.bean.MyPrefixBean?bean.cool=#foo")
    .to("mock:result");

// Camel 2.16 and older
from("direct:start")
    .to("class:org.apache.camel.component.bean.MyPrefixBean?cool=#foo")
    .to("mock:result");
```

Which will lookup a bean from the Registry with the id **foo** and invoke the **setCool** method on the created instance of the **MyPrefixBean** class.

TIP:See more details at the Bean component as the **class** component works in much the same way.

60.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Bean
- Bean Binding
- Bean Integration

CHAPTER 61. CMIS COMPONENT

Available as of Camel version 2.11

The cmis component uses the Apache Chemistry client API and allows you to add/read nodes to/from a CMIS compliant content repositories.

61.1. URI FORMAT

```
cmis://cmisServerUrl[?options]
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

61.2. CMIS OPTIONS

The CMIS component supports 2 options which are listed below.

Name	Description	Default	Type
sessionFacadeFactory (common)	To use a custom CMISSessionFacadeFactory to create the CMISSessionFacade instances		CMISSessionFacade Factory
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The CMIS endpoint is configured using URI syntax:

```
cmis:cmsUrl
```

with the following path and query parameters:

61.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cmsUrl	**Required** URL to the cmis repository		String

61.2.2. Query Parameters (13 parameters):

Name	Description	Default	Type
pageSize (common)	Number of nodes to retrieve per page	100	int
readContent (common)	If set to true, the content of document node will be retrieved in addition to the properties	false	boolean
readCount (common)	Max number of nodes to read		int
repositoryId (common)	The Id of the repository to use. If not specified the first available repository is used		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
query (consumer)	The cmis query to execute against the repository. If not specified, the consumer will retrieve every node from the content repository by iterating the content tree recursively		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
queryMode (producer)	If true, will execute the cmis query from the message body and return result, otherwise will create a node in the cmis repository	false	boolean
sessionFacadeFactory (advanced)	To use a custom CMISSessionFacadeFactory to create the CMISSessionFacade instances		CMISSessionFacade Factory
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
password (security)	Password for the cmis repository		String
username (security)	Username for the cmis repository		String

61.3. USAGE

61.3.1. Message headers evaluated by the producer

Header	Default Value	Description
CamelCMISFolderPath	/	The current folder to use during the execution. If not specified will use the root folder
CamelCMISRetrieveContent	**false**	In **queryMode** this header will force the producer to retrieve the content of document nodes.
CamelCMISReadSize	**0**	Max number of nodes to read.
cmis:path	**null**	If **CamelCMISFolderPath** is not set, will try to find out the path of the node from this cmis property and it is name
cmis:name	**null**	If **CamelCMISFolderPath** is not set, will try to find out the path of the node from this cmis property and it is path
cmis:objectTypeId	**null**	The type of the node
cmis:contentStreamMimeType	**null**	The mimetype to set for a document

61.3.2. Message headers set during querying Producer operation

Header	Type	Description
CamelCMISResultCount	**Integer**	Number of nodes returned from the query.

The message body will contain a list of maps, where each entry in the map is cmis property and its value. If **CamelCMISRetrieveContent** header is set to true, one additional entry in the map with key **CamelCMISContent** will contain **InputStream** of the document type of nodes.

61.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cmis</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.11 or higher).

61.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 62. CM SMS GATEWAY COMPONENT

Available as of Camel version 2.18

Camel-Cm-Sms is an Apache Camel component for the [CM SMS Gateway] (https://www.cmtelecom.com).

It allows to integrate CM SMS APIin an application as a camel component.

You must have a valid account. More information are available at CM Telecom.

```
cm-sms://sgw01.cm.nl/gateway.ashx?
defaultFrom=DefaultSender&defaultMaxNumberOfParts=8&productToken=xxxxx
```

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-cm-sms</artifactId>
 <version>x.x.x</version>
 <!-- use the same version as your Camel core version -->
</dependency>
```

62.1. OPTIONS

The CM SMS Gateway component has no options.

The CM SMS Gateway endpoint is configured using URI syntax:

```
cm-sms:host
```

with the following path and query parameters:

62.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** SMS Provider HOST with scheme		String

62.1.2. Query Parameters (5 parameters):

Name	Description	Default	Type
defaultFrom (producer)	This is the sender name. The maximum length is 11 characters.		String

Name	Description	Default	Type
defaultMaxNumberOfParts (producer)	If it is a multipart message forces the max number. Message can be truncated. Technically the gateway will first check if a message is larger than 160 characters, if so, the message will be cut into multiple 153 characters parts limited by these parameters.	8	int
productToken (producer)	**Required** The unique token to use		String
testConnectionOnStartup (producer)	Whether to test the connection to the SMS Gateway on startup	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

62.2. SAMPLE

You can try this project to see how camel-cm-sms can be integrated in a camel route.

CHAPTER 63. COAP COMPONENT

Available as of Camel version 2.16

Camel-CoAP is an Apache Camel component that allows you to work with CoAP, a lightweight REST-type protocol for machine-to-machine operation. CoAP, Constrained Application Protocol is a specialized web transfer protocol for use with constrained nodes and constrained networks and it is based on RFC 7252.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-coap</artifactId>
 <version>x.x.x</version>
 <!-- use the same version as your Camel core version -->
</dependency>
```

63.1. OPTIONS

The CoAP component has no options.

The CoAP endpoint is configured using URI syntax:

```
coap:uri
```

with the following path and query parameters:

63.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
uri	The URI for the CoAP endpoint		URI

63.1.2. Query Parameters (5 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
coapMethodRestrict (consumer)	Comma separated list of methods that the CoAP consumer will bind to. The default is to bind to all methods (DELETE, GET, POST, PUT).		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

63.2. MESSAGE HEADERS

Name	Type	Description
CamelCoapMethod	**String**	The request method that the CoAP producer should use when calling the target CoAP server URI. Valid options are DELETE, GET, PING, POST & PUT.
CamelCoapResponseCode	**String**	The CoAP response code sent by the external server. See RFC 7252 for details of what each code means.
CamelCoapUri	**String**	The URI of a CoAP server to call. Will override any existing URI configured directly on the endpoint.

63.2.1. Configuring the CoAP producer request method

The following rules determine which request method the CoAP producer will use to invoke the target URI:

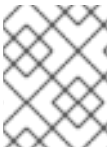
1. The value of the **CamelCoapMethod** header
2. **GET** if a query string is provided on the target CoAP server URI.

3. **POST** if the message exchange body is not null.
4. **GET** otherwise.

CHAPTER 64. CONSTANT LANGUAGE

Available as of Camel version 1.5

The Constant Expression Language is really just a way to specify constant strings as a type of expression.



NOTE

This is a fixed constant value that is only set once during starting up the route, do not use this if you want dynamic values during routing.

64.1. CONSTANT OPTIONS

The Constant language supports 1 options which are listed below.

Name	Default	Java Type	Description
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

64.2. EXAMPLE USAGE

The setHeader element of the Spring DSL can utilize a constant expression like:

```
<route>
  <from uri="seda:a"/>
  <setHeader headerName="theHeader">
    <constant>the value</constant>
  </setHeader>
  <to uri="mock:b"/>
</route>
```

in this case, the Message coming from the seda:a Endpoint will have 'theHeader' header set to the constant value 'the value'.

And the same example using Java DSL:

```
from("seda:a")
  .setHeader("theHeader", constant("the value"))
  .to("mock:b");
```

64.3. DEPENDENCIES

The Constant language is part of **camel-core**.

CHAPTER 65. COMETD COMPONENT

Available as of Camel version 2.0

The **cometd:** component is a transport for working with the jetty implementation of the cometd/bayeux protocol.
Using this component in combination with the dojo toolkit library it's possible to push Camel messages directly into the browser using an AJAX based mechanism.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cometd</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

65.1. URI FORMAT

```
cometd://host:port/channelName[?options]
```

The **channelName** represents a topic that can be subscribed to by the Camel endpoints.

65.2. EXAMPLES

```
cometd://localhost:8080/service/mychannel
cometds://localhost:8443/service/mychannel
```

where **cometds:** represents an SSL configured endpoint.

65.3. OPTIONS

The CometD component supports 8 options which are listed below.

Name	Description	Default	Type
sslKeyPassword (security)	The password for the keystore when using SSL.		String
sslPassword (security)	The password when using SSL.		String
sslKeystore (security)	The path to the keystore.		String
securityPolicy (security)	To use a custom configured SecurityPolicy to control authorization		SecurityPolicy

Name	Description	Default	Type
extensions (common)	To use a list of custom BayeuxServer.Extension that allows modifying incoming and outgoing requests.		List
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The CometD endpoint is configured using URI syntax:

```
cometd:host:port/channelName
```

with the following path and query parameters:

65.3.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	**Required** Hostname		String
port	**Required** Host port number		int
channelName	**Required** The channelName represents a topic that can be subscribed to by the Camel endpoints.		String

65.3.2. Query Parameters (16 parameters):

Name	Description	Default	Type
allowedOrigins (common)	The origins domain that support to cross, if the crosssOriginFilterOn is true	*	String

Name	Description	Default	Type
baseResource (common)	The root directory for the web resources or classpath. Use the protocol file: or classpath: depending if you want that the component loads the resource from file system or classpath. Classpath is required for OSGI deployment where the resources are packaged in the jar		String
crossOriginFilterOn (common)	If true, the server will support for cross-domain filtering	false	boolean
filterPath (common)	The filterPath will be used by the CrossOriginFilter, if the crosssOriginFilterOn is true		String
interval (common)	The client side poll timeout in milliseconds. How long a client will wait between reconnects		int
jsonCommented (common)	If true, the server will accept JSON wrapped in a comment and will generate JSON wrapped in a comment. This is a defence against Ajax Hijacking.	true	boolean
logLevel (common)	Logging level. 0=none, 1=info, 2=debug.	1	int
maxInterval (common)	The max client side poll timeout in milliseconds. A client will be removed if a connection is not received in this time.	30000	int
multiFrameInterval (common)	The client side poll timeout, if multiple connections are detected from the same browser.	1500	int
timeout (common)	The server side poll timeout in milliseconds. This is how long the server will hold a reconnect request before responding.	240000	int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sessionHeadersEnabled (consumer)	Whether to include the server session headers in the Camel message when creating a Camel Message for incoming requests.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
disconnectLocalSession (producer)	Whether to disconnect local sessions after publishing a message to its channel. Disconnecting local session is needed as they are not swept by default by CometD, and therefore you can run out of memory.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

You can append query options to the URI in the following format, **?option=value&option=value&…**

Here is some examples on How to pass the parameters

For file (for webapp resources located in the Web Application directory -→ cometd://localhost:8080?resourceBase=file./webapp
For classpath (when by example the web resources are packaged inside the webapp folder -→ cometd://localhost:8080?resourceBase=classpath:webapp

65.4. AUTHENTICATION

Available as of Camel 2.8

You can configure custom **SecurityPolicy** and **Extension's to the `CometdComponent** which allows you to use authentication as documented here

65.5. SETTING UP SSL FOR COMETD COMPONENT

65.5.1. Using the JSSE Configuration Utility

As of Camel 2.9, the Cometd component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Cometd component. You need to configure SSL on the CometdComponent.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
```

```
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);
scp.setTrustManagers(tmp);

CometdComponent commetdComponent = getContext().getComponent("cometds",
CometdComponent.class);
commetdComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:keyManagers
        keyPassword="keyPassword">
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
    <camel:trustManagers>
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...

  <bean id="cometd" class="org.apache.camel.component.cometd.CometdComponent">
    <property name="sslContextParameters" ref="sslContextParameters"/>
  </bean>
...
  <to uri="cometds://127.0.0.1:443/service/test?baseResource=file:./target/test-
classes/webapp&timeout=240000&interval=0&maxInterval=30000&multiFrameInterval=1500&jsonCom
mented=true&logLevel=2"/>...
```

65.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 66. CONSUL COMPONENT

Available as of Camel version 2.18

The **Consul** component is a component for integrating your application with Consul.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-consul</artifactId>
  <version>${camel-version}</version>
</dependency>
```

66.1. URI FORMAT

```
consul://domain?[options]
```

You can append query options to the URI in the following format:

```
?option=value&option=value&...
```

66.2. OPTIONS

The Consul component supports 9 options which are listed below.

Name	Description	Default	Type
url (common)	The Consul agent URL		String
datacenter (common)	The data center		String
sslContextParameters (common)	SSL configuration using an org.apache.camel.util.jsse.SSLContextParameters instance.		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
aclToken (common)	Sets the ACL token to be used with Consul		String
userName (common)	Sets the username to be used for basic authentication		String

Name	Description	Default	Type
password (common)	Sets the password to be used for basic authentication		String
configuration (advanced)	Sets the common configuration shared among endpoints		ConsulConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Consul endpoint is configured using URI syntax:

```
consul:apiEndpoint
```

with the following path and query parameters:

66.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
apiEndpoint	**Required** The API endpoint		String

66.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

66.3. HEADERS

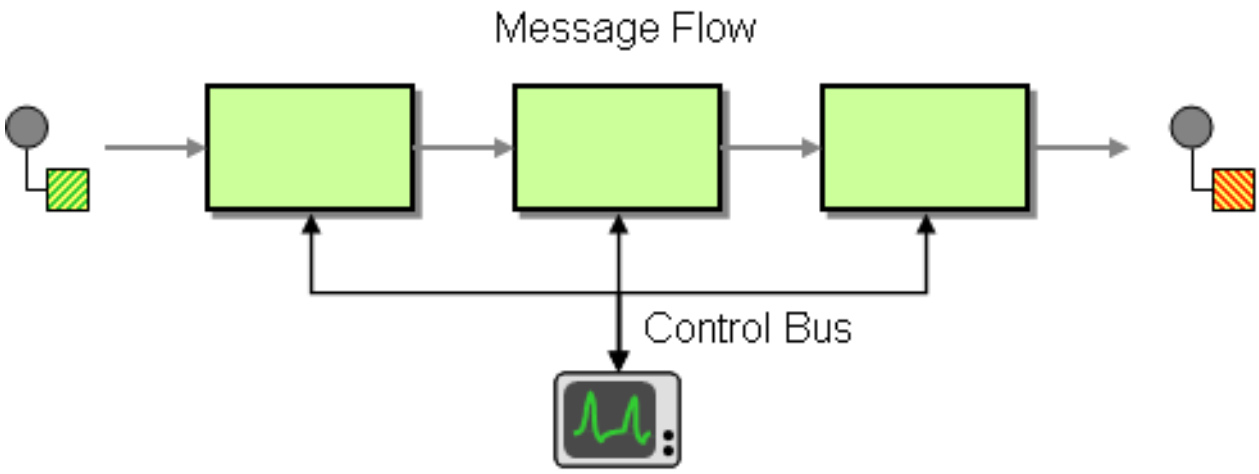
Name	Type	Description
Camel Consul Action	String	The Producer action
Camel Consul Key	String	The Key on which the action should applied
Camel Consul EventId	String	The event id (consumer only)
Camel Consul EventName	String	The event name (consumer only)
Camel Consul EventLTime	Long	The event LTime
Camel Consul NodeFilter	String	The Node filter
Camel Consul TagFilter	String	The tag filter

Name	Type	Description
Camel Consul Session Filter	String	The session filter
Camel Consul Version	int	The data version
Camel Consul Flags	Long	Flags associated with a value
Camel Consul CreateIndex	Long	The internal index value that represents when the entry was created
Camel Consul LockIndex	Long	The number of times this key has successfully been acquired in a lock
Camel Consul ModifyIndex	Long	The last index that modified this key
Camel Consul Options	Object	Options associated to the request
Camel Consul Result	boolean	true if the response has a result
Camel Consul Session	String	The session id
Camel Consul ValueAsString	boolean	To transform values retrieved from Consul i.e. on KV endpoint to string.

CHAPTER 67. CONTROL BUS COMPONENT

Available as of Camel version 2.11

The Control Bus from the EIP patterns allows for the integration system to be monitored and managed from within the framework.



Use a Control Bus to manage an enterprise integration system. The Control Bus uses the same messaging mechanism used by the application data, but uses separate channels to transmit data that is relevant to the management of components involved in the message flow.

In Camel you can manage and monitor using JMX, or by using a Java API from the **CamelContext**, or from the **org.apache.camel.api.management** package,
or use the event notifier which has an example here.

From Camel 2.11 onwards we have introduced a new ControlBus Component that allows you to send messages to a control bus Endpoint that reacts accordingly.

67.1. CONTROLBUS COMPONENT

Available as of Camel 2.11

The **controlbus:** component provides easy management of Camel applications based on the Control Bus EIP pattern. For example, by sending a message to an Endpoint you can control the lifecycle of routes, or gather performance statistics.

```
controlbus:command[?options]
```

Where **command** can be any string to identify which type of command to use.

67.2. COMMANDS

Command	Description
route	To control routes using the **routeId** and **action** parameter.

Command	Description
language	Allows you to specify a Language to use for evaluating the message body. If there is any result from the evaluation, then the result is put in the message body.

67.3. OPTIONS

The Control Bus component has no options.

The Control Bus endpoint is configured using URI syntax:

```
controlbus:command:language
```

with the following path and query parameters:

67.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
command	**Required** Command can be either route or language		String
language	Allows you to specify the name of a Language to use for evaluating the message body. If there is any result from the evaluation, then the result is put in the message body.		Language

67.3.2. Query Parameters (6 parameters):

Name	Description	Default	Type
action (producer)	To denote an action that can be either: start, stop, or status. To either start or stop a route, or to get the status of the route as output in the message body. You can use suspend and resume from Camel 2.11.1 onwards to either suspend or resume a route. And from Camel 2.11.1 onwards you can use stats to get performance statics returned in XML format; the routeId option can be used to define which route to get the performance stats for, if routeId is not defined, then you get statistics for the entire CamelContext. The restart action will restart the route.		String

Name	Description	Default	Type
async (producer)	Whether to execute the control bus task asynchronously. Important: If this option is enabled, then any result from the task is not set on the Exchange. This is only possible if executing tasks synchronously.	false	boolean
loggingLevel (producer)	Logging level used for logging when task is done, or if any exceptions occurred during processing the task.	INFO	LoggingLevel
restartDelay (producer)	The delay in millis to use when restarting a route.	1000	int
routeId (producer)	To specify a route by its id. The special keyword current indicates the current route.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

You can append query options to the URI in the following format, **?option=value&option=value&…**

67.4. USING ROUTE COMMAND

The route command allows you to do common tasks on a given route very easily, for example to start a route, you can send an empty message to this endpoint:

```
template.sendBody("controlbus:route?routeId=foo&action=start", null);
```

To get the status of the route, you can do:

```
String status = template.requestBody("controlbus:route?routeId=foo&action=status", null,
String.class);
```

67.5. GETTING PERFORMANCE STATISTICS

Available as of Camel 2.11.1

This requires JMX to be enabled (is by default) then you can get the performance statics per route, or for the CamelContext. For example to get the statics for a route named foo, we can do:

```
String xml = template.requestBody("controlbus:route?routeId=foo&action=stats", null, String.class);
```

The returned statics is in XML format. Its the same data you can get from JMX with the **dumpRouteStatsAsXml** operation on the **ManagedRouteMBean**.

To get statics for the entire CamelContext you just omit the routeId parameter as shown below:

```
String xml = template.requestBody("controlbus:route?action=stats", null, String.class);
```

67.6. USING SIMPLE LANGUAGE

You can use the Simple language with the control bus, for example to stop a specific route, you can send a message to the **"controlbus:language:simple"** endpoint containing the following message:

```
template.sendBody("controlbus:language:simple", "${camelContext.stopRoute('myRoute')}");
```

As this is a void operation, no result is returned. However, if you want the route status you can do:

```
String status = template.requestBody("controlbus:language:simple",
"${camelContext.getRouteStatus('myRoute')}", String.class);
```

It's easier to use the **route** command to control lifecycle of routes. The **language** command allows you to execute a language script that has stronger powers such as Groovy or to some extend the Simple language.

For example to shutdown Camel itself you can do:

```
template.sendBody("controlbus:language:simple?async=true", "${camelContext.stop()}");
```

We use **async=true** to stop Camel asynchronously as otherwise we would be trying to stop Camel while it was in-flight processing the message we sent to the control bus component.

TIP

You can also use other languages such as Groovy, etc.

CHAPTER 68. COUCHBASE COMPONENT

Available as of Camel version 2.19

The **couchbase:** component allows you to treat CouchBase instances as a producer or consumer of messages.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-couchbase</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

68.1. URI FORMAT

```
couchbase:url
```

68.2. OPTIONS

The Couchbase component has no options.

The Couchbase endpoint is configured using URI syntax:

```
couchbase:protocol:hostname:port
```

with the following path and query parameters:

68.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol to use		String
hostname	**Required** The hostname to use		String
port	The port number to use	8091	int

68.2.2. Query Parameters (47 parameters):

Name	Description	Default	Type
bucket (common)	The bucket to use		String

Name	Description	Default	Type
key (common)	The key to use		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
consumerProcessedStrategy (consumer)	Define the consumer Processed strategy to use	none	String
descending (consumer)	Define if this operation is descending or not	false	boolean
designDocumentName (consumer)	The design document name to use	beer	String
limit (consumer)	The output limit to use	-1	int
rangeEndKey (consumer)	Define a range for the end key		String
rangeStartKey (consumer)	Define a range for the start key		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
skip (consumer)	Define the skip to use	-1	int
viewName (consumer)	The view name to use	brewery_beers	String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
autoStartIdForInserts (producer)	Define if we want an autostart Id when we are doing an insert operation	false	boolean
operation (producer)	The operation to do	CCB_PUT	String
persistTo (producer)	Where to persist the data	0	int
producerRetryAttempts (producer)	Define the number of retry attempts	2	int
producerRetryPause (producer)	Define the retry pause between different attempts	5000	int
replicateTo (producer)	Where to replicate the data	0	int
startingIdForInsertsFrom (producer)	Define the starting Id where we are doing an insert operation		long
additionalHosts (advanced)	The additional hosts		String
maxReconnectDelay (advanced)	Define the max delay during a reconnection	30000	long
obsPollInterval (advanced)	Define the observation polling interval	400	long
obsTimeout (advanced)	Define the observation timeout	-1	long
opQueueMaxBlockTime (advanced)	Define the max time an operation can be in queue blocked	10000	long

Name	Description	Default	Type
opTimeOut (advanced)	Define the operation timeout	2500	long
readBufferSize (advanced)	Define the buffer size	16384	int
shouldOptimize (advanced)	Define if we want to use optimization or not where possible	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
timeoutExceptionThreshold (advanced)	Define the threshold for throwing a timeout Exception	998	int
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel

Name	Description	Default	Type
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
password (security)	The password to use		String
username (security)	The username to use		String

CHAPTER 69. COUCHDB COMPONENT

Available as of Camel version 2.11

The **couchdb:** component allows you to treat CouchDB instances as a producer or consumer of messages. Using the lightweight LightCouch API, this camel component has the following features:

- As a consumer, monitors couch changesets for inserts, updates and deletes and publishes these as messages into camel routes.
- As a producer, can save, update and from Camel 2.18 delete (by using CouchDbMethod with DELETE value) documents into couch.
- Can support as many endpoints as required, eg for multiple databases across multiple instances.
- Ability to have events trigger for only deletes, only inserts/updates or all (default).
- Headers set for sequenceId, document revision, document id, and HTTP method type.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-couchdb</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

69.1. URI FORMAT

```
couchdb:http://hostname[:port]/database?[options]
```

Where **hostname** is the hostname of the running couchdb instance. Port is optional and if not specified then defaults to 5984.

69.2. OPTIONS

The CouchDB component has no options.

The CouchDB endpoint is configured using URI syntax:

```
couchdb:protocol:hostname:port/database
```

with the following path and query parameters:

69.2.1. Path Parameters (4 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol to use for communicating with the database.		String

Name	Description	Default	Type
hostname	**Required** Hostname of the running couchdb instance		String
port	Port number for the running couchdb instance	5984	int
database	**Required** Name of the database to use		String

69.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type
createDatabase (common)	Creates the database if it does not already exist	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
deletes (consumer)	Document deletes are published as events	true	boolean
heartbeat (consumer)	How often to send an empty message to keep socket alive in millis	30000	long
since (consumer)	Start tracking changes immediately after the given update sequence. The default, null, will start monitoring from the latest sequence.		String
style (consumer)	Specifies how many revisions are returned in the changes array. The default, main_only, will only return the current winning revision; all_docs will return all leaf revisions (including conflicts and deleted former conflicts.)	main_only	String
updates (consumer)	Document inserts/updates are published as events	true	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
password (security)	Password for authenticated databases		String
username (security)	Username in case of authenticated databases		String

69.3. HEADERS

The following headers are set on exchanges during message transport.

Property	Value
CouchDbDatabase	the database the message came from
CouchDbSeq	the couchdb changeset sequence number of the update / delete message
CouchDbId	the couchdb document id
CouchDbRev	the couchdb document revision
CouchDbMethod	the method (delete / update)

Headers are set by the consumer once the message is received. The producer will also set the headers for downstream processors once the insert/update has taken place. Any headers set prior to the producer are ignored. That means for example, if you set CouchDbId as a header, it will not be used as the id for insertion, the id of the document will still be used.

69.4. MESSAGE BODY

The component will use the message body as the document to be inserted. If the body is an instance of String, then it will be marshalled into a GSON object before insert. This means that the string must be valid JSON or the insert / update will fail. If the body is an instance of a com.google.gson.JsonElement then it will be inserted as is. Otherwise the producer will throw an exception of unsupported body type.

69.5. SAMPLES

For example if you wish to consume all inserts, updates and deletes from a CouchDB instance running locally, on port 9999 then you could use the following:

```
from("couchdb:http://localhost:9999").process(someProcessor);
```

If you were only interested in deletes, then you could use the following

```
from("couchdb:http://localhost:9999?updates=false").process(someProcessor);
```

If you wanted to insert a message as a document, then the body of the exchange is used

```
from("someProducingEndpoint").process(someProcessor).to("couchdb:http://localhost:9999")
```

CHAPTER 70. CASSANDRA CQL COMPONENT

Available as of Camel version 2.15

Apache Cassandra is an open source NoSQL database designed to handle large amounts on commodity hardware. Like Amazon's DynamoDB, Cassandra has a peer-to-peer and master-less architecture to avoid single point of failure and garanty high availability. Like Google's BigTable, Cassandra data is structured using column families which can be accessed through the Thrift RPC API or a SQL-like API called CQL.

This component aims at integrating Cassandra 2.0+ using the CQL3 API (not the Thrift API). It's based on Cassandra Java Driver provided by DataStax.

Maven users will need to add the following dependency to their **pom.xml**:

pom.xml

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cassandraql</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

70.1. URI FORMAT

The endpoint can initiate the Cassandra connection or use an existing one.

URI	Description
cql:localhost/keyspace	Single host, default port, usual for testing
cql:host1,host2/keyspace	Multi host, default port
cql:host1,host2:9042/keyspace	Multi host, custom port
cql:host1,host2	Default port and keyspace
cql:bean:sessionRef	Provided Session reference
cql:bean:clusterRef/keyspace	Provided Cluster reference

To fine tune the Cassandra connection (SSL options, pooling options, load balancing policy, retry policy, reconnection policy...), create your own Cluster instance and give it to the Camel endpoint.

70.2. CASSANDRA OPTIONS

The Cassandra CQL component has no options.

The Cassandra CQL endpoint is configured using URI syntax:

```
cql:beanRef:hosts:port/keyspace
```

with the following path and query parameters:

70.2.1. Path Parameters (4 parameters):

Name	Description	Default	Type
beanRef	beanRef is defined using bean:id		String
hosts	Hostname(s) cassansdra server(s). Multiple hosts can be separated by comma.		String
port	Port number of cassansdra server(s)		Integer
keyspace	Keyspace to use		String

70.2.2. Query Parameters (29 parameters):

Name	Description	Default	Type
cluster (common)	To use the Cluster instance (you would normally not use this option)		Cluster
clusterName (common)	Cluster name		String
consistencyLevel (common)	Consistency level to use		ConsistencyLevel
cql (common)	CQL query to perform. Can be overridden with the message header with key CamelCqlQuery.		String
loadBalancingPolicy (common)	To use a specific LoadBalancingPolicy		String
password (common)	Password for session authentication		String
prepareStatements (common)	Whether to use PreparedStatements or regular Statements	true	boolean
resultSetConversionStrategy (common)	To use a custom class that implements logic for converting ResultSet into message body ALL, ONE, LIMIT_10, LIMIT_100...		String

Name	Description	Default	Type
session (common)	To use the Session instance (you would normally not use this option)		Session
username (common)	Username for session authentication		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

70.3. MESSAGES

70.3.1. Incoming Message

The Camel Cassandra endpoint expects a bunch of simple objects (**Object** or **Object[]** or **Collection<Object>**) which will be bound to the CQL statement as query parameters. If message body is null or empty, then CQL query will be executed without binding parameters.

Headers:

- **CamelCqlQuery** (optional, **String** or **RegularStatement**): CQL query either as a plain String or built using the **QueryBuilder**.

70.3.2. Outgoing Message

The Camel Cassandra endpoint produces one or many a Cassandra Row objects depending on the **resultSetConversionStrategy**:

- **List<Row>** if **resultSetConversionStrategy** is **ALL** or **LIMIT_[0-9]+**
- Single` Row` if **resultSetConversionStrategy** is **ONE**
- Anything else, if **resultSetConversionStrategy** is a custom implementation of the **ResultSetConversionStrategy**

70.4. REPOSITORIES

Cassandra can be used to store message keys or messages for the idempotent and aggregation EIP.

Cassandra might not be the best tool for queuing use cases yet, read Cassandra anti-patterns queues and queue like datasets. It's advised to use LeveledCompaction and a small GC grace setting for these tables to allow tombstoned rows to be removed quickly.

70.5. IDEMPOTENT REPOSITORY

The **NamedCassandraIdempotentRepository** stores messages keys in a Cassandra table like this:

CAMEL_IDEMPOTENT.cql

```
CREATE TABLE CAMEL_IDEMPOTENT (
  NAME varchar,   -- Repository name
  KEY varchar,    -- Message key
  PRIMARY KEY (NAME, KEY)
) WITH compaction = {'class':'LeveledCompactionStrategy'}
  AND gc_grace_seconds = 86400;
```

This repository implementation uses lightweight transactions (also known as Compare and Set) and requires Cassandra 2.0.7+.

Alternatively, the **CassandraIdempotentRepository** does not have a **NAME** column and can be extended to use a different data model.

Option	Default	Description
table	**CAMEL_IDEMPOTENT**	Table name
pkColumns	**NAME**,` KEY`	Primary key columns
name		Repository name, value used for **NAME** column
ttl		Key time to live
writeConsistencyLevel		Consistency level used to insert/delete key: **ANY**, **ONE**, **TWO**, **QUORUM**, **LOCAL_QUORUM**...
readConsistencyLevel		Consistency level used to read/check key: **ONE**, **TWO**, **QUORUM**, **LOCAL_QUORUM**...

70.6. AGGREGATION REPOSITORY

The **NamedCassandraAggregationRepository** stores exchanges by correlation key in a Cassandra table like this:

CAMEL_AGGREGATION.cql

```
CREATE TABLE CAMEL_AGGREGATION (
  NAME varchar,        -- Repository name
  KEY varchar,         -- Correlation id
  EXCHANGE_ID varchar, -- Exchange id
  EXCHANGE blob,       -- Serialized exchange
  PRIMARY KEY (NAME, KEY)
) WITH compaction = {'class':'LeveledCompactionStrategy'}
  AND gc_grace_seconds = 86400;
```

Alternatively, the **CassandraAggregationRepository** does not have a **NAME** column and can be extended to use a different data model.

Option	Default	Description
table	**CAMEL_AGGREGATION**	Table name
pkColumns	**NAME,KEY**	Primary key columns
exchangeIdColumn	**EXCHANGE_ID**	Exchange Id column

Option	Default	Description
exchangeColumn	**EXCHANGE**	Exchange content column
name		Repository name, value used for **NAME** column
ttl		Exchange time to live
writeConsistencyLevel		Consistency level used to insert/delete exchange: **ANY**, **ONE**, **TWO**, **QUORUM**, **LOCAL_QUORUM**...
readConsistencyLevel		Consistency level used to read/check exchange: **ONE**, **TWO**, **QUORUM**, **LOCAL_QUORUM**...

CHAPTER 71. CRYPTO (JCE) COMPONENT

Available as of Camel version 2.3

With Camel cryptographic endpoints and Java's Cryptographic extension it is easy to create Digital Signatures for Exchanges. Camel provides a pair of flexible endpoints which get used in concert to create a signature for an exchange in one part of the exchange's workflow and then verify the signature in a later part of the workflow.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

71.1. INTRODUCTION

Digital signatures make use of Asymmetric Cryptographic techniques to sign messages. From a (very) high level, the algorithms use pairs of complimentary keys with the special property that data encrypted with one key can only be decrypted with the other. One, the private key, is closely guarded and used to 'sign' the message while the other, public key, is shared around to anyone interested in verifying the signed messages. Messages are signed by using the private key to encrypting a digest of the message. This encrypted digest is transmitted along with the message. On the other side the verifier recalculates the message digest and uses the public key to decrypt the the digest in the signature. If both digests match the verifier knows only the holder of the private key could have created the signature.

Camel uses the Signature service from the Java Cryptographic Extension to do all the heavy cryptographic lifting required to create exchange signatures. The following are some excellent resources for explaining the mechanics of Cryptography, Message digests and Digital Signatures and how to leverage them with the JCE.

- Bruce Schneier's Applied Cryptography
- Beginning Cryptography with Java by David Hook
- The ever insightful Wikipedia Digital_signatures

71.2. URI FORMAT

As mentioned Camel provides a pair of crypto endpoints to create and verify signatures

```
crypto:sign:name[?options]
crypto:verify:name[?options]
```

- **crypto:sign** creates the signature and stores it in the Header keyed by the constant **org.apache.camel.component.crypto.DigitalSignatureConstants.SIGNATURE**, i.e. **"CamelDigitalSignature"**.
- **crypto:verify** will read in the contents of this header and do the verification calculation.

In order to correctly function, the sign and verify process needs a pair of keys to be shared, signing

requiring a **PrivateKey** and verifying a **PublicKey** (or a **Certificate** containing one). Using the JCE it is very simple to generate these key pairs but it is usually most secure to use a KeyStore to house and share your keys. The DSL is very flexible about how keys are supplied and provides a number of mechanisms.

Note a **crypto:sign** endpoint is typically defined in one route and the complimentary **crypto:verify** in another, though for simplicity in the examples they appear one after the other. It goes without saying that both signing and verifying should be configured identically.

71.3. OPTIONS

The Crypto (JCE) component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared DigitalSignatureConfiguration as configuration		DigitalSignatureConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Crypto (JCE) endpoint is configured using URI syntax:

```
crypto:cryptoOperation:name
```

with the following path and query parameters:

71.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
cryptoOperation	**Required** Set the Crypto operation from that supplied after the crypto scheme in the endpoint uri e.g. crypto:sign sets sign as the operation.		CryptoOperation
name	**Required** The logical name of this operation.		String

71.3.2. Query Parameters (19 parameters):

Name	Description	Default	Type
algorithm (producer)	Sets the JCE name of the Algorithm that should be used for the signer.	SHA1WithDSA	String

Name	Description	Default	Type
alias (producer)	Sets the alias used to query the KeyStore for keys and link java.security.cert.Certificate Certificates to be used in signing and verifying exchanges. This value can be provided at runtime via the message header link org.apache.camel.component.crypto.DigitalSignatureConstantsKEYSTORE_ALIAS		String
certificateName (producer)	Sets the reference name for a PrivateKey that can be fond in the registry.		String
keystore (producer)	Sets the KeyStore that can contain keys and Certficates for use in signing and verifying exchanges. A KeyStore is typically used with an alias, either one supplied in the Route definition or dynamically via the message header CamelSignatureKeyStoreAlias. If no alias is supplied and there is only a single entry in the Keystore, then this single entry will be used.		KeyStore
keystoreName (producer)	Sets the reference name for a Keystore that can be fond in the registry.		String
privateKey (producer)	Set the PrivateKey that should be used to sign the exchange		PrivateKey
privateKeyName (producer)	Sets the reference name for a PrivateKey that can be fond in the registry.		String
provider (producer)	Set the id of the security provider that provides the configured Signature algorithm.		String
publicKeyName (producer)	references that should be resolved when the context changes		String
secureRandomName (producer)	Sets the reference name for a SecureRandom that can be fond in the registry.		String
signatureHeaderName (producer)	Set the name of the message header that should be used to store the base64 encoded signature. This defaults to 'CamelDigitalSignature'		String
bufferSize (advanced)	Set the size of the buffer used to read in the Exchange payload data.	2048	Integer
certificate (advanced)	Set the Certificate that should be used to verify the signature in the exchange based on its payload.		Certificate

Name	Description	Default	Type
clearHeaders (advanced)	Determines if the Signature specific headers be cleared after signing and verification. Defaults to true, and should only be made otherwise at your extreme peril as vital private information such as Keys and passwords may escape if unset.	true	boolean
keyStoreParameters (advanced)	Sets the KeyStore that can contain keys and Certficates for use in signing and verifying exchanges based on the given KeyStoreParameters. A KeyStore is typically used with an alias, either one supplied in the Route definition or dynamically via the message header CamelSignatureKeyStoreAlias. If no alias is supplied and there is only a single entry in the Keystore, then this single entry will be used.		KeyStoreParameters
publicKey (advanced)	Set the PublicKey that should be used to verify the signature in the exchange.		PublicKey
secureRandom (advanced)	Set the SecureRandom used to initialize the Signature service		SecureRandom
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
password (security)	Sets the password used to access an aliased PrivateKey in the KeyStore.		String

71.4. USING

71.4.1. Raw keys

The most basic way to way to sign and verify an exchange is with a KeyPair as follows.

The same can be achieved with the Spring XML Extensions using references to keys

71.4.2. KeyStores and Aliases.

The JCE provides a very versatile keystore concept for housing pairs of private keys and certificates, keeping them encrypted and password protected. They can be retrieved by applying an alias to the retrieval APIs. There are a number of ways to get keys and Certificates into a keystore, most often this is done with the external 'keytool' application. This is a good example of using keytool to create a KeyStore with a self signed Cert and Private key.

The examples use a Keystore with a key and cert aliased by 'bob'. The password for the keystore and the key is 'letmein'

The following shows how to use a Keystore via the Fluent builders, it also shows how to load and initialize the keystore.

Again in Spring a ref is used to lookup an actual keystore instance.

71.4.3. Changing JCE Provider and Algorithm

Changing the Signature algorithm or the Security provider is a simple matter of specifying their names. You will need to also use Keys that are compatible with the algorithm you choose.

or

71.4.4. Changing the Signature Message Header

It may be desirable to change the message header used to store the signature. A different header name can be specified in the route definition as follows

or

71.4.5. Changing the buffersize

In case you need to update the size of the buffer...

or

71.4.6. Supplying Keys dynamically.

When using a Recipient list or similar EIP the recipient of an exchange can vary dynamically. Using the same key across all recipients may be neither feasible nor desirable. It would be useful to be able to specify signature keys dynamically on a per-exchange basis. The exchange could then be dynamically enriched with the key of its target recipient prior to signing. To facilitate this the signature mechanisms allow for keys to be supplied dynamically via the message headers below

- **Exchange.SIGNATURE_PRIVATE_KEY**, **"CamelSignaturePrivateKey"**
- **Exchange.SIGNATURE_PUBLIC_KEY_OR_CERT**, **"CamelSignaturePublicKeyOrCert"**

or

Even better would be to dynamically supply a keystore alias. Again the alias can be supplied in a message header

- **Exchange.KEYSTORE_ALIAS**, **"CamelSignatureKeyStoreAlias"**

or

The header would be set as follows

```
Exchange unsigned = getMandatoryEndpoint("direct:alias-sign").createExchange();
unsigned.getIn().setBody(payload);
unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_ALIAS, "bob");
unsigned.getIn().setHeader(DigitalSignatureConstants.KEYSTORE_PASSWORD,
"letmein".toCharArray());
template.send("direct:alias-sign", unsigned);
Exchange signed = getMandatoryEndpoint("direct:alias-sign").createExchange();
```

```
signed.getIn().copyFrom(unsigned.getOut());
signed.getIn().setHeader(KEYSTORE_ALIAS, "bob");
template.send("direct:alias-verify", signed);
```

71.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 72. CRYPTO CMS COMPONENT

Available as of Camel version 2.20

Cryptographic Message Syntax (CMS) is a well established standard for signing and encrypting messages. The Apache Crypto CMS component supports the following parts of this standard: * Content Type "Enveloped Data" with Key Transport (asymmetric key), * Content Type "Signed Data". You can create CMS Enveloped Data instances, decrypt CMS Enveloped Data instances, create CMS Signed Data instances, and validate CMS Signed Data instances.

The component uses the Bouncy Castle libraries bcprov-jdk15on and bcpkix-jdk15on.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto-cms</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

We recommend to register the Bouncy Castle security provider in your application before you call an endpoint of this component:

```
Security.addProvider(new BouncyCastleProvider());
```

If the Bouncy Castle security provider is not registered then the Crypto CMS component will register the provider.

72.1. OPTIONS

The Crypto CMS component supports 3 options which are listed below.

Name	Description	Default	Type
signedDataVerifier Configuration (advanced)	To configure the shared SignedDataVerifierConfiguration, which determines the uri parameters for the verify operation.		SignedDataVerifier Configuration
envelopedDataDecryptor Configuration (advanced)	To configure the shared EnvelopedDataDecryptorConfiguration, which determines the uri parameters for the decrypt operation.		EnvelopedDataDecryptor Configuration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Crypto CMS endpoint is configured using URI syntax:

```
crypto-cms:cryptoOperation:name
```

with the following path and query parameters:

72.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
cryptoOperation	**Required** Set the Crypto operation from that supplied after the crypto scheme in the endpoint uri e.g. crypto-cms:sign sets sign as the operation. Possible values: sign, verify, encrypt, or decrypt.		CryptoOperation
name	**Required** The name part in the URI can be chosen by the user to distinguish between different signer/verifier/encryptor/decryptor endpoints within the camel context.		String

72.1.2. Query Parameters (15 parameters):

Name	Description	Default	Type
keyStore (common)	Keystore which contains signer private keys, verifier public keys, encryptor public keys, decryptor private keys depending on the operation. Use either this parameter or the parameter 'keyStoreParameters'.		KeyStore
keyStoreParameters (common)	Keystore containing signer private keys, verifier public keys, encryptor public keys, decryptor private keys depending on the operation. Use either this parameter or the parameter 'keystore'.		KeyStoreParameters
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
password (decrypt)	Sets the password of the private keys. It is assumed that all private keys in the keystore have the same password. If not set then it is assumed that the password of the private keys is given by the keystore password given in the KeyStoreParameters.		Char[]
fromBase64 (decrypt_verify)	If true then the CMS message is base 64 encoded and must be decoded during the processing. Default value is false.	false	Boolean

Name	Description	Default	Type
contentEncryptionAlgorithm (encrypt)	Encryption algorithm, for example DESede/CBC/PKCS5Padding. Further possible values: DESede/CBC/PKCS5Padding, AES/CBC/PKCS5Padding, Camellia/CBC/PKCS5Padding, CAST5/CBC/PKCS5Padding.		String
originatorInformation Provider (encrypt)	Provider for the originator info. See https://tools.ietf.org/html/rfc5652section-6.1. The default value is null.		OriginatorInformation Provider
recipient (encrypt)	Recipient Info: reference to a bean which implements the interface org.apache.camel.component.crypto.cms.api.TransRecipientInfo		List
secretKeyLength (encrypt)	Key length for the secret symmetric key used for the content encryption. Only used if the specified content-encryption algorithm allows keys of different sizes. If contentEncryptionAlgorithm=AES/CBC/PKCS5Padding or Camellia/CBC/PKCS5Padding then 128; if contentEncryptionAlgorithm=DESede/CBC/PKCS5Padding then 192, 128; if strong encryption is enabled then for AES/CBC/PKCS5Padding and Camellia/CBC/PKCS5Padding also the key lengths 192 and 256 are possible.		int
unprotectedAttributes GeneratorProvider (encrypt)	Provider of the generator for the unprotected attributes. The default value is null which means no unprotected attribute is added to the Enveloped Data object. See https://tools.ietf.org/html/rfc5652section-6.1.		AttributesGenerator Provider
toBase64 (encrypt_sign)	Indicates whether the Signed Data or Enveloped Data instance shall be base 64 encoded. Default value is false.	false	Boolean
includeContent (sign)	Indicates whether the signed content should be included into the Signed Data instance. If false then a detached Signed Data instance is created in the header CamelCryptoCmsSignedData.	true	Boolean

Name	Description	Default	Type
signer (sign)	Signer information: reference to a bean which implements org.apache.camel.component.crypto.cms.api.SignerInfo		List
signedDataHeaderBase64 (verify)	Indicates whether the value in the header CamelCryptoCmsSignedData is base64 encoded. Default value is false. Only relevant for detached signatures. In the detached signature case, the header contains the Signed Data object.	false	Boolean
verifySignaturesOfAll Signers (verify)	If true then the signatures of all signers contained in the Signed Data object are verified. If false then only one signature whose signer info matches with one of the specified certificates is verified. Default value is true.	true	Boolean

72.2. ENVELOPED DATA

Note, that a **crypto-cms:encypt** endpoint is typically defined in one route and the complimentary **crypto-cms:decrypt** in another, though for simplicity in the examples they appear one after the other.

The following example shows how you can create an Enveloped Data message and how you can decrypt an Enveloped Data message.

Basic Example in Java DSL

```
import org.apache.camel.util.jsse.KeyStoreParameters;
import org.apache.camel.component.crypto.cms.crypt.DefaultKeyTransRecipientInfo;
...
KeyStoreParameters keystore  = new KeyStoreParameters();
keystore.setType("JCEKS");
keystore.setResource("keystore/keystore.jceks);
keystore.setPassword("some_password"); // this password will also be used for accessing the private key if not specified in the crypto-cms:decrypt endpoint

DefaultKeyTransRecipientInfo recipient1 = new DefaultKeyTransRecipientInfo();
recipient1.setCertificateAlias("rsa"); // alias of the public key used for the encryption
recipient1.setKeyStoreParameters(keystore);

simpleReg.put("keyStoreParameters", keystore); // register keystore in the registry
simpleReg.put("recipient1", recipient1); // register recipient info in the registry

from("direct:start")
    .to("crypto-cms:encrypt://testencrpyt?toBase64=true&recipient=#recipient1&contentEncryptionAlgorithm=DESede/CBC/PKCS5Padding&secretKeyLength=128")
```

```
    .to("crypto-cms:decrypt://testdecrypt?
fromBase64=true&keyStoreParameters=#keyStoreParameters")
    .to("mock:result");
```

Basic Example in Spring XML

```
  <keyStoreParameters xmlns="http://camel.apache.org/schema/spring"
      id="keyStoreParameters1" resource="./keystore/keystore.jceks"
      password="some_password" type="JCEKS" />
   <bean id="recipient1"
      class="org.apache.camel.component.crypto.cms.crypt.DefaultKeyTransRecipientInfo">
      <property name="keyStoreParameters" ref="keyStoreParameters1" />
      <property name="certificateAlias" value="rsa" />
   </bean>
...
   <route>
      <from uri="direct:start" />
      <to uri="crypto-cms:encrypt://testencrpyt?
toBase64=true&amp;recipient=#recipient1&amp;contentEncryptionAlgorithm=DESede/CBC/PKCS5Pad
ding&amp;secretKeyLength=128" />
      <to uri="crypto-cms:decrypt://testdecrypt?
fromBase64=true&amp;keyStoreParameters=#keyStoreParameters1" />
       <to uri="mock:result" />
   </route>
```

Two Recipients in Java DSL

```
import org.apache.camel.util.jsse.KeyStoreParameters;
import org.apache.camel.component.crypto.cms.crypt.DefaultKeyTransRecipientInfo;
...
KeyStoreParameters keystore  = new KeyStoreParameters();
keystore.setType("JCEKS");
keystore.setResource("keystore/keystore.jceks);
keystore.setPassword("some_password"); // this password will also be used for accessing the private
key if not specified in the crypto-cms:decrypt endpoint

DefaultKeyTransRecipientInfo recipient1 = new DefaultKeyTransRecipientInfo();
recipient1.setCertificateAlias("rsa"); // alias of the public key used for the encryption
recipient1.setKeyStoreParameters(keystore);

DefaultKeyTransRecipientInfo recipient2 = new DefaultKeyTransRecipientInfo();
recipient2.setCertificateAlias("dsa");
recipient2.setKeyStoreParameters(keystore);

simpleReg.put("keyStoreParameters", keystore); // register keystore in the registry
simpleReg.put("recipient1", recipient1); // register recipient info in the registry

from("direct:start")
   .to("crypto-cms:encrypt:/testencrpyt?
toBase64=true&recipient=#recipient1&recipient=#recipient2&contentEncryptionAlgorithm=DESede/CBC
PKCS5Padding&secretKeyLength=128")
   //the decryptor will automatically choose one of the two private keys depending which one is in the
decryptor keystore
```

```
    .to("crypto-cms:decrypt://testdecrypt?
fromBase64=true&keyStoreParameters=#keyStoreParameters")
    .to("mock:result");
```

Two Recipients in Spring XML

```
  <keyStoreParameters xmlns="http://camel.apache.org/schema/spring"
      id="keyStoreParameters1" resource="./keystore/keystore.jceks"
      password="some_password" type="JCEKS" />
   <bean id="recipient1"
      class="org.apache.camel.component.crypto.cms.crypt.DefaultKeyTransRecipientInfo">
      <property name="keyStoreParameters" ref="keyStoreParameters1" />
      <property name="certificateAlias" value="rsa" />
    </bean>
    <bean id="recipient2"
      class="org.apache.camel.component.crypto.cms.crypt.DefaultKeyTransRecipientInfo">
      <property name="keyStoreParameters" ref="keyStoreParameters1" />
      <property name="certificateAlias" value="dsa" />
    </bean>
...
    <route>
      <from uri="direct:start" />
      <to uri="crypto-cms:encrypt://testencrpyt?
toBase64=true&amp;recipient=#recipient1&amp;recipient=#recipient2&amp;contentEncryptionAlgorithm
=DESede/CBC/PKCS5Padding&amp;secretKeyLength=128" />
      <!-- the decryptor will automatically choose one of the two private keys depending which one is in
the decryptor keystore -->
      <to uri="crypto-cms:decrypt://testdecrypt?
fromBase64=true&amp;keyStoreParameters=#keyStoreParameters1" />
       <to uri="mock:result" />
    </route>
```

72.3. SIGNED DATA

Note, that a **crypto-cms:sign** endpoint is typically defined in one route and the complimentary **crypto-cms:verify** in another, though for simplicity in the examples they appear one after the other.

The following example shows how you can create a Signed Data message and how you can validate a Signed Data message.

Basic Example in Java DSL

```
import org.apache.camel.util.jsse.KeyStoreParameters;
import org.apache.camel.component.crypto.cms.sig.DefaultSignerInfo;
...
KeyStoreParameters keystore  = new KeyStoreParameters();
keystore.setType("JCEKS");
keystore.setResource("keystore/keystore.jceks);
keystore.setPassword("some_password"); // this password will also be used for accessing the private
key if not specified in the signerInfo1 bean

//Signer Information, by default the following signed attributes are included: contentType,
signingTime, messageDigest, and cmsAlgorithmProtect; by default no unsigned attribute is included.
// If you want to add your own signed attributes or unsigned attributes, see methods
DefaultSignerInfo.setSignedAttributeGenerator and DefaultSignerInfo.setUnsignedAttributeGenerator.
```

```
DefaultSignerInfo signerInfo1 = new DefaultSignerInfo();
signerInfo1.setIncludeCertificates(true); // if set to true then the certificate chain of the private key will be added to the Signed Data object
signerInfo1.setSignatureAlgorithm("SHA256withRSA"); // signature algorithm; attention, the signature algorithm must fit to the signer private key.
signerInfo1.setPrivateKeyAlias("rsa"); // alias of the private key used for the signing
signerInfo1.setPassword("private_key_pw".toCharArray()); // optional parameter, if not set then the password of the KeyStoreParameters will be used for accessing the private key
signerInfo1.setKeyStoreParameters(keystore);

simpleReg.put("keyStoreParameters", keystore); //register keystore in the registry
simpleReg.put("signer1", signerInfo1); //register signer info in the registry

from("direct:start")
    .to("crypto-cms:sign://testsign?signer=#signer1&includeContent=true&toBase64=true")
    .to("crypto-cms:verify://testverify?keyStoreParameters=#keyStoreParameters&fromBase64=true"")
    .to("mock:result");
```

Basic Example in Spring XML

```
  <keyStoreParameters xmlns="http://camel.apache.org/schema/spring"
      id="keyStoreParameters1" resource="./keystore/keystore.jceks"
      password="some_password" type="JCEKS" />
   <bean id="signer1"
      class="org.apache.camel.component.crypto.cms.sig.DefaultSignerInfo">
      <property name="keyStoreParameters" ref="keyStoreParameters1" />
      <property name="privateKeyAlias" value="rsa" />
      <property name="signatureAlgorithm" value="SHA256withRSA" />
      <property name="includeCertificates" value="true" />
      <!-- optional parameter 'password', if not set then the password of the KeyStoreParameters will be used for accessing the private key -->
      <property name="password" value="private_key_pw" />
   </bean>
...
   <route>
      <from uri="direct:start" />
      <to uri="crypto-cms:sign://testsign?signer=#signer1&amp;includeContent=true&amp;toBase64=true" />
      <to uri="crypto-cms:verify://testverify?keyStoreParameters=#keyStoreParameters1&amp;fromBase64=true" />
      <to uri="mock:result" />
   </route>
```

Example with two Signers in Java DSL

```
import org.apache.camel.util.jsse.KeyStoreParameters;
import org.apache.camel.component.crypto.cms.sig.DefaultSignerInfo;
...
KeyStoreParameters keystore  = new KeyStoreParameters();
keystore.setType("JCEKS");
keystore.setResource("keystore/keystore.jceks);
keystore.setPassword("some_password"); // this password will also be used for accessing the private key if not specified in the signerInfo1 bean

//Signer Information, by default the following signed attributes are included: contentType,
```

```
signingTime, messageDigest, and cmsAlgorithmProtect; by default no unsigned attribute is included.
// If you want to add your own signed attributes or unsigned attributes, see methods
DefaultSignerInfo.setSignedAttributeGenerator and DefaultSignerInfo.setUnsignedAttributeGenerator.
DefaultSignerInfo signerInfo1 = new DefaultSignerInfo();
signerInfo1.setIncludeCertificates(true); // if set to true then the certificate chain of the private key will
be added to the Signed Data object
signerInfo1.setSignatureAlgorithm("SHA256withRSA"); // signature algorithm; attention, the signature
algorithm must fit to the signer private key.
signerInfo1.setPrivateKeyAlias("rsa"); // alias of the private key used for the signing
signerInfo1.setPassword("private_key_pw".toCharArray()); // optional parameter, if not set then the
password of the KeyStoreParameters will be used for accessing the private key
signerInfo1.setKeyStoreParameters(keystore);

DefaultSignerInfo signerInfo2 = new DefaultSignerInfo();
signerInfo2.setIncludeCertificates(true);
signerInfo2.setSignatureAlgorithm("SHA256withDSA");
signerInfo2.setPrivateKeyAlias("dsa");
signerInfo2.setKeyStoreParameters(keystore);


simpleReg.put("keyStoreParameters", keystore); //register keystore in the registry
simpleReg.put("signer1", signerInfo1); //register signer info in the registry
simpleReg.put("signer2", signerInfo2); //register signer info in the registry

from("direct:start")
    .to("crypto-cms:sign://testsign?signer=#signer1&signer=#signer2&includeContent=true")
    .to("crypto-cms:verify://testverify?keyStoreParameters=#keyStoreParameters")
    .to("mock:result");
```

Example with two Signers in Spring XML

```
  <keyStoreParameters xmlns="http://camel.apache.org/schema/spring"
      id="keyStoreParameters1" resource="./keystore/keystore.jceks"
      password="some_password" type="JCEKS" />
  <bean id="signer1"
      class="org.apache.camel.component.crypto.cms.sig.DefaultSignerInfo">
      <property name="keyStoreParameters" ref="keyStoreParameters1" />
      <property name="privateKeyAlias" value="rsa" />
      <property name="signatureAlgorithm" value="SHA256withRSA" />
      <property name="includeCertificates" value="true" />
      <!-- optional parameter 'password', if not set then the password of the KeyStoreParameters will
be used for accessing the private key -->
      <property name="password" value="private_key_pw" />
  </bean>
  <bean id="signer2"
      class="org.apache.camel.component.crypto.cms.sig.DefaultSignerInfo">
      <property name="keyStoreParameters" ref="keyStoreParameters1" />
      <property name="privateKeyAlias" value="dsa" />
      <property name="signatureAlgorithm" value="SHA256withDSA" />
      <!-- optional parameter 'password', if not set then the password of the KeyStoreParameters will
be used for accessing the private key -->
      <property name="password" value="private_key_pw2" />
  </bean>
...
  <route>
      <from uri="direct:start" />
```

```
        <to uri="crypto-cms:sign://testsign?
signer=#signer1&amp;signer=#signer2&amp;includeContent=true" />
        <to uri="crypto-cms:verify://testverify?keyStoreParameters=#keyStoreParameters1" />
        <to uri="mock:result" />
    </route>
```

Detached Signature Example in Java DSL

```
import org.apache.camel.util.jsse.KeyStoreParameters;
import org.apache.camel.component.crypto.cms.sig.DefaultSignerInfo;
...
KeyStoreParameters keystore  = new KeyStoreParameters();
keystore.setType("JCEKS");
keystore.setResource("keystore/keystore.jceks);
keystore.setPassword("some_password"); // this password will also be used for accessing the private
key if not specified in the signerInfo1 bean

//Signer Information, by default the following signed attributes are included: contentType,
signingTime, messageDigest, and cmsAlgorithmProtect; by default no unsigned attribute is included.
// If you want to add your own signed attributes or unsigned attributes, see methods
DefaultSignerInfo.setSignedAttributeGenerator and DefaultSignerInfo.setUnsignedAttributeGenerator.
DefaultSignerInfo signerInfo1 = new DefaultSignerInfo();
signerInfo1.setIncludeCertificates(true); // if set to true then the certificate chain of the private key will
be added to the Signed Data object
signerInfo1.setSignatureAlgorithm("SHA256withRSA"); // signature algorithm; attention, the signature
algorithm must fit to the signer private key.
signerInfo1.setPrivateKeyAlias("rsa"); // alias of the private key used for the signing
signerInfo1.setPassword("private_key_pw".toCharArray()); // optional parameter, if not set then the
password of the KeyStoreParameters will be used for accessing the private key
signerInfo1.setKeyStoreParameters(keystore);

simpleReg.put("keyStoreParameters", keystore); //register keystore in the registry
simpleReg.put("signer1", signerInfo1); //register signer info in the registry

from("direct:start")
    //with the option includeContent=false the SignedData object without the signed text will be written
into the header "CamelCryptoCmsSignedData"
    .to("crypto-cms:sign://testsign?signer=#signer1&includeContent=false&toBase64=true")
    //the verifier reads the Signed Data object form the header CamelCryptoCmsSignedData and
assumes that the signed content is in the message body
    .to("crypto-cms:verify://testverify?
keyStoreParameters=#keyStoreParameters&signedDataHeaderBase64=true")
    .to("mock:result");
```

Detached Signature Example in Spring XML

```
  <keyStoreParameters xmlns="http://camel.apache.org/schema/spring"
      id="keyStoreParameters1" resource="./keystore/keystore.jceks"
      password="some_password" type="JCEKS" />
  <bean id="signer1"
      class="org.apache.camel.component.crypto.cms.sig.DefaultSignerInfo">
      <property name="keyStoreParameters" ref="keyStoreParameters1" />
      <property name="privateKeyAlias" value="rsa" />
      <property name="signatureAlgorithm" value="SHA256withRSA" />
      <property name="includeCertificates" value="true" />
```

```
        <!-- optional parameter 'password', if not set then the password of the KeyStoreParameters will
be used for accessing the private key -->
        <property name="password" value="private_key_pw" />
    </bean>
...
    <route>
        <from uri="direct:start" />
        <!-- with the option includeContent=false the SignedData object without the signed text will be
written into the header "CamelCryptoCmsSignedData" -->
        <to uri="crypto-cms:sign://testsign?
signer=#signer1&amp;includeContent=false&amp;toBase64=true" />
        <!-- the verifier reads the Signed Data object form the header CamelCryptoCmsSignedData and
assumes that the signed content is in the message body -->
        <to uri="crypto-cms:verify://testverify?
keyStoreParameters=#keyStoreParameters1&amp;signedDataHeaderBase64=true" />
        <to uri="mock:result" />
    </route>
```

CHAPTER 73. CRYPTO (JAVA CRYPTOGRAPHIC EXTENSION) DATAFORMAT

Available as of Camel version 2.3

The Crypto Data Format integrates the Java Cryptographic Extension into Camel, allowing simple and flexible encryption and decryption of messages using Camel's familiar marshall and unmarshal formatting mechanism. It assumes marshalling to mean encryption to cyphertext and unmarshalling to mean decryption back to the original plaintext. This data format implements only symmetric (shared-key) encryption and decyption.

73.1. CRYPTODATAFORMAT OPTIONS

The Crypto (Java Cryptographic Extension) dataformat supports 10 options which are listed below.

Name	Default	Java Type	Description
algorithm	**DES/CBC/PKCS5Padding**	**String**	The JCE algorithm name indicating the cryptographic algorithm that will be used. Is by default DES/CBC/PKCS5Padding.
cryptoProvider		**String**	The name of the JCE Security Provider that should be used.
keyRef		**String**	Refers to the secret key to lookup from the register to use.
initVectorRef		**String**	Refers to a byte array containing the Initialization Vector that will be used to initialize the Cipher.
algorithmParameterRef		**String**	A JCE AlgorithmParameterSpec used to initialize the Cipher. Will lookup the type using the given name as a java.security.spec.AlgorithmParameterSpec type.
buffersize		**Integer**	The size of the buffer used in the signature process.
macAlgorithm	**HmacSHA1**	**String**	The JCE algorithm name indicating the Message Authentication algorithm.
shouldAppendHMAC	**false**	**Boolean**	Flag indicating that a Message Authentication Code should be calculated and appended to the encrypted data.
inline	**false**	**Boolean**	Flag indicating that the configured IV should be inlined into the encrypted data stream. Is by default false.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

73.2. BASIC USAGE

At its most basic all that is required to encrypt/decrypt an exchange is a shared secret key. If one or more instances of the Crypto data format are configured with this key the format can be used to encrypt the payload in one route (or part of one) and decrypted in another. For example, using the Java DSL as follows:

```
KeyGenerator generator = KeyGenerator.getInstance("DES");

CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES", generator.generateKey());

from("direct:basic-encryption")
    .marshal(cryptoFormat)
    .to("mock:encrypted")
    .unmarshal(cryptoFormat)
    .to("mock:unencrypted");
```

In Spring the dataformat is configured first and then used in routes

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <crypto id="basic" algorithm="DES" keyRef="desKey" />
  </dataFormats>
    ...
  <route>
    <from uri="direct:basic-encryption" />
    <marshal ref="basic" />
    <to uri="mock:encrypted" />
    <unmarshal ref="basic" />
    <to uri="mock:unencrypted" />
  </route>
</camelContext>
```

73.3. SPECIFYING THE ENCRYPTION ALGORITHM

Changing the algorithm is a matter of supplying the JCE algorithm name. If you change the algorithm you will need to use a compatible key.

```
KeyGenerator generator = KeyGenerator.getInstance("DES");

CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES", generator.generateKey());
cryptoFormat.setShouldAppendHMAC(true);
cryptoFormat.setMacAlgorithm("HmacMD5");
```

```
from("direct:hmac-algorithm")
    .marshal(cryptoFormat)
    .to("mock:encrypted")
    .unmarshal(cryptoFormat)
    .to("mock:unencrypted");
```

A list of the available algorithms in Java 7 is available via the Java Cryptography Architecture Standard Algorithm Name Documentation.

73.4. SPECIFYING AN INITIALIZATION VECTOR

Some crypto algorithms, particularly block algorithms, require configuration with an initial block of data known as an Initialization Vector. In the JCE this is passed as an AlgorithmParameterSpec when the Cipher is initialized. To use such a vector with the CryptoDataFormat you can configure it with a byte[] containing the required data e.g.

```
KeyGenerator generator = KeyGenerator.getInstance("DES");
byte[] initializationVector = new byte[] {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};

CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES/CBC/PKCS5Padding",
generator.generateKey());
cryptoFormat.setInitializationVector(initializationVector);

from("direct:init-vector")
    .marshal(cryptoFormat)
    .to("mock:encrypted")
    .unmarshal(cryptoFormat)
    .to("mock:unencrypted");
```

or with spring, suppling a reference to a byte[]

```
<crypto id="initvector" algorithm="DES/CBC/PKCS5Padding" keyRef="desKey"
initVectorRef="initializationVector" />
```

The same vector is required in both the encryption and decryption phases. As it is not necessary to keep the IV a secret, the DataFormat allows for it to be inlined into the encrypted data and subsequently read out in the decryption phase to initialize the Cipher. To inline the IV set the /oinline flag.

```
KeyGenerator generator = KeyGenerator.getInstance("DES");
byte[] initializationVector = new byte[] {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07};
SecretKey key = generator.generateKey();

CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES/CBC/PKCS5Padding", key);
cryptoFormat.setInitializationVector(initializationVector);
cryptoFormat.setShouldInlineInitializationVector(true);
CryptoDataFormat decryptFormat = new CryptoDataFormat("DES/CBC/PKCS5Padding", key);
decryptFormat.setShouldInlineInitializationVector(true);

from("direct:inline")
    .marshal(cryptoFormat)
    .to("mock:encrypted")
    .unmarshal(decryptFormat)
    .to("mock:unencrypted");
```

or with spring.

```
<crypto id="inline" algorithm="DES/CBC/PKCS5Padding" keyRef="desKey"
initVectorRef="initializationVector"
  inline="true" />
<crypto id="inline-decrypt" algorithm="DES/CBC/PKCS5Padding" keyRef="desKey" inline="true" />
```

For more information of the use of Initialization Vectors, consult

- http://en.wikipedia.org/wiki/Initialization_vector
- http://www.herongyang.com/Cryptography/
- http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

73.5. HASHED MESSAGE AUTHENTICATION CODES (HMAC)

To avoid attacks against the encrypted data while it is in transit the CryptoDataFormat can also calculate a Message Authentication Code for the encrypted exchange contents based on a configurable MAC algorithm. The calculated HMAC is appended to the stream after encryption. It is separated from the stream in the decryption phase. The MAC is recalculated and verified against the transmitted version to insure nothing was tampered with in transit.For more information on Message Authentication Codes see http://en.wikipedia.org/wiki/HMAC

```
KeyGenerator generator = KeyGenerator.getInstance("DES");

CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES", generator.generateKey());
cryptoFormat.setShouldAppendHMAC(true);

from("direct:hmac")
    .marshal(cryptoFormat)
    .to("mock:encrypted")
    .unmarshal(cryptoFormat)
    .to("mock:unencrypted");
```

or with spring.

```
<crypto id="hmac" algorithm="DES" keyRef="desKey" shouldAppendHMAC="true" />
```

By default the HMAC is calculated using the HmacSHA1 mac algorithm though this can be easily changed by supplying a different algorithm name. See here for how to check what algorithms are available through the configured security providers

```
KeyGenerator generator = KeyGenerator.getInstance("DES");

CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES", generator.generateKey());
cryptoFormat.setShouldAppendHMAC(true);
cryptoFormat.setMacAlgorithm("HmacMD5");

from("direct:hmac-algorithm")
    .marshal(cryptoFormat)
    .to("mock:encrypted")
    .unmarshal(cryptoFormat)
    .to("mock:unencrypted");
```

or with spring.

```
<crypto id="hmac-algorithm" algorithm="DES" keyRef="desKey" macAlgorithm="HmacMD5"
shouldAppendHMAC="true" />
```

73.6. SUPPLYING KEYS DYNAMICALLY

When using a Recipient list or similar EIP the recipient of an exchange can vary dynamically. Using the same key across all recipients may neither be feasible or desirable. It would be useful to be able to specify keys dynamically on a per exchange basis. The exchange could then be dynamically enriched with the key of its target recipient before being processed by the data format. To facilitate this the DataFormat allow for keys to be supplied dynamically via the message headers below

- CryptoDataFormat.KEY "CamelCryptoKey"

```
CryptoDataFormat cryptoFormat = new CryptoDataFormat("DES", null);
/**
 * Note: the header containing the key should be cleared after
 * marshalling to stop it from leaking by accident and
 * potentially being compromised. The processor version below is
 * arguably better as the key is left in the header when you use
 * the DSL leaks the fact that camel encryption was used.
 */
from("direct:key-in-header-encrypt")
    .marshal(cryptoFormat)
    .removeHeader(CryptoDataFormat.KEY)
    .to("mock:encrypted");

from("direct:key-in-header-decrypt").unmarshal(cryptoFormat).process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getIn().getHeaders().remove(CryptoDataFormat.KEY);
        exchange.getOut().copyFrom(exchange.getIn());
    }
}).to("mock:unencrypted");
```

or with spring.

```
<crypto id="nokey" algorithm="DES" />
```

73.7. DEPENDENCIES

To use the Crypto dataformat in your camel routes you need to add the following dependency to your pom.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

73.8. SEE ALSO

- Data Format
- Crypto (Digital Signatures)
- http://www.bouncycastle.org/java.html

CHAPTER 74. CSV DATAFORMAT

Available as of Camel version 1.3

The CSV Data Format uses Apache Commons CSV to handle CSV payloads (Comma Separated Values) such as those exported/imported by Excel.

74.1. OPTIONS

The CSV dataformat supports 28 options which are listed below.

Name	Default	Java Type	Description
formatRef		**String**	The reference format to use, it will be updated with the other format options, the default value is CSVFormat.DEFAULT
formatName		**String**	The name of the format to use, the default value is CSVFormat.DEFAULT
commentMarkerDisabled	**false**	**Boolean**	Disables the comment marker of the reference format.
commentMarker		**String**	Sets the comment marker of the reference format.
delimiter		**String**	Sets the delimiter to use. The default value is , (comma)
escapeDisabled	**false**	**Boolean**	Use for disabling using escape character
escape		**String**	Sets the escape character to use
headerDisabled	**false**	**Boolean**	Use for disabling headers
header		**List**	To configure the CSV headers
allowMissingColumnNames	**false**	**Boolean**	Whether to allow missing column names.
ignoreEmptyLines	**false**	**Boolean**	Whether to ignore empty lines.
ignoreSurroundingSpaces	**false**	**Boolean**	Whether to ignore surrounding spaces
nullStringDisabled	**false**	**Boolean**	Used to disable null strings

Name	Default	Java Type	Description
nullString		String	Sets the null string
quoteDisabled	false	Boolean	Used to disable quotes
quote		String	Sets the quote which by default is
recordSeparatorDisabled		String	Used for disabling record separator
recordSeparator		String	Sets the record separator (aka new line) which by default is new line characters (CRLF)
skipHeaderRecord	false	Boolean	Whether to skip the header record in the output
quoteMode		String	Sets the quote mode
ignoreHeaderCase	false	Boolean	Sets whether or not to ignore case when accessing header names.
trim	false	Boolean	Sets whether or not to trim leading and trailing blanks.
trailingDelimiter	false	Boolean	Sets whether or not to add a trailing delimiter.
lazyLoad	false	Boolean	Whether the unmarshalling should produce an iterator that reads the lines on the fly or if all the lines must be read at one.
useMaps	false	Boolean	Whether the unmarshalling should produce maps (HashMap)for the lines values instead of lists. It requires to have header (either defined or collected).
useOrderedMaps	false	Boolean	Whether the unmarshalling should produce ordered maps (LinkedHashMap) for the lines values instead of lists. It requires to have header (either defined or collected).
recordConverterRef		String	Refers to a custom CsvRecordConverter to lookup from the registry to use.
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

74.2. MARSHALLING A MAP TO CSV

The component allows you to marshal a Java Map (or any other message type that can be converted in a Map) into a CSV payload.

Considering the following body

```
Map<String, Object> body = new LinkedHashMap<>();
body.put("foo", "abc");
body.put("bar", 123);
```

and this Java route definition

```
from("direct:start")
    .marshal().csv()
    .to("mock:result");
```

or this XML route definition

```
<route>
   <from uri="direct:start" />
   <marshal>
      <csv />
   </marshal>
   <to uri="mock:result" />
</route>
```

then it will produce

```
abc,123
```

74.3. UNMARSHALLING A CSV MESSAGE INTO A JAVA LIST

Unmarshalling will transform a CSV messsage into a Java List with CSV file lines (containing another List with all the field values).

An example: we have a CSV file with names of persons, their IQ and their current activity.

```
Jack Dalton, 115, mad at Averell
Joe Dalton, 105, calming Joe
William Dalton, 105, keeping Joe from killing Averell
Averell Dalton, 80, playing with Rantanplan
Lucky Luke, 120, capturing the Daltons
```

We can now use the CSV component to unmarshal this file:

```
from("file:src/test/resources/?fileName=daltons.csv&noop=true")
    .unmarshal().csv()
    .to("mock:daltons");
```

The resulting message will contain a **List<List<String>>** like...

```
List<List<String>> data = (List<List<String>>) exchange.getIn().getBody();
for (List<String> line : data) {
    LOG.debug(String.format("%s has an IQ of %s and is currently %s", line.get(0), line.get(1),
line.get(2)));
}
```

74.4. MARSHALLING A LIST<MAP> TO CSV

Available as of Camel 2.1

If you have multiple rows of data you want to be marshalled into CSV format you can now store the message payload as a **List<Map<String, Object>>** object where the list contains a Map for each row.

74.5. FILE POLLER OF CSV, THEN UNMARSHALING

Given a bean which can handle the incoming data...

MyCsvHandler.java

```
// Some comments here
public void doHandleCsvData(List<List<String>> csvData)
{
    // do magic here
}
```

i. your route then looks as follows

```
<route>
        <!-- poll every 10 seconds -->
        <from uri="file:///some/path/to/pickup/csvfiles?delete=true&amp;consumer.delay=10000" />
        <unmarshal><csv /></unmarshal>
        <to uri="bean:myCsvHandler?method=doHandleCsvData" />
</route>
```

74.6. MARSHALING WITH A PIPE AS DELIMITER

Considering the following body

```
Map<String, Object> body = new LinkedHashMap<>();
body.put("foo", "abc");
body.put("bar", 123);
```

and this Java route definition

```
// Camel version < 2.15
CsvDataFormat oldCSV = new CsvDataFormat();
oldCSV.setDelimiter("|");
from("direct:start")
    .marshal(oldCSV)
    .to("mock:result")

// Camel version >= 2.15
```

```
from("direct:start")
    .marshal(new CsvDataFormat().setDelimiter(&#39;|&#39;))
    .to("mock:result")
```

or this XML route definition

```
<route>
  <from uri="direct:start" />
  <marshal>
    <csv delimiter="|" />
  </marshal>
  <to uri="mock:result" />
</route>
```

then it will produce

```
abc|123
```

Using autogenColumns, configRef and strategyRef attributes inside XML # DSL

Available as of Camel 2.9.2 / 2.10 and deleted for Camel 2.15

You can customize the CSV Data Format to make use of your own **CSVConfig** and/or **CSVStrategy**. Also note that the default value of the **autogenColumns** option is true. The following example should illustrate this customization.

```
<route>
  <from uri="direct:start" />
  <marshal>
    <!-- make use of a strategy other than the default one which is
'org.apache.commons.csv.CSVStrategy.DEFAULT_STRATEGY' -->
    <csv autogenColumns="false" delimiter="|" configRef="csvConfig" strategyRef="excelStrategy" />
  </marshal>
  <convertBodyTo type="java.lang.String" />
  <to uri="mock:result" />
</route>

<bean id="csvConfig" class="org.apache.commons.csv.writer.CSVConfig">
  <property name="fields">
    <list>
      <bean class="org.apache.commons.csv.writer.CSVField">
        <property name="name" value="orderId" />
      </bean>
      <bean class="org.apache.commons.csv.writer.CSVField">
        <property name="name" value="amount" />
      </bean>
    </list>
  </property>
</bean>

<bean id="excelStrategy"
class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
  <property name="staticField" value="org.apache.commons.csv.CSVStrategy.EXCEL_STRATEGY"
/>
</bean>
```

74.7. USING SKIPFIRSTLINE OPTION WHILE UNMARSHALING

Available as of Camel 2.10 and deleted for Camel 2.15

You can instruct the CSV Data Format to skip the first line which contains the CSV headers. Using the Spring/XML DSL:

```
<route>
  <from uri="direct:start" />
  <unmarshal>
    <csv skipFirstLine="true" />
  </unmarshal>
  <to uri="bean:myCsvHandler?method=doHandleCsv" />
</route>
```

Or the Java DSL:

```
CsvDataFormat csv = new CsvDataFormat();
csv.setSkipFirstLine(true);

from("direct:start")
  .unmarshal(csv)
.to("bean:myCsvHandler?method=doHandleCsv");
```

74.8. UNMARSHALING WITH A PIPE AS DELIMITER

Using the Spring/XML DSL:

```
<route>
  <from uri="direct:start" />
  <unmarshal>
    <csv delimiter="|" />
  </unmarshal>
  <to uri="bean:myCsvHandler?method=doHandleCsv" />
</route>
```

Or the Java DSL:

```
CsvDataFormat csv = new CsvDataFormat();
CSVStrategy strategy = CSVStrategy.DEFAULT_STRATEGY;
strategy.setDelimiter('|');
csv.setStrategy(strategy);

from("direct:start")
  .unmarshal(csv)
  .to("bean:myCsvHandler?method=doHandleCsv");
```

```
CsvDataFormat csv = new CsvDataFormat();
csv.setDelimiter("|");

from("direct:start")
  .unmarshal(csv)
  .to("bean:myCsvHandler?method=doHandleCsv");
```

```
CsvDataFormat csv = new CsvDataFormat();
CSVConfig csvConfig = new CSVConfig();
csvConfig.setDelimiter(";");
csv.setConfig(csvConfig);

from("direct:start")
  .unmarshal(csv)
  .to("bean:myCsvHandler?method=doHandleCsv");
```

Issue in CSVConfig

It looks like that

```
CSVConfig csvConfig = new CSVConfig();
csvConfig.setDelimiter(';');
```

doesn't work. You have to set the delimiter as a String!

74.9. DEPENDENCIES

To use CSV in your Camel routes you need to add a dependency on **camel-csv**, which implements this data format.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest and greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-csv</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 75. CXF

CXF COMPONENT

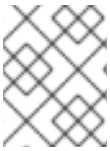
The **cxf:** component provides integration with Apache CXF for connecting to JAX-WS services hosted in CXF.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cxf</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

NOTE

If you want to learn about CXF dependencies, see the **WHICH-JARS** text file.



NOTE

When using CXF in streaming modes (see DataFormat option), then also read about Stream caching.

CAMEL ON EAP DEPLOYMENT

This component is supported by the Camel on EAP (Wildfly Camel) framework, which offers a simplified deployment model on the Red Hat JBoss Enterprise Application Platform (JBoss EAP) container.

The CXF component integrates with the JBoss EAP **webservices** susbsystem that also uses Apache CXF. For more information, see JAX-WS.

NOTE

At present, the Camel on EAP subsystem does **not** support CXF or Restlet consumers. However, it is possible to mimic CXF consumer behaviour, using the **CamelProxy**.

URI FORMAT

```
cxf:bean:cxfEndpoint[?options]
```

Where **cxfEndpoint** represents a bean ID that references a bean in the Spring bean registry. With this URI format, most of the endpoint details are specified in the bean definition.

```
cxf://someAddress[?options]
```

Where **someAddress** specifies the CXF endpoint's address. With this URI format, most of the endpoint details are specified using options.

For either style above, you can append options to the URI as follows:

```
cxf:bean:cxfEndpoint?wsdlURL=wsdl/hello_world.wsdl&dataFormat=PAYLOAD
```

OPTIONS

Name	Required	Description
wsdlURL	No	The location of the WSDL. WSDL is obtained from endpoint address by default. For example: **file://local/wsdl/hello.wsdl** or **wsdl/hello.wsdl**
serviceClass	Yes	The name of the SEI (Service Endpoint Interface) class. This class can have, but does not require, JSR181 annotations. **Since 2.0,** this option is only required by POJO mode. If the wsdlURL option is provided, serviceClass is not required for PAYLOAD and MESSAGE mode. When wsdlURL option is used without serviceClass, the serviceName and portName (endpointName for Spring configuration) options **MUST** be provided. Since 2.0, it is possible to use **\#** notation to reference a **serviceClass** object instance from the registry.. **Please be advised that the referenced object cannot be a Proxy (Spring AOP Proxy is OK)** as it relies on **Object.getClass().getName()** method for non Spring AOP Proxy. **Since 2.8,** it is possible to omit both wsdlURL and serviceClass options for PAYLOAD and MESSAGE mode. When they are omitted, arbitrary XML elements can be put in CxfPayload's body in PAYLOAD mode to facilitate CXF Dispatch Mode. For example: **org.apache.camel.Hello**

serviceName	Only if more than one **serviceName** present in WSDL	The service name this service is implementing, it maps to the **wsdl:service@name**. For example: **{http://org.apache.camel}ServiceName**
endpointName	Only if more than one **portName** under the **serviceName** is present, and it is required for camel-cxf consumer since camel 2.2	The port name this service is implementing, it maps to the **wsdl:port@name**. For example: **{http://org.apache.camel}PortName**
dataFormat	No	Which message data format the CXF endpoint supports. Possible values are: **POJO (default)**, **PAYLOAD**, **MESSAGE**.
relayHeaders	No	Please see the **Description of relayHeaders option** section for this option. Should a CXF endpoint relay headers along the route. Currently only available when **dataFormat=POJO** Default: **true** Example: **true**, **false**
wrapped	No	Which kind of operation the CXF endpoint producer will invoke. Possible values are: **true**, **false (default)**.
wrappedStyle	No	**Since 2.5.0** The WSDL style that describes how parameters are represented in the SOAP body. If the value is **false**, CXF will chose the document-literal unwrapped style, If the value is **true**, CXF will chose the document-literal wrapped style
setDefaultBus	No	**Deprecated:** Specifies whether or not to use the default CXF bus for this endpoint. Possible values are: **true**, **false (default)**. This option is deprecated and you should use **defaultBus** from Camel 2.16 onwards.

defaultBus	No	**Deprecated:** Specifies whether or not to use the default CXF bus for this endpoint. Possible values are: **true**, **false** (default). This option is deprecated and you should use **defaultBus** from Camel 2.16 onwards.
bus	No	Use **\#** notation to reference a bus object from the registry – for example, **bus=\#busName**. The referenced object must be an instance of **org.apache.cxf.Bus**. By default, uses the default bus created by CXF Bus Factory.
cxfBinding	No	Use **\#** notation to reference a CXF binding object from the registry – for example, **cxfBinding=\#bindingName**. The referenced object must be an instance of **org.apache.camel.component.cxf.CxfBinding**.
headerFilterStrategy	No	Use **\#** notation to reference a header filter strategy object from the registry – for example, **headerFilterStrategy=\#strategyName**. The referenced object must be an instance of **org.apache.camel.spi.HeaderFilterStrategy**.
loggingFeatureEnabled	No	New in 2.3, this option enables CXF Logging Feature which writes inbound and outbound SOAP messages to log. Possible values are: **true**, **false** (default).
defaultOperationName	No	New in 2.4, this option will set the default **operationName** that will be used by the **CxfProducer** that invokes the remote service. For example: **defaultOperationName=greetMe**

defaultOperationNamespace	No	New in 2.4, this option will set the default operationNamespace that will be used by the CxfProducer which invokes the remote service. For example: **defaultOperationNamespace=http://apache.org/hello_world_soap_http**
synchronous	No	New in 2.5, this option will let CXF endpoint decide to use sync or async API to do the underlying work. The default value is **false**, which means **camel-cxf** endpoint will try to use async API by default.
publishedEndpointUrl	No	New in 2.5, this option overrides the endpoint URL that appears in the published WSDL that is accessed using the service address URL plus **?wsdl**. For example: **publishedEndpointUrl=http://example.com/service**
properties.propName	No	**Camel 2.8:** Allows you to set custom CXF properties in the endpoint URI. For example, setting **properties.mtom-enabled=true** to enable MTOM. To make sure that CXF does not switch the thread when starting the invocation, you can set **properties.org.apache.cxf.interceptor.OneWayProcessorInterceptor.USE_ORIGINAL_THREAD=true**.
allowStreaming	No	New in 2.8.2. This option controls whether the CXF component, when running in PAYLOAD mode (see below), will DOM parse the incoming messages into DOM Elements or keep the payload as a javax.xml.transform.Source object that would allow streaming in some cases.

skipFaultLogging	No	New in 2.11. This option controls whether the PhaseInterceptorChain skips logging the Fault that it catches.
cxfEndpointConfigurer	No	New in **Camel 2.11**. This option could apply the implementation of **org.apache.camel.component.cxf.CxfEndpointConfigurer which** supports to configure the CXF endpoint in programmatic way. Since **Camel 2.15.0,** user can configure the CXF server and client by implementing configure{Server/Client} method of **CxfEndpointConfigurer**.
username	No	New in **Camel 2.12.3** This option is used to set the basic authentication information of username for the CXF client.
password	No	New in **Camel 2.12.3** This option is used to set the basic authentication information of password for the CXF client.
continuationTimeout	No	New in **Camel 2.14.0** This option is used to set the CXF continuation timeout which could be used in CxfConsumer by default when the CXF server is using Jetty or Servlet transport. (Before **Camel 2.14.0**, CxfConsumer just set the continuation timeout to be 0, which means the continuation suspend operation never timeout.) **Default**: 30000 **Example**: continuation=80000

The **serviceName** and **portName** are QNames, so if you provide them be sure to prefix them with their **{namespace}** as shown in the examples above.

THE DESCRIPTIONS OF THE DATAFORMATS

DataFormat	Description

POJO	POJOs (plain old Java objects) are the Java parameters to the method being invoked on the target server. Both Protocol and Logical JAX-WS handlers are supported.
PAYLOAD	**PAYLOAD** is the message payload (the contents of the **soap:body**) after message configuration in the CXF endpoint is applied. Only Protocol JAX-WS handler is supported. Logical JAX-WS handler is not supported.
MESSAGE	**MESSAGE** is the raw message that is received from the transport layer. It is not suppose to touch or change Stream, some of the CXF interceptors will be removed if you are using this kind of DataFormat so you can't see any soap headers after the camel-cxf consumer and JAX-WS handler is not supported.
CXF_MESSAGE	New in **Camel 2.8.2**, **CXF_MESSAGE** allows for invoking the full capabilities of CXF interceptors by converting the message from the transport layer into a raw SOAP message

You can determine the data format mode of an exchange by retrieving the exchange property, **CamelCXFDataFormat**. The exchange key constant is defined in **org.apache.camel.component.cxf.CxfConstants.DATA_FORMAT_PROPERTY**.

CONFIGURING THE CXF ENDPOINTS WITH APACHE ARIES BLUEPRINT.

Since Camel 2.8, there is support for using Aries blueprint dependency injection for your CXF endpoints. The schema is very similar to the Spring schema, so the transition is fairly transparent.

For example:

```
 <blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
          xmlns:camel-cxf="http://camel.apache.org/schema/blueprint/cxf"
     xmlns:cxfcore="http://cxf.apache.org/blueprint/core"
          xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

       <camel-cxf:cxfEndpoint id="routerEndpoint"
                  address="http://localhost:9001/router"
                  serviceClass="org.apache.servicemix.examples.cxf.HelloWorld">
         <camel-cxf:properties>
            <entry key="dataFormat" value="MESSAGE"/>
         </camel-cxf:properties>
       </camel-cxf:cxfEndpoint>

       <camel-cxf:cxfEndpoint id="serviceEndpoint"
```

```
    address="http://localhost:9000/SoapContext/SoapPort"
                  serviceClass="org.apache.servicemix.examples.cxf.HelloWorld">
    </camel-cxf:cxfEndpoint>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
       <route>
          <from uri="routerEndpoint"/>
          <to uri="log:request"/>
       </route>
    </camelContext>

</blueprint>
```

Currently the endpoint element is the first supported CXF namespacehandler.

You can also use the bean references just as in spring

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
          xmlns:jaxws="http://cxf.apache.org/blueprint/jaxws"
          xmlns:cxf="http://cxf.apache.org/blueprint/core"
          xmlns:camel="http://camel.apache.org/schema/blueprint"
          xmlns:camelcxf="http://camel.apache.org/schema/blueprint/cxf"
          xsi:schemaLocation="
            http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
            http://cxf.apache.org/blueprint/jaxws http://cxf.apache.org/schemas/blueprint/jaxws.xsd
            http://cxf.apache.org/blueprint/core http://cxf.apache.org/schemas/blueprint/core.xsd
            ">

    <camelcxf:cxfEndpoint id="reportIncident"
                  address="/camel-example-cxf-blueprint/webservices/incident"
                  wsdlURL="META-INF/wsdl/report_incident.wsdl"
                  serviceClass="org.apache.camel.example.reportincident.ReportIncidentEndpoint">
    </camelcxf:cxfEndpoint>

    <bean id="reportIncidentRoutes"
class="org.apache.camel.example.reportincident.ReportIncidentRoutes" />

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
       <routeBuilder ref="reportIncidentRoutes"/>
    </camelContext>

</blueprint>
```

HOW TO ENABLE CXF'S LOGGINGOUTINTERCEPTOR IN MESSAGE MODE

CXF's **LoggingOutInterceptor** outputs outbound message that goes on the wire to logging system (**java.util.logging**). Since the **LoggingOutInterceptor** is in **PRE_STREAM** phase (but **PRE_STREAM** phase is removed in **MESSAGE** mode), you have to configure **LoggingOutInterceptor** to be run during the **WRITE** phase. The following is an example.

```
  <bean id="loggingOutInterceptor" class="org.apache.cxf.interceptor.LoggingOutInterceptor">
      <!--  it really should have been user-prestream but CXF does have such phase! -->
      <constructor-arg value="target/write"/>
  </bean>

<cxf:cxfEndpoint id="serviceEndpoint" address="http://localhost:9002/helloworld"
 serviceClass="org.apache.camel.component.cxf.HelloService">
 <cxf:outInterceptors>
    <ref bean="loggingOutInterceptor"/>
 </cxf:outInterceptors>
 <cxf:properties>
  <entry key="dataFormat" value="MESSAGE"/>
 </cxf:properties>
</cxf:cxfEndpoint>
```

DESCRIPTION OF RELAYHEADERS OPTION

There are **in-band** and **out-of-band** on-the-wire headers from the perspective of a JAXWS WSDL-first developer.

The **in-band** headers are headers that are explicitly defined as part of the WSDL binding contract for an endpoint such as SOAP headers.

The **out-of-band** headers are headers that are serialized over the wire, but are not explicitly part of the WSDL binding contract.

Headers relaying/filtering is bi-directional.

When a route has a CXF endpoint and the developer needs to have on-the-wire headers, such as SOAP headers, be relayed along the route to be consumed say by another JAXWS endpoint, then **relayHeaders** should be set to **true**, which is the default value.

AVAILABLE ONLY IN POJO MODE

The **relayHeaders=true** setting expresses an intent to relay the headers. The actual decision on whether a given header is relayed is delegated to a pluggable instance that implements the **MessageHeadersRelay** interface. A concrete implementation of **MessageHeadersRelay** will be consulted to decide if a header needs to be relayed or not. There is already an implementation of **SoapMessageHeadersRelay** which binds itself to well-known SOAP name spaces. Currently only out-of-band headers are filtered, and in-band headers will always be relayed when **relayHeaders=true**. If there is a header on the wire, whose name space is unknown to the runtime, then a fall back **DefaultMessageHeadersRelay** will be used, which simply allows all headers to be relayed.

The **relayHeaders=false** setting asserts that all headers, in-band and out-of-band, will be dropped.

You can plugin your own **MessageHeadersRelay** implementations overriding or adding additional ones to the list of relays. In order to override a preloaded relay instance just make sure that your **MessageHeadersRelay** implementation services the same name spaces as the one you looking to override. Also note, that the overriding relay has to service all of the name spaces as the one you looking to override, or else a runtime exception on route start up will be thrown as this would introduce an ambiguity in name spaces to relay instance mappings.

```
<cxf:cxfEndpoint ...>
  <cxf:properties>
```

```
    <entry key="org.apache.camel.cxf.message.headers.relays">
      <list>
        <ref bean="customHeadersRelay"/>
      </list>
    </entry>
  </cxf:properties>
 </cxf:cxfEndpoint>
 <bean id="customHeadersRelay"
class="org.apache.camel.component.cxf.soap.headers.CustomHeadersRelay"/>
```

Take a look at the tests that show how you'd be able to relay/drop headers here:

link:https://svn.apache.org/repos/asf/camel/branches/camel-1.x/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest.java[https://svn.apache.org/repos/asf/camel/branches/camel-1.x/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTest.java]

CHANGES SINCE RELEASE 2.0

- **POJO** and **PAYLOAD** modes are supported. In **POJO** mode, only out-of-band message headers are available for filtering as the in-band headers have been processed and removed from the header list by CXF. The in-band headers are incorporated into the **MessageContentList** in **POJO** mode. The **camel-cxf** component does make any attempt to remove the in-band headers from the **MessageContentList** If filtering of in-band headers is required, please use **PAYLOAD** mode or plug in a (pretty straightforward) CXF interceptor/JAXWS Handler to the CXF endpoint.

- The Message Header Relay mechanism has been merged into **CxfHeaderFilterStrategy**. The **relayHeaders** option, its semantics, and default value remain the same, but it is a property of **CxfHeaderFilterStrategy**. Here is an example of configuring it.

  ```
  <bean id="dropAllMessageHeadersStrategy"
  class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrategy">

     <!--  Set relayHeaders to false to drop all SOAP headers -->
     <property name="relayHeaders" value="false"/>

  </bean>
  ```

  Then, your endpoint can reference the **CxfHeaderFilterStrategy**.

  ```
  <route>
     <from uri="cxf:bean:routerNoRelayEndpoint?
  headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
     <to uri="cxf:bean:serviceNoRelayEndpoint?
  headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
  </route>
  ```

- The **MessageHeadersRelay** interface has changed slightly and has been renamed to **MessageHeaderFilter**. It is a property of **CxfHeaderFilterStrategy**. Here is an example of configuring user defined Message Header Filters:

  ```
  <bean id="customMessageFilterStrategy"
  class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrategy">
  ```

```
    <property name="messageHeaderFilters">
      <list>
        <!--  SoapMessageHeaderFilter is the built in filter.  It can be removed by omitting it. -
->
        <bean
class="org.apache.camel.component.cxf.common.header.SoapMessageHeaderFilter"/>

        <!--  Add custom filter here -->
        <bean class="org.apache.camel.component.cxf.soap.headers.CustomHeaderFilter"/>
      </list>
    </property>
</bean>
```

- Other than **relayHeaders**, there are new properties that can be configured in **CxfHeaderFilterStrategy**.

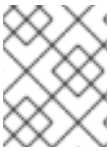
Name	Description	type	Required?	Default value
relayHeaders	All message headers will be processed by Message Header Filters	**boolean**	No	**true** (1.6.1 behavior)
relayAllMessageHeaders	All message headers will be propagated (without processing by Message Header Filters)	**boolean**	No	**false** (1.6.1 behavior)
allowFilterNamespaceClash	If two filters overlap in activation namespace, the property control how it should be handled. If the value is **true**, last one wins. If the value is **false**, it will throw an exception	**boolean**	No	**false** (1.6.1 behavior)

CONFIGURE THE CXF ENDPOINTS WITH SPRING

You can configure the CXF endpoint with the Spring configuration file shown below, and you can also embed the endpoint into the **camelContext** tags. When you are invoking the service endpoint, you can set the **operationName** and **operationNamespace** headers to explicitly state which operation you are calling.

NOTE In Camel 2.x we change to use **http://camel.apache.org/schema/cxf** as the CXF endpoint's target namespace.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://camel.apache.org/schema/cxf"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
">
 ...
```



NOTE

In Apache Camel 2.x, the **http://activemq.apache.org/camel/schema/cxfEndpoint** namespace was changed to **http://camel.apache.org/schema/cxf**.

Be sure to include the JAX-WS **schemaLocation** attribute specified on the root beans element. This allows CXF to validate the file and is required. Also note the namespace declarations at the end of the **<cxf:cxfEndpoint/>** tag—these are required because the combined **{namespace}localName** syntax is presently not supported for this tag's attribute values.

The **cxf:cxfEndpoint** element supports many additional attributes:

Name	Value
PortName	The endpoint name this service is implementing, it maps to the **wsdl:port@name**. In the format of **ns:PORT_NAME** where **ns** is a namespace prefix valid at this scope.
serviceName	The service name this service is implementing, it maps to the **wsdl:service@name**. In the format of **ns:SERVICE_NAME** where **ns** is a namespace prefix valid at this scope.
wsdlURL	The location of the WSDL. Can be on the classpath, file system, or be hosted remotely.
bindingId	The **bindingId** for the service model to use.
address	The service publish address.
bus	The bus name that will be used in the JAX-WS endpoint.
serviceClass	The class name of the SEI (Service Endpoint Interface) class which could have JSR181 annotation or not.

It also supports many child elements:

Name	Value
cxf:inInterceptors	The incoming interceptors for this endpoint. A list of **<bean>** or **<ref>**.
cxf:inFaultInterceptors	The incoming fault interceptors for this endpoint. A list of **<bean>** or **<ref>**.
cxf:outInterceptors	The outgoing interceptors for this endpoint. A list of **<bean>** or **<ref>**.
cxf:outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of **<bean>** or **<ref>**.
cxf:properties	A properties map which should be supplied to the JAX-WS endpoint. See below.
cxf:handlers	A JAX-WS handler list which should be supplied to the JAX-WS endpoint. See below.
cxf:dataBinding	You can specify the which **DataBinding** will be use in the endpoint. This can be supplied using the Spring **<bean class="MyDataBinding"/>** syntax.
cxf:binding	You can specify the **BindingFactory** for this endpoint to use. This can be supplied using the Spring **<bean class="MyBindingFactory"/>** syntax.
cxf:features	The features that hold the interceptors for this endpoint. A list of **<bean>**s or **<ref>**s
cxf:schemaLocations	The schema locations for endpoint to use. A list of **<schemaLocation>**s
cxf:serviceFactory	The service factory for this endpoint to use. This can be supplied using the Spring **<bean class="MyServiceFactory"/>** syntax

You can find more advanced examples which show how to provide interceptors, properties and handlers here: http://cwiki.apache.org/CXF20DOC/jax-ws-configuration.html

NOTE

You can use CXF:properties to set the CXF endpoint's **dataFormat** and **setDefaultBus** properties from a Spring configuration file, as follows:

```
<cxf:cxfEndpoint id="testEndpoint" address="http://localhost:9000/router"
    serviceClass="org.apache.camel.component.cxf.HelloService"
    endpointName="s:PortName"
    serviceName="s:ServiceName"
    xmlns:s="http://www.example.com/test">
    <cxf:properties>
      <entry key="dataFormat" value="MESSAGE"/>
      <entry key="setDefaultBus" value="true"/>
    </cxf:properties>
  </cxf:cxfEndpoint>
```

HOW TO MAKE THE CAMEL-CXF COMPONENT USE LOG4J INSTEAD OF JAVA.UTIL.LOGGING

CXF's default logger is **java.util.logging**. If you want to change it to **log4j**, proceed as follows. Create a file, in the classpath, named **META-INF/cxf/org.apache.cxf.logger**. This file should contain the fully-qualified name of the class, **org.apache.cxf.common.logging.Log4jLogger**, with no comments, on a single line.

HOW TO LET CAMEL-CXF RESPONSE MESSAGE WITH XML START DOCUMENT

If you are using some SOAP client such as PHP, you will get this kind of error, because CXF doesn't add the XML start document **<?xml version="1.0" encoding="utf-8"?>**.

```
Error:sendSms: SoapFault exception: [Client] looks like we got no XML document in [...]
```

To resolved this issue, you just need to tell StaxOutInterceptor to write the XML start document for you.

```
public class WriteXmlDeclarationInterceptor extends AbstractPhaseInterceptor<SoapMessage> {
    public WriteXmlDeclarationInterceptor() {
        super(Phase.PRE_STREAM);
        addBefore(StaxOutInterceptor.class.getName());
    }

    public void handleMessage(SoapMessage message) throws Fault {
        message.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
    }

}
```

You can add a customer interceptor like this and configure it into you **camel-cxf** endpont

```
<cxf:cxfEndpoint id="routerEndpoint"
address="http://localhost:${CXFTestSupport.port2}/CXFGreeterRouterTest/CamelContext/RouterPort"

    serviceClass="org.apache.hello_world_soap_http.GreeterImpl"
    skipFaultLogging="true">
```

```
    <cxf:outInterceptors>
      <!-- This interceptor will force the CXF server send the XML start document to client -->
      <bean class="org.apache.camel.component.cxf.WriteXmlDeclarationInterceptor"/>
    </cxf:outInterceptors>
    <cxf:properties>
      <!-- Set the publishedEndpointUrl which could override the service address from generated
WSDL as you want -->
      <entry key="publishedEndpointUrl" value="http://www.simple.com/services/test" />
    </cxf:properties>
  </cxf:cxfEndpoint>
```

Or adding a message header for it like this if you are using **Camel 2.4**.

```
// set up the response context which force start document
Map<String, Object> map = new HashMap<String, Object>();
map.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
exchange.getOut().setHeader(Client.RESPONSE_CONTEXT, map);
```

HOW TO CONSUME A MESSAGE FROM A CAMEL-CXF ENDPOINT IN POJO DATA FORMAT

The **camel-cxf** endpoint consumer **POJO** data format is based on the cxf invoker, so the message header has a property with the name of **CxfConstants.OPERATION_NAME** and the message body is a list of the SEI method parameters.

```
public class PersonProcessor implements Processor {

    private static final transient Logger LOG = LoggerFactory.getLogger(PersonProcessor.class);

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        LOG.info("processing exchange in camel");

        BindingOperationInfo boi =
(BindingOperationInfo)exchange.getProperty(BindingOperationInfo.class.toString());
        if (boi != null) {
            LOG.info("boi.isUnwrapped" + boi.isUnwrapped());
        }
        // Get the parameters list which element is the holder.
        MessageContentsList msgList = (MessageContentsList)exchange.getIn().getBody();
        Holder<String> personId = (Holder<String>)msgList.get(0);
        Holder<String> ssn = (Holder<String>)msgList.get(1);
        Holder<String> name = (Holder<String>)msgList.get(2);

        if (personId.value == null || personId.value.length() == 0) {
            LOG.info("person id 123, so throwing exception");
            // Try to throw out the soap fault message
            org.apache.camel.wsdl_first.types.UnknownPersonFault personFault =
                new org.apache.camel.wsdl_first.types.UnknownPersonFault();
            personFault.setPersonId("");
            org.apache.camel.wsdl_first.UnknownPersonFault fault =
                new org.apache.camel.wsdl_first.UnknownPersonFault("Get the null value of person name",
personFault);
            // Since camel has its own exception handler framework, we can't throw the exception to
```

```
trigger it
            // We just set the fault message in the exchange for camel-cxf component handling and return
            exchange.getOut().setFault(true);
            exchange.getOut().setBody(fault);
            return;
        }

        name.value = "Bonjour";
        ssn.value = "123";
        LOG.info("setting Bonjour as the response");
        // Set the response message, first element is the return value of the operation,
        // the others are the holders of method parameters
        exchange.getOut().setBody(new Object[] {null, personId, ssn, name});
    }

}
```

HOW TO PREPARE THE MESSAGE FOR THE CAMEL-CXF ENDPOINT IN POJO DATA FORMAT

The **camel-cxf** endpoint producer is based on the cxf client API. First you need to specify the operation name in the message header, then add the method parameters to a list, and initialize the message with this parameter list. The response message's body is a **messageContentsList**, you can get the result from that list.

If you don't specify the operation name in the message header, **CxfProducer** will try to use the **defaultOperationName** from **CxfEndpoint**. If there is no **defaultOperationName** set on **CxfEndpoint**, it will pick up the first operation name from the operation list.

If you want to get the object array from the message body, you can get the body using **message.getbody(Object[].class)**, as follows:

```
Exchange senderExchange = new DefaultExchange(context, ExchangePattern.InOut);
final List<String> params = new ArrayList<String>();
// Prepare the request message for the camel-cxf procedure
params.add(TEST_MESSAGE);
senderExchange.getIn().setBody(params);
senderExchange.getIn().setHeader(CxfConstants.OPERATION_NAME, ECHO_OPERATION);

Exchange exchange = template.send("direct:EndpointA", senderExchange);

org.apache.camel.Message out = exchange.getOut();
// The response message's body is an MessageContentsList which first element is the return value of
the operation,
// If there are some holder parameters, the holder parameter will be filled in the reset of List.
// The result will be extract from the MessageContentsList with the String class type
MessageContentsList result = (MessageContentsList)out.getBody();
LOG.info("Received output text: " + result.get(0));
Map<String, Object> responseContext = CastUtils.cast((Map<?, ?
>)out.getHeader(Client.RESPONSE_CONTEXT));
assertNotNull(responseContext);
assertEquals("We should get the response context here", "UTF-8",
responseContext.get(org.apache.cxf.message.Message.ENCODING));
assertEquals("Reply body on Camel is wrong", "echo " + TEST_MESSAGE, result.get(0));
```

HOW TO DEAL WITH THE MESSAGE FOR A CAMEL-CXF ENDPOINT IN PAYLOAD DATA FORMAT

In Apache Camel 2.0: **CxfMessage.getBody()** will return an **org.apache.camel.component.cxf.CxfPayload** object, which has getters for SOAP message headers and Body elements. This change enables decoupling the native CXF message from the Apache Camel message.

```
protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from(SIMPLE_ENDPOINT_URI + "&dataFormat=PAYLOAD").to("log:info").process(new
Processor() {
                @SuppressWarnings("unchecked")
                public void process(final Exchange exchange) throws Exception {
                    CxfPayload<SoapHeader> requestPayload =
exchange.getIn().getBody(CxfPayload.class);
                    List<Source> inElements = requestPayload.getBodySources();
                    List<Source> outElements = new ArrayList<Source>();
                    // You can use a customer toStringConverter to turn a CxfPayLoad message into String
as you want
                    String request = exchange.getIn().getBody(String.class);
                    XmlConverter converter = new XmlConverter();
                    String documentString = ECHO_RESPONSE;

                    Element in = new XmlConverter().toDOMElement(inElements.get(0));
                    // Just check the element namespace
                    if (!in.getNamespaceURI().equals(ELEMENT_NAMESPACE)) {
                        throw new IllegalArgumentException("Wrong element namespace");
                    }
                    if (in.getLocalName().equals("echoBoolean")) {
                        documentString = ECHO_BOOLEAN_RESPONSE;
                        checkRequest("ECHO_BOOLEAN_REQUEST", request);
                    } else {
                        documentString = ECHO_RESPONSE;
                        checkRequest("ECHO_REQUEST", request);
                    }
                    Document outDocument = converter.toDOMDocument(documentString);
                    outElements.add(new DOMSource(outDocument.getDocumentElement()));
                    // set the payload header with null
                    CxfPayload<SoapHeader> responsePayload = new CxfPayload<SoapHeader>(null,
outElements, null);
                    exchange.getOut().setBody(responsePayload);
                }
            });
        }
    };
}
```

HOW TO GET AND SET SOAP HEADERS IN POJO MODE

POJO means that the data format is a **list of Java objects** when the CXF endpoint produces or consumes Camel exchanges. Even though Apache Camel exposes the message body as POJOs in this mode, the CXF component still provides access to read and write SOAP headers. However, since CXF

interceptors remove in-band SOAP headers from the header list after they have been processed, only out-of-band SOAP headers are available in POJO mode.

The following example illustrates how to get/set SOAP headers. Suppose we have a route that forwards from one CXF endpoint to another. That is, SOAP Client → Apache Camel → CXF service. We can attach two processors to obtain/insert SOAP headers at (1) before request goes out to the CXF service and (2) before response comes back to the SOAP Client. Processor (1) and (2) in this example are InsertRequestOutHeaderProcessor and InsertResponseOutHeaderProcessor. Our route looks like this:

```
<route>
   <from uri="cxf:bean:routerRelayEndpointWithInsertion"/>
   <process ref="InsertRequestOutHeaderProcessor" />
   <to uri="cxf:bean:serviceRelayEndpointWithInsertion"/>
   <process ref="InsertResponseOutHeaderProcessor" />
</route>
```

In 2.x SOAP headers are propagated to and from Apache Camel Message headers. The Apache Camel message header name is **org.apache.cxf.headers.Header.list**, which is a constant defined in CXF (**org.apache.cxf.headers.Header.HEADER_LIST**). The header value is a **List<>** of CXF **SoapHeader** objects (**org.apache.cxf.binding.soap.SoapHeader**). The following snippet is the **InsertResponseOutHeaderProcessor** (that inserts a new SOAP header in the response message). The way to access SOAP headers in both **InsertResponseOutHeaderProcessor** and **InsertRequestOutHeaderProcessor** are actually the same. The only difference between the two processors is setting the direction of the inserted SOAP header.

```
public static class InsertResponseOutHeaderProcessor implements Processor {

   @SuppressWarnings("unchecked")
   public void process(Exchange exchange) throws Exception {
      // You should be able to get the header if exchange is routed from camel-cxf endpoint
      List<SoapHeader> soapHeaders = CastUtils.cast((List<?
>)exchange.getIn().getHeader(Header.HEADER_LIST));
      if (soapHeaders == null) {
         // we just create a new soap headers in case the header is null
         soapHeaders = new ArrayList<SoapHeader>();
      }

      // Insert a new header
      String xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?><outofbandHeader "
         + "xmlns=\"http://cxf.apache.org/outofband/Header\" hdrAttribute=\"testHdrAttribute\" "
         + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\" soap:mustUnderstand=\"1\">"
         + "<name>New_testOobHeader</name><value>New_testOobHeaderValue</value>
</outofbandHeader>";
      SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
               DOMUtils.readXml(new StringReader(xml)).getDocumentElement());
      // make sure direction is OUT since it is a response message.
      newHeader.setDirection(Direction.DIRECTION_OUT);
      //newHeader.setMustUnderstand(false);
      soapHeaders.add(newHeader);

   }

}
```

HOW TO GET AND SET SOAP HEADERS IN PAYLOAD MODE

We have already shown how to access SOAP message (**CxfPayload** object) in **PAYLOAD** mode (see the section called "How to deal with the message for a camel-cxf endpoint in PAYLOAD data format").

Once you obtain a **CxfPayload** object, you can invoke the **CxfPayload.getHeaders()** method that returns a **List** of DOM Elements (SOAP headers).

```
from(getRouterEndpointURI()).process(new Processor() {
    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        CxfPayload<SoapHeader> payload = exchange.getIn().getBody(CxfPayload.class);
        List<Source> elements = payload.getBodySources();
        assertNotNull("We should get the elements here", elements);
        assertEquals("Get the wrong elements size", 1, elements.size());

        Element el = new XmlConverter().toDOMElement(elements.get(0));
        elements.set(0, new DOMSource(el));
        assertEquals("Get the wrong namespace URI", "http://camel.apache.org/pizza/types",
                el.getNamespaceURI());

        List<SoapHeader> headers = payload.getHeaders();
        assertNotNull("We should get the headers here", headers);
        assertEquals("Get the wrong headers size", headers.size(), 1);
        assertEquals("Get the wrong namespace URI",
                ((Element)(headers.get(0).getObject())).getNamespaceURI(),
                "http://camel.apache.org/pizza/types");
    }

})
.to(getServiceEndpointURI());
```

Since Camel 2.16.0, you can use the same approach as described in the section called "How to get and set SOAP headers in POJO mode" to set or get the SOAP headers. You can now use the **org.apache.cxf.headers.Header.list** header to get and set a list of SOAP headers. This means that if you have a route that forwards from one Camel CXF endpoint to another (SOAP Client → Camel → CXF service), the SOAP headers sent by the SOAP client are now also forwarded to the CXF service. If you do not want the headers to be forwarded, remove them from the **org.apache.cxf.headers.Header.list** Camel header.

SOAP HEADERS ARE NOT AVAILABLE IN MESSAGE MODE

SOAP headers are not available in **MESSAGE** mode as SOAP processing is skipped.

HOW TO THROW A SOAP FAULT FROM APACHE CAMEL

If you are using a CXF endpoint to consume the SOAP request, you may need to throw the SOAP **Fault** from the camel context. Basically, you can use the **throwFault** DSL to do that; it works for **POJO**, **PAYLOAD** and **MESSAGE** data format. You can define the soap fault like this:

```
SOAP_FAULT = new SoapFault(EXCEPTION_MESSAGE, SoapFault.FAULT_CODE_CLIENT);
Element detail = SOAP_FAULT.getOrCreateDetail();
Document doc = detail.getOwnerDocument();
```

```
Text tn = doc.createTextNode(DETAIL_TEXT);
detail.appendChild(tn);
```

Then throw it as you like:

```
from(routerEndpointURI).setFaultBody(constant(SOAP_FAULT));
```

If your CXF endpoint is working in the **MESSAGE** data format, you could set the the SOAP Fault message in the message body and set the response code in the message header.

```
from(routerEndpointURI).process(new Processor() {

    public void process(Exchange exchange) throws Exception {
        Message out = exchange.getOut();
        // Set the message body with the
        out.setBody(this.getClass().getResourceAsStream("SoapFaultMessage.xml"));
        // Set the response code here
        out.setHeader(org.apache.cxf.message.Message.RESPONSE_CODE, new Integer(500));
    }

});
```

The same is true for the POJO data format. You can set the SOAP Fault on the **Out** body and also indicate it's a fault by calling **Message.setFault(true)**, as follows:

```
from("direct:start").onException(SoapFault.class).maximumRedeliveries(0).handled(true)
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            SoapFault fault = exchange
                .getProperty(Exchange.EXCEPTION_CAUGHT, SoapFault.class);
            exchange.getOut().setFault(true);
            exchange.getOut().setBody(fault);
        }

    }).end().to(serviceURI);
```

HOW TO PROPAGATE A CXF ENDPOINT'S REQUEST AND RESPONSE CONTEXT

cxf client API provides a way to invoke the operation with request and response context. If you are using a CXF endpoint producer to invoke the external Web service, you can set the request context and get the response context with the following code:

```
        CxfExchange exchange = (CxfExchange)template.send(getJaxwsEndpointUri(), new
Processor() {
            public void process(final Exchange exchange) {
                final List<String> params = new ArrayList<String>();
                params.add(TEST_MESSAGE);
                // Set the request context to the inMessage
                Map<String, Object> requestContext = new HashMap<String, Object>();
                requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
JAXWS_SERVER_ADDRESS);
                exchange.getIn().setBody(params);
```

```
                exchange.getIn().setHeader(Client.REQUEST_CONTEXT , requestContext);
                exchange.getIn().setHeader(CxfConstants.OPERATION_NAME,
GREET_ME_OPERATION);
            }
        });
        org.apache.camel.Message out = exchange.getOut();
        // The output is an object array, the first element of the array is the return value
        Object\[\] output = out.getBody(Object\[\].class);
        LOG.info("Received output text: " + output\[0\]);
        // Get the response context form outMessage
        Map<String, Object> responseContext =
CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
        assertNotNull(responseContext);
        assertEquals("Get the wrong wsdl opertion name", "
{http://apache.org/hello_world_soap_http}greetMe",
                     responseContext.get("javax.xml.ws.wsdl.operation").toString());
```

ATTACHMENT SUPPORT

POJO Mode: Both SOAP with Attachment and MTOM are supported (see example in Payload Mode for enabling MTOM). However, SOAP with Attachment is not tested. Since attachments are marshalled and unmarshalled into POJOs, users typically do not need to deal with the attachment themself. Attachments are propagated to Camel message's attachments since 2.1. So, it is possible to retreive attachments by Camel Message API

```
DataHandler Message.getAttachment(String id)
```

.

Payload Mode: MTOM is supported since 2.1. Attachments can be retrieved by Camel Message APIs mentioned above. SOAP with Attachment is not supported as there is no SOAP processing in this mode.

To enable MTOM, set the CXF endpoint property "mtom_enabled" to **true**. (I believe you can only do it with Spring.)

```
<cxf:cxfEndpoint id="routerEndpoint"
address="http://localhost:${CXFTestSupport.port1}/CxfMtomRouterPayloadModeTest/jaxws-
mtom/hello"
        wsdlURL="mtom.wsdl"
        serviceName="ns:HelloService"
        endpointName="ns:HelloPort"
        xmlns:ns="http://apache.org/camel/cxf/mtom_feature">

    <cxf:properties>
        <!--  enable mtom by setting this property to true -->
        <entry key="mtom-enabled" value="true"/>

        <!--  set the camel-cxf endpoint data fromat to PAYLOAD mode -->
        <entry key="dataFormat" value="PAYLOAD"/>
    </cxf:properties>
```

You can produce a Camel message with attachment to send to a CXF endpoint in Payload mode.

```
Exchange exchange = context.createProducerTemplate().send("direct:testEndpoint", new
```

```
Processor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        List&lt;Source> elements = new ArrayList&lt;Source>();
        elements.add(new DOMSource(DOMUtils.readXml(new
StringReader(MtomTestHelper.REQ_MESSAGE)).getDocumentElement()));
        CxfPayload<SoapHeader> body = new CxfPayload<SoapHeader>(new ArrayList<SoapHeader>
(),
            elements, null);
        exchange.getIn().setBody(body);
        exchange.getIn().addAttachment(MtomTestHelper.REQ_PHOTO_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.REQ_PHOTO_DATA,
"application/octet-stream")));

        exchange.getIn().addAttachment(MtomTestHelper.REQ_IMAGE_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.requestJpeg, "image/jpeg")));

    }

});

// process response

CxfPayload<SoapHeader> out = exchange.getOut().getBody(CxfPayload.class);
Assert.assertEquals(1, out.getBody().size());

Map<String, String> ns = new HashMap<String, String>();
ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
ns.put("xop", MtomTestHelper.XOP_NS);

XPathUtils xu = new XPathUtils(ns);
Element oute = new XmlConverter().toDOMElement(out.getBody().get(0));
Element ele = (Element)xu.getValue("//ns:DetailResponse/ns:photo/xop:Include", oute,
                           XPathConstants.NODE);
String photoId = ele.getAttribute("href").substring(4); // skip "cid:"

ele = (Element)xu.getValue("//ns:DetailResponse/ns:image/xop:Include", oute,
                           XPathConstants.NODE);
String imageId = ele.getAttribute("href").substring(4); // skip "cid:"

DataHandler dr = exchange.getOut().getAttachment(photoId);
Assert.assertEquals("application/octet-stream", dr.getContentType());
MtomTestHelper.assertEquals(MtomTestHelper.RESP_PHOTO_DATA,
IOUtils.readBytesFromStream(dr.getInputStream()));

dr = exchange.getOut().getAttachment(imageId);
Assert.assertEquals("image/jpeg", dr.getContentType());

BufferedImage image = ImageIO.read(dr.getInputStream());
Assert.assertEquals(560, image.getWidth());
Assert.assertEquals(300, image.getHeight());
```

You can also consume a Camel message received from a CXF endpoint in Payload mode.

```
public static class MyProcessor implements Processor {
```

```
    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        CxfPayload<SoapHeader> in = exchange.getIn().getBody(CxfPayload.class);

        // verify request
        assertEquals(1, in.getBody().size());

        Map<String, String> ns = new HashMap<String, String>();
        ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
        ns.put("xop", MtomTestHelper.XOP_NS);

        XPathUtils xu = new XPathUtils(ns);
        Element body = new XmlConverter().toDOMElement(in.getBody().get(0));
        Element ele = (Element)xu.getValue("//ns:Detail/ns:photo/xop:Include", body,
                                   XPathConstants.NODE);
        String photoId = ele.getAttribute("href").substring(4); // skip "cid:"
        assertEquals(MtomTestHelper.REQ_PHOTO_CID, photoId);

        ele = (Element)xu.getValue("//ns:Detail/ns:image/xop:Include", body,
                                   XPathConstants.NODE);
        String imageId = ele.getAttribute("href").substring(4); // skip "cid:"
        assertEquals(MtomTestHelper.REQ_IMAGE_CID, imageId);

        DataHandler dr = exchange.getIn().getAttachment(photoId);
        assertEquals("application/octet-stream", dr.getContentType());
        MtomTestHelper.assertEquals(MtomTestHelper.REQ_PHOTO_DATA,
IOUtils.readBytesFromStream(dr.getInputStream()));

        dr = exchange.getIn().getAttachment(imageId);
        assertEquals("image/jpeg", dr.getContentType());
        MtomTestHelper.assertEquals(MtomTestHelper.requestJpeg,
IOUtils.readBytesFromStream(dr.getInputStream()));

        // create response
        List&lt;Source> elements = new ArrayList&lt;Source>();
        elements.add(new DOMSource(DOMUtils.readXml(new
StringReader(MtomTestHelper.RESP_MESSAGE)).getDocumentElement()));
        CxfPayload&lt;SoapHeader> sbody = new CxfPayload&lt;SoapHeader>(new
ArrayList&lt;SoapHeader>(),
            elements, null);
        exchange.getOut().setBody(sbody);
        exchange.getOut().addAttachment(MtomTestHelper.RESP_PHOTO_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.RESP_PHOTO_DATA,
"application/octet-stream")));

        exchange.getOut().addAttachment(MtomTestHelper.RESP_IMAGE_CID,
            new DataHandler(new ByteArrayDataSource(MtomTestHelper.responseJpeg, "image/jpeg")));

    }
}
```

Message Mode: Attachments are not supported as it does not process the message at all.

CXF_MESSAGE Mode: MTOM is supported, and Attachments can be retrieved by Camel Message APIs mentioned above. Note that when receiving a multipart (that is, MTOM) message the default

SOAPMessage to **String** converter will provide the complete multi-part payload on the body. If you require just the SOAP XML as a **String**, you can set the message body with **message.getSOAPPart()**, and Camel convert can do the rest of work for you.

HOW TO PROPAGATE STACK TRACE INFORMATION

It is possible to configure a CXF endpoint so that, when a Java exception is thrown on the server side, the stack trace for the exception is marshalled into a fault message and returned to the client. To enable this feaure, set the **dataFormat** to **PAYLOAD** and set the **faultStackTraceEnabled** property to **true** in the **cxfEndpoint** element, as follows:

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable sending the stack trace back to client; the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" /> <entry key="dataFormat" value="PAYLOAD"
/>
  </cxf:properties>
</cxf:cxfEndpoint>
```

For security reasons, the stack trace does not include the causing exception (that is, the part of a stack trace that follows **Caused by**). If you want to include the causing exception in the stack trace, set the **exceptionMessageCauseEnabled** property to **true** in the **cxfEndpoint** element, as follows:

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable to show the cause exception message and the default value is false -->
    <entry key="exceptionMessageCauseEnabled" value="true" />
    <!-- enable to send the stack trace back to client,  the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" />
    <entry key="dataFormat" value="PAYLOAD" />
  </cxf:properties>
</cxf:cxfEndpoint>
```

WARNING

You should only enable the **exceptionMessageCauseEnabled** flag for testing and diagnostic purposes. It is normal practice for servers to conceal the original cause of an exception to make it harder for hostile users to probe the server.

STREAMING SUPPORT IN PAYLOAD MODE

In 2.8.2, the camel-cxf component now supports streaming of incoming messages when using PAYLOAD mode. Previously, the incoming messages would have been completely DOM parsed. For large messages, this is time consuming and uses a significant amount of memory. Starting in 2.8.2, the incoming messages can remain as a javax.xml.transform.Source while being routed and, if nothing modifies the payload, can then be directly streamed out to the target destination. For common "simple proxy" use cases (example: from("cxf:...").to("cxf:...")), this can provide very significant performance increases as well as significantly lowered memory requirements.

However, there are cases where streaming may not be appropriate or desired. Due to the streaming nature, invalid incoming XML may not be caught until later in the processing chain. Also, certain actions may require the message to be DOM parsed anyway (like WS-Security or message tracing and such) in which case the advantages of the streaming is limited. At this point, there are two ways to control the streaming:

- Endpoint property: you can add "allowStreaming=false" as an endpoint property to turn the streaming on/off.
- Component property: the CxfComponent object also has an allowStreaming property that can set the default for endpoints created from that component.
- Global system property: you can add a system property of "org.apache.camel.component.cxf.streaming" to "false" to turn if off. That sets the global default, but setting the endpoint property above will override this value for that endpoint.

USING THE GENERIC CXF DISPATCH MODE

From 2.8.0, the camel-cxf component supports the generic CXF dispatch mode that can transport messages of arbitrary structures (i.e., not bound to a specific XML schema). To use this mode, you simply omit specifying the wsdlURL and serviceClass attributes of the CXF endpoint.

```
<cxf:cxfEndpoint id="testEndpoint" address="http://localhost:9000/SoapContext/SoapAnyPort">
  <cxf:properties>
    <entry key="dataFormat" value="PAYLOAD"/>
  </cxf:properties>
</cxf:cxfEndpoint>
```

It is noted that the default CXF dispatch client does not send a specific SOAPAction header. Therefore, when the target service requires a specific SOAPAction value, it is supplied in the Camel header using the key SOAPAction (case-insensitive).

75.1. CXF CONSUMERS ON {WILDFLY}

The configuration of camel-cxf consumers on {wildfly} is different to that of standalone Camel. Producer endpoints work as per normal.

On {wildfly}, camel-cxf consumers leverage the default Undertow HTTP server provided by the container. The server is defined within the undertow subsystem configuration. Here's an excerpt of the default configuration from standalone.xml:

```
<subsystem xmlns="urn:jboss:domain:undertow:4.0">
    <buffer-cache name="default" />
    <server name="default-server">
        <http-listener name="default" socket-binding="http" redirect-socket="https" enable-http2="true" />
        <https-listener name="https" socket-binding="https" security-realm="ApplicationRealm" enable-
http2="true" />
```

```
        <host name="default-host" alias="localhost">
            <location name="/" handler="welcome-content" />
            <filter-ref name="server-header" />
            <filter-ref name="x-powered-by-header" />
            <http-invoker security-realm="ApplicationRealm" />
        </host>
    </server>
</subsystem>
```

In this instance, Undertow is configured to listen on interfaces / ports specified by the http and https socket-binding. By default this is port 8080 for http and 8443 for https.

For example, if you configure an endpoint consumer using different host or port combinations, a warning will appear within the server log file. For example the following host & port configurations would be ignored:

```
<cxf:rsServer id="cxfRsConsumer"
              address="http://somehost:1234/path/to/resource"
              serviceClass="org.example.ServiceClass" />
```

```
<cxf:cxfEndpoint id="cxfWsConsumer"
                 address="http://somehost:1234/path/to/resource"
                 serviceClass="org.example.ServiceClass" />
```

```
[org.wildfly.extension.camel] (pool-2-thread-1) Ignoring configured host:
http://somehost:1234/path/to/resource
```

However, the consumer is still available on the default host & port localhost:8080 or localhost:8443.

NOTE

Applications which use camel-cxf consumers ***must*** be packaged as a WAR. In previous {wildfly-camel} releases, other types of archive such as JAR were permitted, but this is no longer supported.

75.1.1. Configuring alternative ports

If alternative ports are to be accepted, then these must be configured via the {wildfly} subsystem configuration. This is explained in the server documentation:

https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.1/html/configuration_guide/configuring_the_web_ser

75.1.2. Configuring SSL

To configure SSL, refer to the {wildfly} SSL configuration guide:

https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.1/html-single/how_to_configure_server_security/#configure_one_way_and_two_way_ssl_tls_for_application

75.1.3. Configuring security with Elytron

{wildfly-camel} supports securing camel-cxf consumer endpoints with the Elytron security framework.

75.1.3.1. Configuring a security domain

To secure a {wildfly-camel} application with Elytron, an application security domain needs to be referenced within **WEB-INF/jboss-web.xml** of your WAR deployment:

```
<jboss-web>
  <security-domain>my-application-security-domain</security-domain>
</jboss-web>
```

The **<security-domain>** configuration references the name of an **<application-security-domain>** defined by the Undertow subsystem. For example, the Undertow subsystem **<application-security-domain>** is configured within the {wildfly} server **standalone.xml** configuration file as follows:

```
<subsystem xmlns="urn:jboss:domain:undertow:6.0">
    ...
    <application-security-domains>
        <application-security-domain name="my-application-security-domain" http-authentication-factory="application-http-authentication"/>
    </application-security-domains>
</subsystem>
```

The **<http-authentication-factory> application-http-authentication** is defined within the Elytron subsystem. **application-http-authentication** is available by default in both the **standalone.xml** and **standalone-full.xml** server configuration files. For example:

```
<subsystem xmlns="urn:wildfly:elytron:1.2">
    ...
    <http>
        ...
        <http-authentication-factory name="application-http-authentication" http-server-mechanism-factory="global" security-domain="ApplicationDomain">
            <mechanism-configuration>
                <mechanism mechanism-name="BASIC">
                    <mechanism-realm realm-name="Application Realm" />
                </mechanism>
                <mechanism mechanism-name="FORM" />
            </mechanism-configuration>
        </http-authentication-factory>
        <provider-http-server-mechanism-factory name="global" />
    </http>
    ...
</subsystem>
```

The **<http-authentication-factory>** named **application-http-authentication**, holds a reference to a Elytron security domain called **ApplicationDomain**.

For more information on how to configure the Elytron subsystem, refer to the Elytron documentation.

75.1.3.2. Configuring security constraints, authentication methods and security roles

Security constraints, authentication methods and security roles for camel-cxf consumer endpoints can be configured within your WAR deployment **WEB-INF/web.xml**. For example, to configure BASIC Authentication:

```
<web-app>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>/webservices/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>my-role</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <description>The role that is required to log in to /webservices/*</description>
    <role-name>my-role</role-name>
  </security-role>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>my-realm</realm-name>
  </login-config>
</web-app>
```

Note that the **<url-pattern>** defined by the Servlet Specification is relative to the context path of the web application. If your application is packaged as **my-app.war**, {wildfly} will make it accessible under the context path **/my-app** and the **<url-patternpattern> /webservices/*** will be applied to paths relative to **/my-app**.

For example, requests against **http://my-server/my-app/webservices/my-endpoint** will match the **/webservices/*** pattern, while **http://my-server/webservices/my-endpoint** will not match.

This is important because {wildfly-camel} allows the creation of camel-cxf endpoint consumers whose base path is outside of the host web application context path. For example, it is possible to create a camel-cxf consumer for **http://my-server/webservices/my-endpoint** inside **my-app.war**.

In order to define security constraints for such out-of-context endpoints, {wildfly-camel} supports a custom, ***non-standard* <url-pattern>** convention where prefixing the pattern with three forward slashes /// will be interpreted as absolute to server host name. For example, to secure **http://my-server/webservices/my-endpoint** inside **my-app.war**, you would add the following configuration to **web.xml**:

```
<web-app>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>///webservices/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>my-role</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <description>The role that is required to log in to /webservices/*</description>
    <role-name>my-role</role-name>
  </security-role>
```

```
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>my-realm</realm-name>
  </login-config>
</web-app>
```

CHAPTER 76. CXF-RS COMPONENT

Available as of Camel version 2.0

The **cxfrs:** component provides integration with Apache CXF for connecting to JAX-RS 1.1 and 2.0 services hosted in CXF.

When using CXF as a consumer, the CXF Bean Component allows you to factor out how message payloads are received from their processing as a RESTful or SOAP web service. This has the potential of using a multitude of transports to consume web services. The bean component's configuration is also simpler and provides the fastest method to implement web services using Camel and CXF.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cxf</artifactId>
  <version>x.x.x</version>  <!-- use the same version as your Camel core version -->
</dependency>
```

76.1. URI FORMAT

```
cxfrs://address?options
```

Where **address** represents the CXF endpoint's address

```
cxfrs:bean:rsEndpoint
```

Where **rsEndpoint** represents the spring bean's name which presents the CXFRS client or server

For either style above, you can append options to the URI as follows:

```
cxfrs:bean:cxfEndpoint?resourceClasses=org.apache.camel.rs.Example
```

76.2. OPTIONS

The CXF-RS component supports 3 options which are listed below.

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The CXF-RS endpoint is configured using URI syntax:

```
cxfrs:beanId:address
```

with the following path and query parameters:

76.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
beanId	To lookup an existing configured CxfRsEndpoint. Must used bean: as prefix.		String
address	The service publish address.		String

76.2.2. Query Parameters (29 parameters):

Name	Description	Default	Type
features (common)	Set the feature list to the CxfRs endpoint.		List
loggingFeatureEnabled (common)	This option enables CXF Logging Feature which writes inbound and outbound REST messages to log.	false	boolean
loggingSizeLimit (common)	To limit the total size of number of bytes the logger will output when logging feature has been enabled.		int
modelRef (common)	This option is used to specify the model file which is useful for the resource class without annotation. When using this option, then the service class can be omitted, to emulate document-only endpoints		String
providers (common)	Set custom JAX-RS provider(s) list to the CxfRs endpoint. You can specify a string with a list of providers to lookup in the registy separated by comma.		String

Name	Description	Default	Type
resourceClasses (common)	The resource classes which you want to export as REST service. Multiple classes can be separated by comma.		List
schemaLocations (common)	Sets the locations of the schema(s) which can be used to validate the incoming XML or JAXB-driven JSON.		List
skipFaultLogging (common)	This option controls whether the PhaseInterceptorChain skips logging the Fault that it catches.	false	boolean
bindingStyle (consumer)	Sets how requests and responses will be mapped to/from Camel. Two values are possible: SimpleConsumer: This binding style processes request parameters, multiparts, etc. and maps them to IN headers, IN attachments and to the message body. It aims to eliminate low-level processing of org.apache.cxf.message.MessageContentsList. It also also adds more flexibility and simplicity to the response mapping. Only available for consumers. Default: The default style. For consumers this passes on a MessageContentsList to the route, requiring low-level processing in the route. This is the traditional binding style, which simply dumps the org.apache.cxf.message.MessageContentsList coming in from the CXF stack onto the IN message body. The user is then responsible for processing it according to the contract defined by the JAX-RS method signature. Custom: allows you to specify a custom binding through the binding option.	Default	BindingStyle
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
publishedEndpointUrl (consumer)	This option can override the endpointUrl that published from the WADL which can be accessed with resource address url plus _wadl		String

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
hostnameVerifier (producer)	The hostname verifier to be used. Use the notation to reference a HostnameVerifier from the registry.		HostnameVerifier
sslContextParameters (producer)	The Camel SSL setting reference. Use the notation to reference the SSL Context.		SSLContextParameters
throwExceptionOnFailure (producer)	This option tells the CxfRsProducer to inspect return codes and will generate an Exception if the return code is larger than 207.	true	boolean
httpClientAPI (producer)	If it is true, the CxfRsProducer will use the HttpClientAPI to invoke the service. If it is false, the CxfRsProducer will use the ProxyClientAPI to invoke the service	true	boolean
ignoreDeleteMethodMessageBody (producer)	This option is used to tell CxfRsProducer to ignore the message body of the DELETE method when using HTTP API.	false	boolean
maxClientCacheSize (producer)	This option allows you to configure the maximum size of the cache. The implementation caches CXF clients or ClientFactoryBean in CxfProvider and CxfRsProvider.	10	int
binding (advanced)	To use a custom CxfBinding to control the binding between Camel Message and CXF Message.		CxfRsBinding
bus (advanced)	To use a custom configured CXF Bus.		Bus
continuationTimeout (advanced)	This option is used to set the CXF continuation timeout which could be used in CxfConsumer by default when the CXF server is using Jetty or Servlet transport.	30000	long

Name	Description	Default	Type
cxfRsEndpointConfigurer (advanced)	This option could apply the implementation of org.apache.camel.component.cxf.jaxrs.CxfRsEndpointConfigurer which supports to configure the CXF endpoint in programmatic way. User can configure the CXF server and client by implementing configureServer/Client method of CxfEndpointConfigurer.		CxfRsEndpointConfigurer
defaultBus (advanced)	Will set the default bus when CXF endpoint create a bus by itself	false	boolean
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
performInvocation (advanced)	When the option is true, Camel will perform the invocation of the resource class instance and put the response object into the exchange for further processing.	false	boolean
propagateContexts (advanced)	When the option is true, JAXRS UriInfo, HttpHeaders, Request and SecurityContext contexts will be available to custom CXFRS processors as typed Camel exchange properties. These contexts can be used to analyze the current requests using JAX-RS API.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

You can also configure the CXF REST endpoint through the spring configuration. Since there are lots of difference between the CXF REST client and CXF REST Server, we provide different configuration for them. Please check out the schema file and CXF JAX-RS documentation for more information.

76.3. HOW TO CONFIGURE THE REST ENDPOINT IN CAMEL

In camel-cxf schema file, there are two elements for the REST endpoint definition. **cxf:rsServer** for REST consumer, **cxf:rsClient** for REST producer.
You can find a Camel REST service route configuration example here.

76.4. HOW TO OVERRIDE THE CXF PRODUCER ADDRESS FROM MESSAGE HEADER

The **camel-cxfrs** producer supports to override the services address by setting the message with the key of "CamelDestinationOverrideUrl".

```
// set up the service address from the message header to override the setting of CXF endpoint
 exchange.getIn().setHeader(Exchange.DESTINATION_OVERRIDE_URL,
constant(getServiceAddress()));
```

76.5. CONSUMING A REST REQUEST - SIMPLE BINDING STYLE

Available as of Camel 2.11

The **Default** binding style is rather low-level, requiring the user to manually process the **MessageContentsList** object coming into the route. Thus, it tightly couples the route logic with the method signature and parameter indices of the JAX-RS operation. Somewhat inelegant, difficult and error-prone.

In contrast, the **SimpleConsumer** binding style performs the following mappings, in order to **make the request data more accessible** to you within the Camel Message:

- JAX-RS Parameters (@HeaderParam, @QueryParam, etc.) are injected as IN message headers. The header name matches the value of the annotation.
- The request entity (POJO or other type) becomes the IN message body. If a single entity cannot be identified in the JAX-RS method signature, it falls back to the original **MessageContentsList**.
- Binary **@Multipart** body parts become IN message attachments, supporting **DataHandler**, **InputStream**, **DataSource** and CXF's **Attachment** class.
- Non-binary **@Multipart** body parts are mapped as IN message headers. The header name matches the Body Part name.

Additionally, the following rules apply to the **Response mapping**:

- If the message body type is different to **javax.ws.rs.core.Response** (user-built response), a new **Response** is created and the message body is set as the entity (so long it's not null). The response status code is taken from the **Exchange.HTTP_RESPONSE_CODE** header, or defaults to 200 OK if not present.
- If the message body type is equal to **javax.ws.rs.core.Response**, it means that the user has built a custom response, and therefore it is respected and it becomes the final response.
- In all cases, Camel headers permitted by custom or default **HeaderFilterStrategy** are added to the HTTP response.

76.5.1. Enabling the Simple Binding Style

This binding style can be activated by setting the **bindingStyle** parameter in the consumer endpoint to value **SimpleConsumer**:

```
from("cxfrs:bean:rsServer?bindingStyle=SimpleConsumer")
  .to("log:TEST?showAll=true");
```

76.5.2. Examples of request binding with different method signatures

Below is a list of method signatures along with the expected result from the Simple binding.

public Response doAction(BusinessObject request);
Request payload is placed in IN message body, replacing the original MessageContentsList.

public Response doAction(BusinessObject request, @HeaderParam("abcd") String abcd, @QueryParam("defg") String defg); Request payload placed in IN message body, replacing the original MessageContentsList. Both request params mapped as IN message headers with names abcd and defg.

public Response doAction(@HeaderParam("abcd") String abcd, @QueryParam("defg") String defg); Both request params mapped as IN message headers with names abcd and defg. The original MessageContentsList is preserved, even though it only contains the 2 parameters.

public Response doAction(@Multipart(value="body1") BusinessObject request, @Multipart(value="body2") BusinessObject request2); The first parameter is transferred as a header with name body1, and the second one is mapped as header body2. The original MessageContentsList is preserved as the IN message body.

public Response doAction(InputStream abcd); The InputStream is unwrapped from the MessageContentsList and preserved as the IN message body.

public Response doAction(DataHandler abcd); The DataHandler is unwrapped from the MessageContentsList and preserved as the IN message body.

76.5.3. More examples of the Simple Binding Style

Given a JAX-RS resource class with this method:

```
    @POST @Path("/customers/{type}")
    public Response newCustomer(Customer customer, @PathParam("type") String type,
@QueryParam("active") @DefaultValue("true") boolean active) {
        return null;
    }
```

Serviced by the following route:

```
    from("cxfrs:bean:rsServer?bindingStyle=SimpleConsumer")
        .recipientList(simple("direct:${header.operationName}"));

    from("direct:newCustomer")
        .log("Request: type=${header.type}, active=${header.active}, customerData=${body}");
```

The following HTTP request with XML payload (given that the Customer DTO is JAXB-annotated):

```
POST /customers/gold?active=true

Payload:
<Customer>
  <fullName>Raul Kripalani</fullName>
  <country>Spain</country>
  <project>Apache Camel</project>
</Customer>
```

Will print the message:

```
Request: type=gold, active=true, customerData=<Customer.toString() representation>
```

For more examples on how to process requests and write responses can be found here.

76.6. CONSUMING A REST REQUEST - DEFAULT BINDING STYLE

The CXF JAXRS front end implements the JAX-RS (JSR-311) API, so we can export the resources classes as a REST service. And we leverage the CXF Invoker API to turn a REST request into a normal Java object method invocation.
Unlike the Camel Restlet component, you don't need to specify the URI template within your endpoint, CXF takes care of the REST request URI to resource class method mapping according to the JSR-311 specification. All you need to do in Camel is delegate this method request to a right processor or endpoint.

Here is an example of a CXFRS route...

```
private static final String CXF_RS_ENDPOINT_URI =
        "cxfrs://http://localhost:" + CXT + "/rest?
resourceClasses=org.apache.camel.component.cxf.jaxrs.testbean.CustomerServiceResource";
private static final String CXF_RS_ENDPOINT_URI2 =
        "cxfrs://http://localhost:" + CXT + "/rest2?
resourceClasses=org.apache.camel.component.cxf.jaxrs.testbean.CustomerService";
private static final String CXF_RS_ENDPOINT_URI3 =
        "cxfrs://http://localhost:" + CXT + "/rest3?"
        +
"resourceClasses=org.apache.camel.component.cxf.jaxrs.testbean.CustomerServiceNoAnnotations&"

        + "modelRef=classpath:/org/apache/camel/component/cxf/jaxrs/CustomerServiceModel.xml";
private static final String CXF_RS_ENDPOINT_URI4 =
        "cxfrs://http://localhost:" + CXT + "/rest4?"
        +
"modelRef=classpath:/org/apache/camel/component/cxf/jaxrs/CustomerServiceDefaultHandlerModel.xm
";
private static final String CXF_RS_ENDPOINT_URI5 =
        "cxfrs://http://localhost:" + CXT + "/rest5?"
        + "propagateContexts=true&"
        +
"modelRef=classpath:/org/apache/camel/component/cxf/jaxrs/CustomerServiceDefaultHandlerModel.xm
";
protected RouteBuilder createRouteBuilder() throws Exception {
    final Processor testProcessor = new TestProcessor();
    final Processor testProcessor2 = new TestProcessor2();
    final Processor testProcessor3 = new TestProcessor3();
    return new RouteBuilder() {
        public void configure() {
            errorHandler(new NoErrorHandlerBuilder());
            from(CXF_RS_ENDPOINT_URI).process(testProcessor);
            from(CXF_RS_ENDPOINT_URI2).process(testProcessor);
            from(CXF_RS_ENDPOINT_URI3).process(testProcessor);
            from(CXF_RS_ENDPOINT_URI4).process(testProcessor2);
            from(CXF_RS_ENDPOINT_URI5).process(testProcessor3);
        }
    };
}
```

And the corresponding resource class used to configure the endpoint...

INFO:*Note about resource classes*

By default, JAX-RS resource classes are **only*used to configure JAX-RS properties. Methods will *not** be executed during routing of messages to the endpoint. Instead, it is the responsibility of the route to do all processing.

Note that starting from Camel 2.15 it is also sufficient to provide an interface only as opposed to a no-op service implementation class for the default mode.

Starting from Camel 2.15, if a **performInvocation** option is enabled, the service implementation will be invoked first, the response will be set on the Camel exchange and the route execution will continue as usual. This can be useful for integrating the existing JAX-RS implementations into Camel routes and for post-processing JAX-RS Responses in custom processors.

76.7. HOW TO INVOKE THE REST SERVICE THROUGH CAMEL-CXFRS PRODUCER

The CXF JAXRS front end implements a proxy-based client API, with this API you can invoke the remote REST service through a proxy. The **camel-cxfrs** producer is based on this proxy API.
You just need to specify the operation name in the message header and prepare the parameter in the message body, the camel-cxfrs producer will generate right REST request for you.

Here is an example:

```
Exchange exchange = template.send("direct://proxy", new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        Message inMessage = exchange.getIn();
        // set the operation name
        inMessage.setHeader(CxfConstants.OPERATION_NAME, "getCustomer");
        // using the proxy client API
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_USING_HTTP_API, Boolean.FALSE);
        // set a customer header
        inMessage.setHeader("key", "value");
        // setup the accept content type
        inMessage.setHeader(Exchange.ACCEPT_CONTENT_TYPE, "application/json");
        // set the parameters , if you just have one parameter
        // camel will put this object into an Object[] itself
        inMessage.setBody("123");
    }
});

// get the response message
Customer response = (Customer) exchange.getOut().getBody();

assertNotNull("The response should not be null ", response);
assertEquals("Get a wrong customer id ", 123, response.getId());
assertEquals("Get a wrong customer name", "John", response.getName());
assertEquals("Get a wrong response code", 200,
exchange.getOut().getHeader(Exchange.HTTP_RESPONSE_CODE));
assertEquals("Get a wrong header value", "value", exchange.getOut().getHeader("key"));
```

The CXF JAXRS front end also provides a http centric client API . You can also invoke this API from **camel-cxfrs** producer. You need to specify the HTTP_PATH and the HTTP_METHOD and let the producer use the http centric client API by using the URI option **httpClientAPI** or by setting the

message header CxfConstants.CAMEL_CXF_RS_USING_HTTP_API. You can turn the response object to the type class specified with the message header CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS.

From Camel 2.1, we also support to specify the query parameters from cxfrs URI for the CXFRS http centric client.

Error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20

To support the Dynamical routing, you can override the URI's query parameters by using the CxfConstants.CAMEL_CXF_RS_QUERY_MAP header to set the parameter map for it.

76.8. WHAT'S THE CAMEL TRANSPORT FOR CXF

In CXF you offer or consume a webservice by defining its address. The first part of the address specifies the protocol to use. For example address="http://localhost:9000" in an endpoint configuration means your service will be offered using the http protocol on port 9000 of localhost. When you integrate Camel Tranport into CXF you get a new transport "camel". So you can specify address="camel://direct:MyEndpointName" to bind the CXF service address to a camel direct endpoint.

Technically speaking Camel transport for CXF is a component which implements the CXF transport API with the Camel core library. This allows you to easily use Camel's routing engine and integration patterns support together with your CXF services.

76.9. INTEGRATE CAMEL INTO CXF TRANSPORT LAYER

To include the Camel Tranport into your CXF bus you use the CamelTransportFactory. You can do this in Java as well as in Spring.

76.9.1. Setting up the Camel Transport in Spring

You can use the following snippet in your applicationcontext if you want to configure anything special. If you only want to activate the camel transport you do not have to do anything in your application context. As soon as you include the camel-cxf-transport jar (or camel-cxf.jar if your camel version is less than 2.7.x) in your app, cxf will scan the jar and load a CamelTransportFactory for you.

```
<!-- you don't need to specify the CamelTransportFactory configuration as it is auto load by CXF bus
-->
<bean class="org.apache.camel.component.cxf.transport.CamelTransportFactory">
  <property name="bus" ref="cxf" />
  <property name="camelContext" ref="camelContext" />
  <!-- checkException new added in Camel 2.1 and Camel 1.6.2 -->
  <!-- If checkException is true , CamelDestination will check the outMessage's
     exception and set it into camel exchange. You can also override this value
     in CamelDestination's configuration. The default value is false.
     This option should be set true when you want to leverage the camel's error
     handler to deal with fault message -->
  <property name="checkException" value="true" />
  <property name="transportIds">
    <list>
      <value>http://cxf.apache.org/transports/camel</value>
    </list>
  </property>
</bean>
```

76.9.2. Integrating the Camel Transport in a programmatic way

Camel transport provides a setContext method that you could use to set the Camel context into the transport factory. If you want this factory take effect, you need to register the factory into the CXF bus. Here is a full example for you.

```
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.transport.ConduitInitiatorManager;
import org.apache.cxf.transport.DestinationFactoryManager;
...

BusFactory bf = BusFactory.newInstance();
Bus bus = bf.createBus();
CamelTransportFactory camelTransportFactory = new CamelTransportFactory();
// set up the CamelContext which will be use by the CamelTransportFactory
camelTransportFactory.setCamelContext(context)
// if you are using CXF higher then 2.4.x the
camelTransportFactory.setBus(bus);

// if you are lower CXF, you need to register the ConduitInitiatorManager and
DestinationFactoryManager like below
// register the conduit initiator
ConduitInitiatorManager cim = bus.getExtension(ConduitInitiatorManager.class);
cim.registerConduitInitiator(CamelTransportFactory.TRANSPORT_ID, camelTransportFactory);
// register the destination factory
DestinationFactoryManager dfm = bus.getExtension(DestinationFactoryManager.class);
dfm.registerDestinationFactory(CamelTransportFactory.TRANSPORT_ID, camelTransportFactory);
// set or bus as the default bus for cxf
BusFactory.setDefaultBus(bus);
```

76.10. CONFIGURE THE DESTINATION AND CONDUIT WITH SPRING

76.10.1. Namespace

The elements used to configure an Camel transport endpoint are defined in the namespace **http://cxf.apache.org/transports/camel**. It is commonly referred to using the prefix **camel**. In order to use the Camel transport configuration elements, you will need to add the lines shown below to the beans element of your endpoint's configuration file. In addition, you will need to add the configuration elements' namespace to the **xsi:schemaLocation** attribute.

Adding the Configuration Namespace

```
<beans ...
      xmlns:camel="http://cxf.apache.org/transports/camel
      ...
      xsi:schemaLocation="...
                          http://cxf.apache.org/transports/camel
                          http://cxf.apache.org/transports/camel.xsd
                         ...>
```

76.10.2. The destination element

You configure an Camel transport server endpoint using the **camel:destination** element and its children. The **camel:destination** element takes a single attribute, **name**, that specifies the WSDL port element that corresponds to the endpoint. The value for the **name** attribute takes the form *portQName*`.camel-destination`. The example below shows the **camel:destination** element that would be used to add configuration for an endpoint that was specified by the WSDL fragment **<port binding="widgetSOAPBinding" name="widgetSOAPPort">** if the endpoint's target namespace was **http://widgets.widgetvendor.net**.

camel:destination Element

```
...
  <camel:destination name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-destination>
    <camelContext id="context" xmlns="http://activemq.apache.org/camel/schema/spring">
        <route>
          <from uri="direct:EndpointC" />
          <to uri="direct:EndpointD" />
        </route>
    </camelContext>
  </camel:destination>

  <!-- new added feature since Camel 2.11.x
  <camel:destination name="{http://widgets/widgetvendor.net}widgetSOAPPort.camel-destination"
camelContextId="context" />

...
```

The **camel:destination** element for Spring has a number of child elements that specify configuration information. They are described below.

Element

Description

camel-spring:camelContext

You can specify the camel context in the camel destination

camel:camelContextRef

The camel context id which you want inject into the camel destination

76.10.3. The conduit element

You configure a Camel transport client using the **camel:conduit** element and its children. The **camel:conduit** element takes a single attribute, **name**, that specifies the WSDL port element that corresponds to the endpoint. The value for the **name** attribute takes the form *portQName*`.camel-conduit`. For example, the code below shows the **camel:conduit** element that would be used to add configuration for an endpoint that was specified by the WSDL fragment **<port binding="widgetSOAPBinding" name="widgetSOAPPort">** if the endpoint's target namespace was **http://widgets.widgetvendor.net**.

http-conf:conduit Element

```
...
  <camelContext id="conduit_context" xmlns="http://activemq.apache.org/camel/schema/spring">
```

```
        <route>
            <from uri="direct:EndpointA" />
            <to uri="direct:EndpointB" />
        </route>
    </camelContext>

    <camel:conduit name="{http://widgets/widgetvendor.net}widgetSOAPPort.camel-conduit">
      <camel:camelContextRef>conduit_context</camel:camelContextRef>
    </camel:conduit>

    <!-- new added feature since Camel 2.11.x
    <camel:conduit name="{http://widgets/widgetvendor.net}widgetSOAPPort.camel-conduit"
  camelContextId="conduit_context" />


    <camel:conduit name="*.camel-conduit">
    <!-- you can also using the wild card to specify the camel-conduit that you want to configure -->
      ...
    </camel:conduit>
  ...
```

The **camel:conduit** element has a number of child elements that specify configuration information. They are described below.

Element

Description

camel-spring:camelContext

You can specify the camel context in the camel conduit

camel:camelContextRef

The camel context id which you want inject into the camel conduit

76.11. CONFIGURE THE DESTINATION AND CONDUIT WITH BLUEPRINT

From **Camel 2.11.x**, Camel Transport supports to be configured with Blueprint.

If you are using blueprint, you should use the the namespace **http://cxf.apache.org/transports/camel/blueprint** and import the schema like the blow.

Adding the Configuration Namespace for blueprint

```
<beans ...
      xmlns:camel="http://cxf.apache.org/transports/camel/blueprint"
      ...
      xsi:schemaLocation="...
                          http://cxf.apache.org/transports/camel/blueprint
                          http://cxf.apache.org/schmemas/blueprint/camel.xsd
                         ...>
```

In blueprint **camel:conduit camel:destination** only has one camelContextId attribute, they doesn't support to specify the camel context in the camel destination.

```
<camel:conduit id="*.camel-conduit" camelContextId="camel1" />
<camel:destination id="*.camel-destination" camelContextId="camel1" />
```

76.12. EXAMPLE USING CAMEL AS A LOAD BALANCER FOR CXF

This example shows how to use the camel load balancing feature in CXF. You need to load the configuration file in CXF and publish the endpoints on the address "camel://direct:EndpointA" and "camel://direct:EndpointB"

76.13. COMPLETE HOWTO AND EXAMPLE FOR ATTACHING CAMEL TO CXF

Better JMS Transport for CXF Webservice using Apache Camel

CHAPTER 77. DATA FORMAT COMPONENT

Available as of Camel version 2.12

The **dataformat:** component allows to use Data Format as a Camel Component.

77.1. URI FORMAT

```
dataformat:name:(marshal|unmarshal)[?options]
```

Where **name** is the name of the Data Format. And then followed by the operation which must either be **marshal** or **unmarshal**. The options is used for configuring the Data Format in use. See the Data Format documentation for which options it support.

77.2. DATAFORMAT OPTIONS

The Data Format component has no options.

The Data Format endpoint is configured using URI syntax:

```
dataformat:name:operation
```

with the following path and query parameters:

77.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
name	**Required** Name of data format		String
operation	**Required** Operation to use either marshal or unmarshal		String

77.2.2. Query Parameters (1 parameters):

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

77.3. SAMPLES

For example to use the JAXB Data Format we can do as follows:

```
from("activemq:My.Queue").
  to("dataformat:jaxb:unmarshal?contextPath=com.acme.model").
  to("mqseries:Another.Queue");
```

And in XML DSL you do:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="dataformat:jaxb:unmarshal?contextPath=com.acme.model"/>
    <to uri="mqseries:Another.Queue"/>
  </route>
</camelContext>
```

CHAPTER 78. DATASET COMPONENT

Available as of Camel version 1.3

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Camel Testing Framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Bean Integration.

The DataSet component provides a mechanism to easily perform load & soak testing of your system. It works by allowing you to create DataSet instances both as a source of messages and as a way to assert that the data set is received.

Camel will use the throughput logger when sending dataset's.

78.1. URI FORMAT

```
dataset:name[?options]
```

Where **name** is used to find the DataSet instance in the Registry

Camel ships with a support implementation of **org.apache.camel.component.dataset.DataSet**, the **org.apache.camel.component.dataset.DataSetSupport** class, that can be used as a base for implementing your own DataSet. Camel also ships with some implementations that can be used for testing: **org.apache.camel.component.dataset.SimpleDataSet**, **org.apache.camel.component.dataset.ListDataSet** and **org.apache.camel.component.dataset.FileDataSet**, all of which extend **DataSetSupport**.

78.2. OPTIONS

The Dataset component has no options.

The Dataset endpoint is configured using URI syntax:

```
dataset:name
```

with the following path and query parameters:

78.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of DataSet to lookup in the registry		DataSet

78.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
dataSetIndex (common)	Controls the behaviour of the CamelDataSetIndex header. For Consumers: - off = the header will not be set - strict/lenient = the header will be set For Producers: - off = the header value will not be verified, and will not be set if it is not present = strict = the header value must be present and will be verified = lenient = the header value will be verified if it is present, and will be set if it is not present	lenient	String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
initialDelay (consumer)	Time period in millis to wait before starting sending messages.	1000	long
minRate (consumer)	Wait until the DataSet contains at least this number of messages	0	int
preloadSize (consumer)	Sets how many messages should be preloaded (sent) before the route completes its initialization	0	long
produceDelay (consumer)	Allows a delay to be specified which causes a delay when a message is sent by the consumer (to simulate slow processing)	3	long
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern

Name	Description	Default	Type
assertPeriod (producer)	Sets a grace period after which the mock endpoint will re-assert to ensure the preliminary assertion is still valid. This is used for example to assert that exactly a number of messages arrives. For example if link expectedMessageCount(int) was set to 5, then the assertion is satisfied when 5 or more message arrives. To ensure that exactly 5 messages arrives, then you would need to wait a little period to ensure no further message arrives. This is what you can use this link setAssertPeriod(long) method for. By default this period is disabled.	0	long
consumeDelay (producer)	Allows a delay to be specified which causes a delay when a message is consumed by the producer (to simulate slow processing)	0	long
expectedCount (producer)	Specifies the expected number of message exchanges that should be received by this endpoint. Beware: If you want to expect that 0 messages, then take extra care, as 0 matches when the tests starts, so you need to set a assert period time to let the test run for a while to make sure there are still no messages arrived; for that use link setAssertPeriod(long). An alternative is to use NotifyBuilder, and use the notifier to know when Camel is done routing some messages, before you call the link assertIsSatisfied() method on the mocks. This allows you to not use a fixed assert period, to speedup testing times. If you want to assert that exactly n'th message arrives to this mock endpoint, then see also the link setAssertPeriod(long) method for further details.	-1	int
reportGroup (producer)	A number that is used to turn on throughput logging based on groups of the size.		int
resultMinimumWaitTime (producer)	Sets the minimum expected amount of time (in millis) the link assertIsSatisfied() will wait on a latch until it is satisfied	0	long
resultWaitTime (producer)	Sets the maximum amount of time (in millis) the link assertIsSatisfied() will wait on a latch until it is satisfied	0	long

Name	Description	Default	Type
retainFirst (producer)	Specifies to only retain the first n'th number of received Exchanges. This is used when testing with big data, to reduce memory consumption by not storing copies of every Exchange this mock endpoint receives. Important: When using this limitation, then the link getReceivedCounter() will still return the actual number of received Exchanges. For example if we have received 5000 Exchanges, and have configured to only retain the first 10 Exchanges, then the link getReceivedCounter() will still return 5000 but there is only the first 10 Exchanges in the link getExchanges() and link getReceivedExchanges() methods. When using this method, then some of the other expectation methods is not supported, for example the link expectedBodiesReceived(Object...) sets a expectation on the first number of bodies received. You can configure both link setRetainFirst(int) and link setRetainLast(int) methods, to limit both the first and last received.	-1	int
retainLast (producer)	Specifies to only retain the last n'th number of received Exchanges. This is used when testing with big data, to reduce memory consumption by not storing copies of every Exchange this mock endpoint receives. Important: When using this limitation, then the link getReceivedCounter() will still return the actual number of received Exchanges. For example if we have received 5000 Exchanges, and have configured to only retain the last 20 Exchanges, then the link getReceivedCounter() will still return 5000 but there is only the last 20 Exchanges in the link getExchanges() and link getReceivedExchanges() methods. When using this method, then some of the other expectation methods is not supported, for example the link expectedBodiesReceived(Object...) sets a expectation on the first number of bodies received. You can configure both link setRetainFirst(int) and link setRetainLast(int) methods, to limit both the first and last received.	-1	int
sleepForEmptyTest (producer)	Allows a sleep to be specified to wait to check that this endpoint really is empty when link expectedMessageCount(int) is called with zero	0	long
copyOnExchange (producer)	Sets whether to make a deep copy of the incoming Exchange when received at this mock endpoint. Is by default true.	true	boolean

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

You can append query options to the URI in the following format, **?option=value&option=value&…**

78.3. CONFIGURING DATASET

Camel will lookup in the Registry for a bean implementing the DataSet interface. So you can register your own DataSet as:

```
<bean id="myDataSet" class="com.mycompany.MyDataSet">
  <property name="size" value="100"/>
</bean>
```

78.4. EXAMPLE

For example, to test that a set of messages are sent to a queue and then consumed from the queue without losing any messages:

```
// send the dataset to a queue
from("dataset:foo").to("activemq:SomeQueue");

// now lets test that the messages are consumed correctly
from("activemq:SomeQueue").to("dataset:foo");
```

The above would look in the Registry to find the **foo** DataSet instance which is used to create the messages.

Then you create a DataSet implementation, such as using the **SimpleDataSet** as described below, configuring things like how big the data set is and what the messages look like etc.

78.5. DATASETSUPPORT (ABSTRACT CLASS)

The DataSetSupport abstract class is a nice starting point for new DataSets, and provides some useful features to derived classes.

78.5.1. Properties on DataSetSupport

Property	Type	Default	Description
defaultHeaders	**Map<String,Object>**	**null**	Specifies the default message body. For SimpleDataSet it is a constant payload; though if you want to create custom payloads per message, create your own derivation of **DataSetSupport**.

Property	Type	Default	Description
outputTransformer	**org.apache.camel.Processor**	null	
size	**long**	**10**	Specifies how many messages to send/consume.
reportCount	**long**	**-1**	Specifies the number of messages to be received before reporting progress. Useful for showing progress of a large load test. If < 0, then **size** / 5, if is 0 then **size**, else set to **reportCount** value.

78.6. SIMPLEDATASET

The **SimpleDataSet** extends **DataSetSupport**, and adds a default body.

78.6.1. Additional Properties on SimpleDataSet

Property	Type	Default	Description
defaultBody	**Object**	**<hello>world!</hello>**	Specifies the default message body. By default, the **SimpleDataSet** produces the same constant payload for each exchange. If you want to customize the payload for each exchange, create a Camel **Processor** and configure the **SimpleDataSet** to use it by setting the **outputTransformer** property.

78.7. LISTDATASET

Available since Camel 2.17

The List`DataSet` extends **DataSetSupport**, and adds a list of default bodies.

78.7.1. Additional Properties on ListDataSet

Property	Type	Default	Description
defaultBodies	**List<Object>**	**empty LinkedList<Object>**	Specifies the default message body. By default, the **ListDataSet** selects a constant payload from the list of **defaultBodies** using the **CamelDataSetIndex**. If you want to customize the payload, create a Camel **Processor** and configure the **ListDataSet** to use it by setting the **outputTransformer** property.

Property	Type	Default	Description
size	**long**	the size of the default Bodies list	Specifies how many messages to send/consume. This value can be different from the size of the **defaultBodies** list. If the value is less than the size of the **defaultBodies** list, some of the list elements will not be used. If the value is greater than the size of the **defaultBodies** list, the payload for the exchange will be selected using the modulus of the **CamelDataSetIndex** and the size of the**defaultBodies** list (i.e. **CamelDataSetIndex % defaultBodies.size()**)

78.8. FILEDATASET

Available since Camel 2.17

The **FileDataSet** extends **ListDataSet**, and adds support for loading the bodies from a file.

78.8.1. Additional Properties on FileDataSet

Property	Type	Default	Description
sourceFile	**File**	null	Specifies the source file for payloads
delimiter	**String**	\z	Specifies the delimiter pattern used by a **java.util.Scanner** to split the file into multiple payloads.

CHAPTER 79. DIGITALOCEAN COMPONENT

Available as of Camel version 2.19

The **DigitalOcean** component allows you to manage Droplets and resources within the DigitalOcean cloud with **Camel** by encapsulating [digitalocean-api-java] (https://www.digitalocean.com/community/projects/api-client-in-java). All of the functionality that you are familiar with in the DigitalOcean control panel is also available through this Camel component.

79.1. PREREQUISITES

You must have a valid DigitalOcean account and a valid OAuth token. You can generate an OAuth token by visiting the [Apps & API](https://cloud.digitalocean.com/settings/applications) section of the DigitalOcean control panel for your account.

79.2. URI FORMAT

The **DigitalOcean Component** uses the following URI format:

```
digitalocean://endpoint?[options]
```

where **endpoint** is a DigitalOcean resource type.

Example : to list your droplets:

```
digitalocean://droplets?operation=list&oAuthToken=XXXXXX&page=1&perPage=10
```

The DigitalOcean component only supports producer endpoints so you cannot use this component at the beginning of a route to listen to messages in a channel.

79.3. OPTIONS

The DigitalOcean component has no options.

The DigitalOcean endpoint is configured using URI syntax:

```
digitalocean:operation
```

with the following path and query parameters:

79.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
operation	The operation to perform to the given resource.		DigitalOceanOperations

79.3.2. Query Parameters (10 parameters):

Name	Description	Default	Type
page (producer)	Use for pagination. Force the page number.	1	Integer
perPage (producer)	Use for pagination. Set the number of item per request. The maximum number of results per page is 200.	25	Integer
resource (producer)	**Required** The DigitalOcean resource type on which perform the operation.		DigitalOceanResources
digitalOceanClient (advanced)	To use a existing configured DigitalOceanClient as client		DigitalOceanClient
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
httpProxyHost (proxy)	Set a proxy host if needed		String
httpProxyPassword (proxy)	Set a proxy password if needed		String
httpProxyPort (proxy)	Set a proxy port if needed		Integer
httpProxyUser (proxy)	Set a proxy host if needed		String
oAuthToken (security)	DigitalOcean OAuth Token		String

You have to provide an **operation** value for each endpoint, with the **operation** URI option or the **CamelDigitalOceanOperation** message header.

All **operation** values are defined in **DigitalOceanOperations** enumeration.

All **header** names used by the component are defined in **DigitalOceanHeaders** enumeration.

79.4. MESSAGE BODY RESULT

All message bodies returned are using objects provided by the **digitalocean-api-java** library.

79.5. API RATE LIMITS

DigitalOcean REST API encapsulated by camel-digitalocean component is subjected to API Rate Limiting. You can find the per method limits in the [API Rate Limits documentation] (https://developers.digitalocean.com/documentation/v2/#rate-limit).

79.6. ACCOUNT ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **get** | get account info | | **com.myjeeva.digitalocean.pojo.Account** |

79.7. BLOCKSTORAGES ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all of the Block Storage volumes available on your account | | **List<com.myjeeva.digitalocean.pojo.Volume>** | | **get** | show information about a Block Storage volume| **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Volume** | | **get** | show information about a Block Storage volume by name| **CamelDigitalOceanName** *String*
`CamelDigitalOceanRegion` *String*| **com.myjeeva.digitalocean.pojo.Volume** | | **listSnapshots** | retrieve the snapshots that have been created from a volume | **CamelDigitalOceanId** *Integer*| **List<com.myjeeva.digitalocean.pojo.Snapshot>** | | **create** | create a new volume | **CamelDigitalOceanVolumeSizeGigabytes** *Integer*
`CamelDigitalOceanName` *String*
`CamelDigitalOceanDescription`* *String*
`CamelDigitalOceanRegion`* *String*| **com.myjeeva.digitalocean.pojo.Volume** | | **delete** | delete a Block Storage volume, destroying all data and removing it from your account| **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Delete**| | **delete** | delete a Block Storage volume by name| **CamelDigitalOceanName** *String*
`CamelDigitalOceanRegion` *String*| **com.myjeeva.digitalocean.pojo.Delete** | **attach** | attach a Block Storage volume to a Droplet| **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanDropletId` *Integer*
`CamelDigitalOceanDropletRegion` *String*| **com.myjeeva.digitalocean.pojo.Action** | **attach** | attach a Block Storage volume to a Droplet by name| **CamelDigitalOceanName** *String*
`CamelDigitalOceanDropletId` *Integer*
`CamelDigitalOceanDropletRegion` *String*| **com.myjeeva.digitalocean.pojo.Action** | **detach** | detach a Block Storage volume from a Droplet| **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanDropletId` *Integer*
`CamelDigitalOceanDropletRegion` *String*| **com.myjeeva.digitalocean.pojo.Action** | **attach** | detach a Block Storage volume from a Droplet by name| **CamelDigitalOceanName** *String*
`CamelDigitalOceanDropletId` *Integer*
`CamelDigitalOceanDropletRegion` *String*| **com.myjeeva.digitalocean.pojo.Action** | **resize** | resize a Block Storage volume | **CamelDigitalOceanVolumeSizeGigabytes** *Integer*
`CamelDigitalOceanRegion` *String*| **com.myjeeva.digitalocean.pojo.Action** | | **listActions** | retrieve all actions that have been executed on a volume | **CamelDigitalOceanId** *Integer*| **List<com.myjeeva.digitalocean.pojo.Action>** |

79.8. DROPLETS ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all Droplets in your account | | **List<com.myjeeva.digitalocean.pojo.Droplet>** | | **get** | show an individual droplet | **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Droplet** | | **create** | create a new Droplet | **CamelDigitalOceanName** *String*
`CamelDigitalOceanDropletImage` *String*
`CamelDigitalOceanRegion` *String*
`CamelDigitalOceanDropletSize` *String*
`CamelDigitalOceanDropletSSHKeys`* *List\<String\>*
`CamelDigitalOceanDropletEnableBackups`* *Boolean*
`CamelDigitalOceanDropletEnableIpv6`* *Boolean*
`CamelDigitalOceanDropletEnablePrivateNetworking`* *Boolean*
`CamelDigitalOceanDropletUserData`* *String*
`CamelDigitalOceanDropletVolumes`* *List\<String\>*
`CamelDigitalOceanDropletTags` *List\<String\>*| **com.myjeeva.digitalocean.pojo.Droplet** | | **create** | create multiple Droplets | **CamelDigitalOceanNames** *List\<String\>*
`CamelDigitalOceanDropletImage` *String*
`CamelDigitalOceanRegion` *String*
`CamelDigitalOceanDropletSize` *String*
`CamelDigitalOceanDropletSSHKeys`* *List\<String\>*
`CamelDigitalOceanDropletEnableBackups`* *Boolean*
`CamelDigitalOceanDropletEnableIpv6`* *Boolean*

`CamelDigitalOceanDropletEnablePrivateNetworking`* *Boolean*

`CamelDigitalOceanDropletUserData`* *String*
`CamelDigitalOceanDropletVolumes`* *List*\
<*String*\>
`CamelDigitalOceanDropletTags` *List*\<*String*\>|
com.myjeeva.digitalocean.pojo.Droplet | | **delete** | delete a Droplet, | **CamelDigitalOceanId** *Integer*|
com.myjeeva.digitalocean.pojo.Delete | | **enableBackups** | enable backups on an existing Droplet |
CamelDigitalOceanId *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **disableBackups** | disable backups on an existing Droplet | **CamelDigitalOceanId** *Integer*|
com.myjeeva.digitalocean.pojo.Action | | **enableIpv6** | enable IPv6 networking on an existing Droplet | **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **enablePrivateNetworking** | enable private networking on an existing Droplet | **CamelDigitalOceanId** *Integer*|
com.myjeeva.digitalocean.pojo.Action | | **reboot** | reboot a Droplet | **CamelDigitalOceanId** *Integer*|
com.myjeeva.digitalocean.pojo.Action | | **powerCycle** | power cycle a Droplet |
CamelDigitalOceanId *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **shutdown** | shutdown a Droplet | **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **powerOff** | power off a Droplet | **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **powerOn** | power on a Droplet | **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **restore** | shutdown a Droplet | **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanImageId` *Integer*|
com.myjeeva.digitalocean.pojo.Action | | **passwordReset** | reset the password for a Droplet |
CamelDigitalOceanId *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **resize** | resize a Droplet |
CamelDigitalOceanId *Integer*
`CamelDigitalOceanDropletSize` *String*|
com.myjeeva.digitalocean.pojo.Action | | **rebuild** | rebuild a Droplet | **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanImageId` *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **rename** | rename a Droplet | **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanName` *String*|
com.myjeeva.digitalocean.pojo.Action | | **changeKernel** | change the kernel of a Droplet |
CamelDigitalOceanId *Integer*
`CamelDigitalOceanKernelId` *Integer*|
com.myjeeva.digitalocean.pojo.Action | | **takeSnapshot** | snapshot a Droplet | **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanName`* *String*| **com.myjeeva.digitalocean.pojo.Action** | | **tag** | tag a Droplet | **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanName` *String*|
com.myjeeva.digitalocean.pojo.Response | | **untag** | untag a Droplet | **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanName` *String*| **com.myjeeva.digitalocean.pojo.Response** | | **listKernels** | retrieve a list of all kernels available to a Droplet | **CamelDigitalOceanId** *Integer* |
List<com.myjeeva.digitalocean.pojo.Kernel> | | **listSnapshots** | retrieve the snapshots that have been created from a Droplet | **CamelDigitalOceanId** *Integer* |
List<com.myjeeva.digitalocean.pojo.Snapshot> | | **listBackups** | retrieve any backups associated with a Droplet | **CamelDigitalOceanId** *Integer* | **List<com.myjeeva.digitalocean.pojo.Backup>** | | **listActions** | retrieve all actions that have been executed on a Droplet | **CamelDigitalOceanId** *Integer* |
List<com.myjeeva.digitalocean.pojo.Action> | | **listNeighbors** | retrieve a list of droplets that are running on the same physical server | **CamelDigitalOceanId** *Integer* |
List<com.myjeeva.digitalocean.pojo.Droplet> | | **listAllNeighbors** | retrieve a list of any droplets that are running on the same physical hardware | | **List<com.myjeeva.digitalocean.pojo.Droplet>** |

79.9. IMAGES ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ----------- | | **list** | list images available on your account | **CamelDigitalOceanType*** *DigitalOceanImageTypes* |
List<com.myjeeva.digitalocean.pojo.Image> | | **ownList** | retrieve only the private images of a user | |
List<com.myjeeva.digitalocean.pojo.Image> | | **listActions** | retrieve all actions that have been executed on a Image | **CamelDigitalOceanId** *Integer* | **List<com.myjeeva.digitalocean.pojo.Action>** |
| **get** | retrieve information about an image (public or private) by id| **CamelDigitalOceanId** *Integer*|
com.myjeeva.digitalocean.pojo.Image | | **get** | retrieve information about an public image by slug|
CamelDigitalOceanDropletImage *String*| **com.myjeeva.digitalocean.pojo.Image** | | **update** | update an image| **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanName` *String*|
com.myjeeva.digitalocean.pojo.Image | | **delete** | delete an image| **CamelDigitalOceanId** *Integer* |
com.myjeeva.digitalocean.pojo.Delete | | **transfer** | transfer an image to another region|

CamelDigitalOceanId *Integer*
`CamelDigitalOceanRegion` *String*| **com.myjeeva.digitalocean.pojo.Action** | | **convert** | convert an image, for example, a backup to a snapshot| **CamelDigitalOceanId** *Integer* | **com.myjeeva.digitalocean.pojo.Action** |

79.10. SNAPSHOTS ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all of the snapshots available on your account | **CamelDigitalOceanType*** *DigitalOceanSnapshotTypes* | **List<com.myjeeva.digitalocean.pojo.Snapshot>** | | **get** | retrieve information about a snapshot| **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Snapshot** | | **delete** | delete an snapshot| **CamelDigitalOceanId** *Integer* | **com.myjeeva.digitalocean.pojo.Delete** |

79.11. KEYS ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all of the keys in your account | | **List<com.myjeeva.digitalocean.pojo.Key>** | | **get** | retrieve information about a key by id| **CamelDigitalOceanId** *Integer*| **com.myjeeva.digitalocean.pojo.Key** | | **get** | retrieve information about a key by fingerprint| **CamelDigitalOceanKeyFingerprint** *String*| **com.myjeeva.digitalocean.pojo.Key** | | **update** | update a key by id| **CamelDigitalOceanId** *Integer*
`CamelDigitalOceanName` *String*| **com.myjeeva.digitalocean.pojo.Key** | | **update** | update a key by fingerprint| **CamelDigitalOceanKeyFingerprint** *String*
`CamelDigitalOceanName` *String*| **com.myjeeva.digitalocean.pojo.Key** | | **delete** | delete a key by id| **CamelDigitalOceanId** *Integer* | **com.myjeeva.digitalocean.pojo.Delete** | | **delete** | delete a key by fingerprint| **CamelDigitalOceanKeyFingerprint** *String* | **com.myjeeva.digitalocean.pojo.Delete** |

79.12. REGIONS ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all of the regions that are available | | **List<com.myjeeva.digitalocean.pojo.Region>** |

79.13. SIZES ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all of the sizes that are available | | **List<com.myjeeva.digitalocean.pojo.Size>** |

79.14. FLOATING IPS ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all of the Floating IPs available on your account | | **List<com.myjeeva.digitalocean.pojo.FloatingIP>** | | **create** | create a new Floating IP assigned to a Droplet | **CamelDigitalOceanId** *Integer* | **List<com.myjeeva.digitalocean.pojo.FloatingIP>** | | **create** | create a new Floating IP assigned to a Region | **CamelDigitalOceanRegion** *String* | **List<com.myjeeva.digitalocean.pojo.FloatingIP>** | | **get** | retrieve information about a Floating IP| **CamelDigitalOceanFloatingIPAddress** *String*| **com.myjeeva.digitalocean.pojo.Key** | | **delete** | delete a Floating IP and remove it from your account| **CamelDigitalOceanFloatingIPAddress** *String*| **com.myjeeva.digitalocean.pojo.Delete** | | **assign** | assign a Floating IP to a Droplet| **CamelDigitalOceanFloatingIPAddress** *String*
`CamelDigitalOceanDropletId` *Integer*| **com.myjeeva.digitalocean.pojo.Action** | | **unassign** | unassign a Floating IP | **CamelDigitalOceanFloatingIPAddress** *String* | **com.myjeeva.digitalocean.pojo.Action** | | **listActions** | retrieve all actions that have been executed on a Floating IP | **CamelDigitalOceanFloatingIPAddress** *String* | **List<com.myjeeva.digitalocean.pojo.Action>** |

79.15. TAGS ENDPOINT

| operation | Description | Headers | Result | | ------ | ---- | ------- | ---------- | | **list** | list all of your tags | | **List<com.myjeeva.digitalocean.pojo.Tag>** | | **create** | create a Tag | **CamelDigitalOceanName** *String* | **com.myjeeva.digitalocean.pojo.Tag** | | **get** | retrieve an individual tag | **CamelDigitalOceanName** *String* | **com.myjeeva.digitalocean.pojo.Tag** | | **delete** | delete a tag | **CamelDigitalOceanName** *String* | **com.myjeeva.digitalocean.pojo.Delete** | | **update** | update a tag | **CamelDigitalOceanName** *String*
`CamelDigitalOceanNewName` *String*| **com.myjeeva.digitalocean.pojo.Tag** |

79.16. EXAMPLES

Get your account info

```
from("direct:getAccountInfo")
    .setHeader(DigitalOceanConstants.OPERATION, constant(DigitalOceanOperations.get))
    .to("digitalocean:account?oAuthToken=XXXXXX")
```

Create a droplet

```
from("direct:createDroplet")
    .setHeader(DigitalOceanConstants.OPERATION, constant("create"))
    .setHeader(DigitalOceanHeaders.NAME, constant("myDroplet"))
    .setHeader(DigitalOceanHeaders.REGION, constant("fra1"))
    .setHeader(DigitalOceanHeaders.DROPLET_IMAGE, constant("ubuntu-14-04-x64"))
    .setHeader(DigitalOceanHeaders.DROPLET_SIZE, constant("512mb"))
    .to("digitalocean:droplet?oAuthToken=XXXXXX")
```

List all your droplets

```
from("direct:getDroplets")
    .setHeader(DigitalOceanConstants.OPERATION, constant("list"))
    .to("digitalocean:droplets?oAuthToken=XXXXXX")
```

Retrieve information for the Droplet (dropletId = 34772987)

```
from("direct:getDroplet")
    .setHeader(DigitalOceanConstants.OPERATION, constant("get"))
    .setHeader(DigitalOceanConstants.ID, 34772987)
    .to("digitalocean:droplet?oAuthToken=XXXXXX")
```

Shutdown information for the Droplet (dropletId = 34772987)

```
from("direct:shutdown")
    .setHeader(DigitalOceanConstants.ID, 34772987)
    .to("digitalocean:droplet?operation=shutdown&oAuthToken=XXXXXX")
```

CHAPTER 80. DIRECT COMPONENT

Available as of Camel version 1.0

The **direct:** component provides direct, synchronous invocation of any consumers when a producer sends a message exchange.
This endpoint can be used to connect existing routes in the **same** camel context.

TIP

Asynchronous The SEDA component provides asynchronous invocation of any consumers when a producer sends a message exchange.

TIP

Connection to other camel contexts The VM component provides connections between Camel contexts as long they run in the same **JVM**.

80.1. URI FORMAT

```
direct:someName[?options]
```

Where **someName** can be any string to uniquely identify the endpoint

80.2. OPTIONS

The Direct component supports 3 options which are listed below.

Name	Description	Default	Type
block (producer)	If sending a message to a direct endpoint which has no active consumer, then we can tell the producer to block and wait for the consumer to become active.	true	boolean
timeout (producer)	The timeout value to use if block is enabled.	30000	long
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Direct endpoint is configured using URI syntax:

```
direct:name
```

with the following path and query parameters:

80.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of direct endpoint		String

80.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
block (producer)	If sending a message to a direct endpoint which has no active consumer, then we can tell the producer to block and wait for the consumer to become active.	true	boolean
failIfNoConsumers (producer)	Whether the producer should fail by throwing an exception, when sending to a DIRECT endpoint with no active consumers.	false	boolean
timeout (producer)	The timeout value to use if block is enabled.	30000	long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

80.3. SAMPLES

In the route below we use the direct component to link the two routes together:

```
from("activemq:queue:order.in")
```

```
    .to("bean:orderServer?method=validate")
    .to("direct:processOrder");

from("direct:processOrder")
    .to("bean:orderService?method=process")
    .to("activemq:queue:order.out");
```

And the sample using spring DSL:

```
<route>
 <from uri="activemq:queue:order.in"/>
 <to uri="bean:orderService?method=validate"/>
 <to uri="direct:processOrder"/>
</route>

<route>
 <from uri="direct:processOrder"/>
 <to uri="bean:orderService?method=process"/>
 <to uri="activemq:queue:order.out"/>
</route>
```

See also samples from the SEDA component, how they can be used together.

80.4. SEE ALSO

- SEDA
- VM

CHAPTER 81. DIRECT VM COMPONENT

Available as of Camel version 2.10

The **direct-vm:** component provides direct, synchronous invocation of any consumers in the JVM when a producer sends a message exchange.
This endpoint can be used to connect existing routes in the same camel context, as well from other camel contexts in the **same** JVM.

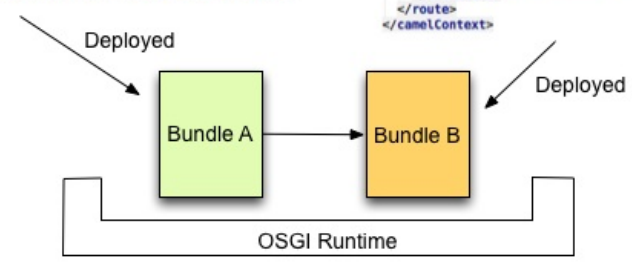
This component differs from the Direct component in that Direct-VM supports communication across CamelContext instances - so you can use this mechanism to communicate across web applications (provided that camel-core.jar is on the system/boot classpath).

At runtime you can swap in new consumers, by stopping the existing consumer(s) and start new consumers.
But at any given time there can be at most only one active consumer for a given endpoint.

This component allows also to connect routes deployed in different OSGI Bundles as you can see here after. Even if they are running in different bundles, the camel routes will use
the same thread. That autorises to develop applications using Transactions - Tx.

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="timer:foo?period=5000"/>
    <log message=">> Here is a message from direct-vm (bundle1)"/>
    <log message=">> Using ${threadName} of bundle 1"/>
    <to uri="direct-vm:hello"/>
    <log message=">> Response received from bundle 2 : ${body}"/>
  </route>
</camelContext>
```

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct-vm:hello"/>
    <log message=">> Prepare response from bundle 2"/>
    <log message=">> Using ${threadName} of bundle 2"/>
    <transform>
        <method ref="helloBean" method="hello"/>
    </transform>
  </route>
</camelContext>
```




```
INFO  | 14 - timer://foo | route7  >> Here is a message from direct-vm (bundle1)
INFO  | 14 - timer://foo | route7  >> Using Camel (89-camel-23) thread #14 - timer://foo of bundle 1
INFO  | 14 - timer://foo | route8  >> Prepare response from bundle 2
INFO  | 14 - timer://foo | route8  >> Using Camel (89-camel-23) thread #14 - timer://foo of bundle 2
INFO  | 14 - timer://foo | route7  >> Response received from bundle 2 : Hi from Camel direct-vm at 2012-06-21 15:21:22
```

81.1. URI FORMAT

```
direct-vm:someName
```

Where **someName** can be any string to uniquely identify the endpoint

81.2. OPTIONS

The Direct VM component supports 5 options which are listed below.

Name	Description	Default	Type

Name	Description	Default	Type
block (producer)	If sending a message to a direct endpoint which has no active consumer, then we can tell the producer to block and wait for the consumer to become active.	true	boolean
timeout (producer)	The timeout value to use if block is enabled.	30000	long
headerFilterStrategy (advanced)	Sets a HeaderFilterStrategy that will only be applied on producer endpoints (on both directions: request and response). Default value: none.		HeaderFilterStrategy
propagateProperties (advanced)	Whether to propagate or not properties from the producer side to the consumer side, and vice versa. Default value: true.	true	boolean
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Direct VM endpoint is configured using URI syntax:

```
direct-vm:name
```

with the following path and query parameters:

81.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of direct-vm endpoint		String

81.2.2. Query Parameters (9 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
block (producer)	If sending a message to a direct endpoint which has no active consumer, then we can tell the producer to block and wait for the consumer to become active.	true	boolean
failIfNoConsumers (producer)	Whether the producer should fail by throwing an exception, when sending to a Direct-VM endpoint with no active consumers.	false	boolean
timeout (producer)	The timeout value to use if block is enabled.	30000	long
headerFilterStrategy (producer)	Sets a HeaderFilterStrategy that will only be applied on producer endpoints (on both directions: request and response). Default value: none.		HeaderFilterStrategy
propagateProperties (advanced)	Whether to propagate or not properties from the producer side to the consumer side, and vice versa. Default value: true.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

81.3. SAMPLES

In the route below we use the direct component to link the two routes together:

-

```
from("activemq:queue:order.in")
    .to("bean:orderServer?method=validate")
    .to("direct-vm:processOrder");
```

And now in another CamelContext, such as another OSGi bundle

```
from("direct-vm:processOrder")
    .to("bean:orderService?method=process")
    .to("activemq:queue:order.out");
```

And the sample using spring DSL:

```
<route>
 <from uri="activemq:queue:order.in"/>
 <to uri="bean:orderService?method=validate"/>
 <to uri="direct-vm:processOrder"/>
</route>

<route>
 <from uri="direct-vm:processOrder"/>
 <to uri="bean:orderService?method=process"/>
 <to uri="activemq:queue:order.out"/>
</route>
```

81.4. SEE ALSO

- Direct
- SEDA
- VM

CHAPTER 82. DISRUPTOR COMPONENT

Available as of Camel version 2.12

The **disruptor:** component provides asynchronous SEDA behavior much as the standard SEDA Component, but utilizes a Disruptor instead of a BlockingQueue utilized by the standard SEDA. Alternatively, a

disruptor-vm: endpoint is supported by this component, providing an alternative to the standard VM. As with the SEDA component, buffers of the **disruptor:** endpoints are only visible within a **single** CamelContext and no support is provided for persistence or recovery. The buffers of the **disruptor-vm:** endpoints also provides support for communication across CamelContexts instances so you can use this mechanism to communicate across web applications (provided that **camel-disruptor.jar** is on the **system/boot** classpath).

The main advantage of choosing to use the Disruptor Component over the SEDA or the VM Component is performance in use cases where there is high contention between producer(s) and/or multicasted or concurrent Consumers. In those cases, significant increases of throughput and reduction of latency has been observed. Performance in scenarios without contention is comparable to the SEDA and VM Components.

The Disruptor is implemented with the intention of mimicing the behaviour and options of the SEDA and VM Components as much as possible. The main differences with the them are the following:

- The buffer used is always bounded in size (default 1024 exchanges).
- As a the buffer is always bouded, the default behaviour for the Disruptor is to block while the buffer is full instead of throwing an exception. This default behaviour may be configured on the component (see options).
- The Disruptor enpoints don't implement the BrowsableEndpoint interface. As such, the exchanges currently in the Disruptor can't be retrieved, only the amount of exchanges.
- The Disruptor requires its consumers (multicasted or otherwise) to be statically configured. Adding or removing consumers on the fly requires complete flushing of all pending exchanges in the Disruptor.
- As a result of the reconfiguration: Data sent over a Disruptor is directly processed and 'gone' if there is at least one consumer, late joiners only get new exchanges published after they've joined.
- The **pollTimeout** option is not supported by the Disruptor Component.
- When a producer blocks on a full Disruptor, it does not respond to thread interrupts.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-disruptor</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

82.1. URI FORMAT

```
disruptor:someName[?options]
```

or

```
disruptor-vm:someName[?options]
```

Where **someName** can be any string that uniquely identifies the endpoint within the current CamelContext (or across contexts in case of **disruptor-vm:**).
You can append query options to the URI in the following format:

```
?option=value&option=value&…
```

82.2. OPTIONS

All the following options are valid for both the **disruptor:** and **disruptor-vm:** components.

The Disruptor component supports 8 options which are listed below.

Name	Description	Default	Type
defaultConcurrent Consumers (consumer)	To configure the default number of concurrent consumers	1	int
defaultMultiple Consumers (consumer)	To configure the default value for multiple consumers	false	boolean
defaultProducerType (producer)	To configure the default value for DisruptorProducerType The default value is Multi.	Multi	DisruptorProducerType
defaultWaitStrategy (consumer)	To configure the default value for DisruptorWaitStrategy The default value is Blocking.	Blocking	DisruptorWaitStrategy
defaultBlockWhenFull (producer)	To configure the default value for block when full The default value is true.	true	boolean
queueSize (common)	**Deprecated** To configure the ring buffer size		int
bufferSize (common)	To configure the ring buffer size	1024	int
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Disruptor endpoint is configured using URI syntax:

```
disruptor:name
```

with the following path and query parameters:

82.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of queue		String

82.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type
size (common)	The maximum capacity of the Disruptors ringbuffer Will be effectively increased to the nearest power of two. Notice: Mind if you use this option, then its the first endpoint being created with the queue name, that determines the size. To make sure all endpoints use same size, then configure the size option on all of them, or the first endpoint being created.	1024	int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
concurrentConsumers (consumer)	Number of concurrent threads processing exchanges.	1	int
multipleConsumers (consumer)	Specifies whether multiple consumers are allowed. If enabled, you can use Disruptor for Publish-Subscribe messaging. That is, you can send a message to the queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.	false	boolean
waitStrategy (consumer)	Defines the strategy used by consumer threads to wait on new exchanges to be published. The options allowed are:Blocking, Sleeping, BusySpin and Yielding.	Blocking	DisruptorWaitStrategy

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
blockWhenFull (producer)	Whether a thread that sends messages to a full Disruptor will block until the ringbuffer's capacity is no longer exhausted. By default, the calling thread will block and wait until the message can be accepted. By disabling this option, an exception will be thrown stating that the queue is full.	false	boolean
producerType (producer)	Defines the producers allowed on the Disruptor. The options allowed are: Multi to allow multiple producers and Single to enable certain optimizations only allowed when one concurrent producer (on one thread or otherwise synchronized) is active.	Multi	DisruptorProducerType
timeout (producer)	Timeout (in milliseconds) before a producer will stop waiting for an asynchronous task to complete. You can disable timeout by using 0 or a negative value.	30000	long
waitForTaskToComplete (producer)	Option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: Always, Never or IfReplyExpected. The first two values are self-explanatory. The last value, IfReplyExpected, will only wait if the message is Request Reply based.	IfReplyExpected	WaitForTaskToComplete
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

82.3. WAIT STRATEGIES

The wait strategy effects the type of waiting performed by the consumer threads that are currently waiting for the next exchange to be published. The following strategies can be chosen:

Name	Description	Advice
Blocking	Blocking strategy that uses a lock and condition variable for Consumers waiting on a barrier.	This strategy can be used when throughput and low-latency are not as important as CPU resource.

Name	Description	Advice
Sleeping	Sleeping strategy that initially spins, then uses a Thread.yield(), and eventually for the minimum number of nanos the OS and JVM will allow while the Consumers are waiting on a barrier.	This strategy is a good compromise between performance and CPU resource. Latency spikes can occur after quiet periods.
BusySpin	Busy Spin strategy that uses a busy spin loop for Consumers waiting on a barrier.	This strategy will use CPU resource to avoid syscalls which can introduce latency jitter. It is best used when threads can be bound to specific CPU cores.
Yielding	Yielding strategy that uses a Thread.yield() for Consumers waiting on a barrier after an initially spinning.	This strategy is a good compromise between performance and CPU resource without incurring significant latency spikes.

82.4. USE OF REQUEST REPLY

The Disruptor component supports using Request Reply, where the caller will wait for the Async route to complete. For instance:

```
from("mina:tcp://0.0.0.0:9876?textline=true&sync=true").to("disruptor:input");
from("disruptor:input").to("bean:processInput").to("bean:createResponse");
```

In the route above, we have a TCP listener on port 9876 that accepts incoming requests. The request is routed to the *disruptor:input* buffer. As it is a Request Reply message, we wait for the response. When the consumer on the *disruptor:input* buffer is complete, it copies the response to the original message response.

82.5. CONCURRENT CONSUMERS

By default, the Disruptor endpoint uses a single consumer thread, but you can configure it to use concurrent consumer threads. So instead of thread pools you can use:

```
from("disruptor:stageName?concurrentConsumers=5").process(...)
```

As for the difference between the two, note a thread pool can increase/shrink dynamically at runtime depending on load, whereas the number of concurrent consumers is always fixed and supported by the Disruptor internally so performance will be higher.

82.6. THREAD POOLS

Be aware that adding a thread pool to a Disruptor endpoint by doing something like:

```
from("disruptor:stageName").thread(5).process(...)
```

Can wind up with adding a normal BlockingQueue to be used in conjunction with the Disruptor, effectively negating part of the performance gains achieved by using the Disruptor. Instead, it is advices to directly configure number of threads that process messages on a Disruptor endpoint using the concurrentConsumers option.

82.7. SAMPLE

In the route below we use the Disruptor to send the request to this async queue to be able to send a fire-and-forget message for further processing in another thread, and return a constant reply in this thread to the original caller.

```
public void configure() throws Exception {
    from("direct:start")
        // send it to the disruptor that is async
        .to("disruptor:next")
        // return a constant response
        .transform(constant("OK"));

    from("disruptor:next").to("mock:result");
}
```

Here we send a Hello World message and expects the reply to be OK.

```
Object out = template.requestBody("direct:start", "Hello World");
assertEquals("OK", out);
```

The "Hello World" message will be consumed from the Disruptor from another thread for further processing. Since this is from a unit test, it will be sent to a mock endpoint where we can do assertions in the unit test.

82.8. USING MULTIPLECONSUMERS

In this example we have defined two consumers and registered them as spring beans.

```
<!-- define the consumers as spring beans -->
<bean id="consumer1" class="org.apache.camel.spring.example.FooEventConsumer"/>

<bean id="consumer2" class="org.apache.camel.spring.example.AnotherFooEventConsumer"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define a shared endpoint which the consumers can refer to instead of using url -->
    <endpoint id="foo" uri="disruptor:foo?multipleConsumers=true"/>
</camelContext>
```

Since we have specified multipleConsumers=true on the Disruptor foo endpoint we can have those two or more consumers receive their own copy of the message as a kind of pub-sub style messaging. As the beans are part of an unit test they simply send the message to a mock endpoint, but notice how we can use @Consume to consume from the Disruptor.

```
public class FooEventConsumer {

    @EndpointInject(uri = "mock:result")
    private ProducerTemplate destination;

    @Consume(ref = "foo")
    public void doSomething(String body) {
        destination.sendBody("foo" + body);
```

```
    }

}
```

82.9. EXTRACTING DISRUPTOR INFORMATION

If needed, information such as buffer size, etc. can be obtained without using JMX in this fashion:

```
DisruptorEndpoint disruptor = context.getEndpoint("disruptor:xxxx");
int size = disruptor.getBufferSize();
```

CHAPTER 83. DNS COMPONENT

Available as of Camel version 2.7

This is an additional component for Camel to run DNS queries, using DNSJava. The component is a thin layer on top of DNSJava.
The component offers the following operations:

- ip, to resolve a domain by its ip
- lookup, to lookup information about the domain
- dig, to run DNS queries

INFO:*Requires SUN JVM* The DNSJava library requires running on the SUN JVM.
If you use Apache ServiceMix or Apache Karaf, you'll need to adjust the **etc/jre.properties** file, to add **sun.net.spi.nameservice** to the list of Java platform packages exported. The server will need restarting before this change takes effect.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-dns</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

83.1. URI FORMAT

The URI scheme for a DNS component is as follows

```
dns://operation[?options]
```

This component only supports producers.

83.2. OPTIONS

The DNS component has no options.

The DNS endpoint is configured using URI syntax:

```
dns:dnsType
```

with the following path and query parameters:

83.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
dnsType	**Required** The type of the lookup.		DnsType

83.2.2. Query Parameters (1 parameters):

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

83.3. HEADERS

Header	Type	Operations	Description
dns.domain	String	ip	The domain name. Mandatory.
dns.name	String	lookup	The name to lookup. Mandatory.
dns.type		lookup, dig	The type of the lookup. Should match the values of **org.xbill.dns.Type**. Optional.
dns.class		lookup, dig	The DNS class of the lookup. Should match the values of **org.xbill.dns.DClass**. Optional.
dns.query	String	dig	The query itself. Mandatory.
dns.server	String	dig	The server in particular for the query. If none is given, the default one specified by the OS will be used. Optional.

83.4. EXAMPLES

83.4.1. IP lookup

```
<route id="IPCheck">
    <from uri="direct:start"/>
    <to uri="dns:ip"/>
</route>
```

This looks up a domain's IP. For example, www.example.com resolves to 192.0.32.10.
The IP address to lookup must be provided in the header with key **"dns.domain"**.

83.4.2. DNS lookup

```
<route id="IPCheck">
    <from uri="direct:start"/>
```

```
        <to uri="dns:lookup"/>
    </route>
```

This returns a set of DNS records associated with a domain.
The name to lookup must be provided in the header with key **"dns.name"**.

83.4.3. DNS Dig

Dig is a Unix command-line utility to run DNS queries.

```
    <route id="IPCheck">
        <from uri="direct:start"/>
        <to uri="dns:dig"/>
    </route>
```

The query must be provided in the header with key **"dns.query"**.

83.5. DNS ACTIVATION POLICY

DnsActivationPolicy can be used to dynamically start and stop routes based on dns state.

If you have instances of the same component running in different regions you can configure a route in each region to activate only if dns is pointing to its region.

i.e. You may have an instance in NYC and an instance in SFO. You would configure a service CNAME service.example.com to point to nyc-service.example.com to bring NYC instance up and SFO instance down. When you change the CNAME service.example.com to point to sfo-service.example.com – nyc instance would stop its routes and sfo will bring its routes up. This allows you to switch regions without restarting actual components.

```
 <bean id="dnsActivationPolicy"
class="org.apache.camel.component.dns.policy.DnsActivationPolicy">
  <property name="hostname" value="service.example.com" />
  <property name="resolvesTo" value="nyc-service.example.com" />
  <property name="ttl" value="60000" />
 </bean>

 <route id="routeId" autoStartup="false" routePolicyRef="dnsActivationPolicy">
 </route>
```

CHAPTER 84. DOCKER COMPONENT

Available as of Camel version 2.15

Camel component for communicating with Docker.

The Docker Camel component leverages the docker-java via the Docker Remote API.

84.1. URI FORMAT

```
docker://[operation]?[options]
```

Where **operation** is the specific action to perform on Docker.

84.2. GENERAL OPTIONS

The Docker component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared docker configuration		DockerConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Docker endpoint is configured using URI syntax:

```
docker:operation
```

with the following path and query parameters:

84.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
operation	**Required** Which operation to use		DockerOperation

84.2.2. Query Parameters (20 parameters):

Name	Description	Default	Type
email (common)	Email address associated with the user		String
host (common)	**Required** Docker host	localhost	String
port (common)	**Required** Docker port	2375	Integer
requestTimeout (common)	Request timeout for response (in seconds)		Integer
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
cmdExecFactory (advanced)	The fully qualified class name of the DockerCmdExecFactory implementation to use	com.github.dockerjava.netty.NettyDockerCmdExecFactory	String
followRedirectFilter (advanced)	Whether to follow redirect filter	false	boolean
loggingFilter (advanced)	Whether to use logging filter	false	boolean
maxPerRouteConnections (advanced)	Maximum route connections	100	Integer

Name	Description	Default	Type
maxTotalConnections (advanced)	Maximum total connections	100	Integer
serverAddress (advanced)	Server address for docker registry.	https://index.docker.io/v1/	String
socket (advanced)	Socket connection mode	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
certPath (security)	Location containing the SSL certificate chain		String
password (security)	Password to authenticate with		String
secure (security)	Use HTTPS communication	false	boolean
tlsVerify (security)	Check TLS	false	boolean
username (security)	User name to authenticate with		String

84.3. HEADER STRATEGY

All URI option can be passed as Header properties. Values found in a message header take precedence over URI parameters. A header property takes the form of a URI option prefixed with **CamelDocker** as shown below

URI Option	Header Property
containerId	CamelDockerContainerId

84.4. EXAMPLES

The following example consumes events from Docker:

```
from("docker://events?host=192.168.59.103&port=2375").to("log:event");
```

The following example queries Docker for system wide information

```
from("docker://info?host=192.168.59.103&port=2375").to("log:info");
```

84.5. DEPENDENCIES

To use Docker in your Camel routes you need to add a dependency on **camel-docker**, which implements the component.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest and greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-docker</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 85. DOZER COMPONENT

Available as of Camel version 2.15

The **dozer:** component provides the ability to map between Java beans using the Dozer mapping framework since **Camel 2.15.0**. Camel also supports the ability to trigger Dozer mappings as a type converter. The primary differences between using a Dozer endpoint and a Dozer converter are:

- The ability to manage Dozer mapping configuration on a per-endpoint basis vs. global configuration via the converter registry.
- A Dozer endpoint can be configured to marshal/unmarshal input and output data using Camel data formats to support a single, any-to-any transformation endpoint
- The Dozer component allows for fine-grained integration and extension of Dozer to support additional functionality (e.g. mapping literal values, using expressions for mappings, etc.).

In order to use the Dozer component, Maven users will need to add the following dependency to their **pom.xml**:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-dozer</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

85.1. URI FORMAT

The Dozer component only supports producer endpoints.

```
dozer:endpointId[?options]
```

Where **endpointId** is a name used to uniquely identify the Dozer endpoint configuration.

An example Dozer endpoint URI:

```
from("direct:orderInput").
  to("dozer:transformOrder?mappingFile=orderMapping.xml&targetModel=example.XYZOrder").
  to("direct:orderOutput");
```

85.2. OPTIONS

The Dozer component has no options.

The Dozer endpoint is configured using URI syntax:

```
dozer:name
```

with the following path and query parameters:

85.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** A human readable name of the mapping.		String

85.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
mappingConfiguration (producer)	The name of a DozerBeanMapperConfiguration bean in the Camel registry which should be used for configuring the Dozer mapping. This is an alternative to the mappingFile option that can be used for fine-grained control over how Dozer is configured. Remember to use a prefix in the value to indicate that the bean is in the Camel registry (e.g. myDozerConfig).		DozerBeanMapperConfiguration
mappingFile (producer)	The location of a Dozer configuration file. The file is loaded from the classpath by default, but you can use file:, classpath:, or http: to load the configuration from a specific location.	dozerBeanMapping.xml	String
marshalId (producer)	The id of a dataFormat defined within the Camel Context to use for marshalling the mapping output to a non-Java type.		String
sourceModel (producer)	Fully-qualified class name for the source type used in the mapping. If specified, the input to the mapping is converted to the specified type before being mapped with Dozer.		String
targetModel (producer)	**Required** Fully-qualified class name for the target type used in the mapping.		String
unmarshalId (producer)	The id of a dataFormat defined within the Camel Context to use for unmarshalling the mapping input from a non-Java type.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

85.3. USING DATA FORMATS WITH DOZER

Dozer does not support non-Java sources and targets for mappings, so it cannot, for example, map an XML document to a Java object on its own. Luckily, Camel has extensive support for marshalling between Java and a wide variety of formats using data formats. The Dozer component takes advantage of this support by allowing you to specify that input and output data should be passed through a data

format prior to processing via Dozer. You can always do this on your own outside the call to Dozer, but supporting it directly in the Dozer component allows you to use a single endpoints to configure any-to-any transformation within Camel.

As an example, let's say you wanted to map between an XML data structure and a JSON data structure using the Dozer component. If you had the following data formats defined in a Camel Context:

```
<dataFormats>
  <json library="Jackson" id="myjson"/>
  <jaxb contextPath="org.example" id="myjaxb"/>
</dataFormats>
```

You could then configure a Dozer endpoint to unmarshal the input XML using a JAXB data format and marshal the mapping output using Jackson.

```
<endpoint uri="dozer:xml2json?
marshalId=myjson&amp;unmarshalId=myjaxb&amp;targetModel=org.example.Order"/>
```

85.4. CONFIGURING DOZER

All Dozer endpoints require a Dozer mapping configuration file which defines mappings between source and target objects. The component will default to a location of META-INF/dozerBeanMapping.xml if the mappingFile or mappingConfiguration options are not specified on an endpoint. If you need to supply multiple mapping configuration files for a single endpoint or specify additional configuration options (e.g. event listeners, custom converters, etc.), then you can use an instance of **org.apache.camel.converter.dozer.DozerBeanMapperConfiguration**.

```
<bean id="mapper" class="org.apache.camel.converter.dozer.DozerBeanMapperConfiguration">
  <property name="mappingFiles">
    <list>
      <value>mapping1.xml</value>
      <value>mapping2.xml</value>
    </list>
  </property>
</bean>
```

85.5. MAPPING EXTENSIONS

The Dozer component implements a number of extensions to the Dozer mapping framework as custom converters. These converters implement mapping functions that are not supported directly by Dozer itself.

85.5.1. Variable Mappings

Variable mappings allow you to map the value of a variable definition within a Dozer configuration into a target field instead of using the value of a source field. This is equivalent to constant mapping in other mapping frameworks, where can you assign a literal value to a target field. To use a variable mapping, simply define a variable within your mapping configuration and then map from the VariableMapper class into your target field of choice:

```
<mappings xmlns="http://dozermapper.github.io/schema/bean-mapping"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://dozermapper.github.io/schema/bean-mapping
```

```
http://dozermapper.github.io/schema/bean-mapping.xsd">
  <configuration>
    <variables>
      <variable name="CUST_ID">ACME-SALES</variable>
    </variables>
  </configuration>
  <mapping>
    <class-a>org.apache.camel.component.dozer.VariableMapper</class-a>
    <class-b>org.example.Order</class-b>
    <field custom-converter-id="_variableMapping" custom-converter-param="${CUST_ID}">
      <a>literal</a>
      <b>custId</b>
    </field>
  </mapping>
</mappings>
```

85.5.2. Custom Mappings

Custom mappings allow you to define your own logic for how a source field is mapped to a target field. They are similar in function to Dozer customer converters, with two notable differences:

- You can have multiple converter methods in a single class with custom mappings.
- There is no requirement to implement a Dozer-specific interface with custom mappings.

A custom mapping is declared by using the built-in '_customMapping' converter in your mapping configuration. The parameter to this converter has the following syntax:

```
[class-name][,method-name]
```

Method name is optional - the Dozer component will search for a method that matches the input and output types required for a mapping. An example custom mapping and configuration are provided below.

```
public class CustomMapper {
    // All customer ids must be wrapped in "[ ]"
    public Object mapCustomer(String customerId) {
        return "[" + customerId + "]";
    }
}
```

```
<mappings xmlns="http://dozermapper.github.io/schema/bean-mapping"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://dozermapper.github.io/schema/bean-mapping
http://dozermapper.github.io/schema/bean-mapping.xsd">
  <mapping>
    <class-a>org.example.A</class-a>
    <class-b>org.example.B</class-b>
    <field custom-converter-id="_customMapping"
        custom-converter-param="org.example.CustomMapper,mapCustomer">
      <a>header.customerNum</a>
      <b>custId</b>
```

```
    </field>
  </mapping>
</mappings>
```

85.5.3. Expression Mappings

Expression mappings allow you to use the powerful language capabilities of Camel to evaluate an expression and assign the result to a target field in a mapping. Any language that Camel supports can be used in an expression mapping. Basic examples of expressions include the ability to map a Camel message header or exchange property to a target field or to concatenate multiple source fields into a target field. The syntax of a mapping expression is:

```
[language]:[expression]
```

An example of mapping a message header into a target field:

```
<mappings xmlns="http://dozermapper.github.io/schema/bean-mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://dozermapper.github.io/schema/bean-mapping
http://dozermapper.github.io/schema/bean-mapping.xsd">
  <mapping>
    <class-a>org.apache.camel.component.dozer.ExpressionMapper</class-a>
    <class-b>org.example.B</class-b>
    <field custom-converter-id="_expressionMapping" custom-converter-
param="simple:\${header.customerNumber}">
      <a>expression</a>
      <b>custId</b>
    </field>
  </mapping>
</mappings>
```

Note that any properties within your expression must be escaped with "\" to prevent an error when Dozer attempts to resolve variable values defined using the EL.

CHAPTER 86. DRILL COMPONENT

Available as of Camel version 2.19

The **drill:** component gives you the ability to querying to Apache Drill Cluster

Drill is an Apache open-source SQL query engine for Big Data exploration. Drill is designed from the ground up to support high-performance analysis on the semi-structured and rapidly evolving data coming from modern Big Data applications, while still providing the familiarity and ecosystem of ANSI SQL, the industry-standard query language

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-drill</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

86.1. URI FORMAT

```
drill://host[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

86.2. DRILL PRODUCER

The producer execute query using **CamelDrillQuery** header and put results into body.

86.3. OPTIONS

The Drill component has no options.

The Drill endpoint is configured using URI syntax:

```
drill:host
```

with the following path and query parameters:

86.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** ZooKeeper host name or IP address. Use local instead of a host name or IP address to connect to the local Drillbit		String

86.3.2. Query Parameters (5 parameters):

Name	Description	Default	Type
clusterId (producer)	Cluster ID https://drill.apache.org/docs/using-the-jdbc-driver/determining-the-cluster-id		String
directory (producer)	Drill directory in ZooKeeper		String
mode (producer)	Connection mode: zk: Zookeeper drillbit: Drillbit direct connection https://drill.apache.org/docs/using-the-jdbc-driver/	ZK	DrillConnectionMode
port (producer)	ZooKeeper port number		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

86.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 87. DROPBOX COMPONENT

Available as of Camel version 2.14

The **dropbox:** component allows you to treat Dropbox remote folders as a producer or consumer of messages. Using the Dropbox Java Core API (reference version for this component is 1.7.x), this camel component has the following features:

- As a consumer, download files and search files by queries
- As a producer, download files, move files between remote directories, delete files/dir, upload files and search files by queries

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-dropbox</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

87.1. URI FORMAT

```
dropbox://[operation]?[options]
```

Where **operation** is the specific action (typically is a CRUD action) to perform on Dropbox remote folder.

87.2. OPERATIONS

Operation	Description
del	deletes files or directories on Dropbox
get	download files from Dropbox
move	move files from folders on Dropbox
put	upload files on Dropbox
search	search files on Dropbox based on string queries

Operations require additional options to work, some are mandatory for the specific operation.

87.3. OPTIONS

In order to work with Dropbox API you need to obtain an **accessToken** and a **clientIdentifier.** You can refer to the Dropbox documentation that explains how to get them.

The Dropbox component has no options.

The Dropbox endpoint is configured using URI syntax:

```
dropbox:operation
```

with the following path and query parameters:

87.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
operation	**Required** The specific action (typically is a CRUD action) to perform on Dropbox remote folder.		DropboxOperation

87.3.2. Query Parameters (12 parameters):

Name	Description	Default	Type
accessToken (common)	**Required** The access token to make API requests for a specific Dropbox user		String
client (common)	To use an existing DbxClient instance as DropBox client.		DbxClientV2
clientIdentifier (common)	Name of the app registered to make API requests		String
localPath (common)	Optional folder or file to upload on Dropbox from the local filesystem. If this option has not been configured then the message body is used as the content to upload.		String
newRemotePath (common)	Destination file or folder		String
query (common)	A space-separated list of sub-strings to search for. A file matches only if it contains all the sub-strings. If this option is not set, all files will be matched.		String
remotePath (common)	Original file or folder to move		String

Name	Description	Default	Type
uploadMode (common)	Which mode to upload. in case of add the new file will be renamed if a file with the same name already exists on dropbox. in case of force if a file with the same name already exists on dropbox, this will be overwritten.		DropboxUploadMode
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

87.4. DEL OPERATION

Delete files on Dropbox.

Works only as Camel producer.

Below are listed the options for this operation:

Property	Mandatory	Description
remotePath	**true**	Folder or file to delete on Dropbox

87.4.1. Samples

```
from("direct:start")
  .to("dropbox://del?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1")
  .to("mock:result");
```

```
from("direct:start")
  .to("dropbox://del?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1/file1.tar.gz")
  .to("mock:result");
```

87.4.2. Result Message Headers

The following headers are set on message result:

Property	Value
DELETED_PATH	name of the path deleted on dropbox

87.4.3. Result Message Body

The following objects are set on message body result:

Object type	Description
String	name of the path deleted on dropbox

87.5. GET (DOWNLOAD) OPERATION

Download files from Dropbox.

Works as Camel producer or Camel consumer.

Below are listed the options for this operation:

Property	Mandatory	Description
remotePath	**true**	Folder or file to download from Dropbox

87.5.1. Samples

```
from("direct:start")
  .to("dropbox://get?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1/file1.tar.gz")
  .to("file:///home/kermit/?fileName=file1.tar.gz");

from("direct:start")
  .to("dropbox://get?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1")
  .to("mock:result");

from("dropbox://get?accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1")
  .to("file:///home/kermit/");
```

87.5.2. Result Message Headers

The following headers are set on message result:

Property	Value
DOWNLOADED_FILE	in case of single file download, path of the remote file downloaded
DOWNLOADED_FILES	in case of multiple files download, path of the remote files downloaded

87.5.3. Result Message Body

The following objects are set on message body result:

Object type	Description
ByteArrayOutputStream	in case of single file download, stream representing the file downloaded
Map<String, ByteArrayOutputStream>	in case of multiple files download, a map with as key the path of the remote file downloaded and as value the stream representing the file downloaded

87.6. MOVE OPERATION

Move files on Dropbox between one folder to another.

Works only as Camel producer.

Below are listed the options for this operation:

Property	Mandatory	Description
remotePath	**true**	Original file or folder to move
newRemotePath	**true**	Destination file or folder

87.6.1. Samples

```
from("direct:start")
  .to("dropbox://move?
accessToken=XXX&clientIdentifier=XXX&remotePath=/root/folder1&newRemotePath=/root/folder2")
  .to("mock:result");
```

87.6.2. Result Message Headers

The following headers are set on message result:

Property	Value
MOVED_PATH	name of the path moved on dropbox

87.6.3. Result Message Body

The following objects are set on message body result:

Object type	Description
String	name of the path moved on dropbox

87.7. PUT (UPLOAD) OPERATION

Upload files on Dropbox.

Works as Camel producer.

Below are listed the options for this operation:

Property	Mandatory	Description
uploadMode	**true**	add or force this option specifies how a file should be saved on dropbox: in case of "add" the new file will be renamed if a file with the same name already exists on dropbox. In case of "force" if a file with the same name already exists on dropbox, this will be overwritten.
localPath	**false**	Folder or file to upload on Dropbox from the local filesystem. If this option has been configured then it takes precedence over uploading as a single file with content from the Camel message body (message body is converted into a byte array).
remotePath	**false**	Folder destination on Dropbox. If the property is not set, the component will upload the file on a remote path equal to the local path. With Windows or without an absolute localPath you may run into an exception like the following: Caused by: java.lang.IllegalArgumentException: 'path': bad path: must start with "/": "C:/My/File" OR Caused by: java.lang.IllegalArgumentException: 'path': bad path: must start with "/": "MyFile"

87.7.1. Samples

```
from("direct:start").to("dropbox://put?
accessToken=XXX&clientIdentifier=XXX&uploadMode=add&localPath=/root/folder1")
  .to("mock:result");
```

```
from("direct:start").to("dropbox://put?
accessToken=XXX&clientIdentifier=XXX&uploadMode=add&localPath=/root/folder1&remotePath=/root/f
older2")
  .to("mock:result");
```

And to upload a single file with content from the message body

```
from("direct:start")
   .setHeader(DropboxConstants.HEADER_PUT_FILE_NAME, constant("myfile.txt"))
   .to("dropbox://put?
accessToken=XXX&clientIdentifier=XXX&uploadMode=add&remotePath=/root/folder2")
   .to("mock:result");
```

The name of the file can be provided in the header **DropboxConstants.HEADER_PUT_FILE_NAME** or **Exchange.FILE_NAME** in that order of precedence. If no header has been provided then the message id (uuid) is used as the file name.

87.7.2. Result Message Headers

The following headers are set on message result:

Property	Value
UPLOADED_FILE	in case of single file upload, path of the remote path uploaded
UPLOADED_FILES	in case of multiple files upload, string with the remote paths uploaded

87.7.3. Result Message Body

The following objects are set on message body result:

Object type	Description
String	in case of single file upload, result of the upload operation, OK or KO
Map<String, DropboxResultCode>	in case of multiple files upload, a map with as key the path of the remote file uploaded and as value the result of the upload operation, OK or KO

87.8. SEARCH OPERATION

Search inside a remote Dropbox folder including its sub directories.

Works as Camel producer and as Camel consumer.

Below are listed the options for this operation:

Property	Mandatory	Description
remotePath	**true**	Folder on Dropbox where to search in.
query	**true**	A space-separated list of sub-strings to search for. A file matches only if it contains all the sub-strings. If this option is not set, all files will be matched. The query is required to be provided in either the endpoint configuration or as a header **CamelDropboxQuery** on the Camel message.

87.8.1. Samples

```
from("dropbox://search?accessToken=XXX&clientIdentifier=XXX&remotePath=/XXX&query=XXX")
  .to("mock:result");

from("direct:start")
  .setHeader("CamelDropboxQuery", constant("XXX"))
  .to("dropbox://search?accessToken=XXX&clientIdentifier=XXX&remotePath=/XXX")
  .to("mock:result");
```

87.8.2. Result Message Headers

The following headers are set on message result:

Property	Value
FOUNDED_FILES	list of file path founded

87.8.3. Result Message Body

The following objects are set on message body result:

Object type	Description
List<DbxEntry>	list of file path founded. For more information on this object refer to Dropbox documentation,

CHAPTER 88. EHCACHE COMPONENT

Available as of Camel version 2.18

The **ehcache** component enables you to perform caching operations using Ehcache 3 as the Cache Implementation.

This component supports producer and event based consumer endpoints.

The Cache consumer is an event based consumer and can be used to listen and respond to specific cache activities.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ehcache</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

88.1. URI FORMAT

```
ehcache://cacheName[?options]
```

You can append query options to the URI in the following format, **?option=value&option=#beanRef&…**

88.2. OPTIONS

The Ehcache component supports 7 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	Sets the global component configuration		EhcacheConfiguration
cacheManager (common)	The cache manager		CacheManager
cacheManagerConfiguration (common)	The cache manager configuration		Configuration
cacheConfiguration (common)	The default cache configuration to be used to create caches.		CacheConfiguration<?,?>
cachesConfigurations (common)	A map of caches configurations to be used to create caches.		Map

Name	Description	Default	Type
cacheConfigurationUri (common)	URI pointing to the Ehcache XML configuration file's location		String
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ehcache endpoint is configured using URI syntax:

```
ehcache:cacheName
```

with the following path and query parameters:

88.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** the cache name		String

88.2.2. Query Parameters (17 parameters):

Name	Description	Default	Type
cacheManager (common)	The cache manager		CacheManager
cacheManagerConfiguration (common)	The cache manager configuration		Configuration
configurationUri (common)	URI pointing to the Ehcache XML configuration file's location		String
createCacheIfNotExist (common)	Configure if a cache need to be created if it does exist or can't be pre-configured.	true	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
eventFiring (consumer)	Set the the delivery mode (synchronous, asynchronous)	ASYNCHRONOUS	EventFiring
eventOrdering (consumer)	Set the the delivery mode (ordered, unordered)	ORDERED	EventOrdering
eventTypes (consumer)	Set the type of events to listen for	EVICTED,EXPIRED,REMOVED,CREATED,UPDATED	Set
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
action (producer)	To configure the default cache action. If an action is set in the message header, then the operation from the header takes precedence.		String
key (producer)	To configure the default action key. If a key is set in the message header, then the key from the header takes precedence.		Object
configuration (advanced)	The default cache configuration to be used to create caches.		CacheConfiguration<?,?>

Name	Description	Default	Type
configurations (advanced)	A map of cache configuration to be used to create caches.		Map
keyType (advanced)	The cache key type, default java.lang.Object	java.lang.Object	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
valueType (advanced)	The cache value type, default java.lang.Object	java.lang.Object	String

88.2.3. Message Headers Camel

Header	Type	Description
CamelEhcacheAction	**String**	The operation to be perfomed on the cache, valid options are: * CLEAR * PUT * PUT_ALL * PUT_IF_ABSENT * GET * GET_ALL * REMOVE * REMOVE_ALL * REPLACE
CamelEhcacheActionHasResult	Boolean	Set to true if the action has a result
CamelEhcacheActionSucceeded	**Boolean**	Set to true if the actionsuccedded
CamelEhcacheKey	Object	The cache key used for an action
CamelEhcacheKeys	Set<Object>	A list of keys, used in * PUT_ALL * GET_ALL * REMOVE_ALL

Header	Type	Description
CamelEhcacheValue	Object	The value to put in the cache or the result of an operation
CamelEhcacheOldValue	Object	The old value associated to a key for actions like PUT_IF_ABSENT or the Object used for comparison for actions like REPLACE
CamelEhcacheEventType	EventType	The type of event received

88.3. EHCACHE BASED IDEMPOTENT REPOSITORY EXAMPLE:

```
CacheManager manager = CacheManagerBuilder.newCacheManager(new
XmlConfiguration("ehcache.xml"));
EhcacheIdempotentRepository repo = new EhcacheIdempotentRepository(manager, "idempotent-
cache");

from("direct:in")
    .idempotentConsumer(header("messageId"), idempotentRepo)
    .to("mock:out");
```

88.4. EHCACHE BASED AGGREGATION REPOSITORY EXAMPLE:

```
public class EhcacheAggregationRepositoryRoutesTest extends CamelTestSupport {
    private static final String ENDPOINT_MOCK = "mock:result";
    private static final String ENDPOINT_DIRECT = "direct:one";
    private static final int[] VALUES = generateRandomArrayOfInt(10, 0, 30);
    private static final int SUM = IntStream.of(VALUES).reduce(0, (a, b) -> a + b);
    private static final String CORRELATOR = "CORRELATOR";

    @EndpointInject(uri = ENDPOINT_MOCK)
    private MockEndpoint mock;

    @Produce(uri = ENDPOINT_DIRECT)
    private ProducerTemplate producer;

    @Test
    public void checkAggregationFromOneRoute() throws Exception {
        mock.expectedMessageCount(VALUES.length);
```

```
        mock.expectedBodiesReceived(SUM);

        IntStream.of(VALUES).forEach(
            i -> producer.sendBodyAndHeader(i, CORRELATOR, CORRELATOR)
        );

        mock.assertIsSatisfied();
    }

    private Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        if (oldExchange == null) {
            return newExchange;
        } else {
            Integer n = newExchange.getIn().getBody(Integer.class);
            Integer o = oldExchange.getIn().getBody(Integer.class);
            Integer v = (o == null ? 0 : o) + (n == null ? 0 : n);

            oldExchange.getIn().setBody(v, Integer.class);

            return oldExchange;
        }
    }

    @Override
    protected RoutesBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                from(ENDPOINT_DIRECT)
                    .routeId("AggregatingRouteOne")
                    .aggregate(header(CORRELATOR))
                    .aggregationRepository(createAggregateRepository())
                    .aggregationStrategy(EhcacheAggregationRepositoryRoutesTest.this::aggregate)
                    .completionSize(VALUES.length)

.to("log:org.apache.camel.component.ehcache.processor.aggregate.level=INFO&showAll=true&mulltilin
e=true")
                        .to(ENDPOINT_MOCK);
            }
        };
    }

    protected EhcacheAggregationRepository createAggregateRepository() throws Exception {
        CacheManager cacheManager = CacheManagerBuilder.newCacheManager(new
XmlConfiguration("ehcache.xml"));
        cacheManager.init();

        EhcacheAggregationRepository repository = new EhcacheAggregationRepository();
        repository.setCacheManager(cacheManager);
        repository.setCacheName("aggregate");

        return repository;
    }
}
```

CHAPTER 89. EJB COMPONENT

Available as of Camel version 2.4

The **ejb:** component binds EJBs to Camel message exchanges.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ejb</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

89.1. URI FORMAT

```
ejb:ejbName[?options]
```

Where **ejbName** can be any string which is used to look up the EJB in the Application Server JNDI Registry

89.2. OPTIONS

The EJB component supports 3 options which are listed below.

Name	Description	Default	Type
context (producer)	The Context to use for looking up the EJBs		Context
properties (producer)	Properties for creating javax.naming.Context if a context has not been configured.		Properties
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The EJB endpoint is configured using URI syntax:

```
ejb:beanName
```

with the following path and query parameters:

89.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
beanName	**Required** Sets the name of the bean to invoke		String

89.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
method (producer)	Sets the name of the method to invoke on the bean		String
cache (advanced)	If enabled, Camel will cache the result of the first Registry look-up. Cache can be enabled if the bean in the Registry is defined as a singleton scope.	false	boolean
multiParameterArray (advanced)	**Deprecated** How to treat the parameters which are passed from the message body.true means the message body should be an array of parameters.. Deprecation note: This option is used internally by Camel, and is not intended for end users to use. Deprecation note: This option is used internally by Camel, and is not intended for end users to use.	false	boolean
parameters (advanced)	Used for configuring additional properties on the bean		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

89.3. BEAN BINDING

How bean methods to be invoked are chosen (if they are not specified explicitly through the **method** parameter) and how parameter values are constructed from the Message are all defined by the Bean Binding mechanism which is used throughout all of the various Bean Integration mechanisms in Camel.

89.4. EXAMPLES

In the following examples we use the Greater EJB which is defined as follows:

GreaterLocal.java

```
public interface GreaterLocal {

    String hello(String name);
```

```
    String bye(String name);

}
```

And the implementation

GreaterImpl.java

```
@Stateless
public class GreaterImpl implements GreaterLocal {

    public String hello(String name) {
        return "Hello " + name;
    }

    public String bye(String name) {
        return "Bye " + name;
    }

}
```

89.4.1. Using Java DSL

In this example we want to invoke the **hello** method on the EJB. Since this example is based on an unit test using Apache OpenEJB we have to set a **JndiContext** on the EJB component with the OpenEJB settings.

```
@Override
protected CamelContext createCamelContext() throws Exception {
    CamelContext answer = new DefaultCamelContext();

    // enlist EJB component using the JndiContext
    EjbComponent ejb = answer.getComponent("ejb", EjbComponent.class);
    ejb.setContext(createEjbContext());

    return answer;
}

private static Context createEjbContext() throws NamingException {
    // here we need to define our context factory to use OpenEJB for our testing
    Properties properties = new Properties();
    properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.openejb.client.LocalInitialContextFactory");

    return new InitialContext(properties);
}
```

Then we are ready to use the EJB in the Camel route:

```
from("direct:start")
    // invoke the greeter EJB using the local interface and invoke the hello method
    .to("ejb:GreaterImplLocal?method=hello")
    .to("mock:result");
```

In a real application server

In a real application server you most likely do not have to setup a **JndiContext** on the EJB component as it will create a default **JndiContext** on the same JVM as the application server, which usually allows it to access the JNDI registry and lookup the EJBs. However if you need to access a application server on a remote JVM or the likes, you have to prepare the properties beforehand.

89.4.2. Using Spring XML

And this is the same example using Spring XML instead:

Again since this is based on an unit test we need to setup the EJB component:

```
<!-- setup Camel EJB component -->
<bean id="ejb" class="org.apache.camel.component.ejb.EjbComponent">
    <property name="properties" ref="jndiProperties"/>
</bean>

<!-- use OpenEJB context factory -->
<p:properties id="jndiProperties">
    <prop
key="java.naming.factory.initial">org.apache.openejb.client.LocalInitialContextFactory</prop>
</p:properties>
```

Before we are ready to use EJB in the Camel routes:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <to uri="ejb:GreaterImplLocal?method=hello"/>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

89.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Bean
- Bean Binding
- Bean Integration

CHAPTER 90. ELASTICSEARCH COMPONENT (DEPRECATED)

Available as of Camel version 2.11

The ElasticSearch component allows you to interface with an ElasticSearch server.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-elasticsearch</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

90.1. URI FORMAT

```
elasticsearch://clusterName[?options]
```

90.2. ENDPOINT OPTIONS

The Elasticsearch component supports 2 options which are listed below.

Name	Description	Default	Type
client (advanced)	To use an existing configured Elasticsearch client, instead of creating a client per endpoint.		Client
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Elasticsearch endpoint is configured using URI syntax:

```
elasticsearch:clusterName
```

with the following path and query parameters:

90.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
clusterName	**Required** Name of cluster or use local for local mode		String

90.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
clientTransportSniff (producer)	Is the client allowed to sniff the rest of the cluster or not (default true). This setting map to the client.transport.sniff setting.	true	Boolean
consistencyLevel (producer)	The write consistency level to use with INDEX and BULK operations (can be any of ONE, QUORUM, ALL or DEFAULT)	DEFAULT	WriteConsistencyLevel
data (producer)	Is the node going to be allowed to allocate data (shards) to it or not. This setting map to the node.data setting.		Boolean
indexName (producer)	The name of the index to act against		String
indexType (producer)	The type of the index to act against		String
ip (producer)	The TransportClient remote host ip to use		String
operation (producer)	What operation to perform		String
pathHome (producer)	The path.home property of ElasticSearch configuration. You need to provide a valid path, otherwise the default, $user.home/.elasticsearch, will be used.	${user.home}/.elasticsearch	String
port (producer)	The TransportClient remote port to use (defaults to 9300)	9300	int
transportAddresses (producer)	Comma separated list with ip:port formatted remote transport addresses to use. The ip and port options must be left blank for transportAddresses to be considered instead.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

90.3. LOCAL TESTING

If you want to run against a local (in JVM/classloader) ElasticSearch server, just set the clusterName value in the URI to "local". See the client guide for more details.

90.4. MESSAGE OPERATIONS

The following ElasticSearch operations are currently supported. Simply set an endpoint URI option or exchange header with a key of "operation" and a value set to one of the following. Some operations also require other parameters or the message body to be set.

operation	message body	description
INDEX	Map, String, byte[] or XContentBuilder content to index	adds content to an index and returns the content's indexId in the body. **Camel 2.15,** you can set the indexId by setting the message header with the key "indexId".
GET_BY_ID	index id of content to retrieve	retrieves the specified index and returns a GetResult object in the body
DELETE	index id of content to delete	deletes the specified indexId and returns a DeleteResult object in the body
BULK_INDEX	a **List** or **Collection** of any type that is already accepted (XContentBuilder, Map, byte[], String)	*Camel 2.14,*adds content to an index and return a List of the id of the successfully indexed documents in the body

operation	message body	description
BULK	a **List** or **Collection** of any type that is already accepted (XContentBuilder, Map, byte[], String)	**Camel 2.15:** Adds content to an index and returns the BulkResponse object in the body
SEARCH	Map or SearchRequest Object	**Camel 2.15:** search the content with the map of query string
MULTIGET	List of MultigetRequest.Item object	**Camel 2.17:** retrieves the specified indexes, types etc. in MultigetRequest and returns a MultigetResponse object in the body
MULTISEARCH	List of SearchRequest object	**Camel 2.17:** search for parameters specified in MultiSearchRequest and returns a MultiSearchResponse object in the body
EXISTS	Index name as header	**Camel 2.17:** Returns a Boolean object in the body

operation	message body	description
UPDATE	Map, String, byte[] or XContentBuilder content to update	**Camel 2.17:** Updates content to an index and returns the content's indexId in the body.

90.5. INDEX EXAMPLE

Below is a simple INDEX example

```
from("direct:index")
.to("elasticsearch://local?operation=INDEX&indexName=twitter&indexType=tweet");
```

```
<route>
   <from uri="direct:index" />
   <to uri="elasticsearch://local?operation=INDEX&indexName=twitter&indexType=tweet"/>
</route>
```

A client would simply need to pass a body message containing a Map to the route. The result body contains the indexId created.

```
Map<String, String> map = new HashMap<String, String>();
map.put("content", "test");
String indexId = template.requestBody("direct:index", map, String.class);
```

90.6. FOR MORE INFORMATION, SEE THESE RESOURCES

ElasticSearch Main Site

ElasticSearch Java API

90.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 91. ELASTICSEARCH5 COMPONENT (DEPRECATED)

Available as of Camel version 2.19

The ElasticSearch component allows you to interface with an ElasticSearch 5.x API.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-elasticsearch5</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

91.1. URI FORMAT

```
elasticsearch5://clusterName[?options]
```

91.2. ENDPOINT OPTIONS

The Elasticsearch5 component supports 2 options which are listed below.

Name	Description	Default	Type
client (advanced)	To use an existing configured Elasticsearch client, instead of creating a client per endpoint. This allow to customize the client with specific settings.		TransportClient
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Elasticsearch5 endpoint is configured using URI syntax:

```
elasticsearch5:clusterName
```

with the following path and query parameters:

91.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
clusterName	**Required** Name of the cluster		String

91.2.2. Query Parameters (16 parameters):

Name	Description	Default	Type
clientTransportSniff (producer)	Is the client allowed to sniff the rest of the cluster or not. This setting map to the client.transport.sniff setting.	false	boolean
indexName (producer)	The name of the index to act against		String
indexType (producer)	The type of the index to act against		String
ip (producer)	The TransportClient remote host ip to use		String
operation (producer)	What operation to perform		ElasticsearchOperation
pingSchedule (producer)	The time(in unit) the client ping the cluster.	5s	String
pingTimeout (producer)	The time(in unit) to wait for a ping response from a node too return.	5s	String
port (producer)	The TransportClient remote port to use (defaults to 9300)	9300	int
tcpCompress (producer)	true if compression (LZF) enable between all nodes.	false	boolean
tcpConnectTimeout (producer)	The time(in unit) to wait for connection timeout.	30s	String
transportAddresses (producer)	Comma separated list with ip:port formatted remote transport addresses to use. The ip and port options must be left blank for transportAddresses to be considered instead.		String
waitForActiveShards (producer)	Index creation waits for the write consistency number of shards to be available	1	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
enableSSL (security)	Enable SSL. Require XPack client jar on the classpath	false	boolean

Name	Description	Default	Type
password (authentication)	Password for authenticate against the cluster. Require XPack client jar on the classpath		String
user (authentication)	User for authenticate against the cluster. Requires transport_client role for accessing the cluster. Require XPack client jar on the classpath		String

91.3. MESSAGE OPERATIONS

The following ElasticSearch operations are currently supported. Simply set an endpoint URI option or exchange header with a key of "operation" and a value set to one of the following. Some operations also require other parameters or the message body to be set.

operation	message body	description
INDEX	Map, String, byte[] or XContentBuilder content to index	Adds content to an index and returns the content's indexId in the body. You can set the indexId by setting the message header with the key "indexId".
GET_BY_ID	index id of content to retrieve	Retrieves the specified index and returns a GetResult object in the body
DELETE	index name and type of content to delete	Deletes the specified indexName and indexType and returns a DeleteResponse object in the body

operation	message body	description
DELETE_INDEX	index name of content to delete	Deletes the specified indexName and returns a DeleteIndexResponse object in the body
BULK_INDEX	a **List** or **Collection** of any type that is already accepted (XContentBuilder, Map, byte[], String)	Adds content to an index and return a List of the id of the successfully indexed documents in the body
BULK	a **List** or **Collection** of any type that is already accepted (XContentBuilder, Map, byte[], String)	Adds content to an index and returns the BulkResponse object in the body

operation	message body	description
SEARCH	Map, String or SearchRequest Object	Search the content with the map of query string
MULTIGET	List of MultigetRequest.Item object	Retrieves the specified indexes, types etc. in MultigetRequest and returns a MultigetResponse object in the body
MULTISEARCH	List of SearchRequest object	Search for parameters specified in MultiSearchRequest and returns a MultiSearchResponse object in the body
EXISTS	Index name as header	Checks the index exists or not and returns a Boolean flag in the body
UPDATE	Map, String, byte[] or XContentBuilder content to update	Updates content to an index and returns the content's indexId in the body.

91.4. INDEX EXAMPLE

Below is a simple INDEX example

```
from("direct:index")
.to("elasticsearch5://elasticsearch?operation=INDEX&indexName=twitter&indexType=tweet");
```

```
<route>
   <from uri="direct:index" />
   <to uri="elasticsearch5://elasticsearch?operation=INDEX&indexName=twitter&indexType=tweet"/>
</route>
```

A client would simply need to pass a body message containing a Map to the route. The result body contains the indexId created.

```
Map<String, String> map = new HashMap<String, String>();
map.put("content", "test");
String indexId = template.requestBody("direct:index", map, String.class);
```

91.5. FOR MORE INFORMATION, SEE THESE RESOURCES

Elastic Main Site

ElasticSearch Java API

91.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 92. ELASTICHSEARCH REST COMPONENT

Available as of Camel version 2.21

The ElasticSearch component allows you to interface with an ElasticSearch 6.x API using the REST Client library.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-elasticsearch-rest</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

92.1. URI FORMAT

```
elasticsearch-rest://clusterName[?options]
```

92.2. ENDPOINT OPTIONS

The Elastichsearch Rest component supports 12 options which are listed below.

Name	Description	Default	Type
client (advanced)	To use an existing configured Elasticsearch client, instead of creating a client per endpoint. This allow to customize the client with specific settings.		RestClient
hostAddresses (advanced)	Comma separated list with ip:port formatted remote transport addresses to use. The ip and port options must be left blank for hostAddresses to be considered instead.		String
socketTimeout (advanced)	The timeout in ms to wait before the socket will timeout.	30000	int
connectionTimeout (advanced)	The time in ms to wait before connection will timeout.	30000	int
user (advance)	Basic authenticate user		String
password (producer)	Password for authenticate		String
enableSSL (advanced)	Enable SSL	false	Boolean

Name	Description	Default	Type
maxRetryTimeout (advanced)	The time in ms before retry	30000	int
enableSniffer (advanced)	Enable automatically discover nodes from a running Elasticsearch cluster	false	Boolean
snifferInterval (advanced)	The interval between consecutive ordinary sniff executions in milliseconds. Will be honoured when sniffOnFailure is disabled or when there are no failures between consecutive sniff executions	300000	int
sniffAfterFailureDelay (advanced)	The delay of a sniff execution scheduled after a failure (in milliseconds)	60000	int
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Elastichsearch Rest endpoint is configured using URI syntax:

```
elasticsearch-rest:clusterName
```

with the following path and query parameters:

92.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
clusterName	**Required** Name of the cluster		String

92.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
connectionTimeout (producer)	The time in ms to wait before connection will timeout.	30000	int
disconnect (producer)	Disconnect after it finish calling the producer	false	boolean

Name	Description	Default	Type
enableSSL (producer)	Enable SSL	false	boolean
hostAddresses (producer)	**Required** Comma separated list with ip:port formatted remote transport addresses to use. The ip and port options must be left blank for hostAddresses to be considered instead.		String
indexName (producer)	The name of the index to act against		String
indexType (producer)	The type of the index to act against		String
maxRetryTimeout (producer)	The time in ms before retry	30000	int
operation (producer)	What operation to perform		ElasticsearchOperation
socketTimeout (producer)	The timeout in ms to wait before the socket will timeout.	30000	int
waitForActiveShards (producer)	Index creation waits for the write consistency number of shards to be available	1	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

92.3. MESSAGE OPERATIONS

The following ElasticSearch operations are currently supported. Simply set an endpoint URI option or exchange header with a key of "operation" and a value set to one of the following. Some operations also require other parameters or the message body to be set.

operation	message body	description

operation	message body	description
Index	**Map**, **String**, **byte[]**, **XContentBuilder** or **IndexRequest** content to index	Adds content to an index and returns the content's indexId in the body. You can set the indexId by setting the message header with the key "indexId".
GetById	**String** or **GetRequest** index id of content to retrieve	Retrieves the specified index and returns a GetResult object in the body
Delete	**String** or **DeleteRequest** index name and type of content to delete	Deletes the specified indexName and indexType and returns a DeleteResponse object in the body
DeleteIndex	**String** or **DeleteRequest** index name of the index to delete	Deletes the specified indexName and returns a status code the body

operation	message body	description
BulkIndex	a **List**, **BulkRequest**, or **Collection** of any type that is already accepted (XContentBuilder, Map, byte[], String)	Adds content to an index and return a List of the id of the successfully indexed documents in the body
Bulk	a **List**, **BulkRequest**, or **Collection** of any type that is already accepted (XContentBuilder, Map, byte[], String)	Adds content to an index and returns the BulkItemResponse[] object in the body
Search	**Map**, **String** or **SearchRequest**	Search the content with the map of query string

operation	message body	description
Exists	Index name(indexName) as header	Checks the index exists or not and returns a Boolean flag in the body
Update	**Map**, **UpdateRequest**, **String**, **byte[]** or **XContentBuilder** content to update	Updates content to an index and returns the content's indexId in the body.
Ping	None	Pings the remote Elasticsearch cluster and returns true if the ping succeeded, false otherwise

92.4. CONFIGURE THE COMPONENT AND ENABLE BASIC AUTHENTICATION

To use the Elasticsearch component is has to be configured with a minimum configuration.

```
ElasticsearchComponent elasticsearchComponent = new ElasticsearchComponent();
elasticsearchComponent.setHostAddresses("myelkhost:9200");
camelContext.addComponent("elasticsearch-rest", elasticsearchComponent);
```

For basic authentication with elasticsearch or using reverse http proxy in front of the elasticsearch cluster, simply setup basic authentication and SSL on the component like the example below

```
ElasticsearchComponent elasticsearchComponent = new ElasticsearchComponent();
elasticsearchComponent.setHostAddresses("myelkhost:9200");
elasticsearchComponent.setUser("elkuser");
elasticsearchComponent.setPassword("secure!!");
elasticsearchComponent.setEnableSSL(true);

camelContext.addComponent("elasticsearch-rest", elasticsearchComponent);
```

92.5. INDEX EXAMPLE

Below is a simple INDEX example

```
from("direct:index")
  .to("elasticsearch-rest://elasticsearch?operation=Index&indexName=twitter&indexType=tweet");
```

```
<route>
    <from uri="direct:index" />
    <to uri="elasticsearch-rest://elasticsearch?
operation=Index&indexName=twitter&indexType=tweet"/>
</route>
```

A client would simply need to pass a body message containing a Map to the route. The result body contains the indexId created.

```
Map<String, String> map = new HashMap<String, String>();
map.put("content", "test");
String indexId = template.requestBody("direct:index", map, String.class);
```

92.6. SEARCH EXAMPLE

Searching on specific field(s) and value use the Operation ´Search´. Pass in the query JSON String or the Map

```
from("direct:search")
  .to("elasticsearch-rest://elasticsearch?operation=Search&indexName=twitter&indexType=tweet");
```

```
<route>
    <from uri="direct:search" />
    <to uri="eelasticsearch-rest://elasticsearch?
operation=Search&indexName=twitter&indexType=tweet"/>
</route>
```

```
String query = "{\"query\":{\"match\":{\"content\":\"new release of ApacheCamel\"}}}";
SearchHits response = template.requestBody("direct:search", query, SearchHits.class);
```

Search on specific field(s) using Map.

```
Map<String, Object> actualQuery = new HashMap<>();
actualQuery.put("content", "new release of ApacheCamel");

Map<String, Object> match = new HashMap<>();
match.put("match", actualQuery);

Map<String, Object> query = new HashMap<>();
query.put("query", match);
SearchHits response = template.requestBody("direct:search", query, SearchHits.class);
```

CHAPTER 93. ELSQL COMPONENT

Available as of Camel version 2.16

The **elsql:** component is an extension to the existing SQL Component that uses ElSql to define the SQL queries.

This component uses **spring-jdbc** behind the scenes for the actual SQL handling.

This component can be used as a Transactional Client.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-elsql</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

The SQL component uses the following endpoint URI notation:

```
sql:elSqlName:resourceUri[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

The parameters to the SQL queries are named parameters in the elsql mapping files, and maps to corresponding keys from the Camel message, in the given precedence:

1. **Camel 2.16.1:** from message body if Simple expression.
2. from message body if its a `java.util.Map`3. from message headers

If a named parameter cannot be resolved, then an exception is thrown.

93.1. OPTIONS

The ElSQL component supports 5 options which are listed below.

Name	Description	Default	Type
databaseVendor (common)	To use a vendor specific com.opengamma.elsql.ElSqlConfig		ElSqlDatabaseVendor
dataSource (common)	Sets the DataSource to use to communicate with the database.		DataSource
elSqlConfig (advanced)	To use a specific configured ElSqlConfig. It may be better to use the databaseVendor option instead.		ElSqlConfig

Name	Description	Default	Type
resourceUri (common)	The resource file which contains the elsql SQL statements to use. You can specify multiple resources separated by comma. The resources are loaded on the classpath by default, you can prefix with file: to load from file system. Notice you can set this option on the component and then you do not have to configure this on the endpoint.		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The ElSQL endpoint is configured using URI syntax:

```
elsql:elsqlName:resourceUri
```

with the following path and query parameters:

93.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
elsqlName	**Required** The name of the elsql to use (is NAMED in the elsql file)		String
resourceUri	The resource file which contains the elsql SQL statements to use. You can specify multiple resources separated by comma. The resources are loaded on the classpath by default, you can prefix with file: to load from file system. Notice you can set this option on the component and then you do not have to configure this on the endpoint.		String

93.1.2. Query Parameters (47 parameters):

Name	Description	Default	Type
allowNamedParameters (common)	Whether to allow using named parameters in the queries.	true	boolean

Name	Description	Default	Type
databaseVendor (common)	To use a vendor specific com.opengamma.elsql.ElSqlConfig		ElSqlDatabaseVendor
dataSource (common)	Sets the DataSource to use to communicate with the database.		DataSource
dataSourceRef (common)	**Deprecated** Sets the reference to a DataSource to lookup from the registry, to use for communicating with the database.		String
outputClass (common)	Specify the full package and class name to use as conversion when outputType=SelectOne.		String
outputHeader (common)	Store the query result in a header instead of the message body. By default, outputHeader == null and the query result is stored in the message body, any existing content in the message body is discarded. If outputHeader is set, the value is used as the name of the header to store the query result and the original message body is preserved.		String
outputType (common)	Make the output of consumer or producer to SelectList as List of Map, or SelectOne as single Java object in the following way:a) If the query has only single column, then that JDBC Column object is returned. (such as SELECT COUNT( ) FROM PROJECT will return a Long object.b) If the query has more than one column, then it will return a Map of that result.c) If the outputClass is set, then it will convert the query result into an Java bean object by calling all the setters that match the column names.It will assume your class has a default constructor to create instance with.d) If the query resulted in more than one rows, it throws an non-unique result exception.StreamList streams the result of the query using an Iterator. This can be used with the Splitter EIP in streaming mode to process the ResultSet in streaming fashion.	SelectList	SqlOutputType
separator (common)	The separator to use when parameter values is taken from message body (if the body is a String type), to be inserted at placeholders.Notice if you use named parameters, then a Map type is used instead. The default value is comma	,	char
breakBatchOnConsumeFail (consumer)	Sets whether to break batch if onConsume failed.	false	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
expectedUpdateCount (consumer)	Sets an expected update count to validate when using onConsume.	-1	int
maxMessagesPerPoll (consumer)	Sets the maximum number of messages to poll		int
onConsume (consumer)	After processing each row then this query can be executed, if the Exchange was processed successfully, for example to mark the row as processed. The query can have parameter.		String
onConsumeBatchComplete (consumer)	After processing the entire batch, this query can be executed to bulk update rows etc. The query cannot have parameters.		String
onConsumeFailed (consumer)	After processing each row then this query can be executed, if the Exchange failed, for example to mark the row as failed. The query can have parameter.		String
routeEmptyResultSet (consumer)	Sets whether empty resultset should be allowed to be sent to the next hop. Defaults to false. So the empty resultset will be filtered out.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
transacted (consumer)	Enables or disables transaction. If enabled then if processing an exchange failed then the consumerbreak out processing any further exchanges to cause a rollback eager.	false	boolean
useIterator (consumer)	Sets how resultset should be delivered to route. Indicates delivery as either a list or individual object. defaults to true.	true	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
processingStrategy (consumer)	Allows to plugin to use a custom org.apache.camel.component.sql.SqlProcessingStrategy to execute queries when the consumer has processed the rows/batch.		SqlProcessingStrategy
batch (producer)	Enables or disables batch mode	false	boolean
noop (producer)	If set, will ignore the results of the SQL query and use the existing IN message as the OUT message for the continuation of processing	false	boolean
useMessageBodyForSql (producer)	Whether to use the message body as the SQL and then headers for parameters. If this option is enabled then the SQL in the uri is not used.	false	boolean
alwaysPopulateStatement (producer)	If enabled then the populateStatement method from org.apache.camel.component.sql.SqlPrepareStatementStrategy is always invoked, also if there is no expected parameters to be prepared. When this is false then the populateStatement is only invoked if there is 1 or more expected parameters to be set; for example this avoids reading the message body/headers for SQL queries with no parameters.	false	boolean
parametersCount (producer)	If set greater than zero, then Camel will use this count value of parameters to replace instead of querying via JDBC metadata API. This is useful if the JDBC vendor could not return correct parameters count, then user may override instead.		int
elSqlConfig (advanced)	To use a specific configured ElSqlConfig. It may be better to use the databaseVendor option instead.		ElSqlConfig

Name	Description	Default	Type
placeholder (advanced)	Specifies a character that will be replaced to in SQL query. Notice, that it is simple String.replaceAll() operation and no SQL parsing is involved (quoted strings will also change).	#	String
prepareStatementStrategy (advanced)	Allows to plugin to use a custom org.apache.camel.component.sql.SqlPrepareStatementStrategy to control preparation of the query and prepared statement.		SqlPrepareStatement Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
templateOptions (advanced)	Configures the Spring JdbcTemplate with the key/values from the Map		Map
usePlaceholder (advanced)	Sets whether to use placeholder and replace all placeholder characters with sign in the SQL queries.	true	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long

Name	Description	Default	Type
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

93.2. RESULT OF THE QUERY

For **select** operations, the result is an instance of **List<Map<String, Object>>** type, as returned by the JdbcTemplate.queryForList() method. For **update** operations, the result is the number of updated rows, returned as an **Integer**.

By default, the result is placed in the message body. If the outputHeader parameter is set, the result is placed in the header. This is an alternative to using a full message enrichment pattern to add headers, it provides a concise syntax for querying a sequence or some other small value into a header. It is convenient to use outputHeader and outputType together:

93.3. HEADER VALUES

When performing **update** operations, the SQL Component stores the update count in the following message headers:

Header	Description
CamelSqlUpdateCount	The number of rows updated for **update** operations, returned as an **Integer** object.
CamelSqlRowCount	The number of rows returned for **select** operations, returned as an **Integer** object.

93.3.1. Sample

In the given route below, we want to get all the projects from the projects table. Notice the SQL query has 2 named parameters, :#lic and :#min.

Camel will then lookup for these parameters from the message body or message headers. Notice in the example above we set two headers with constant value
for the named parameters:

```
from("direct:projects")
  .setHeader("lic", constant("ASF"))
  .setHeader("min", constant(123))
  .to("elsql:projects:com/foo/orders.elsql")
```

And the elsql mapping file

```
@NAME(projects)
  SELECT *
  FROM projects
  WHERE license = :lic AND id > :min
  ORDER BY id
```

Though if the message body is a **java.util.Map** then the named parameters will be taken from the body.

```
from("direct:projects")
  .to("elsql:projects:com/foo/orders.elsql")
```

In from Camel 2.16.1 onwards you can use Simple expressions as well, which allows to use an OGNL like notation on the message body, where it assumes to have **getLicense** and **getMinimum** methods:

```
@NAME(projects)
  SELECT *
  FROM projects
  WHERE license = :${body.license} AND id > :${body.minimum}
  ORDER BY id
```

93.4. SEE ALSO

- Configuring Camel

- Component
- Endpoint
- Getting Started
- SQL Component
- MyBatis
- JDBC

CHAPTER 94. ETCD COMPONENT

Available as of Camel version 2.18

The camel etcd component allows you to work with Etcd, a distributed reliable key-value store.

94.1. URI FORMAT

```
etcd:namespace/path[?options]
```

94.2. URI OPTIONS

The etcd component supports 7 options which are listed below.

Name	Description	Default	Type
uris (common)	To set the URIs the client connects.		String
sslContextParameters (common)	To configure security using SSLContextParameters.		SSLContextParameters
userName (common)	The user name to use for basic authentication.		String
password (common)	The password to use for basic authentication.		String
configuration (advanced)	Sets the common configuration shared among endpoints		EtcdConfiguration
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The etcd endpoint is configured using URI syntax:

```
etcd:namespace/path
```

with the following path and query parameters:

94.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
namespace	**Required** The API namespace to use		EtcdNamespace
path	The path the endpoint refers to		String

94.2.2. Query Parameters (29 parameters):

Name	Description	Default	Type
recursive (common)	To apply an action recursively.	false	boolean
servicePath (common)	The path to look for for service discovery	/services/	String
timeout (common)	To set the maximum time an action could take to complete.		Long
uris (common)	To set the URIs the client connects.	http://localhost:2379,http://localhost:4001	String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyExchangeOnTimeout (consumer)	To send an empty message in case of timeout watching for a key.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
fromIndex (consumer)	The index to watch from	0	Long
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
timeToLive (producer)	To set the lifespan of a key in milliseconds.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long

Name	Description	Default	Type
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
password (security)	The password to use for basic authentication.		String
sslContextParameters (security)	To configure security using SSLContextParameters.		SSLContextParameters
userName (security)	The user name to use for basic authentication.		String

CHAPTER 95. OSGI EVENTADMIN COMPONENT

Available as of Camel version 2.6

The **eventadmin** component can be used in an OSGi environment to receive OSGi EventAdmin events and process them.

95.1. DEPENDENCIES

Maven users need to add the following dependency to their **pom.xml**

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-eventadmin</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.6.0 or higher).

95.2. URI FORMAT

```
eventadmin:topic[?options]
```

where **topic** is the name of the topic to listen too.

95.3. URI OPTIONS

The OSGi EventAdmin component supports 2 options which are listed below.

Name	Description	Default	Type
bundleContext (common)	The OSGi BundleContext is automatic injected by Camel		BundleContext
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The OSGi EventAdmin endpoint is configured using URI syntax:

```
eventadmin:topic
```

with the following path and query parameters:

95.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
topic	Name of topic to listen or send to		String

95.3.2. Query Parameters (5 parameters):

Name	Description	Default	Type
send (common)	Whether to use 'send' or 'synchronous' deliver. Default false (async delivery)	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

95.4. MESSAGE HEADERS

Name	Type	Message
Description		

95.5. MESSAGE BODY

The **in** message body will be set to the received Event.

95.6. EXAMPLE USAGE

```
<route>
    <from uri="eventadmin:*"/>
    <to uri="stream:out"/>
</route>
```

CHAPTER 96. EXEC COMPONENT

Available as of Camel version 2.3

The **exec** component can be used to execute system commands.

96.1. DEPENDENCIES

Maven users need to add the following dependency to their **pom.xml**

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-exec</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.3.0 or higher).

96.2. URI FORMAT

```
exec://executable[?options]
```

where **executable** is the name, or file path, of the system command that will be executed. If executable name is used (e.g. **exec:java**), the executable must in the system path.

96.3. URI OPTIONS

The Exec component has no options.

The Exec endpoint is configured using URI syntax:

```
exec:executable
```

with the following path and query parameters:

96.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
executable	**Required** Sets the executable to be executed. The executable must not be empty or null.		String

96.3.2. Query Parameters (8 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
args (producer)	The arguments may be one or many whitespace-separated tokens.		String
binding (producer)	A reference to a org.apache.commons.exec.ExecBinding in the Registry.		ExecBinding
commandExecutor (producer)	A reference to a org.apache.commons.exec.ExecCommandExecutor in the Registry that customizes the command execution. The default command executor utilizes the commons-exec library, which adds a shutdown hook for every executed command.		ExecCommandExecutor
outFile (producer)	The name of a file, created by the executable, that should be considered as its output. If no outFile is set, the standard output (stdout) of the executable will be used instead.		String
timeout (producer)	The timeout, in milliseconds, after which the executable should be terminated. If execution has not completed within the timeout, the component will send a termination request.		long
useStderrOnEmptyStdout (producer)	A boolean indicating that when stdout is empty, this component will populate the Camel Message Body with stderr. This behavior is disabled (false) by default.	false	boolean
workingDir (producer)	The directory in which the command should be executed. If null, the working directory of the current process will be used.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

96.4. MESSAGE HEADERS

The supported headers are defined in **org.apache.camel.component.exec.ExecBinding**.

Name	Type	Message	Description
ExecBinding.EXEC_COMMAND_EXECUTABLE	**String**	**in**	The name of the system command that will be executed. Overrides **executable** in the URI.
ExecBinding.EXEC_COMMAND_ARGS	**java.util.List<String>**	**in**	Command-line arguments to pass to the executed process. The arguments are used literally - no quoting is applied. Overrides any existing **args** in the URI.
ExecBinding.EXEC_COMMAND_ARGS	**String**	**in**	**Camel 2.5:** The arguments of the executable as a Single string where each argument is whitespace separated (see **args** in URI option). The arguments are used literally, no quoting is applied. Overrides any existing **args** in the URI.
ExecBinding.EXEC_COMMAND_OUT_FILE	**String**	**in**	The name of a file, created by the executable, that should be considered as its output. Overrides any existing **outFile** in the URI.
ExecBinding.EXEC_COMMAND_TIMEOUT	**long**	**in**	The timeout, in milliseconds, after which the executable should be terminated. Overrides any existing **timeout** in the URI.
ExecBinding.EXEC_COMMAND_WORKING_DIR	**String**	**in**	The directory in which the command should be executed. Overrides any existing **workingDir** in the URI.

Name	Type	Message	Description
ExecBinding.EXEC_EXIT_VALUE	**int**	**out**	The value of this header is the *exit value* of the executable. Non-zero exit values typically indicate abnormal termination. Note that the exit value is OS-dependent.
ExecBinding.EXEC_STDERR	**java.io.InputStream**	**out**	The value of this header points to the standard error stream (stderr) of the executable. If no stderr is written, the value is **null**.
ExecBinding.EXEC_USE_STDERR_ON_EMPTY_STDOUT	**boolean**	**in**	Indicates that when **stdout** is empty, this component will populate the Camel Message Body with **stderr**. This behavior is disabled (**false**) by default.

96.5. MESSAGE BODY

If the **Exec** component receives an **in** message body that is convertible to **java.io.InputStream**, it is used to feed input to the executable via its stdin. After execution, the message body is the result of the execution,- that is, an **org.apache.camel.components.exec.ExecResult** instance containing the stdout, stderr, exit value, and out file. This component supports the following **ExecResult** type converters for convenience:

From	To
ExecResult	**java.io.InputStream**
ExecResult	**String**
ExecResult	**byte []**
ExecResult	**org.w3c.dom.Document**

If an out file is specified (in the endpoint via **outFile** or the message headers via **ExecBinding.EXEC_COMMAND_OUT_FILE**), converters will return the content of the out file. If no out file is used, then this component will convert the stdout of the process to the target type. For more details, please refer to the usage examples below.

96.6. USAGE EXAMPLES

96.6.1. Executing word count (Linux)

The example below executes **wc** (word count, Linux) to count the words in file **/usr/share/dict/words**. The word count (output) is written to the standard output stream of **wc**.

```
from("direct:exec")
.to("exec:wc?args=--words /usr/share/dict/words")
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
      // By default, the body is ExecResult instance
      assertIsInstanceOf(ExecResult.class, exchange.getIn().getBody());
      // Use the Camel Exec String type converter to convert the ExecResult to String
      // In this case, the stdout is considered as output
      String wordCountOutput = exchange.getIn().getBody(String.class);
      // do something with the word count
    }
});
```

96.6.2. Executing java

The example below executes **java** with 2 arguments: **-server** and **-version**, provided that **java** is in the system path.

```
from("direct:exec")
.to("exec:java?args=-server -version")
```

The example below executes **java** in **c:\temp** with 3 arguments: **-server**, **-version** and the sytem property **user.name**.

```
from("direct:exec")
.to("exec:c:/program files/jdk/bin/java?args=-server -version -
Duser.name=Camel&workingDir=c:/temp")
```

96.6.3. Executing Ant scripts

The following example executes Apache Ant (Windows only) with the build file **CamelExecBuildFile.xml**, provided that **ant.bat** is in the system path, and that **CamelExecBuildFile.xml** is in the current directory.

```
from("direct:exec")
.to("exec:ant.bat?args=-f CamelExecBuildFile.xml")
```

In the next example, the **ant.bat** command redirects its output to **CamelExecOutFile.txt** with **-l**. The file **CamelExecOutFile.txt** is used as the out file with **outFile=CamelExecOutFile.txt**. The example assumes that **ant.bat** is in the system path, and that **CamelExecBuildFile.xml** is in the current directory.

```
from("direct:exec")
.to("exec:ant.bat?args=-f CamelExecBuildFile.xml -l
CamelExecOutFile.txt&outFile=CamelExecOutFile.txt")
```

```
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        InputStream outFile = exchange.getIn().getBody(InputStream.class);
        assertIsInstanceOf(InputStream.class, outFile);
        // do something with the out file here
    }
});
```

96.6.4. Executing echo (Windows)

Commands such as **echo** and **dir** can be executed only with the command interpreter of the operating system. This example shows how to execute such a command - **echo** - in Windows.

```
from("direct:exec").to("exec:cmd?args=/C echo echoString")
```

96.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 97. FACEBOOK COMPONENT

Available as of Camel version 2.14

The Facebook component provides access to all of the Facebook APIs accessible using Facebook4J. It allows producing messages to retrieve, add, and delete posts, likes, comments, photos, albums, videos, photos, checkins, locations, links, etc. It also supports APIs that allow polling for posts, users, checkins, groups, locations, etc.

Facebook requires the use of OAuth for all client application authentication. In order to use camel-facebook with your account, you'll need to create a new application within Facebook at https://developers.facebook.com/apps and grant the application access to your account. The Facebook application's id and secret will allow access to Facebook APIs which do not require a current user. A user access token is required for APIs that require a logged in user. More information on obtaining a user access token can be found at https://developers.facebook.com/docs/facebook-login/access-tokens/.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-facebook</artifactId>
  <version>${camel-version}</version>
</dependency>
```

97.1. URI FORMAT

```
facebook://[endpoint]?[options]
```

97.2. FACEBOOKCOMPONENT

The facebook component can be configured with the Facebook account settings below, which are mandatory. The values can be provided to the component using the bean property **configuration** of type **org.apache.camel.component.facebook.config.FacebookConfiguration**. The **oAuthAccessToken** option may be ommited but that will only allow access to application APIs.

The Facebook component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared configuration		FacebookConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Facebook endpoint is configured using URI syntax:

```
facebook:methodName
```

with the following path and query parameters:

97.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
methodName	**Required** What operation to perform		String

97.2.2. Query Parameters (102 parameters):

Name	Description	Default	Type
achievementURL (common)	The unique URL of the achievement		URL
albumId (common)	The album ID		String
albumUpdate (common)	The facebook Album to be created or updated		AlbumUpdate
appId (common)	The ID of the Facebook Application		String
center (common)	Location latitude and longitude		GeoLocation
checkinId (common)	The checkin ID		String
checkinUpdate (common)	**Deprecated** The checkin to be created. Deprecated, instead create a Post with an attached location		CheckinUpdate
clientURL (common)	Facebook4J API client URL		String
clientVersion (common)	Facebook4J client API version		String
commentId (common)	The comment ID		String
commentUpdate (common)	The facebook Comment to be created or updated		CommentUpdate

Name	Description	Default	Type
debugEnabled (common)	Enables deubg output. Effective only with the embedded logger	false	Boolean
description (common)	The description text		String
distance (common)	Distance in meters		Integer
domainId (common)	The domain ID		String
domainName (common)	The domain name		String
domainNames (common)	The domain names		List
eventId (common)	The event ID		String
eventUpdate (common)	The event to be created or updated		EventUpdate
friendId (common)	The friend ID		String
friendlistId (common)	The friend list ID		String
friendlistName (common)	The friend list Name		String
friendUserId (common)	The friend user ID		String
groupId (common)	The group ID		String
gzipEnabled (common)	Use Facebook GZIP encoding	true	Boolean
httpConnectionTimeout (common)	Http connection timeout in milliseconds	20000	Integer

Name	Description	Default	Type
httpDefaultMaxPerRoute (common)	HTTP maximum connections per route	2	Integer
httpMaxTotalConnections (common)	HTTP maximum total connections	20	Integer
httpReadTimeout (common)	Http read timeout in milliseconds	120000	Integer
httpRetryCount (common)	Number of HTTP retries	0	Integer
httpRetryIntervalSeconds (common)	HTTP retry interval in seconds	5	Integer
httpStreamingReadTimeout (common)	HTTP streaming read timeout in milliseconds	40000	Integer
ids (common)	The ids of users		List
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
includeRead (common)	Enables notifications that the user has already read in addition to unread ones		Boolean
isHidden (common)	Whether hidden		Boolean
jsonStoreEnabled (common)	If set to true, raw JSON forms will be stored in DataObjectFactory	false	Boolean
link (common)	Link URL		URL
linkId (common)	Link ID		String
locale (common)	Desired FQL locale		Locale
mbeanEnabled (common)	If set to true, Facebook4J mbean will be registerd	false	Boolean

Name	Description	Default	Type
message (common)	The message text		String
messageId (common)	The message ID		String
metric (common)	The metric name		String
milestoneId (common)	The milestone id		String
name (common)	Test user name, must be of the form 'first last'		String
noteId (common)	The note ID		String
notificationId (common)	The notification ID		String
objectId (common)	The insight object ID		String
offerId (common)	The offer id		String
optionDescription (common)	The question's answer option description		String
pageId (common)	The page id		String
permissionName (common)	The permission name		String
permissions (common)	Test user permissions in the format perm1,perm2,...		String
photoId (common)	The photo ID		String
pictureId (common)	The picture id		Integer
pictureId2 (common)	The picture2 id		Integer
pictureSize (common)	The picture size		PictureSize

Name	Description	Default	Type
placeId (common)	The place ID		String
postId (common)	The post ID		String
postUpdate (common)	The post to create or update		PostUpdate
prettyDebugEnabled (common)	Prettify JSON debug output if set to true	false	Boolean
queries (common)	FQL queries		Map
query (common)	FQL query or search terms for search endpoints		String
questionId (common)	The question id		String
reading (common)	Optional reading parameters. See Reading Options(reading)		Reading
readingOptions (common)	To configure Reading using key/value pairs from the Map.		Map
restBaseURL (common)	API base URL	https://graph.facebook.com/	String
scoreValue (common)	The numeric score with value		Integer
size (common)	The picture size, one of large, normal, small or square		PictureSize
source (common)	The media content from either a java.io.File or java.io.Inputstream		Media
subject (common)	The note of the subject		String
tabId (common)	The tab id		String
tagUpdate (common)	Photo tag information		TagUpdate
testUser1 (common)	Test user 1		TestUser

Name	Description	Default	Type
testUser2 (common)	Test user 2		TestUser
testUserId (common)	The ID of the test user		String
title (common)	The title text		String
toUserId (common)	The ID of the user to tag		String
toUserIds (common)	The IDs of the users to tag		List
userId (common)	The Facebook user ID		String
userId1 (common)	The ID of a user 1		String
userId2 (common)	The ID of a user 2		String
userIds (common)	The IDs of users to invite to event		List
userLocale (common)	The test user locale		String
useSSL (common)	Use SSL	true	Boolean
videoBaseURL (common)	Video API base URL	https://graph-video.facebook.com/	String
videoId (common)	The video ID		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
httpProxyHost (proxy)	HTTP proxy server host name		String
httpProxyPassword (proxy)	HTTP proxy server password		String
httpProxyPort (proxy)	HTTP proxy server port		Integer
httpProxyUser (proxy)	HTTP proxy server user name		String
oAuthAccessToken (security)	The user access token		String
oAuthAccessTokenURL (security)	OAuth access token URL	https://graph.facebook.com/oauth/access_token	String
oAuthAppId (security)	The application Id		String
oAuthAppSecret (security)	The application Secret		String

Name	Description	Default	Type
oAuthAuthorizationURL (security)	OAuth authorization URL	https://www.facebook.com/dialog/oauth	String
oAuthPermissions (security)	Default OAuth permissions. Comma separated permission names. See https://developers.facebook.com/docs/reference/login/permissions for the detail		String

97.3. PRODUCER ENDPOINTS:

Producer endpoints can use endpoint names and options from the table below. Endpoints can also use the short name without the **get** or **search** prefix, except **checkin** due to ambiguity between **getCheckin** and **searchCheckin**. Endpoint options that are not mandatory are denoted by [].

Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message. For example, the facebook endpoint in the following route retrieves activities for the user id value in the incoming message body.

```
from("direct:test").to("facebook://activities?inBody=userId")...
```

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelFacebook.https://cwiki.apache.org/confluence/pages/createpage.action?spaceKey=CAMEL&title=option&linkCreation=true&fromPageId=34020899[option]**. For example, the **userId** option value in the previous route could alternately be provided in the message header **CamelFacebook.userId**. Note that the inBody option overrides message header, e.g. the endpoint option **inBody=user** would override a **CamelFacebook.userId** header.

Endpoints that return a String return an Id for the created or modified entity, e.g. **addAlbumPhoto** returns the new album Id. Endpoints that return a boolean, return true for success and false otherwise. In case of Facebook API errors the endpoint will throw a RuntimeCamelException with a facebook4j.FacebookException cause.

97.4. CONSUMER ENDPOINTS:

Any of the producer endpoints that take a reading#reading parameter can be used as a consumer endpoint. The polling consumer uses the **since** and **until** fields to get responses within the polling interval. In addition to other reading fields, an initial **since** value can be provided in the endpoint for the first poll.

Rather than the endpoints returning a List (or **facebook4j.ResponseList**) through a single route exchange, camel-facebook creates one route exchange per returned object. As an example, if **"facebook://home"** results in five posts, the route will be executed five times (once for each Post).

97.5. READING OPTIONS

The **reading** option of type **facebook4j.Reading** adds support for reading parameters, which allow selecting specific fields, limits the number of results, etc. For more information see Graph API#reading - Facebook Developers.

It is also used by consumer endpoints to poll Facebook data to avoid sending duplicate messages across polls.

The reading option can be a reference or value of type **facebook4j.Reading**, or can be specified using the following reading options in either the endpoint URI or exchange header with **CamelFacebook.** prefix.

97.6. MESSAGE HEADER

Any of the URI options#urioptions can be provided in a message header for producer endpoints with **CamelFacebook.** prefix.

97.7. MESSAGE BODY

All result message bodies utilize objects provided by the Facebook4J API. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint parameter.

For endpoints that return an array, or **facebook4j.ResponseList**, or **java.util.List**, a consumer endpoint will map every elements in the list to distinct messages.

97.8. USE CASES

To create a post within your Facebook profile, send this producer a facebook4j.PostUpdate body.

```
from("direct:foo")
    .to("facebook://postFeed/inBody=postUpdate);
```

To poll, every 5 sec (You can set the polling consumer options by adding a prefix of "consumer"), all statuses on your home feed:

```
from("facebook://home?consumer.delay=5000")
    .to("bean:blah");
```

Searching using a producer with dynamic options from header.

In the bar header we have the Facebook search string we want to execute in public posts, so we need to assign this value to the CamelFacebook.query header.

```
from("direct:foo")
    .setHeader("CamelFacebook.query", header("bar"))
    .to("facebook://posts");
```

CHAPTER 98. FHIR JSON DATAFORMAT

Available as of Camel version 2.21Available as of Camel version 2.21Available as of Camel version 2.21

The FHIR-JSON Data Format leverages HAPI-FHIR's JSON parser to parse to/from JSON format to/from a HAPI-FHIR's **IBaseResource**.

98.1. FHIR JSON FORMAT OPTIONS

The FHIR JSon dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
fhirVersion	**DSTU3**	**String**	The version of FHIR to use. Possible values are: DSTU2,DSTU2_HL7ORG,DSTU2_1,DSTU3,R4
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

CHAPTER 99. FHIR XML DATAFORMAT

Available as of Camel version 2.21**Available as of Camel version 2.21**

The FHIR-XML Data Format leverages HAPI-FHIR's XML parser to parse to/from XML format to/from a HAPI-FHIR's **IBaseResource**.

99.1. FHIR XML FORMAT OPTIONS

The FHIR XML dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
fhirVersion	**DSTU3**	**String**	The version of FHIR to use. Possible values are: DSTU2,DSTU2_HL7ORG,DSTU2_1,DSTU3,R4
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

CHAPTER 100. FILE COMPONENT

Available as of Camel version 1.0

The File component provides access to file systems, allowing files to be processed by any other Camel Components or messages from other components to be saved to disk.

100.1. URI FORMAT

```
file:directoryName[?options]
```

or

```
file://directoryName[?options]
```

Where **directoryName** represents the underlying file directory.

You can append query options to the URI in the following format, **?option=value&option=value&…**

Only directories

Camel supports only endpoints configured with a starting directory. So the **directoryName** must be a directory.
If you want to consume a single file only, you can use the **fileName** option, e.g. by setting **fileName=thefilename**.
Also, the starting directory must not contain dynamic expressions with $\{ } placeholders. Again use the **fileName** option to specify the dynamic part of the filename.

WARNING

Avoid reading files currently being written by another application Beware the JDK File IO API is a bit limited in detecting whether another application is currently writing/copying a file. And the implementation can be different depending on OS platform as well. This could lead to that Camel thinks the file is not locked by another process and start consuming it. Therefore you have to do you own investigation what suites your environment. To help with this Camel provides different **readLock** options and **doneFileName** option that you can use. See also the section *Consuming files from folders where others drop files directly* .

100.2. URI OPTIONS

The File component has no options.

The File endpoint is configured using URI syntax:

```
file:directoryName
```

with the following path and query parameters:

100.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
directoryName	**Required** The starting directory		File

100.2.2. Query Parameters (81 parameters):

Name	Description	Default	Type
charset (common)	This option is used to specify the encoding of the file. You can use this on the consumer, to specify the encodings of the files, which allow Camel to know the charset it should load the file content in case the file content is being accessed. Likewise when writing a file, you can use this option to specify which charset to write the file as well. Do mind that when writing the file Camel may have to read the message content into memory to be able to convert the data into the configured charset, so do not use this if you have big messages.		String
doneFileName (common)	Producer: If provided, then Camel will write a 2nd done file when the original file has been written. The done file will be empty. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The done file will always be written in the same folder as the original file. Consumer: If provided, Camel will only consume files if a done file exists. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders.The done file is always expected in the same folder as the original file. Only $file.name and $file.name.noext is supported as dynamic placeholders.		String

Name	Description	Default	Type
fileName (common)	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the CamelFileName header. (Note: The header itself can also be an Expression). The expression options support both String and Expression types. If the expression is a String type, it is always evaluated using the File Language. If the expression is an Expression type, the specified Expression type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: mydata-$date:now:yyyyMMdd.txt. The producers support the CamelOverruleFileName header which takes precedence over any existing CamelFileName header; the CamelOverruleFileName is a header that is used only once, and makes it easier as this avoids to temporary store CamelFileName and have to restore it afterwards.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
delete (consumer)	If true, the file will be deleted after it is processed successfully.	false	boolean
moveFailed (consumer)	Sets the move failure expression based on Simple language. For example, to move files into a .error subdirectory use: .error. Note: When moving the files to the fail location Camel will handle the error and will not pick up the file again.		String
noop (consumer)	If true, the file is not moved or deleted in any way. This option is good for readonly data, or for ETL type requirements. If noop=true, Camel will set idempotent=true as well, to avoid consuming the same files over and over again.	false	boolean

Name	Description	Default	Type
preMove (consumer)	Expression (such as File Language) used to dynamically set the filename when moving it before processing. For example to move in-progress files into the order directory set this value to order.		String
preSort (consumer)	When pre-sort is enabled then the consumer will sort the file and directory names during polling, that was retrieved from the file system. You may want to do this in case you need to operate on the files in a sorted order. The pre-sort is executed before the consumer starts to filter, and accept files to process by Camel. This option is default=false meaning disabled.	false	boolean
recursive (consumer)	If a directory, will look for files in all the sub-directories as well.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
directoryMustExist (consumer)	Similar to startingDirectoryMustExist but this applies during polling recursive sub directories.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
extendedAttributes (consumer)	To define which file attributes of interest. Like posix:permissions,posix:owner,basic:lastAccessTime, it supports basic wildcard like posix:, basic:lastAccessTime		String
inProgressRepository (consumer)	A pluggable in-progress repository org.apache.camel.spi.IdempotentRepository. The in-progress repository is used to account the current in progress files being consumed. By default a memory based repository is used.		String>

Name	Description	Default	Type
localWorkDirectory (consumer)	When consuming, a local work directory can be used to store the remote file content directly in local files, to avoid loading the content into memory. This is beneficial, if you consume a very big remote file and thus can conserve memory.		String
onCompletionException Handler (consumer)	To use a custom org.apache.camel.spi.ExceptionHandler to handle any thrown exceptions that happens during the file on completion process where the consumer does either a commit or rollback. The default implementation will log any exception at WARN level and ignore.		ExceptionHandler
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel. In other words the error occurred while the polling was gathering information, for instance access to a file network failed so Camel cannot access it to scan for files. The default implementation will log the caused exception at WARN level and ignore it.		PollingConsumerPoll Strategy
probeContentType (consumer)	Whether to enable probing of the content type. If enable then the consumer uses link FilesprobeContentType(java.nio.file.Path) to determine the content-type of the file, and store that as a header with key link ExchangeFILE_CONTENT_TYPE on the Message.	false	boolean
processStrategy (consumer)	A pluggable org.apache.camel.component.file.GenericFileProcessStrategy allowing you to implement your own readLock option or similar. Can also be used when special conditions must be met before a file can be consumed, such as a special ready file exists. If this option is set then the readLock option does not apply.		GenericFileProcess Strategy<T>

Name	Description	Default	Type
startingDirectoryMustExist (consumer)	Whether the starting directory must exist. Mind that the autoCreate option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable autoCreate and enable this to ensure the starting directory must exist. Will thrown an exception if the directory doesn't exist.	false	boolean
fileExist (producer)	What to do if a file already exists with the same name. Override, which is the default, replaces the existing file. Append - adds content to the existing file. Fail - throws a GenericFileOperationException, indicating that there is already an existing file. Ignore - silently ignores the problem and does not override the existing file, but assumes everything is okay. Move - option requires to use the moveExisting option to be configured as well. The option eagerDeleteTargetFile can be used to control what to do if an moving the file, and there exists already an existing file, otherwise causing the move operation to fail. The Move option will move any existing files, before writing the target file. TryRename is only applicable if tempFileName option is in use. This allows to try renaming the file from the temporary name to the actual name, without doing any exists check. This check may be faster on some file systems and especially FTP servers.	Override	GenericFileExist
flatten (producer)	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to true on the producer enforces that any file name in CamelFileName header will be stripped for any leading paths.	false	boolean
moveExisting (producer)	Expression (such as File Language) used to compute file name to use when fileExist=Move is configured. To move files into a backup subdirectory just enter backup. This option only supports the following File Language tokens: file:name, file:name.ext, file:name.noext, file:onlyname, file:onlyname.noext, file:ext, and file:parent. Notice the file:parent is not supported by the FTP component, as the FTP component can only move any existing files to a relative directory based on current dir as base.		String

Name	Description	Default	Type
tempFileName (producer)	The same as tempPrefix option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language.		String
tempPrefix (producer)	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP when uploading big files.		String
allowNullBody (producer)	Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a GenericFileWriteException of 'Cannot write null body to file.' will be thrown. If the fileExist option is set to 'Override', then the file will be truncated, and if set to append the file will remain unchanged.	false	boolean
chmod (producer)	Specify the file permissions which is sent by the producer, the chmod value must be between 000 and 777; If there is a leading digit like in 0755 we will ignore it.		String
chmodDirectory (producer)	Specify the directory permissions used when the producer creates missing directories, the chmod value must be between 000 and 777; If there is a leading digit like in 0755 we will ignore it.		String
eagerDeleteTargetFile (producer)	Whether or not to eagerly delete any existing target file. This option only applies when you use fileExists=Override and the tempFileName option as well. You can use this to disable (set it to false) deleting the target file before the temp file is written. For example you may write big files and want the target file to exists during the temp file is being written. This ensure the target file is only deleted until the very last moment, just before the temp file is being renamed to the target filename. This option is also used to control whether to delete any existing files when fileExist=Move is enabled, and an existing file exists. If this option copyAndDeleteOnRenameFails false, then an exception will be thrown if an existing file existed, if its true, then the existing file is deleted before the move operation.	true	boolean

Name	Description	Default	Type
forceWrites (producer)	Whether to force syncing writes to the file system. You can turn this off if you do not want this level of guarantee, for example if writing to logs / audit logs etc; this would yield better performance.	true	boolean
keepLastModified (producer)	Will keep the last modified timestamp from the source file (if any). Will use the Exchange.FILE_LAST_MODIFIED header to located the timestamp. This header can contain either a java.util.Date or long with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. Note: This option only applies to the file producer. You cannot use this option with any of the ftp producers.	false	boolean
autoCreate (advanced)	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means the directory the files should be written to.	true	boolean
bufferSize (advanced)	Write buffer sized in bytes.	131072	int
copyAndDeleteOnRenameFail (advanced)	Whether to fallback and do a copy and delete file, in case the file could not be renamed directly. This option is not available for the FTP component.	true	boolean
renameUsingCopy (advanced)	Perform rename operations using a copy and delete strategy. This is primarily used in environments where the regular rename operation is unreliable (e.g. across different file systems or networks). This option takes precedence over the copyAndDeleteOnRenameFail parameter that will automatically fall back to the copy and delete strategy, but only after additional delays.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
antExclude (filter)	Ant style filter exclusion. If both antInclude and antExclude are used, antExclude takes precedence over antInclude. Multiple exclusions may be specified in comma-delimited format.		String
antFilterCaseSensitive (filter)	Sets case sensitive flag on ant filter	true	boolean

Name	Description	Default	Type
antInclude (filter)	Ant style filter inclusion. Multiple inclusions may be specified in comma-delimited format.		String
eagerMaxMessagesPerPoll (filter)	Allows for controlling whether the limit from maxMessagesPerPoll is eager or not. If eager then the limit is during the scanning of files. Where as false would scan all files, and then perform sorting. Setting this option to false allows for sorting all files first, and then limit the poll. Mind that this requires a higher memory usage as all file details are in memory to perform the sorting.	true	boolean
exclude (filter)	Is used to exclude files, if filename matches the regex pattern (matching is case in-senstive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
filter (filter)	Pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. Will skip files if filter returns false in its accept() method.		GenericFileFilter< T>
filterDirectory (filter)	Filters the directory based on Simple language. For example to filter on current date, you can use a simple date pattern such as $date:now:yyyMMdd		String
filterFile (filter)	Filters the file based on Simple language. For example to filter on file size, you can use $file:size 5000		String
idempotent (filter)	Option to use the Idempotent Consumer EIP pattern to let Camel skip already processed files. Will by default use a memory based LRUCache that holds 1000 entries. If noop=true then idempotent will be enabled as well to avoid consuming the same files over and over again.	false	Boolean
idempotentKey (filter)	To use a custom idempotent key. By default the absolute path of the file is used. You can use the File Language, for example to use the file name and file size, you can do: idempotentKey=$file:name-$file:size		String
idempotentRepository (filter)	A pluggable repository org.apache.camel.spi.IdempotentRepository which by default use MemoryMessageIdRepository if none is specified and idempotent is true.		String>

Name	Description	Default	Type
include (filter)	Is used to include files, if filename matches the regex pattern (matching is case in-sensitive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
maxDepth (filter)	The maximum depth to traverse when recursively processing a directory.	2147483647	int
maxMessagesPerPoll (filter)	To define a maximum messages to gather per poll. By default no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disabled it. Notice: If this option is in use then the File and FTP components will limit before any sorting. For example if you have 100000 files and use maxMessagesPerPoll=500, then only the first 500 files will be picked up, and then sorted. You can use the eagerMaxMessagesPerPoll option and set this to false to allow to scan all files first and then sort afterwards.		int
minDepth (filter)	The minimum depth to start processing when recursively processing a directory. Using minDepth=1 means the base directory. Using minDepth=2 means the first sub directory.		int
move (filter)	Expression (such as Simple Language) used to dynamically set the filename when moving it after processing. To move files into a .done subdirectory just enter .done.		String
exclusiveReadLockStrategy (lock)	Pluggable read-lock as a org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy implementation.		GenericFileExclusiveReadLockStrategy<T>

Name	Description	Default	Type
readLock (lock)	Used by consumer, to only poll the files if it has exclusive read-lock on the file (i.e. the file is not in-progress or being written). Camel will wait until the file lock is granted. This option provides the build in strategies: none - No read lock is in use markerFile - Camel creates a marker file (fileName.camelLock) and then holds a lock on it. This option is not available for the FTP component changed - Changed is using file length/modification timestamp to detect whether the file is currently being copied or not. Will at least use 1 sec to determine this, so this option cannot consume files as fast as the others, but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option readLockCheckInterval can be used to set the check frequency. fileLock - is for using java.nio.channels.FileLock. This option is not avail for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks. rename - rename is for using a try to rename the file as a test if we can get exclusive read-lock. idempotent - (only for file component) idempotent is for using a idempotentRepository as the read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-changed - (only for file component) idempotent-changed is for using a idempotentRepository and changed as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-rename - (only for file component) idempotent-rename is for using a idempotentRepository and rename as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. Notice: The various read locks is not all suited to work in clustered mode, where concurrent consumers on different nodes is competing for the same files on a shared file system. The markerFile using a close to atomic operation to create the empty marker file, but its not guaranteed to work in a cluster. The fileLock may work better but then the file system need to support distributed file locks, and so on. Using the idempotent read lock can support clustering if the idempotent repository supports clustering, such as Hazelcast Component or Infinispan.	none	String

Name	Description	Default	Type
readLockCheckInterval (lock)	Interval in millis for the read-lock, if supported by the read lock. This interval is used for sleeping between attempts to acquire the read lock. For example when using the changed read lock, you can set a higher interval period to cater for slow writes. The default of 1 sec. may be too fast if the producer is very slow writing the file. Notice: For FTP the default readLockCheckInterval is 5000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	1000	long
readLockDeleteOrphanLock Files (lock)	Whether or not read lock with marker files should upon startup delete any orphan read lock files, which may have been left on the file system, if Camel was not properly shutdown (such as a JVM crash). If turning this option to false then any orphaned lock file will cause Camel to not attempt to pickup that file, this could also be due another node is concurrently reading files from the same shared directory.	true	boolean
readLockLoggingLevel (lock)	Logging level used when a read lock could not be acquired. By default a WARN is logged. You can change this level, for example to OFF to not have any logging. This option is only applicable for readLock of types: changed, fileLock, idempotent, idempotent-changed, idempotent-rename, rename.	DEBUG	LoggingLevel
readLockMarkerFile (lock)	Whether to use marker file with the changed, rename, or exclusive read lock types. By default a marker file is used as well to guard against other processes picking up the same files. This behavior can be turned off by setting this option to false. For example if you do not want to write marker files to the file systems by the Camel application.	true	boolean
readLockMinAge (lock)	This option applied only for readLock=change. This option allows to specify a minimum age the file must be before attempting to acquire the read lock. For example use readLockMinAge=300s to require the file is at last 5 minutes old. This can speedup the changed read lock as it will only attempt to acquire files which are at least that given age.	0	long

Name	Description	Default	Type
readLockMinLength (lock)	This option applied only for readLock=changed. This option allows you to configure a minimum file length. By default Camel expects the file to contain data, and thus the default value is 1. You can set this option to zero, to allow consuming zero-length files.	1	long
readLockRemoveOnCommit (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file is succeeded and a commit happens. By default the file is not removed which ensures that any race-condition do not occur so another active node may attempt to grab the file. Instead the idempotent repository may support eviction strategies that you can configure to evict the file name entry after X minutes - this ensures no problems with race conditions.	false	boolean
readLockRemoveOnRollback (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file failed and a rollback happens. If this option is false, then the file name entry is confirmed (as if the file did a commit).	true	boolean
readLockTimeout (lock)	Optional timeout in millis for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout triggered, then Camel will skip the file. At next poll Camel, will try the file again, and this time maybe the read-lock could be granted. Use a value of 0 or lower to indicate forever. Currently fileLock, changed and rename support the timeout. Notice: For FTP the default readLockTimeout value is 20000 instead of 10000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	10000	long
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. The default value is 500. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. The default value is 1000. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool. This option allows you to share a thread pool among multiple consumers.		ScheduledExecutor Service
scheduler (scheduler)	Allow to plugin a custom org.apache.camel.spi.ScheduledPollConsumerScheduler to use as the scheduler for firing when the polling consumer runs. The default implementation uses the ScheduledExecutorService and there is a Quartz2, and Spring based which supports CRON expressions. Notice: If using a custom scheduler then the options for initialDelay, useFixedDelay, timeUnit, and scheduledExecutorService may not be in use. Use the text quartz2 to refer to use the Quartz2 scheduler; and use the text spring to use the Spring based; and use the text myScheduler to refer to a custom scheduler by its id in the Registry. See Quartz2 page for an example.	none	ScheduledPollConsumer Scheduler

Name	Description	Default	Type
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
shuffle (sort)	To shuffle the list of files (sort in random order)	false	boolean
sortBy (sort)	Built-in sort by using the File Language. Supports nested sorts, so you can have a sort by file name and as a 2nd group sort by modified date.		String
sorter (sort)	Pluggable sorter as a java.util.Comparator class.		GenericFile<T>>

TIP

Default behavior for file producer By default it will override any existing file, if one exist with the same name.

100.3. MOVE AND DELETE OPERATIONS

Any move or delete operations is executed after (post command) the routing has completed; so during processing of the **Exchange** the file is still located in the inbox folder.

Lets illustrate this with an example:

```
from("file://inbox?move=.done").to("bean:handleOrder");
```

When a file is dropped in the **inbox** folder, the file consumer notices this and creates a new **FileExchange** that is routed to the **handleOrder** bean. The bean then processes the **File** object. At this point in time the file is still located in the **inbox** folder. After the bean completes, and thus the route is completed, the file consumer will perform the move operation and move the file to the **.done** sub-folder.

The **move** and the **preMove** options are considered as a directory name (though if you use an expression such as File Language, or Simple then the result of the expression evaluation is the file name to be used - eg if you set

```
move=../backup/copy-of-${file:name}
```

then that's using the File Language which we use return the file name to be used), which can be either relative or absolute. If relative, the directory is created as a sub-folder from within the folder where the file was consumed.

By default, Camel will move consumed files to the **.camel** sub-folder relative to the directory where the file was consumed.

If you want to delete the file after processing, the route should be:

```
from("file://inobox?delete=true").to("bean:handleOrder");
```

We have introduced a **pre** move operation to move files **before** they are processed. This allows you to mark which files have been scanned as they are moved to this sub folder before being processed.

```
from("file://inbox?preMove=inprogress").to("bean:handleOrder");
```

You can combine the **pre** move and the regular move:

```
from("file://inbox?preMove=inprogress&move=.done").to("bean:handleOrder");
```

So in this situation, the file is in the **inprogress** folder when being processed and after it's processed, it's moved to the **.done** folder.

100.4. FINE GRAINED CONTROL OVER MOVE AND PREMOVE OPTION

The **move** and **preMove** options are Expression-based, so we have the full power of the File Language to do advanced configuration of the directory and name pattern.
Camel will, in fact, internally convert the directory name you enter into a File Language expression. So when we enter **move=.done** Camel will convert this into: **${file:parent}/.done/${``file:onlyname**}. This is only done if Camel detects that you have not provided a $\{ } in the option value yourself. So when you enter a $\{ } Camel will **not** convert it and thus you have the full power.

So if we want to move the file into a backup folder with today's date as the pattern, we can do:

```
move=backup/${date:now:yyyyMMdd}/${file:name}
```

100.5. ABOUT MOVEFAILED

The **moveFailed** option allows you to move files that **could not** be processed succesfully to another location such as a error folder of your choice. For example to move the files in an error folder with a timestamp you can use **moveFailed=/error/${file:name.noext}-${date:now:yyyyMMddHHmmssSSS}.${``file:ext**}.

See more examples at File Language

100.6. MESSAGE HEADERS

The following headers are supported by this component:

100.6.1. File producer only

Header	Description
CamelFileName	Specifies the name of the file to write (relative to the endpoint directory). This name can be a **String**; a **String** with a File Language or Simple expression; or an Expression object. If it's **null** then Camel will auto-generate a filename based on the message unique ID.
CamelFileNameProduced	The actual absolute filepath (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users with the name of the file that was written.
CamelOverruleFileName	**Camel 2.11:** Is used for overruling **CamelFileName** header and use the value instead (but only once, as the producer will remove this header after writing the file). The value can be only be a String. Notice that if the option **fileName** has been configured, then this is still being evaluated.

100.6.2. File consumer only

Header	Description
CamelFileName	Name of the consumed file as a relative file path with offset from the starting directory configured on the endpoint.
CamelFileNameOnly	Only the file name (the name with no leading paths).
CamelFileAbsolute	A **boolean** option specifying whether the consumed file denotes an absolute path or not. Should normally be **false** for relative paths. Absolute paths should normally not be used but we added to the move option to allow moving files to absolute paths. But can be used elsewhere as well.
CamelFileAbsolutePath	The absolute path to the file. For relative files this path holds the relative path instead.
CamelFilePath	The file path. For relative files this is the starting directory + the relative filename. For absolute files this is the absolute path.
CamelFileRelativePath	The relative path.

Header	Description
CamelFileParent	The parent path.
CamelFileLength	A **long** value containing the file size.
CamelFileLastModified	A **Long** value containing the last modified timestamp of the file. In**Camel 2.10.3 and older** the type is **Date**.

100.7. BATCH CONSUMER

This component implements the Batch Consumer.

100.8. EXCHANGE PROPERTIES, FILE CONSUMER ONLY

As the file consumer implements the **BatchConsumer** it supports batching the files it polls. By batching we mean that Camel will add the following additional properties to the Exchange, so you know the number of files polled, the current index, and whether the batch is already completed.

Property	Description
CamelBatchSize	The total number of files that was polled in this batch.
CamelBatchIndex	The current index of the batch. Starts from 0.
CamelBatchComplete	A **boolean** value indicating the last Exchange in the batch. Is only**true** for the last entry.

This allows you for instance to know how many files exist in this batch and for instance let the Aggregator2 aggregate this number of files.

100.9. USING CHARSET

Available as of Camel 2.9.3
The charset option allows for configuring an encoding of the files on both the consumer and producer endpoints. For example if you read utf-8 files, and want to convert the files to iso-8859-1, you can do:

```
from("file:inbox?charset=utf-8")
  .to("file:outbox?charset=iso-8859-1")
```

You can also use the **convertBodyTo** in the route. In the example below we have still input files in utf-8 format, but we want to convert the file content to a byte array in iso-8859-1 format. And then let a bean process the data. Before writing the content to the outbox folder using the current charset.

```
from("file:inbox?charset=utf-8")
  .convertBodyTo(byte[].class, "iso-8859-1")
  .to("bean:myBean")
  .to("file:outbox");
```

If you omit the charset on the consumer endpoint, then Camel does not know the charset of the file, and would by default use "UTF-8". However you can configure a JVM system property to override and use a different default encoding with the key **org.apache.camel.default.charset**.

In the example below this could be a problem if the files is not in UTF-8 encoding, which would be the default encoding for read the files.
In this example when writing the files, the content has already been converted to a byte array, and thus would write the content directly as is (without any further encodings).

```
from("file:inbox")
  .convertBodyTo(byte[].class, "iso-8859-1")
  .to("bean:myBean")
  .to("file:outbox");
```

You can also override and control the encoding dynamic when writing files, by setting a property on the exchange with the key **Exchange.CHARSET_NAME**. For example in the route below we set the property with a value from a message header.

```
from("file:inbox")
  .convertBodyTo(byte[].class, "iso-8859-1")
  .to("bean:myBean")
  .setProperty(Exchange.CHARSET_NAME, header("someCharsetHeader"))
  .to("file:outbox");
```

We suggest to keep things simpler, so if you pickup files with the same encoding, and want to write the files in a specific encoding, then favor to use the **charset** option on the endpoints.

Notice that if you have explicit configured a **charset** option on the endpoint, then that configuration is used, regardless of the **Exchange.CHARSET_NAME** property.

If you have some issues then you can enable DEBUG logging on **org.apache.camel.component.file**, and Camel logs when it reads/write a file using a specific charset.
For example the route below will log the following:

```
from("file:inbox?charset=utf-8")
  .to("file:outbox?charset=iso-8859-1")
```

And the logs:

```
DEBUG GenericFileConverter          - Read file /Users/davsclaus/workspace/camel/camel-
core/target/charset/input/input.txt with charset utf-8
DEBUG FileOperations                - Using Reader to write file: target/charset/output.txt with charset:
iso-8859-1
```

100.10. COMMON GOTCHAS WITH FOLDER AND FILENAMES

When Camel is producing files (writing files) there are a few gotchas affecting how to set a filename of your choice. By default, Camel will use the message ID as the filename, and since the message ID is normally a unique generated ID, you will end up with filenames such as: **ID-MACHINENAME-2443-1211718892437-1-0**. If such a filename is not desired, then you must provide a filename in the **CamelFileName** message header. The constant, **Exchange.FILE_NAME**, can also be used.

The sample code below produces files using the message ID as the filename:

```
from("direct:report").to("file:target/reports");
```

To use **report.txt** as the filename you have to do:

```
from("direct:report").setHeader(Exchange.FILE_NAME, constant("report.txt")).to( "file:target/reports");
```

i. the same as above, but with **CamelFileName**:

```
from("direct:report").setHeader("CamelFileName", constant("report.txt")).to( "file:target/reports");
```

And a syntax where we set the filename on the endpoint with the **fileName** URI option.

```
from("direct:report").to("file:target/reports/?fileName=report.txt");
```

100.11. FILENAME EXPRESSION

Filename can be set either using the **expression** option or as a string-based File Language expression in the **CamelFileName** header. See the File Language for syntax and samples.

100.12. CONSUMING FILES FROM FOLDERS WHERE OTHERS DROP FILES DIRECTLY

Beware if you consume files from a folder where other applications write files to directly. Take a look at the different readLock options to see what suits your use cases. The best approach is however to write to another folder and after the write move the file in the drop folder. However if you write files directly to the drop folder then the option changed could better detect whether a file is currently being written/copied as it uses a file changed algorithm to see whether the file size / modification changes over a period of time. The other readLock options rely on Java File API that sadly is not always very good at detecting this. You may also want to look at the doneFileName option, which uses a marker file (done file) to signal when a file is done and ready to be consumed.

100.13. USING DONE FILES

Available as of Camel 2.6

See also section *writing done files* below.

If you want only to consume files when a done file exists, then you can use the **doneFileName** option on the endpoint.

```
from("file:bar?doneFileName=done");
```

Will only consume files from the bar folder, if a done *file* exists in the same directory as the target files. Camel will automatically delete the *done file* when it's done consuming the files. From Camel **2.9.3** onwards Camel will not automatically delete the *done file* if **noop=true** is configured.

However it is more common to have one *done file* per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the **doneFileName** option. Currently Camel supports the following two dynamic tokens: **file:name** and **file:name.noext** which must be enclosed in $\{ }. The consumer only supports the static part of the *done file* name as either prefix or suffix (not both).

```
from("file:bar?doneFileName=${file:name}.done");
```

In this example only files will be polled if there exists a done file with the name *file name*.done. For example

- **hello.txt** - is the file to be consumed
- **hello.txt.done** - is the associated done file

You can also use a prefix for the done file, such as:

```
from("file:bar?doneFileName=ready-${file:name}");
```

- **hello.txt** - is the file to be consumed
- **ready-hello.txt** - is the associated done file

100.14. WRITING DONE FILES

Available as of Camel 2.6

After you have written a file you may want to write an additional *donefile* as a kind of marker, to indicate to others that the file is finished and has been written. To do that you can use the **doneFileName** option on the file producer endpoint.

```
.to("file:bar?doneFileName=done");
```

Will simply create a file named **done** in the same directory as the target file.

However it is more common to have one done file per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the **doneFileName** option. Currently Camel supports the following two dynamic tokens: **file:name** and **file:name.noext** which must be enclosed in $\{ }.

```
.to("file:bar?doneFileName=done-${file:name}");
```

Will for example create a file named **done-foo.txt** if the target file was **foo.txt** in the same directory as the target file.

```
.to("file:bar?doneFileName=${file:name}.done");
```

Will for example create a file named **foo.txt.done** if the target file was **foo.txt** in the same directory as the target file.

```
.to("file:bar?doneFileName=${file:name.noext}.done");
```

Will for example create a file named **foo.done** if the target file was **foo.txt** in the same directory as the target file.

100.15. SAMPLES

#=== Read from a directory and write to another directory

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

100.15.1. Read from a directory and write to another directory using a overrule dynamic name

```
from("file://inputdir/?delete=true").to("file://outputdir?overruleFile=copy-of-${file:name}")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the **outputdir** and delete the file in the **inputdir**.

100.15.2. Reading recursively from a directory and writing to another

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the **outputdir** and delete the file in the **inputdir**. Will scan recursively into sub-directories. Will lay out the files in the same directory structure in the **outputdir** as the **inputdir**, including any sub-directories.

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

Will result in the following output layout:

```
outputdir/foo.txt
outputdir/sub/bar.txt
```

100.16. USING FLATTEN

If you want to store the files in the outputdir directory in the same directory, disregarding the source directory layout (e.g. to flatten out the path), you just add the **flatten=true** option on the file producer side:

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir?flatten=true")
```

Will result in the following output layout:

```
outputdir/foo.txt
outputdir/bar.txt
```

100.17. READING FROM A DIRECTORY AND THE DEFAULT MOVE OPERATION

Camel will by default move any processed file into a **.camel** subdirectory in the directory the file was consumed from.

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Affects the layout as follows:
before

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

after

```
inputdir/.camel/foo.txt
inputdir/sub/.camel/bar.txt
outputdir/foo.txt
outputdir/sub/bar.txt
```

100.18. READ FROM A DIRECTORY AND PROCESS THE MESSAGE IN JAVA

```
from("file://inputdir/").process(new Processor() {
  public void process(Exchange exchange) throws Exception {
    Object body = exchange.getIn().getBody();
    // do some business logic with the input body
  }
});
```

The body will be a **File** object that points to the file that was just dropped into the **inputdir** directory.

100.19. WRITING TO FILES

Camel is of course also able to write files, i.e. produce files. In the sample below we receive some reports on the SEDA queue that we process before they are being written to a directory.

100.19.1. Write to subdirectory using Exchange.FILE_NAME

Using a single route, it is possible to write a file to any number of subdirectories. If you have a route setup as such:

```
<route>
  <from uri="bean:myBean"/>
  <to uri="file:/rootDirectory"/>
</route>
```

You can have **myBean** set the header **Exchange.FILE_NAME** to values such as:

```
Exchange.FILE_NAME = hello.txt => /rootDirectory/hello.txt
Exchange.FILE_NAME = foo/bye.txt => /rootDirectory/foo/bye.txt
```

This allows you to have a single route to write files to multiple destinations.

100.19.2. Writing file through the temporary directory relative to the final destination

Sometime you need to temporarily write the files to some directory relative to the destination directory. Such situation usually happens when some external process with limited filtering capabilities is reading from the directory you are writing to. In the example below files will be written to the **/var/myapp/filesInProgress** directory and after data transfer is done, they will be atomically moved to the ` /var/myapp/finalDirectory `directory.

```
from("direct:start").
  to("file:///var/myapp/finalDirectory?tempPrefix=/../filesInProgress/");
```

100.20. USING EXPRESSION FOR FILENAMES

In this sample we want to move consumed files to a backup folder using today's date as a sub-folder name:

```
from("file://inbox?move=backup/${date:now:yyyyMMdd}/${file:name}").to("...");
```

See File Language for more samples.

100.21. AVOIDING READING THE SAME FILE MORE THAN ONCE (IDEMPOTENT CONSUMER)

Camel supports Idempotent Consumer directly within the component so it will skip already processed files. This feature can be enabled by setting the **idempotent=true** option.

```
from("file://inbox?idempotent=true").to("...");
```

Camel uses the absolute file name as the idempotent key, to detect duplicate files. From **Camel 2.11** onwards you can customize this key by using an expression in the idempotentKey option. For example to use both the name and the file size as the key

```
<route>
  <from uri="file://inbox?idempotent=true&amp;idempotentKey=${file:name}-${file:size}"/>
  <to uri="bean:processInbox"/>
</route>
```

By default Camel uses a in memory based store for keeping track of consumed files, it uses a least recently used cache holding up to 1000 entries. You can plugin your own implementation of this store by using the **idempotentRepository** option using the **#** sign in the value to indicate it's a referring to a bean in the Registry with the specified **id**.

```
<!-- define our store as a plain spring bean -->
<bean id="myStore" class="com.mycompany.MyIdempotentStore"/>
```

```
<route>
  <from uri="file://inbox?idempotent=true&amp;idempotentRepository=#myStore"/>
  <to uri="bean:processInbox"/>
</route>
```

Camel will log at **DEBUG** level if it skips a file because it has been consumed before:

```
DEBUG FileConsumer is idempotent and the file has been consumed before. Will skip this file:
target\idempotent\report.txt
```

100.22. USING A FILE BASED IDEMPOTENT REPOSITORY

In this section we will use the file based idempotent repository **org.apache.camel.processor.idempotent.FileIdempotentRepository** instead of the in-memory based that is used as default.
This repository uses a 1st level cache to avoid reading the file repository. It will only use the file repository to store the content of the 1st level cache. Thereby the repository can survive server restarts. It will load the content of the file into the 1st level cache upon startup. The file structure is very simple as it stores the key in separate lines in the file. By default, the file store has a size limit of 1mb. When the file grows larger Camel will truncate the file store, rebuilding the content by flushing the 1st level cache into a fresh empty file.

We configure our repository using Spring XML creating our file idempotent repository and define our file consumer to use our repository with the **idempotentRepository** using **#** sign to indicate Registry lookup:

100.23. USING A JPA BASED IDEMPOTENT REPOSITORY

In this section we will use the JPA based idempotent repository instead of the in-memory based that is used as default.

First we need a persistence-unit in **META-INF/persistence.xml** where we need to use the class **org.apache.camel.processor.idempotent.jpa.MessageProcessed** as model.

```
<persistence-unit name="idempotentDb" transaction-type="RESOURCE_LOCAL">
  <class>org.apache.camel.processor.idempotent.jpa.MessageProcessed</class>

  <properties>
    <property name="openjpa.ConnectionURL" value="jdbc:derby:target/idempotentTest;create=true"/>
    <property name="openjpa.ConnectionDriverName"
value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema"/>
    <property name="openjpa.Log" value="DefaultLevel=WARN, Tool=INFO"/>
    <property name="openjpa.Multithreaded" value="true"/>
  </properties>
</persistence-unit>
```

Next, we can create our JPA idempotent repository in the spring XML file as well:

```
<!-- we define our jpa based idempotent repository we want to use in the file consumer -->
<bean id="jpaStore" class="org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository">
    <!-- Here we refer to the entityManagerFactory -->
    <constructor-arg index="0" ref="entityManagerFactory"/>
    <!-- This 2nd parameter is the name  (= a category name).
```

```
        You can have different repositories with different names -->
    <constructor-arg index="1" value="FileConsumer"/>
</bean>
```

And yes then we just need to refer to the **jpaStore** bean in the file consumer endpoint using the **idempotentRepository** using the **#** syntax option:

```
<route>
  <from uri="file://inbox?idempotent=true&amp;idempotentRepository=#jpaStore"/>
  <to uri="bean:processInbox"/>
</route>
```

100.24. FILTER USING ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER

Camel supports pluggable filtering strategies. You can then configure the endpoint with such a filter to skip certain files being processed.

In the sample we have built our own filter that skips files starting with **skip** in the filename:

And then we can configure our route using the **filter** attribute to reference our filter (using **#** notation) that we have defined in the spring XML file:

```
<!-- define our filter as a plain spring bean -->
<bean id="myFilter" class="com.mycompany.MyFileFilter"/>

<route>
  <from uri="file://inbox?filter=#myFilter"/>
  <to uri="bean:processInbox"/>
</route>
```

100.25. FILTERING USING ANT PATH MATCHER

The ANT path matcher is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven.
The reasons is that we leverage Spring's AntPathMatcher to do the actual matching.

The file paths is matched with the following rules:

- **?** matches one character
- ***** matches zero or more characters
- ****** matches zero or more directories in a path

TIP

New options from Camel 2.10 onwards There are now **antInclude** and **antExclude** options to make it easy to specify ANT style include/exclude without having to define the filter. See the URI options above for more information.

The sample below demonstrates how to use it:

100.25.1. Sorting using Comparator

Camel supports pluggable sorting strategies. This strategy it to use the build in **java.util.Comparator** in Java. You can then configure the endpoint with such a comparator and have Camel sort the files before being processed.

In the sample we have built our own comparator that just sorts by file name:

And then we can configure our route using the **sorter** option to reference to our sorter (**mySorter**) we have defined in the spring XML file:

```
<!-- define our sorter as a plain spring bean -->
<bean id="mySorter" class="com.mycompany.MyFileSorter"/>

<route>
  <from uri="file://inbox?sorter=#mySorter"/>
  <to uri="bean:processInbox"/>
</route>
```

TIP

URI options can reference beans using the # syntax In the Spring DSL route above notice that we can refer to beans in the Registry by prefixing the id with **#**. So writing **sorter=#mySorter**, will instruct Camel to go look in the Registry for a bean with the ID, **mySorter**.

100.25.2. Sorting using sortBy

Camel supports pluggable sorting strategies. This strategy it to use the File Language to configure the sorting. The **sortBy** option is configured as follows:

```
sortBy=group 1;group 2;group 3;...
```

Where each group is separated with semi colon. In the simple situations you just use one group, so a simple example could be:

```
sortBy=file:name
```

This will sort by file name, you can reverse the order by prefixing **reverse:** to the group, so the sorting is now Z..A:

```
sortBy=reverse:file:name
```

As we have the full power of File Language we can use some of the other parameters, so if we want to sort by file size we do:

```
sortBy=file:length
```

You can configure to ignore the case, using **ignoreCase:** for string comparison, so if you want to use file name sorting but to ignore the case then we do:

```
sortBy=ignoreCase:file:name
```

You can combine ignore case and reverse, however reverse must be specified first:

```
sortBy=reverse:ignoreCase:file:name
```

In the sample below we want to sort by last modified file, so we do:

```
sortBy=file:modified
```

And then we want to group by name as a 2nd option so files with same modifcation is sorted by name:

```
sortBy=file:modified;file:name
```

Now there is an issue here, can you spot it? Well the modified timestamp of the file is too fine as it will be in milliseconds, but what if we want to sort by date only and then subgroup by name?
Well as we have the true power of File Language we can use its date command that supports patterns. So this can be solved as:

```
sortBy=date:file:yyyyMMdd;file:name
```

Yeah, that is pretty powerful, oh by the way you can also use reverse per group, so we could reverse the file names:

```
sortBy=date:file:yyyyMMdd;reverse:file:name
```

100.26. USING GENERICFILEPROCESSSTRATEGY

The option **processStrategy** can be used to use a custom **GenericFileProcessStrategy** that allows you to implement your own *begin*, *commit* and *rollback* logic.
For instance lets assume a system writes a file in a folder you should consume. But you should not start consuming the file before another *ready* file has been written as well.

So by implementing our own **GenericFileProcessStrategy** we can implement this as:

- In the **begin()** method we can test whether the special *ready* file exists. The begin method returns a **boolean** to indicate if we can consume the file or not.
- In the **abort()** method (Camel 2.10) special logic can be executed in case the **begin** operation returned **false**, for example to cleanup resources etc.
- in the **commit()** method we can move the actual file and also delete the *ready* file.

100.27. USING FILTER

The **filter** option allows you to implement a custom filter in Java code by implementing the **org.apache.camel.component.file.GenericFileFilter** interface. This interface has an **accept** method that returns a boolean. Return **true** to include the file, and **false** to skip the file. From Camel 2.10 onwards, there is a **isDirectory** method on **GenericFile** whether the file is a directory. This allows you to filter unwanted directories, to avoid traversing down unwanted directories.

For example to skip any directories which starts with **"skip"** in the name, can be implemented as follows:

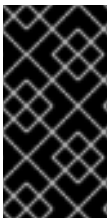
100.28. USING CONSUMER.BRIDGEERRORHANDLER

Available as of Camel 2.10

If you want to use the Camel Error Handler to deal with any exception occurring in the file consumer, then you can enable the **consumer.bridgeErrorHandler** option as shown below:

```
// to handle any IOException being thrown
onException(IOException.class)
    .handled(true)
    .log("IOException occurred due: ${exception.message}")
    .transform().simple("Error ${exception.message}")
    .to("mock:error");

// this is the file route that pickup files, notice how we bridge the consumer to use the Camel routing error handler
// the exclusiveReadLockStrategy is only configured because this is from an unit test, so we use that to simulate exceptions
from("file:target/nospace?consumer.bridgeErrorHandler=true")
    .convertBodyTo(String.class)
    .to("mock:result");
```

So all you have to do is to enable this option, and the error handler in the route will take it from there.



IMPORTANT

Important when using consumer.bridgeErrorHandler When using consumer.bridgeErrorHandler, then interceptors, OnCompletions does **not** apply. The Exchange is processed directly by the Camel Error Handler, and does not allow prior actions such as interceptors, onCompletion to take action.

100.29. DEBUG LOGGING

This component has log level **TRACE** that can be helpful if you have problems.

100.30. SEE ALSO

- File Language
- FTP
- Polling Consumer

CHAPTER 101. FILE LANGUAGE

Available as of Camel version 1.1

INFO:*File language is now merged with Simple language* From Camel 2.2 onwards, the file language is now merged with Simple language which means you can use all the file syntax directly within the simple language.

The File Expression Language is an extension to the Simple language, adding file related capabilities. These capabilities are related to common use cases working with file path and names. The goal is to allow expressions to be used with the File and FTP components for setting dynamic file patterns for both consumer and producer.

101.1. FILE LANGUAGE OPTIONS

The File language supports 2 options which are listed below.

Name	Default	Java Type	Description
resultType		**String**	Sets the class name of the result type (type from output)
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

101.2. SYNTAX

This language is an **extension** to the Simple language so the Simple syntax applies also. So the table below only lists the additional.
As opposed to Simple language File Language also supports Constant expressions so you can enter a fixed filename.

All the file tokens use the same expression name as the method on the **java.io.File** object, for instance **file:absolute** refers to the **java.io.File.getAbsolute()** method. Notice that not all expressions are supported by the current Exchange. For instance the FTP component supports some of the options, where as the File component supports all of them.

Expression	Type	File Consumer	File Producer	FTP Consumer	FTP Producer	Description
file:name	String	yes	no	yes	no	refers to the file name (is relative to the starting directory, see note below)
file:name.ext	String	yes	no	yes	no	**Camel 2.3:** refers to the file extension only

Expression	Type	File Consumer	File Producer	FTP Consumer	FTP Producer	Description
file:name.ext.single	String	yes	no	yes	no	**Camel 2.14.4/2.15.3:** refers to the file extension. If the file extension has mutiple dots, then this expression strips and only returns the last part.
file:name.noext	String	yes	no	yes	no	refers to the file name with no extension (is relative to the starting directory, see note below)
file:name.noext.single	String	yes	no	yes	no	**Camel 2.14.4/2.15.3:** refers to the file name with no extension (is relative to the starting directory, see note below). If the file extension has multiple dots, then this expression strips only the last part, and keep the others.
file:onlyname	String	yes	no	yes	no	refers to the file name only with no leading paths.
file:onlyname.noext	String	yes	no	yes	no	refers to the file name only with no extension and with no leading paths.
file:onlyname.noext.single	String	yes	no	yes	no	*Camel 2.14.4/2.15.3:*refers to the file name only with no extension and with no leading paths. If the file extension has multiple dots, then this expression strips only the last part, and keep the others.

Expression	Type	File Consumer	File Producer	FTP Consumer	FTP Producer	Description
file:ext	String	yes	no	yes	no	refers to the file extension only
file:parent	String	yes	no	yes	no	refers to the file parent
file:path	String	yes	no	yes	no	refers to the file path
file:absolute	Boolean	yes	no	no	no	refers to whether the file is regarded as absolute or relative
file:absolute.path	String	yes	no	no	no	refers to the absolute file path
file:length	Long	yes	no	yes	no	refers to the file length returned as a Long type
file:size	Long	yes	no	yes	no	**Camel 2.5:** refers to the file length returned as a Long type
file:modified	Date	yes	no	yes	no	Refers to the file last modified returned as a Date type
date:_command:pattern_	String	yes	yes	yes	yes	for date formatting using the **java.text.SimpleDateFormat** patterns. Is an **extension** to the Simple language. Additional command is: **file** (consumers only) for the last modified timestamp of the file. Notice: all the commands from the Simple language can also be used.

101.3. FILE TOKEN EXAMPLE

101.3.1. Relative paths

We have a **java.io.File** handle for the file **hello.txt** in the following **relative** directory: **.\filelanguage\test**. And we configure our endpoint to use this starting directory **.\filelanguage**. The file tokens will return as:

Expression	Returns
file:name	test\hello.txt
file:name.ext	txt
file:name.noext	test\hello
file:onlyname	hello.txt
file:onlyname.noext	hello
file:ext	txt
file:parent	filelanguage\test
file:path	filelanguage\test\hello.txt
file:absolute	false
file:absolute.path	\workspace\camel\camel-core\target\filelanguage\test\hello.txt

101.3.2. Absolute paths

We have a **java.io.File** handle for the file **hello.txt** in the following **absolute** directory: **\workspace\camel\camel-core\target\filelanguage\test**. And we configure out endpoint to use the absolute starting directory **\workspace\camel\camel-core\target\filelanguage**. The file tokens will return as:

Expression	Returns
file:name	test\hello.txt
file:name.ext	txt
file:name.noext	test\hello
file:onlyname	hello.txt
file:onlyname.noext	hello
file:ext	txt

Expression	Returns
file:parent	\workspace\camel\camel-core\target\filelanguage\test
file:path	\workspace\camel\camel-core\target\filelanguage\test\hello.txt
file:absolute	true
file:absolute.path	\workspace\camel\camel-core\target\filelanguage\test\hello.txt

101.4. SAMPLES

You can enter a fixed Constant expression such as **myfile.txt**:

```
fileName="myfile.txt"
```

Lets assume we use the file consumer to read files and want to move the read files to backup folder with the current date as a sub folder. This can be archieved using an expression like:

```
fileName="backup/${date:now:yyyyMMdd}/${file:name.noext}.bak"
```

relative folder names are also supported so suppose the backup folder should be a sibling folder then you can append .. as:

```
fileName="../backup/${date:now:yyyyMMdd}/${file:name.noext}.bak"
```

As this is an extension to the Simple language we have access to all the goodies from this language also, so in this use case we want to use the in.header.type as a parameter in the dynamic expression:

```
fileName="../backup/${date:now:yyyyMMdd}/type-${in.header.type}/backup-of-${file:name.noext}.bak"
```

If you have a custom Date you want to use in the expression then Camel supports retrieving dates from the message header.

```
fileName="orders/order-${in.header.customerId}-${date:in.header.orderDate:yyyyMMdd}.xml"
```

And finally we can also use a bean expression to invoke a POJO class that generates some String output (or convertible to String) to be used:

```
fileName="uniquefile-${bean:myguidgenerator.generateid}.txt"
```

And of course all this can be combined in one expression where you can use the File Language, Simple and the Bean language in one combined expression. This is pretty powerful for those common file path patterns.

101.5. USING SPRING PROPERTYPLACEHOLDERCONFIGURER TOGETHER WITH THE FILE COMPONENT

In Camel you can use the File Language directly from the Simple language which makes a Content Based Router easier to do in Spring XML, where we can route based on file extensions as shown below:

```
<from uri="file://input/orders"/>
  <choice>
    <when>
      <simple>${file:ext} == 'txt'</simple>
      <to uri="bean:orderService?method=handleTextFiles"/>
    </when>
    <when>
      <simple>${file:ext} == 'xml'</simple>
      <to uri="bean:orderService?method=handleXmlFiles"/>
    </when>
    <otherwise>
      <to uri="bean:orderService?method=handleOtherFiles"/>
    </otherwise>
  </choice>
```

If you use the **fileName** option on the File endpoint to set a dynamic filename using the File Language then make sure you
use the alternative syntax (available from Camel 2.5 onwards) to avoid clashing with Springs **PropertyPlaceholderConfigurer**.

bundle-context.xml

```
<bean id="propertyPlaceholder"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:bundle-context.cfg" />
</bean>

<bean id="sampleRoute" class="SampleRoute">
    <property name="fromEndpoint" value="${fromEndpoint}" />
    <property name="toEndpoint" value="${toEndpoint}" />
</bean>
```

bundle-context.cfg

```
fromEndpoint=activemq:queue:test
toEndpoint=file://fileRoute/out?fileName=test-$simple{date:now:yyyyMMdd}.txt
```

Notice how we use the $simple\{ } syntax in the **toEndpoint** above.
If you don't do this, there is a clash and Spring will throw an exception like

```
org.springframework.beans.factory.BeanDefinitionStoreException:
Invalid bean definition with name 'sampleRoute' defined in class path resource [bundle-context.xml]:
Could not resolve placeholder 'date:now:yyyyMMdd'
```

101.6. DEPENDENCIES

The File language is part of **camel-core**.

CHAPTER 102. FLATPACK COMPONENT

Available as of Camel version 1.4

The Flatpack component supports fixed width and delimited file parsing via the FlatPack library.
Notice: This component only supports consuming from flatpack files to Object model. You can not (yet) write from Object model to flatpack format.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-flatpack</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

102.1. URI FORMAT

```
flatpack:[delim|fixed]:flatPackConfig.pzmap.xml[?options]
```

Or for a delimited file handler with no configuration file just use

```
flatpack:someName[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

102.2. URI OPTIONS

The Flatpack component has no options.

The Flatpack endpoint is configured using URI syntax:

```
flatpack:type:resourceUri
```

with the following path and query parameters:

102.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
type	Whether to use fixed or delimiter	delim	FlatpackType
resourceUri	**Required** URL for loading the flatpack mapping file from classpath or file system		String

102.2.2. Query Parameters (25 parameters):

Name	Description	Default	Type
allowShortLines (common)	Allows for lines to be shorter than expected and ignores the extra characters	false	boolean
delimiter (common)	The default character delimiter for delimited files.	,	char
ignoreExtraColumns (common)	Allows for lines to be longer than expected and ignores the extra characters	false	boolean
ignoreFirstRecord (common)	Whether the first line is ignored for delimited files (for the column headers).	true	boolean
splitRows (common)	Sets the Component to send each row as a separate exchange once parsed	true	boolean
textQualifier (common)	The text qualifier for delimited files.		char
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler

Name	Description	Default	Type
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

102.3. EXAMPLES

- **flatpack:fixed:foo.pzmap.xml** creates a fixed-width endpoint using the **foo.pzmap.xml** file configuration.
- **flatpack:delim:bar.pzmap.xml** creates a delimited endpoint using the **bar.pzmap.xml** file configuration.
- **flatpack:foo** creates a delimited endpoint called **foo** with no file configuration.

102.4. MESSAGE HEADERS

Camel will store the following headers on the IN message:

Header	Description
camelFlatpackCounter	The current row index. For **splitRows=false** the counter is the total number of rows.

102.5. MESSAGE BODY

The component delivers the data in the IN message as a **org.apache.camel.component.flatpack.DataSetList** object that has converters for **java.util.Map** or **java.util.List**.
Usually you want the **Map** if you process one row at a time (**splitRows=true**). Use **List** for the entire content (**splitRows=false**), where each element in the list is a **Map**.
Each **Map** contains the key for the column name and its corresponding value.

For example to get the firstname from the sample below:

```
Map row = exchange.getIn().getBody(Map.class);
String firstName = row.get("FIRSTNAME");
```

However, you can also always get it as a **List** (even for **splitRows=true**). The same example:

```
List data = exchange.getIn().getBody(List.class);
Map row = (Map)data.get(0);
String firstName = row.get("FIRSTNAME");
```

102.6. HEADER AND TRAILER RECORDS

The header and trailer notions in Flatpack are supported. However, you **must** use fixed record IDs:

- **header** for the header record (must be lowercase)
- **trailer** for the trailer record (must be lowercase)

The example below illustrates this fact that we have a header and a trailer. You can omit one or both of them if not needed.

```
<RECORD id="header" startPosition="1" endPosition="3" indicator="HBT">
   <COLUMN name="INDICATOR" length="3"/>
   <COLUMN name="DATE" length="8"/>
</RECORD>

<COLUMN name="FIRSTNAME" length="35" />
<COLUMN name="LASTNAME" length="35" />
<COLUMN name="ADDRESS" length="100" />
<COLUMN name="CITY" length="100" />
<COLUMN name="STATE" length="2" />
<COLUMN name="ZIP" length="5" />

<RECORD id="trailer" startPosition="1" endPosition="3" indicator="FBT">
   <COLUMN name="INDICATOR" length="3"/>
   <COLUMN name="STATUS" length="7"/>
</RECORD>
```

102.7. USING THE ENDPOINT

A common use case is sending a file to this endpoint for further processing in a separate route. For example:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="file://someDirectory"/>
    <to uri="flatpack:foo"/>
  </route>

  <route>
    <from uri="flatpack:foo"/>
    ...
  </route>
</camelContext>
```

You can also convert the payload of each message created to a **Map** for easy Bean Integration

102.8. FLATPACK DATAFORMAT

The Flatpack component ships with the Flatpack data format that can be used to format between fixed width or delimited text messages to a **List** of rows as **Map**.

- marshal = from **List<Map<String, Object>>** to **OutputStream** (can be converted to **String**)
- unmarshal = from **java.io.InputStream** (such as a **File** or **String**) to a **java.util.List** as an **org.apache.camel.component.flatpack.DataSetList** instance.
  The result of the operation will contain all the data. If you need to process each row one by one you can split the exchange, using Splitter.

Notice: The Flatpack library does currently not support header and trailers for the marshal operation.

102.9. OPTIONS

The data format has the following options:

Option	Default	Description
definition	**null**	The flatpack pzmap configuration file. Can be omitted in simpler situations, but its preferred to use the pzmap.
fixed	**false**	Delimited or fixed.
ignoreFirstRecord	**true**	Whether the first line is ignored for delimited files (for the column headers).
textQualifier	**"**	If the text is qualified with a char such as **"**.
delimiter	**,**	The delimiter char (could be **;** , or similar)
parserFactory	**null**	Uses the default Flatpack parser factory.
allowShortLines	**false**	**Camel 2.9.7 and 2.10.5 onwards**: Allows for lines to be shorter than expected and ignores the extra characters.
ignoreExtraColumns	**false**	**Camel 2.9.7 and 2.10.5 onwards**: Allows for lines to be longer than expected and ignores the extra characters.

102.10. USAGE

To use the data format, simply instantiate an instance and invoke the marshal or unmarshal operation in the route builder:

```
FlatpackDataFormat fp = new FlatpackDataFormat();
fp.setDefinition(new ClassPathResource("INVENTORY-Delimited.pzmap.xml"));
...
from("file:order/in").unmarshal(df).to("seda:queue:neworder");
```

The sample above will read files from the **order/in** folder and unmarshal the input using the Flatpack configuration file **INVENTORY-Delimited.pzmap.xml** that configures the structure of the files. The result is a **DataSetList** object we store on the SEDA queue.

```
FlatpackDataFormat df = new FlatpackDataFormat();
df.setDefinition(new ClassPathResource("PEOPLE-FixedLength.pzmap.xml"));
df.setFixed(true);
df.setIgnoreFirstRecord(false);

from("seda:people").marshal(df).convertBodyTo(String.class).to("jms:queue:people");
```

In the code above we marshal the data from a Object representation as a **List** of rows as **Maps**. The rows as **Map** contains the column name as the key, and the the corresponding value. This structure can be created in Java code from e.g. a processor. We marshal the data according to the Flatpack format and convert the result as a **String** object and store it on a JMS queue.

102.11. DEPENDENCIES

To use Flatpack in your camel routes you need to add the a dependency on **camel-flatpack** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-flatpack</artifactId>
  <version>x.x.x</version>
</dependency>
```

102.12. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 103. FLATPACK DATAFORMAT

Available as of Camel version 2.1

The Flatpack component ships with the Flatpack data format that can be used to format between fixed width or delimited text messages to a **List** of rows as **Map**.

- marshal = from **List<Map<String, Object>>** to **OutputStream** (can be converted to **String**)
- unmarshal = from **java.io.InputStream** (such as a **File** or **String**) to a **java.util.List** as an **org.apache.camel.component.flatpack.DataSetList** instance.
  The result of the operation will contain all the data. If you need to process each row one by one you can split the exchange, using Splitter.

Notice: The Flatpack library does currently not support header and trailers for the marshal operation.

103.1. OPTIONS

The Flatpack dataformat supports 9 options which are listed below.

Name	Default	Java Type	Description
definition		**String**	The flatpack pzmap configuration file. Can be omitted in simpler situations, but its preferred to use the pzmap.
fixed	**false**	**Boolean**	Delimited or fixed. Is by default false = delimited
ignoreFirstRecord	**true**	**Boolean**	Whether the first line is ignored for delimited files (for the column headers). Is by default true.
textQualifier		**String**	If the text is qualified with a character. Uses quote character by default.
delimiter	**,**	**String**	The delimiter char (could be ; , or similar)
allowShortLines	**false**	**Boolean**	Allows for lines to be shorter than expected and ignores the extra characters
ignoreExtraColumns	**false**	**Boolean**	Allows for lines to be longer than expected and ignores the extra characters.
parserFactoryRef		**String**	References to a custom parser factory to lookup in the registry
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

103.2. USAGE

To use the data format, simply instantiate an instance and invoke the marshal or unmarshal operation in the route builder:

```
FlatpackDataFormat fp = new FlatpackDataFormat();
fp.setDefinition(new ClassPathResource("INVENTORY-Delimited.pzmap.xml"));
...
from("file:order/in").unmarshal(df).to("seda:queue:neworder");
```

The sample above will read files from the **order/in** folder and unmarshal the input using the Flatpack configuration file **INVENTORY-Delimited.pzmap.xml** that configures the structure of the files. The result is a **DataSetList** object we store on the SEDA queue.

```
FlatpackDataFormat df = new FlatpackDataFormat();
df.setDefinition(new ClassPathResource("PEOPLE-FixedLength.pzmap.xml"));
df.setFixed(true);
df.setIgnoreFirstRecord(false);

from("seda:people").marshal(df).convertBodyTo(String.class).to("jms:queue:people");
```

In the code above we marshal the data from a Object representation as a **List** of rows as **Maps**. The rows as **Map** contains the column name as the key, and the the corresponding value. This structure can be created in Java code from e.g. a processor. We marshal the data according to the Flatpack format and convert the result as a **String** object and store it on a JMS queue.

103.3. DEPENDENCIES

To use Flatpack in your camel routes you need to add the a dependency on **camel-flatpack** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-flatpack</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 104. APACHE FLINK COMPONENT

Available as of Camel version 2.18

This documentation page covers the Apache Flink component for the Apache Camel. The **camel-flink** component provides a bridge between Camel connectors and Flink tasks.
This Camel Flink connector provides a way to route message from various transports, dynamically choosing a flink task to execute, use incoming message as input data for the task and finally deliver the results back to the Camel pipeline.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-flink</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

104.1. URI FORMAT

Currently, the Flink Component supports only Producers. One can create DataSet, DataStream jobs.

```
flink:dataset?dataset=#myDataSet&dataSetCallback=#dataSetCallback
flink:datastream?datastream=#myDataStream&dataStreamCallback=#dataStreamCallback
```

FlinkEndpoint Options

The Apache Flink endpoint is configured using URI syntax:

```
flink:endpointType
```

with the following path and query parameters:

104.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
endpointType	**Required** Type of the endpoint (dataset, datastream).		EndpointType

104.1.2. Query Parameters (6 parameters):

Name	Description	Default	Type
collect (producer)	Indicates if results should be collected or counted.	true	boolean

Name	Description	Default	Type
dataSet (producer)	DataSet to compute against.		DataSet
dataSetCallback (producer)	Function performing action against a DataSet.		DataSetCallback
dataStream (producer)	DataStream to compute against.		DataStream
dataStreamCallback (producer)	Function performing action against a DataStream.		DataStreamCallback
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

104.2. FLINKCOMPONENT OPTIONS

The Apache Flink component supports 5 options which are listed below.

Name	Description	Default	Type
dataSet (producer)	DataSet to compute against.		DataSet
dataStream (producer)	DataStream to compute against.		DataStream
dataSetCallback (producer)	Function performing action against a DataSet.		DataSetCallback
dataStreamCallback (producer)	Function performing action against a DataStream.		DataStreamCallback
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

104.3. FLINK DATASET CALLBACK

```
@Bean
public DataSetCallback<Long> dataSetCallback() {
    return new DataSetCallback<Long>() {
```

```
        public Long onDataSet(DataSet dataSet, Object... objects) {
            try {
                dataSet.print();
                return new Long(0);
            } catch (Exception e) {
                return new Long(-1);
            }
        }
    };
}
```

104.4. FLINK DATASTREAM CALLBACK

```
@Bean
public VoidDataStreamCallback dataStreamCallback() {
    return new VoidDataStreamCallback() {
        @Override
        public void doOnDataStream(DataStream dataStream, Object... objects) throws Exception {
            dataStream.flatMap(new Splitter()).print();

            environment.execute("data stream test");
        }
    };
}
```

104.5. CAMEL-FLINK PRODUCER CALL

```
CamelContext camelContext = new SpringCamelContext(context);

String pattern = "foo";

try {
    ProducerTemplate template = camelContext.createProducerTemplate();
    camelContext.start();
    Long count = template.requestBody("flink:dataSet?
dataSet=#myDataSet&dataSetCallback=#countLinesContaining", pattern, Long.class);
    } finally {
        camelContext.stop();
    }
```

104.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 105. FOP COMPONENT

Available as of Camel version 2.10

The FOP component allows you to render a message into different output formats using Apache FOP.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-fop</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

105.1. URI FORMAT

```
fop://outputFormat?[options]
```

105.2. OUTPUT FORMATS

The primary output format is PDF but other output formats are also supported:

name	output Format	description
PDF	application/pdf	Portable Document Format
PS	application/postscript	Adobe Postscript
PCL	application/x-pcl	Printer Control Language
PNG	image/png	PNG images
JPEG	image/jpeg	JPEG images
SVG	image/svg+xml	Scalable Vector Graphics

name	outputFormat	description
XML	application/X-fop-areatree	Area tree representation
MIF	application/mif	FrameMaker's MIF
RTF	application/rtf	Rich Text Format
TXT	text/plain	Text

The complete list of valid output formats can be found here

105.3. ENDPOINT OPTIONS

The FOP component has no options.

The FOP endpoint is configured using URI syntax:

```
fop:outputType
```

with the following path and query parameters:

105.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
outputType	**Required** The primary output format is PDF but other output formats are also supported.		FopOutputType

105.3.2. Query Parameters (3 parameters):

Name	Description	Default	Type
fopFactory (producer)	Allows to use a custom configured or implementation of org.apache.fop.apps.FopFactory.		FopFactory

Name	Description	Default	Type
userConfigURL (producer)	The location of a configuration file which can be loaded from classpath or file system.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

The location of a configuration file with the following structure. From **Camel 2.12** onwards the file is loaded from the classpath by default. You can use **file:**, or **classpath:** as prefix to load the resource from file or classpath. In previous releases the file is always loaded from file system.

fopFactory

Allows to use a custom configured or implementation of **org.apache.fop.apps.FopFactory**.

105.4. MESSAGE OPERATIONS

name	default value	description
CamelFop.Output.Format		Overrides the output format for that message
CamelFop.Encrypt.userPassword		PDF user password
CamelFop.Encrypt.ownerPassword		PDF owner passoword
CamelFop.Encrypt.allowPrint	true	Allows printing the PDF

name	default value	description
CamelFop.Encrypt.allowCopyContent	true	Allows copying content of the PDF
CamelFop.Encrypt.allowEditContent	true	Allows editing content of the PDF
CamelFop.Encrypt.allowEditAnnotations	true	Allows editing annotation of the PDF
CamelFop.Render.producer	Apache FOP	Metadata element for the system/software that produces the document
CamelFop.Render.creator		Metadata element for the user that created the document
CamelFop.Render.creationDate		Creation Date
CamelFop.Render.author		Author of the content of the document
CamelFop.Render.title		Title of the document

name	default value	description
CamelFop.Render.subject		Subject of the document
CamelFop.Render.keywords		Set of keywords applicable to this document

105.5. EXAMPLE

Below is an example route that renders PDFs from xml data and xslt template and saves the PDF files in target folder:

```
from("file:source/data/xml")
    .to("xslt:xslt/template.xsl")
    .to("fop:application/pdf")
    .to("file:target/data");
```

For more information, see these resources...

105.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 106. FREEMARKER COMPONENT

Available as of Camel version 2.10

The **freemarker:** component allows for processing a message using a FreeMarker template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-freemarker</artifactId>
  <version>x.x.x</version> <!-- use the same version as your Camel core version -->
</dependency>
```

106.1. URI FORMAT

```
freemarker:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: file://folder/myfile.ftl).

You can append query options to the URI in the following format, **?option=value&option=value&...**

106.2. OPTIONS

The Freemarker component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use an existing freemarker.template.Configuration instance as the configuration.		Configuration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Freemarker endpoint is configured using URI syntax:

```
freemarker:resourceUri
```

with the following path and query parameters:

106.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

106.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
configuration (producer)	Sets the Freemarker configuration to use		Configuration
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
encoding (producer)	Sets the encoding to be used for loading the template file.		String
templateUpdateDelay (producer)	Number of seconds the loaded template resource will remain in the cache.		int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

106.3. HEADERS

Headers set during the FreeMarker evaluation are returned to the message and added as headers. This provides a mechanism for the FreeMarker component to return values to the Message.

An example: Set the header value of **fruit** in the FreeMarker template:

```
${request.setHeader('fruit', 'Apple')}
```

The header, **fruit**, is now accessible from the **message.out.headers**.

106.4. FREEMARKER CONTEXT

Camel will provide exchange information in the FreeMarker context (just a **Map**). The **Exchange** is transferred as:

key	value
exchange	The **Exchange** itself.
exchange.properties	The **Exchange** properties.
headers	The headers of the In message.
camelContext	The Camel Context.
request	The In message.
body	The In message body.
response	The Out message (only for InOut message exchange pattern).

From Camel 2.14, you can setup your custom FreeMarker context in the message header with the key "**CamelFreemarkerDataModel**" just like this

```
Map<String, Object> variableMap = new HashMap<String, Object>();
variableMap.put("headers", headersMap);
variableMap.put("body", "Monday");
variableMap.put("exchange", exchange);
exchange.getIn().setHeader("CamelFreemarkerDataModel", variableMap);
```

106.5. HOT RELOADING

The FreeMarker template resource is by default **not** hot reloadable for both file and classpath resources (expanded jar). If you set **contentCache=false**, then Camel will not cache the resource and hot reloading is thus enabled. This scenario can be used in development.

106.6. DYNAMIC TEMPLATES

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description	Support Version
FreemarkerConstants.FREEMARKER_RESOURCE	org.springframework.core.io.Resource	The template resource	⇐ 2.1

Header	Type	Description	Support Version
FreemarkerConstants.FREEMARKER_RESOURCE_URI	String	A URI for the template resource to use instead of the endpoint configured.	>= 2.1
FreemarkerConstants.FREEMARKER_TEMPLATE	String	The template to use instead of the endpoint configured.	>= 2.1

106.7. SAMPLES

For example you could use something like:

```
from("activemq:My.Queue").
  to("freemarker:com/acme/MyResponse.ftl");
```

To use a FreeMarker template to formulate a response for a message for InOut message exchanges (where there is a **JMSReplyTo** header).

If you want to use InOnly and consume the message and send it to another destination you could use:

```
from("activemq:My.Queue").
  to("freemarker:com/acme/MyResponse.ftl").
  to("activemq:Another.Queue");
```

And to disable the content cache, e.g. for development usage where the **.ftl** template should be hot reloaded:

```
from("activemq:My.Queue").
  to("freemarker:com/acme/MyResponse.ftl?contentCache=false").
  to("activemq:Another.Queue");
```

And a file-based resource:

```
from("activemq:My.Queue").
  to("freemarker:file://myfolder/MyResponse.ftl?contentCache=false").
  to("activemq:Another.Queue");
```

In **Camel 2.1** it's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").

setHeader(FreemarkerConstants.FREEMARKER_RESOURCE_URI).constant("path/to/my/template.ftl").
  to("freemarker:dummy");
```

106.8. THE EMAIL SAMPLE

In this sample we want to use FreeMarker templating for an order confirmation email. The email template is laid out in FreeMarker as:

```
Dear ${headers.lastName}, ${headers.firstName}

Thanks for the order of ${headers.item}.

Regards Camel Riders Bookstore
${body}
```

And the java code:

106.9. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 107. FTP COMPONENT

Available as of Camel version 1.1

This component provides access to remote file systems over the FTP and SFTP protocols.

When consuming from remote FTP server make sure you read the section titled *Default when consuming files* further below for details related to consuming files.

Absolute path is **not** supported. **Camel 2.16** will translate absolute path to relative by trimming all leading slashes from **directoryname**. There'll be WARN message printed in the logs.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>x.x.x</version>See the documentation of the Apache Commons
  <!-- use the same version as your Camel core version -->
</dependency>
```

107.1. URI FORMAT

```
ftp://[username@]hostname[:port]/directoryname[?options]
sftp://[username@]hostname[:port]/directoryname[?options]
ftps://[username@]hostname[:port]/directoryname[?options]
```

Where **directoryname** represents the underlying directory. The directory name is a relative path. Absolute path's is **not** supported. The relative path can contain nested folders, such as /inbox/us.

For Camel versions before **Camel 2.16**, the directoryName **must** exist already as this component does not support the **autoCreate** option (which the file component does). The reason is that its the FTP administrator (FTP server) task to properly setup user accounts, and home directories with the right file permissions etc.

For **Camel 2.16**, **autoCreate** option is supported. When consumer starts, before polling is scheduled, there's additional FTP operation performed to create the directory configured for endpoint. The default value for **autoCreate** is **true**.

If no **username** is provided, then **anonymous** login is attempted using no password.
If no **port** number is provided, Camel will provide default values according to the protocol (ftp = 21, sftp = 22, ftps = 2222).

You can append query options to the URI in the following format, **?option=value&option=value&…**

This component uses two different libraries for the actual FTP work. FTP and FTPS uses Apache Commons Net while SFTP uses JCraft JSCH.

The FTPS component is only available in Camel 2.2 or newer.
FTPS (also known as FTP Secure) is an extension to FTP that adds support for the Transport Layer Security (TLS) and the Secure Sockets Layer (SSL) cryptographic protocols.

107.2. URI OPTIONS

The options below are exclusive for the FTP component.

The FTP component has no options.

The FTP endpoint is configured using URI syntax:

```
ftp:host:port/directoryName
```

with the following path and query parameters:

107.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	**Required** Hostname of the FTP server		String
port	Port of the FTP server		int
directoryName	The starting directory		String

107.2.2. Query Parameters (108 parameters):

Name	Description	Default	Type
binary (common)	Specifies the file transfer mode, BINARY or ASCII. Default is ASCII (false).	false	boolean
charset (common)	This option is used to specify the encoding of the file. You can use this on the consumer, to specify the encodings of the files, which allow Camel to know the charset it should load the file content in case the file content is being accessed. Likewise when writing a file, you can use this option to specify which charset to write the file as well. Do mind that when writing the file Camel may have to read the message content into memory to be able to convert the data into the configured charset, so do not use this if you have big messages.		String
disconnect (common)	Whether or not to disconnect from remote FTP server right after use. Disconnect will only disconnect the current connection to the FTP server. If you have a consumer which you want to stop, then you need to stop the consumer/route instead.	false	boolean

Name	Description	Default	Type
doneFileName (common)	Producer: If provided, then Camel will write a 2nd done file when the original file has been written. The done file will be empty. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The done file will always be written in the same folder as the original file. Consumer: If provided, Camel will only consume files if a done file exists. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders.The done file is always expected in the same folder as the original file. Only $file.name and $file.name.noext is supported as dynamic placeholders.		String
fileName (common)	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the CamelFileName header. (Note: The header itself can also be an Expression). The expression options support both String and Expression types. If the expression is a String type, it is always evaluated using the File Language. If the expression is an Expression type, the specified Expression type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: mydata-$date:now:yyyyMMdd.txt. The producers support the CamelOverruleFileName header which takes precedence over any existing CamelFileName header; the CamelOverruleFileName is a header that is used only once, and makes it easier as this avoids to temporary store CamelFileName and have to restore it afterwards.		String
passiveMode (common)	Sets passive mode connections. Default is active mode connections.	false	boolean
separator (common)	Sets the path separator to be used. UNIX = Uses unix style path separator Windows = Uses windows style path separator Auto = (is default) Use existing path separator in file name	UNIX	PathSeparator
transferLoggingInterval Seconds (common)	Configures the interval in seconds to use when logging the progress of upload and download operations that are in-flight. This is used for logging progress when operations takes longer time.	5	int

Name	Description	Default	Type
transferLoggingLevel (common)	Configure the logging level to use when logging the progress of upload and download operations.	DEBUG	LoggingLevel
transferLoggingVerbose (common)	Configures whether the perform verbose (fine grained) logging of the progress of upload and download operations.	false	boolean
fastExistsCheck (common)	If set this option to be true, camel-ftp will use the list file directly to check if the file exists. Since some FTP server may not support to list the file directly, if the option is false, camel-ftp will use the old way to list the directory and check if the file exists. This option also influences readLock=changed to control whether it performs a fast check to update file information or not. This can be used to speed up the process if the FTP server has a lot of files.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delete (consumer)	If true, the file will be deleted after it is processed successfully.	false	boolean
moveFailed (consumer)	Sets the move failure expression based on Simple language. For example, to move files into a .error subdirectory use: .error. Note: When moving the files to the fail location Camel will handle the error and will not pick up the file again.		String
noop (consumer)	If true, the file is not moved or deleted in any way. This option is good for readonly data, or for ETL type requirements. If noop=true, Camel will set idempotent=true as well, to avoid consuming the same files over and over again.	false	boolean
preMove (consumer)	Expression (such as File Language) used to dynamically set the filename when moving it before processing. For example to move in-progress files into the order directory set this value to order.		String

Name	Description	Default	Type
preSort (consumer)	When pre-sort is enabled then the consumer will sort the file and directory names during polling, that was retrieved from the file system. You may want to do this in case you need to operate on the files in a sorted order. The pre-sort is executed before the consumer starts to filter, and accept files to process by Camel. This option is default=false meaning disabled.	false	boolean
recursive (consumer)	If a directory, will look for files in all the sub-directories as well.	false	boolean
resumeDownload (consumer)	Configures whether resume download is enabled. This must be supported by the FTP server (almost all FTP servers support it). In addition the options localWorkDirectory must be configured so downloaded files are stored in a local directory, and the option binary must be enabled, which is required to support resuming of downloads.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
streamDownload (consumer)	Sets the download method to use when not using a local working directory. If set to true, the remote files are streamed to the route as they are read. When set to false, the remote files are loaded into memory before being sent into the route.	false	boolean
directoryMustExist (consumer)	Similar to startingDirectoryMustExist but this applies during polling recursive sub directories.	false	boolean
download (consumer)	Whether the FTP consumer should download the file. If this option is set to false, then the message body will be null, but the consumer will still trigger a Camel Exchange that has details about the file such as file name, file size, etc. It's just that the file will not be downloaded.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
handleDirectoryParserAbsoluteResult (consumer)	Allows you to set how the consumer will handle subfolders and files in the path if the directory parser results in with absolute paths The reason for this is that some FTP servers may return file names with absolute paths, and if so then the FTP component needs to handle this by converting the returned path into a relative path.	false	boolean
ignoreFileNotFoundOrPermissionError (consumer)	Whether to ignore when (trying to list files in directories or when downloading a file), which does not exist or due to permission error. By default when a directory or file does not exists or insufficient permission, then an exception is thrown. Setting this option to true allows to ignore that instead.	false	boolean
inProgressRepository (consumer)	A pluggable in-progress repository org.apache.camel.spi.IdempotentRepository. The in-progress repository is used to account the current in progress files being consumed. By default a memory based repository is used.		String>
localWorkDirectory (consumer)	When consuming, a local work directory can be used to store the remote file content directly in local files, to avoid loading the content into memory. This is beneficial, if you consume a very big remote file and thus can conserve memory.		String
onCompletionException Handler (consumer)	To use a custom org.apache.camel.spi.ExceptionHandler to handle any thrown exceptions that happens during the file on completion process where the consumer does either a commit or rollback. The default implementation will log any exception at WARN level and ignore.		ExceptionHandler
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy

Name	Description	Default	Type
processStrategy (consumer)	A pluggable org.apache.camel.component.file.GenericFileProcessStrategy allowing you to implement your own readLock option or similar. Can also be used when special conditions must be met before a file can be consumed, such as a special ready file exists. If this option is set then the readLock option does not apply.		GenericFileProcess s Strategy<T>
receiveBufferSize (consumer)	The receive (download) buffer size Used only by FTPClient	32768	int
startingDirectory MustExist (consumer)	Whether the starting directory must exist. Mind that the autoCreate option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable autoCreate and enable this to ensure the starting directory must exist. Will thrown an exception if the directory doesn't exist.	false	boolean
useList (consumer)	Whether to allow using LIST command when downloading a file. Default is true. In some use cases you may want to download a specific file and are not allowed to use the LIST command, and therefore you can set this option to false. Notice when using this option, then the specific file to download does not include meta-data information such as file size, timestamp, permissions etc, because those information is only possible to retrieve when LIST command is in use.	true	boolean
fileExist (producer)	What to do if a file already exists with the same name. Override, which is the default, replaces the existing file. Append - adds content to the existing file. Fail - throws a GenericFileOperationException, indicating that there is already an existing file. Ignore - silently ignores the problem and does not override the existing file, but assumes everything is okay. Move - option requires to use the moveExisting option to be configured as well. The option eagerDeleteTargetFile can be used to control what to do if an moving the file, and there exists already an existing file, otherwise causing the move operation to fail. The Move option will move any existing files, before writing the target file. TryRename is only applicable if tempFileName option is in use. This allows to try renaming the file from the temporary name to the actual name, without doing any exists check. This check may be faster on some file systems and especially FTP servers.	Override	GenericFileExist

Name	Description	Default	Type
flatten (producer)	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to true on the producer enforces that any file name in CamelFileName header will be stripped for any leading paths.	false	boolean
moveExisting (producer)	Expression (such as File Language) used to compute file name to use when fileExist=Move is configured. To move files into a backup subdirectory just enter backup. This option only supports the following File Language tokens: file:name, file:name.ext, file:name.noext, file:onlyname, file:onlyname.noext, file:ext, and file:parent. Notice the file:parent is not supported by the FTP component, as the FTP component can only move any existing files to a relative directory based on current dir as base.		String
tempFileName (producer)	The same as tempPrefix option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language.		String
tempPrefix (producer)	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP when uploading big files.		String
allowNullBody (producer)	Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a GenericFileWriteException of 'Cannot write null body to file.' will be thrown. If the fileExist option is set to 'Override', then the file will be truncated, and if set to append the file will remain unchanged.	false	boolean
chmod (producer)	Allows you to set chmod on the stored file. For example chmod=640.		String
disconnectOnBatchComplete (producer)	Whether or not to disconnect from remote FTP server right after a Batch upload is complete. disconnectOnBatchComplete will only disconnect the current connection to the FTP server.	false	boolean

Name	Description	Default	Type
eagerDeleteTargetFile (producer)	Whether or not to eagerly delete any existing target file. This option only applies when you use fileExists=Override and the tempFileName option as well. You can use this to disable (set it to false) deleting the target file before the temp file is written. For example you may write big files and want the target file to exists during the temp file is being written. This ensure the target file is only deleted until the very last moment, just before the temp file is being renamed to the target filename. This option is also used to control whether to delete any existing files when fileExist=Move is enabled, and an existing file exists. If this option copyAndDeleteOnRenameFails false, then an exception will be thrown if an existing file existed, if its true, then the existing file is deleted before the move operation.	true	boolean
keepLastModified (producer)	Will keep the last modified timestamp from the source file (if any). Will use the Exchange.FILE_LAST_MODIFIED header to located the timestamp. This header can contain either a java.util.Date or long with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. Note: This option only applies to the file producer. You cannot use this option with any of the ftp producers.	false	boolean
sendNoop (producer)	Whether to send a noop command as a pre-write check before uploading files to the FTP server. This is enabled by default as a validation of the connection is still valid, which allows to silently re-connect to be able to upload the file. However if this causes problems, you can turn this option off.	true	boolean
activePortRange (advanced)	Set the client side port range in active mode. The syntax is: minPort-maxPort Both port numbers are inclusive, eg 10000-19999 to include all 1xxxx ports.		String
autoCreate (advanced)	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means the directory the files should be written to.	true	boolean
bufferSize (advanced)	Write buffer sized in bytes.	131072	int

Name	Description	Default	Type
connectTimeout (advanced)	Sets the connect timeout for waiting for a connection to be established Used by both FTPClient and JSCH	10000	int
ftpClient (advanced)	To use a custom instance of FTPClient		FTPClient
ftpClientConfig (advanced)	To use a custom instance of FTPClientConfig to configure the FTP client the endpoint should use.		FTPClientConfig
ftpClientConfigParameters (advanced)	Used by FtpComponent to provide additional parameters for the FTPClientConfig		Map
ftpClientParameters (advanced)	Used by FtpComponent to provide additional parameters for the FTPClient		Map
maximumReconnectAttempts (advanced)	Specifies the maximum reconnect attempts Camel performs when it tries to connect to the remote FTP server. Use 0 to disable this behavior.		int
reconnectDelay (advanced)	Delay in millis Camel will wait before performing a reconnect attempt.		long
siteCommand (advanced)	Sets optional site command(s) to be executed after successful login. Multiple site commands can be separated using a new line character.		String
soTimeout (advanced)	Sets the so timeout FTP and FTPS Only for Camel 2.4. SFTP for Camel 2.14.3/2.15.3/2.16 onwards. Is the SocketOptions.SO_TIMEOUT value in millis. Recommended option is to set this to 300000 so as not have a hanged connection. On SFTP this option is set as timeout on the JSCH Session instance.	300000	int
stepwise (advanced)	Sets whether we should stepwise change directories while traversing file structures when downloading files, or as well when uploading a file to a directory. You can disable this if you for example are in a situation where you cannot change directory on the FTP server due security reasons.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
throwExceptionOnConnectFailed (advanced)	Should an exception be thrown if connection failed (exhausted) By default exception is not thrown and a WARN is logged. You can use this to enable exception being thrown and handle the thrown exception from the org.apache.camel.spi.PollingConsumerPollStrategy rollback method.	false	boolean
timeout (advanced)	Sets the data timeout for waiting for reply Used only by FTPClient	30000	int
antExclude (filter)	Ant style filter exclusion. If both antInclude and antExclude are used, antExclude takes precedence over antInclude. Multiple exclusions may be specified in comma-delimited format.		String
antFilterCaseSensitive (filter)	Sets case sensitive flag on ant filter	true	boolean
antInclude (filter)	Ant style filter inclusion. Multiple inclusions may be specified in comma-delimited format.		String
eagerMaxMessagesPerPoll (filter)	Allows for controlling whether the limit from maxMessagesPerPoll is eager or not. If eager then the limit is during the scanning of files. Where as false would scan all files, and then perform sorting. Setting this option to false allows for sorting all files first, and then limit the poll. Mind that this requires a higher memory usage as all file details are in memory to perform the sorting.	true	boolean
exclude (filter)	Is used to exclude files, if filename matches the regex pattern (matching is case in-senstive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
filter (filter)	Pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. Will skip files if filter returns false in its accept() method.		GenericFileFilter<T>
filterDirectory (filter)	Filters the directory based on Simple language. For example to filter on current date, you can use a simple date pattern such as $date:now:yyyMMdd		String

Name	Description	Default	Type
filterFile (filter)	Filters the file based on Simple language. For example to filter on file size, you can use $file:size 5000		String
idempotent (filter)	Option to use the Idempotent Consumer EIP pattern to let Camel skip already processed files. Will by default use a memory based LRUCache that holds 1000 entries. If noop=true then idempotent will be enabled as well to avoid consuming the same files over and over again.	false	Boolean
idempotentKey (filter)	To use a custom idempotent key. By default the absolute path of the file is used. You can use the File Language, for example to use the file name and file size, you can do: idempotentKey=$file:name-$file:size		String
idempotentRepository (filter)	A pluggable repository org.apache.camel.spi.IdempotentRepository which by default use MemoryMessageIdRepository if none is specified and idempotent is true.		String>
include (filter)	Is used to include files, if filename matches the regex pattern (matching is case in-sensitive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
maxDepth (filter)	The maximum depth to traverse when recursively processing a directory.	2147483647	int
maxMessagesPerPoll (filter)	To define a maximum messages to gather per poll. By default no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disabled it. Notice: If this option is in use then the File and FTP components will limit before any sorting. For example if you have 100000 files and use maxMessagesPerPoll=500, then only the first 500 files will be picked up, and then sorted. You can use the eagerMaxMessagesPerPoll option and set this to false to allow to scan all files first and then sort afterwards.		int
minDepth (filter)	The minimum depth to start processing when recursively processing a directory. Using minDepth=1 means the base directory. Using minDepth=2 means the first sub directory.		int

Name	Description	Default	Type
move (filter)	Expression (such as Simple Language) used to dynamically set the filename when moving it after processing. To move files into a .done subdirectory just enter .done.		String
exclusiveReadLockStrategy (lock)	Pluggable read-lock as a org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy implementation.		GenericFileExclusiveReadLockStrategy<T>

Name	Description	Default	Type
readLock (lock)	Used by consumer, to only poll the files if it has exclusive read-lock on the file (i.e. the file is not in-progress or being written). Camel will wait until the file lock is granted. This option provides the build in strategies: none - No read lock is in use markerFile - Camel creates a marker file (fileName.camelLock) and then holds a lock on it. This option is not available for the FTP component changed - Changed is using file length/modification timestamp to detect whether the file is currently being copied or not. Will at least use 1 sec to determine this, so this option cannot consume files as fast as the others, but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option readLockCheckInterval can be used to set the check frequency. fileLock - is for using java.nio.channels.FileLock. This option is not avail for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks. rename - rename is for using a try to rename the file as a test if we can get exclusive read-lock. idempotent - (only for file component) idempotent is for using a idempotentRepository as the read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-changed - (only for file component) idempotent-changed is for using a idempotentRepository and changed as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-rename - (only for file component) idempotent-rename is for using a idempotentRepository and rename as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. Notice: The various read locks is not all suited to work in clustered mode, where concurrent consumers on different nodes is competing for the same files on a shared file system. The markerFile using a close to atomic operation to create the empty marker file, but its not guaranteed to work in a cluster. The fileLock may work better but then the file system need to support distributed file locks, and so on. Using the idempotent read lock can support clustering if the idempotent repository supports clustering, such as Hazelcast Component or Infinispan.	none	String

Name	Description	Default	Type
readLockCheckInterval (lock)	Interval in millis for the read-lock, if supported by the read lock. This interval is used for sleeping between attempts to acquire the read lock. For example when using the changed read lock, you can set a higher interval period to cater for slow writes. The default of 1 sec. may be too fast if the producer is very slow writing the file. Notice: For FTP the default readLockCheckInterval is 5000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	1000	long
readLockDeleteOrphanLock Files (lock)	Whether or not read lock with marker files should upon startup delete any orphan read lock files, which may have been left on the file system, if Camel was not properly shutdown (such as a JVM crash). If turning this option to false then any orphaned lock file will cause Camel to not attempt to pickup that file, this could also be due another node is concurrently reading files from the same shared directory.	true	boolean
readLockLoggingLevel (lock)	Logging level used when a read lock could not be acquired. By default a WARN is logged. You can change this level, for example to OFF to not have any logging. This option is only applicable for readLock of types: changed, fileLock, idempotent, idempotent-changed, idempotent-rename, rename.	DEBUG	LoggingLevel
readLockMarkerFile (lock)	Whether to use marker file with the changed, rename, or exclusive read lock types. By default a marker file is used as well to guard against other processes picking up the same files. This behavior can be turned off by setting this option to false. For example if you do not want to write marker files to the file systems by the Camel application.	true	boolean
readLockMinAge (lock)	This option applied only for readLock=change. This option allows to specify a minimum age the file must be before attempting to acquire the read lock. For example use readLockMinAge=300s to require the file is at last 5 minutes old. This can speedup the changed read lock as it will only attempt to acquire files which are at least that given age.	0	long

Name	Description	Default	Type
readLockMinLength (lock)	This option applied only for readLock=changed. This option allows you to configure a minimum file length. By default Camel expects the file to contain data, and thus the default value is 1. You can set this option to zero, to allow consuming zero-length files.	1	long
readLockRemoveOnCommit (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file is succeeded and a commit happens. By default the file is not removed which ensures that any race-condition do not occur so another active node may attempt to grab the file. Instead the idempotent repository may support eviction strategies that you can configure to evict the file name entry after X minutes - this ensures no problems with race conditions.	false	boolean
readLockRemoveOnRollback (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file failed and a rollback happens. If this option is false, then the file name entry is confirmed (as if the file did a commit).	true	boolean
readLockTimeout (lock)	Optional timeout in millis for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout triggered, then Camel will skip the file. At next poll Camel, will try the file again, and this time maybe the read-lock could be granted. Use a value of 0 or lower to indicate forever. Currently fileLock, changed and rename support the timeout. Notice: For FTP the default readLockTimeout value is 20000 instead of 10000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	10000	long
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
shuffle (sort)	To shuffle the list of files (sort in random order)	false	boolean

Name	Description	Default	Type
sortBy (sort)	Built-in sort by using the File Language. Supports nested sorts, so you can have a sort by file name and as a 2nd group sort by modified date.		String
sorter (sort)	Pluggable sorter as a java.util.Comparator class.		GenericFile<T>>
account (security)	Account to use for login		String
password (security)	Password to use for login		String
username (security)	Username to use for login		String

107.3. FTPS COMPONENT DEFAULT TRUST STORE

When using the **ftpClient.** properties related to SSL with the FTPS component, the trust store accept all certificates. If you only want trust selective certificates, you have to configure the trust store with the **ftpClient.trustStore.xxx** options or by configuring a custom **ftpClient**.

When using **sslContextParameters**, the trust store is managed by the configuration of the provided SSLContextParameters instance.

You can configure additional options on the **ftpClient** and **ftpClientConfig** from the URI directly by using the **ftpClient.** or **ftpClientConfig.** prefix.

For example to set the **setDataTimeout** on the **FTPClient** to 30 seconds you can do:

```
from("ftp://foo@myserver?password=secret&ftpClient.dataTimeout=30000").to("bean:foo");
```

You can mix and match and have use both prefixes, for example to configure date format or timezones.

```
from("ftp://foo@myserver?
password=secret&ftpClient.dataTimeout=30000&ftpClientConfig.serverLanguageCode=fr").to("bean:f
oo");
```

You can have as many of these options as you like.

See the documentation of the Apache Commons FTP FTPClientConfig for possible options and more details. And as well for Apache Commons FTP FTPClient.

If you do not like having many and long configuration in the url you can refer to the **ftpClient** or **ftpClientConfig** to use by letting Camel lookup in the Registry for it.

For example:

```
<bean id="myConfig" class="org.apache.commons.net.ftp.FTPClientConfig">
  <property name="lenientFutureDates" value="true"/>
  <property name="serverLanguageCode" value="fr"/>
```

```
    </bean>
```

And then let Camel lookup this bean when you use the # notation in the url.

```
from("ftp://foo@myserver?password=secret&ftpClientConfig=#myConfig").to("bean:foo");
```

107.4. EXAMPLES

ftp://someone@someftpserver.com/public/upload/images/holiday2008?password=secret&binary=true

ftp://someoneelse@someotherftpserver.co.uk:12049/reports/2008/password=secret&binary=false
ftp://publicftpserver.com/download

107.5. CONCURRENCY

FTP Consumer does not support concurrency

The FTP consumer (with the same endpoint) does not support concurrency (the backing FTP client is not thread safe).
You can use multiple FTP consumers to poll from different endpoints. It is only a single endpoint that does not support concurrent consumers.

The FTP producer does **not** have this issue, it supports concurrency.

107.6. MORE INFORMATION

This component is an extension of the File component. So there are more samples and details on the File component page.

107.7. DEFAULT WHEN CONSUMING FILES

The FTP consumer will by default leave the consumed files untouched on the remote FTP server. You have to configure it explicitly if you want it to delete the files or move them to another location. For example you can use **delete=true** to delete the files, or use **move=.done** to move the files into a hidden done sub directory.

The regular File consumer is different as it will by default move files to a **.camel** sub directory. The reason Camel does **not** do this by default for the FTP consumer is that it may lack permissions by default to be able to move or delete files.

107.7.1. limitations

The option **readLock** can be used to force Camel **not** to consume files that is currently in the progress of being written. However, this option is turned off by default, as it requires that the user has write access. See the options table at File2 for more details about read locks.
There are other solutions to avoid consuming files that are currently being written over FTP; for instance, you can write to a temporary destination and move the file after it has been written.

When moving files using **move** or **preMove** option the files are restricted to the FTP_ROOT folder. That prevents you from moving files outside the FTP area. If you want to move files to another area you can use soft links and move files into a soft linked folder.

107.8. MESSAGE HEADERS

The following message headers can be used to affect the behavior of the component

Header	Description
CamelFileName	Specifies the output file name (relative to the endpoint directory) to be used for the output message when sending to the endpoint. If this is not present and no expression either, then a generated message ID is used as the filename instead.
CamelFileNameProduced	The actual filepath (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users the name of the file that was written.
CamelFileIndex	Current index out of total number of files being consumed in this batch.
CamelFileSize	Total number of files being consumed in this batch.
CamelFileHost	The remote hostname.
CamelFileLocalWorkPath	Path to the local work file, if local work directory is used.

In addition the FTP/FTPS consumer and producer will enrich the Camel **Message** with the following headers

Header	Description
CamelFtpReplyCode	**Camel 2.11.1:** The FTP client reply code (the type is a integer)
CamelFtpReplyString	**Camel 2.11.1:** The FTP client reply string

107.9. ABOUT TIMEOUTS

The two set of libraries (see top) has different API for setting timeout. You can use the **connectTimeout** option for both of them to set a timeout in millis to establish a network connection. An individual **soTimeout** can also be set on the FTP/FTPS, which corresponds to using **ftpClient.soTimeout**. Notice SFTP will automatically use **connectTimeout** as its **soTimeout**. The **timeout** option only applies for FTP/FTSP as the data timeout, which corresponds to the **ftpClient.dataTimeout** value. All timeout values are in millis.

107.10. USING LOCAL WORK DIRECTORY

Camel supports consuming from remote FTP servers and downloading the files directly into a local work directory. This avoids reading the entire remote file content into memory as it is streamed directly into the local file using **FileOutputStream**.

Camel will store to a local file with the same name as the remote file, though with **.inprogress** as extension while the file is being downloaded. Afterwards, the file is renamed to remove the **.inprogress** suffix. And finally, when the Exchange is complete the local file is deleted.

So if you want to download files from a remote FTP server and store it as files then you need to route to a file endpoint such as:

```
from("ftp://someone@someserver.com?password=secret&localWorkDirectory=/tmp").to("file://inbox");
```

TIP

The route above is ultra efficient as it avoids reading the entire file content into memory. It will download the remote file directly to a local file stream. The **java.io.File** handle is then used as the Exchange body. The file producer leverages this fact and can work directly on the work file **java.io.File** handle and perform a **java.io.File.rename** to the target filename. As Camel knows it's a local work file, it can optimize and use a rename instead of a file copy, as the work file is meant to be deleted anyway.

107.11. STEPWISE CHANGING DIRECTORIES

Camel FTP can operate in two modes in terms of traversing directories when consuming files (eg downloading) or producing files (eg uploading)

- stepwise
- not stepwise

You may want to pick either one depending on your situation and security issues. Some Camel end users can only download files if they use stepwise, while others can only download if they do not. At least you have the choice to pick (from Camel 2.6 onwards).

In Camel 2.0 - 2.5 there is only one mode and it is:

- before 2.5 not stepwise
- 2.5 stepwise

From Camel 2.6 onwards there is now an option **stepwise** you can use to control the behavior.

Note that stepwise changing of directory will in most cases only work when the user is confined to it's home directory and when the home directory is reported as **"/"**.

The difference between the two of them is best illustrated with an example. Suppose we have the following directory structure on the remote FTP server we need to traverse and download files:

```
/
/one
/one/two
/one/two/sub-a
/one/two/sub-b
```

And that we have a file in each of sub-a (a.txt) and sub-b (b.txt) folder.

107.11.1. Using stepwise=true (default mode)

```
TYPE A
200 Type set to A
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,17,94
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,95
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,96
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
PORT 127,0,0,1,17,97
200 Port command successful
RETR foo.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
```

```
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,98
200 Port command successful
RETR a.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,99
200 Port command successful
RETR b.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
QUIT
221 Goodbye
disconnected.
```

As you can see when stepwise is enabled, it will traverse the directory structure using CD xxx.

107.11.2. Using stepwise=false

```
230 Logged on
TYPE A
200 Type set to A
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,4,122
200 Port command successful
LIST one/two
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,123
200 Port command successful
LIST one/two/sub-a
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,124
200 Port command successful
LIST one/two/sub-b
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,125
200 Port command successful
RETR one/two/foo.txt
```

```
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,126
200 Port command successful
RETR one/two/sub-a/a.txt
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,127
200 Port command successful
RETR one/two/sub-b/b.txt
150 Opening data channel for file transfer.
226 Transfer OK
QUIT
221 Goodbye
disconnected.
```

As you can see when not using stepwise, there are no CD operation invoked at all.

107.12. SAMPLES

In the sample below we set up Camel to download all the reports from the FTP server once every hour (60 min) as BINARY content and store it as files on the local file system.

And the route using Spring DSL:

```
<route>
  <from uri="ftp://scott@localhost/public/reports?
password=tiger&amp;binary=true&amp;delay=60000"/>
  <to uri="file://target/test-reports"/>
</route>
```

107.12.1. Consuming a remote FTPS server (implicit SSL) and client authentication

```
from("ftps://admin@localhost:2222/public/camel?
password=admin&securityProtocol=SSL&isImplicit=true
    &ftpClient.keyStore.file=./src/test/resources/server.jks
    &ftpClient.keyStore.password=password&ftpClient.keyStore.keyPassword=password")
  .to("bean:foo");
```

107.12.2. Consuming a remote FTPS server (explicit TLS) and a custom trust store configuration

```
from("ftps://admin@localhost:2222/public/camel?
password=admin&ftpClient.trustStore.file=./src/test/resources/server.jks&ftpClient.trustStore.password=
password")
  .to("bean:foo");
```

107.13. FILTER USING ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER

Camel supports pluggable filtering strategies. This strategy it to use the build in **org.apache.camel.component.file.GenericFileFilter** in Java. You can then configure the endpoint with such a filter to skip certain filters before being processed.

In the sample we have built our own filter that only accepts files starting with report in the filename.

And then we can configure our route using the **filter** attribute to reference our filter (using **#** notation) that we have defined in the spring XML file:

```
<!-- define our sorter as a plain spring bean -->
<bean id="myFilter" class="com.mycompany.MyFileFilter"/>

<route>
  <from uri="ftp://someuser@someftpserver.com?password=secret&amp;filter=#myFilter"/>
  <to uri="bean:processInbox"/>
</route>
```

107.14. FILTERING USING ANT PATH MATCHER

The ANT path matcher is a filter that is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven.
The reason is that we leverage Spring's AntPathMatcher to do the actual matching.

The file paths are matched with the following rules:

- **?** matches one character
- ***** matches zero or more characters
- ****** matches zero or more directories in a path

The sample below demonstrates how to use it:

107.15. USING A PROXY WITH SFTP

To use an HTTP proxy to connect to your remote host, you can configure your route in the following way:

```
<!-- define our sorter as a plain spring bean -->
<bean id="proxy" class="com.jcraft.jsch.ProxyHTTP">
  <constructor-arg value="localhost"/>
  <constructor-arg value="7777"/>
</bean>

<route>
  <from uri="sftp://localhost:9999/root?username=admin&password=admin&proxy=#proxy"/>
  <to uri="bean:processFile"/>
</route>
```

You can also assign a user name and password to the proxy, if necessary. Please consult the documentation for **com.jcraft.jsch.Proxy** to discover all options.

107.16. SETTING PREFERRED SFTP AUTHENTICATION METHOD

If you want to explicitly specify the list of authentication methods that should be used by **sftp** component, use **preferredAuthentications** option. If for example you would like Camel to attempt to authenticate with private/public SSH key and fallback to user/password authentication in the case when no public key is available, use the following route configuration:

```
from("sftp://localhost:9999/root?
username=admin&password=admin&preferredAuthentications=publickey,password").
  to("bean:processFile");
```

107.17. CONSUMING A SINGLE FILE USING A FIXED NAME

When you want to download a single file and knows the file name, you can use **fileName=myFileName.txt** to tell Camel the name of the file to download. By default the consumer will still do a FTP LIST command to do a directory listing and then filter these files based on the **fileName** option. Though in this use-case it may be desirable to turn off the directory listing by setting **useList=false**. For example the user account used to login to the FTP server may not have permission to do a FTP LIST command. So you can turn off this with **useList=false**, and then provide the fixed name of the file to download with **fileName=myFileName.txt**, then the FTP consumer can still download the file. If the file for some reason does not exist, then Camel will by default throw an exception, you can turn this off and ignore this by setting **ignoreFileNotFoundOrPermissionError=true**.

For example to have a Camel route that pickup a single file, and delete it after use you can do

```
from("ftp://admin@localhost:21/nolist/?
password=admin&stepwise=false&useList=false&ignoreFileNotFoundOrPermissionError=true&fileName
=report.txt&delete=true")
  .to("activemq:queue:report");
```

Notice that we have use all the options we talked above above.

You can also use this with **ConsumerTemplate**. For example to download a single file (if it exists) and grab the file content as a String type:

```
String data = template.retrieveBodyNoWait("ftp://admin@localhost:21/nolist/?
password=admin&stepwise=false&useList=false&ignoreFileNotFoundOrPermissionError=true&fileName
=report.txt&delete=true", String.class);
```

107.18. DEBUG LOGGING

This component has log level **TRACE** that can be helpful if you have problems.

107.19. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- File2

CHAPTER 108. FTPS COMPONENT

Available as of Camel version 2.2

This component provides access to remote file systems over the FTP and SFTP protocols.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

For more information you can look at FTP component

108.1. URI OPTIONS

The options below are exclusive for the FTPS component.

The FTPS component supports 2 options which are listed below.

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The FTPS endpoint is configured using URI syntax:

```
ftps:host:port/directoryName
```

with the following path and query parameters:

108.1.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	**Required** Hostname of the FTP server		String
port	Port of the FTP server		int

Name	Description	Default	Type
directoryName	The starting directory		String

108.1.2. Query Parameters (116 parameters):

Name	Description	Default	Type
binary (common)	Specifies the file transfer mode, BINARY or ASCII. Default is ASCII (false).	false	boolean
charset (common)	This option is used to specify the encoding of the file. You can use this on the consumer, to specify the encodings of the files, which allow Camel to know the charset it should load the file content in case the file content is being accessed. Likewise when writing a file, you can use this option to specify which charset to write the file as well. Do mind that when writing the file Camel may have to read the message content into memory to be able to convert the data into the configured charset, so do not use this if you have big messages.		String
disconnect (common)	Whether or not to disconnect from remote FTP server right after use. Disconnect will only disconnect the current connection to the FTP server. If you have a consumer which you want to stop, then you need to stop the consumer/route instead.	false	boolean
doneFileName (common)	Producer: If provided, then Camel will write a 2nd done file when the original file has been written. The done file will be empty. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The done file will always be written in the same folder as the original file. Consumer: If provided, Camel will only consume files if a done file exists. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders.The done file is always expected in the same folder as the original file. Only $file.name and $file.name.noext is supported as dynamic placeholders.		String

Name	Description	Default	Type
fileName (common)	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the CamelFileName header. (Note: The header itself can also be an Expression). The expression options support both String and Expression types. If the expression is a String type, it is always evaluated using the File Language. If the expression is an Expression type, the specified Expression type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: mydata-$date:now:yyyyMMdd.txt. The producers support the CamelOverruleFileName header which takes precedence over any existing CamelFileName header; the CamelOverruleFileName is a header that is used only once, and makes it easier as this avoids to temporary store CamelFileName and have to restore it afterwards.		String
passiveMode (common)	Sets passive mode connections. Default is active mode connections.	false	boolean
separator (common)	Sets the path separator to be used. UNIX = Uses unix style path separator Windows = Uses windows style path separator Auto = (is default) Use existing path separator in file name	UNIX	PathSeparator
transferLoggingInterval Seconds (common)	Configures the interval in seconds to use when logging the progress of upload and download operations that are in-flight. This is used for logging progress when operations takes longer time.	5	int
transferLoggingLevel (common)	Configure the logging level to use when logging the progress of upload and download operations.	DEBUG	LoggingLevel
transferLoggingVerbose (common)	Configures whether the perform verbose (fine grained) logging of the progress of upload and download operations.	false	boolean

Name	Description	Default	Type
fastExistsCheck (common)	If set this option to be true, camel-ftp will use the list file directly to check if the file exists. Since some FTP server may not support to list the file directly, if the option is false, camel-ftp will use the old way to list the directory and check if the file exists. This option also influences readLock=changed to control whether it performs a fast check to update file information or not. This can be used to speed up the process if the FTP server has a lot of files.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delete (consumer)	If true, the file will be deleted after it is processed successfully.	false	boolean
moveFailed (consumer)	Sets the move failure expression based on Simple language. For example, to move files into a .error subdirectory use: .error. Note: When moving the files to the fail location Camel will handle the error and will not pick up the file again.		String
noop (consumer)	If true, the file is not moved or deleted in any way. This option is good for readonly data, or for ETL type requirements. If noop=true, Camel will set idempotent=true as well, to avoid consuming the same files over and over again.	false	boolean
preMove (consumer)	Expression (such as File Language) used to dynamically set the filename when moving it before processing. For example to move in-progress files into the order directory set this value to order.		String

Name	Description	Default	Type
preSort (consumer)	When pre-sort is enabled then the consumer will sort the file and directory names during polling, that was retrieved from the file system. You may want to do this in case you need to operate on the files in a sorted order. The pre-sort is executed before the consumer starts to filter, and accept files to process by Camel. This option is default=false meaning disabled.	false	boolean
recursive (consumer)	If a directory, will look for files in all the sub-directories as well.	false	boolean
resumeDownload (consumer)	Configures whether resume download is enabled. This must be supported by the FTP server (almost all FTP servers support it). In addition the options localWorkDirectory must be configured so downloaded files are stored in a local directory, and the option binary must be enabled, which is required to support resuming of downloads.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
streamDownload (consumer)	Sets the download method to use when not using a local working directory. If set to true, the remote files are streamed to the route as they are read. When set to false, the remote files are loaded into memory before being sent into the route.	false	boolean
directoryMustExist (consumer)	Similar to startingDirectoryMustExist but this applies during polling recursive sub directories.	false	boolean
download (consumer)	Whether the FTP consumer should download the file. If this option is set to false, then the message body will be null, but the consumer will still trigger a Camel Exchange that has details about the file such as file name, file size, etc. It's just that the file will not be downloaded.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
handleDirectoryParserAbsoluteResult (consumer)	Allows you to set how the consumer will handle subfolders and files in the path if the directory parser results in with absolute paths The reason for this is that some FTP servers may return file names with absolute paths, and if so then the FTP component needs to handle this by converting the returned path into a relative path.	false	boolean
ignoreFileNotFoundOrPermissionError (consumer)	Whether to ignore when (trying to list files in directories or when downloading a file), which does not exist or due to permission error. By default when a directory or file does not exists or insufficient permission, then an exception is thrown. Setting this option to true allows to ignore that instead.	false	boolean
inProgressRepository (consumer)	A pluggable in-progress repository org.apache.camel.spi.IdempotentRepository. The in-progress repository is used to account the current in progress files being consumed. By default a memory based repository is used.		String>
localWorkDirectory (consumer)	When consuming, a local work directory can be used to store the remote file content directly in local files, to avoid loading the content into memory. This is beneficial, if you consume a very big remote file and thus can conserve memory.		String
onCompletionException Handler (consumer)	To use a custom org.apache.camel.spi.ExceptionHandler to handle any thrown exceptions that happens during the file on completion process where the consumer does either a commit or rollback. The default implementation will log any exception at WARN level and ignore.		ExceptionHandler
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy

Name	Description	Default	Type
processStrategy (consumer)	A pluggable org.apache.camel.component.file.GenericFileProcessStrategy allowing you to implement your own readLock option or similar. Can also be used when special conditions must be met before a file can be consumed, such as a special ready file exists. If this option is set then the readLock option does not apply.		GenericFileProcess s Strategy<T>
receiveBufferSize (consumer)	The receive (download) buffer size Used only by FTPClient	32768	int
startingDirectory MustExist (consumer)	Whether the starting directory must exist. Mind that the autoCreate option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable autoCreate and enable this to ensure the starting directory must exist. Will thrown an exception if the directory doesn't exist.	false	boolean
useList (consumer)	Whether to allow using LIST command when downloading a file. Default is true. In some use cases you may want to download a specific file and are not allowed to use the LIST command, and therefore you can set this option to false. Notice when using this option, then the specific file to download does not include meta-data information such as file size, timestamp, permissions etc, because those information is only possible to retrieve when LIST command is in use.	true	boolean
fileExist (producer)	What to do if a file already exists with the same name. Override, which is the default, replaces the existing file. Append - adds content to the existing file. Fail - throws a GenericFileOperationException, indicating that there is already an existing file. Ignore - silently ignores the problem and does not override the existing file, but assumes everything is okay. Move - option requires to use the moveExisting option to be configured as well. The option eagerDeleteTargetFile can be used to control what to do if an moving the file, and there exists already an existing file, otherwise causing the move operation to fail. The Move option will move any existing files, before writing the target file. TryRename is only applicable if tempFileName option is in use. This allows to try renaming the file from the temporary name to the actual name, without doing any exists check. This check may be faster on some file systems and especially FTP servers.	Override	GenericFileExist

Name	Description	Default	Type
flatten (producer)	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to true on the producer enforces that any file name in CamelFileName header will be stripped for any leading paths.	false	boolean
moveExisting (producer)	Expression (such as File Language) used to compute file name to use when fileExist=Move is configured. To move files into a backup subdirectory just enter backup. This option only supports the following File Language tokens: file:name, file:name.ext, file:name.noext, file:onlyname, file:onlyname.noext, file:ext, and file:parent. Notice the file:parent is not supported by the FTP component, as the FTP component can only move any existing files to a relative directory based on current dir as base.		String
tempFileName (producer)	The same as tempPrefix option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language.		String
tempPrefix (producer)	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP when uploading big files.		String
allowNullBody (producer)	Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a GenericFileWriteException of 'Cannot write null body to file.' will be thrown. If the fileExist option is set to 'Override', then the file will be truncated, and if set to append the file will remain unchanged.	false	boolean
chmod (producer)	Allows you to set chmod on the stored file. For example chmod=640.		String
disconnectOnBatchComplete (producer)	Whether or not to disconnect from remote FTP server right after a Batch upload is complete. disconnectOnBatchComplete will only disconnect the current connection to the FTP server.	false	boolean

Name	Description	Default	Type
eagerDeleteTargetFile (producer)	Whether or not to eagerly delete any existing target file. This option only applies when you use fileExists=Override and the tempFileName option as well. You can use this to disable (set it to false) deleting the target file before the temp file is written. For example you may write big files and want the target file to exists during the temp file is being written. This ensure the target file is only deleted until the very last moment, just before the temp file is being renamed to the target filename. This option is also used to control whether to delete any existing files when fileExist=Move is enabled, and an existing file exists. If this option copyAndDeleteOnRenameFails false, then an exception will be thrown if an existing file existed, if its true, then the existing file is deleted before the move operation.	true	boolean
keepLastModified (producer)	Will keep the last modified timestamp from the source file (if any). Will use the Exchange.FILE_LAST_MODIFIED header to located the timestamp. This header can contain either a java.util.Date or long with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. Note: This option only applies to the file producer. You cannot use this option with any of the ftp producers.	false	boolean
sendNoop (producer)	Whether to send a noop command as a pre-write check before uploading files to the FTP server. This is enabled by default as a validation of the connection is still valid, which allows to silently re-connect to be able to upload the file. However if this causes problems, you can turn this option off.	true	boolean
activePortRange (advanced)	Set the client side port range in active mode. The syntax is: minPort-maxPort Both port numbers are inclusive, eg 10000-19999 to include all 1xxxx ports.		String
autoCreate (advanced)	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means the directory the files should be written to.	true	boolean
bufferSize (advanced)	Write buffer sized in bytes.	131072	int

Name	Description	Default	Type
connectTimeout (advanced)	Sets the connect timeout for waiting for a connection to be established Used by both FTPClient and JSCH	10000	int
ftpClient (advanced)	To use a custom instance of FTPClient		FTPClient
ftpClientConfig (advanced)	To use a custom instance of FTPClientConfig to configure the FTP client the endpoint should use.		FTPClientConfig
ftpClientConfigParameters (advanced)	Used by FtpComponent to provide additional parameters for the FTPClientConfig		Map
ftpClientParameters (advanced)	Used by FtpComponent to provide additional parameters for the FTPClient		Map
maximumReconnectAttempts (advanced)	Specifies the maximum reconnect attempts Camel performs when it tries to connect to the remote FTP server. Use 0 to disable this behavior.		int
reconnectDelay (advanced)	Delay in millis Camel will wait before performing a reconnect attempt.		long
siteCommand (advanced)	Sets optional site command(s) to be executed after successful login. Multiple site commands can be separated using a new line character.		String
soTimeout (advanced)	Sets the so timeout FTP and FTPS Only for Camel 2.4. SFTP for Camel 2.14.3/2.15.3/2.16 onwards. Is the SocketOptions.SO_TIMEOUT value in millis. Recommended option is to set this to 300000 so as not have a hanged connection. On SFTP this option is set as timeout on the JSCH Session instance.	300000	int
stepwise (advanced)	Sets whether we should stepwise change directories while traversing file structures when downloading files, or as well when uploading a file to a directory. You can disable this if you for example are in a situation where you cannot change directory on the FTP server due security reasons.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
throwExceptionOnConnect Failed (advanced)	Should an exception be thrown if connection failed (exhausted) By default exception is not thrown and a WARN is logged. You can use this to enable exception being thrown and handle the thrown exception from the org.apache.camel.spi.PollingConsumerPollStrategy rollback method.	false	boolean
timeout (advanced)	Sets the data timeout for waiting for reply Used only by FTPClient	30000	int
antExclude (filter)	Ant style filter exclusion. If both antInclude and antExclude are used, antExclude takes precedence over antInclude. Multiple exclusions may be specified in comma-delimited format.		String
antFilterCaseSensitive (filter)	Sets case sensitive flag on ant filter	true	boolean
antInclude (filter)	Ant style filter inclusion. Multiple inclusions may be specified in comma-delimited format.		String
eagerMaxMessagesPerPoll (filter)	Allows for controlling whether the limit from maxMessagesPerPoll is eager or not. If eager then the limit is during the scanning of files. Where as false would scan all files, and then perform sorting. Setting this option to false allows for sorting all files first, and then limit the poll. Mind that this requires a higher memory usage as all file details are in memory to perform the sorting.	true	boolean
exclude (filter)	Is used to exclude files, if filename matches the regex pattern (matching is case in-senstive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
filter (filter)	Pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. Will skip files if filter returns false in its accept() method.		GenericFileFilter<T>
filterDirectory (filter)	Filters the directory based on Simple language. For example to filter on current date, you can use a simple date pattern such as $date:now:yyyMMdd		String

Name	Description	Default	Type
filterFile (filter)	Filters the file based on Simple language. For example to filter on file size, you can use $file:size 5000		String
idempotent (filter)	Option to use the Idempotent Consumer EIP pattern to let Camel skip already processed files. Will by default use a memory based LRUCache that holds 1000 entries. If noop=true then idempotent will be enabled as well to avoid consuming the same files over and over again.	false	Boolean
idempotentKey (filter)	To use a custom idempotent key. By default the absolute path of the file is used. You can use the File Language, for example to use the file name and file size, you can do: idempotentKey=$file:name-$file:size		String
idempotentRepository (filter)	A pluggable repository org.apache.camel.spi.IdempotentRepository which by default use MemoryMessageIdRepository if none is specified and idempotent is true.		String>
include (filter)	Is used to include files, if filename matches the regex pattern (matching is case in-sensitive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
maxDepth (filter)	The maximum depth to traverse when recursively processing a directory.	2147483647	int
maxMessagesPerPoll (filter)	To define a maximum messages to gather per poll. By default no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disabled it. Notice: If this option is in use then the File and FTP components will limit before any sorting. For example if you have 100000 files and use maxMessagesPerPoll=500, then only the first 500 files will be picked up, and then sorted. You can use the eagerMaxMessagesPerPoll option and set this to false to allow to scan all files first and then sort afterwards.		int

Name	Description	Default	Type
minDepth (filter)	The minimum depth to start processing when recursively processing a directory. Using minDepth=1 means the base directory. Using minDepth=2 means the first sub directory.		int
move (filter)	Expression (such as Simple Language) used to dynamically set the filename when moving it after processing. To move files into a .done subdirectory just enter .done.		String
exclusiveReadLockStrategy (lock)	Pluggable read-lock as a org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy implementation.		GenericFileExclusiveReadLockStrategy<T>

Name	Description	Default	Type
readLock (lock)	Used by consumer, to only poll the files if it has exclusive read-lock on the file (i.e. the file is not in-progress or being written). Camel will wait until the file lock is granted. This option provides the build in strategies: none - No read lock is in use markerFile - Camel creates a marker file (fileName.camelLock) and then holds a lock on it. This option is not available for the FTP component changed - Changed is using file length/modification timestamp to detect whether the file is currently being copied or not. Will at least use 1 sec to determine this, so this option cannot consume files as fast as the others, but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option readLockCheckInterval can be used to set the check frequency. fileLock - is for using java.nio.channels.FileLock. This option is not avail for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks. rename - rename is for using a try to rename the file as a test if we can get exclusive read-lock. idempotent - (only for file component) idempotent is for using a idempotentRepository as the read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-changed - (only for file component) idempotent-changed is for using a idempotentRepository and changed as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-rename - (only for file component) idempotent-rename is for using a idempotentRepository and rename as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. Notice: The various read locks is not all suited to work in clustered mode, where concurrent consumers on different nodes is competing for the same files on a shared file system. The markerFile using a close to atomic operation to create the empty marker file, but its not guaranteed to work in a cluster. The fileLock may work better but then the file system need to support distributed file locks, and so on. Using the idempotent read lock can support clustering if the idempotent repository supports clustering, such as Hazelcast Component or Infinispan.	none	String

Name	Description	Default	Type
readLockCheckInterval (lock)	Interval in millis for the read-lock, if supported by the read lock. This interval is used for sleeping between attempts to acquire the read lock. For example when using the changed read lock, you can set a higher interval period to cater for slow writes. The default of 1 sec. may be too fast if the producer is very slow writing the file. Notice: For FTP the default readLockCheckInterval is 5000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	1000	long
readLockDeleteOrphanLock Files (lock)	Whether or not read lock with marker files should upon startup delete any orphan read lock files, which may have been left on the file system, if Camel was not properly shutdown (such as a JVM crash). If turning this option to false then any orphaned lock file will cause Camel to not attempt to pickup that file, this could also be due another node is concurrently reading files from the same shared directory.	true	boolean
readLockLoggingLevel (lock)	Logging level used when a read lock could not be acquired. By default a WARN is logged. You can change this level, for example to OFF to not have any logging. This option is only applicable for readLock of types: changed, fileLock, idempotent, idempotent-changed, idempotent-rename, rename.	DEBUG	LoggingLevel
readLockMarkerFile (lock)	Whether to use marker file with the changed, rename, or exclusive read lock types. By default a marker file is used as well to guard against other processes picking up the same files. This behavior can be turned off by setting this option to false. For example if you do not want to write marker files to the file systems by the Camel application.	true	boolean
readLockMinAge (lock)	This option applied only for readLock=change. This option allows to specify a minimum age the file must be before attempting to acquire the read lock. For example use readLockMinAge=300s to require the file is at last 5 minutes old. This can speedup the changed read lock as it will only attempt to acquire files which are at least that given age.	0	long

Name	Description	Default	Type
readLockMinLength (lock)	This option applied only for readLock=changed. This option allows you to configure a minimum file length. By default Camel expects the file to contain data, and thus the default value is 1. You can set this option to zero, to allow consuming zero-length files.	1	long
readLockRemoveOnCommit (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file is succeeded and a commit happens. By default the file is not removed which ensures that any race-condition do not occur so another active node may attempt to grab the file. Instead the idempotent repository may support eviction strategies that you can configure to evict the file name entry after X minutes - this ensures no problems with race conditions.	false	boolean
readLockRemoveOnRollback (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file failed and a rollback happens. If this option is false, then the file name entry is confirmed (as if the file did a commit).	true	boolean
readLockTimeout (lock)	Optional timeout in millis for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout triggered, then Camel will skip the file. At next poll Camel, will try the file again, and this time maybe the read-lock could be granted. Use a value of 0 or lower to indicate forever. Currently fileLock, changed and rename support the timeout. Notice: For FTP the default readLockTimeout value is 20000 instead of 10000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	10000	long
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
shuffle (sort)	To shuffle the list of files (sort in random order)	false	boolean

Name	Description	Default	Type
sortBy (sort)	Built-in sort by using the File Language. Supports nested sorts, so you can have a sort by file name and as a 2nd group sort by modified date.		String
sorter (sort)	Pluggable sorter as a java.util.Comparator class.		GenericFile<T>>
account (security)	Account to use for login		String
disableSecureDataChannelDefaults (security)	Use this option to disable default options when using secure data channel. This allows you to be in full control what the execPbsz and execProt setting should be used. Default is false	false	boolean
execPbsz (security)	When using secure data channel you can set the exec protection buffer size		Long
execProt (security)	The exec protection level PROT command. C - Clear S - Safe(SSL protocol only) E - Confidential(SSL protocol only) P - Private		String
ftpClientKeyStore Parameters (security)	Set the key store parameters		Map
ftpClientTrustStore Parameters (security)	Set the trust store parameters		Map
isImplicit (security)	Set the security mode(Implicit/Explicit). true - Implicit Mode / False - Explicit Mode	false	boolean
password (security)	Password to use for login		String
securityProtocol (security)	Set the underlying security protocol.	TLS	String
sslContextParameters (security)	Gets the JSSE configuration that overrides any settings in link FtpsEndpointftpClientKeyStoreParameters, link ftpClientTrustStoreParameters, and link FtpsConfigurationgetSecurityProtocol().		SSLContextParameters
username (security)	Username to use for login		String

CHAPTER 109. GANGLIA COMPONENT

Available as of Camel version 2.15

Provides a mechanism to send a value (the message body) as a metric to the Ganglia monitoring system. Uses the gmetric4j library. Can be used in conjunction with standard Ganglia and JMXetric for monitoring metrics from the OS, JVM and business processes through a single platform.

You should have a Ganglia gmond agent running on the machine where your JVM runs. The gmond sends a heartbeat to the Ganglia infrastructure, camel-ganglia can't send the heartbeat itself currently.

On most Linux systems (Debian, Ubuntu, Fedora and RHEL/CentOS with EPEL) you can just install the Ganglia agent package and it runs automatically using multicast configuration. You can configure it to use regular UDP unicast if you prefer.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ganglia</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

109.1. URI FORMAT

```
ganglia:address:port[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

109.2. GANGLIA COMPONENT AND ENDPOINT URI OPTIONS

The Ganglia component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared configuration		GangliaConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ganglia endpoint is configured using URI syntax:

```
ganglia:host:port
```

with the following path and query parameters:

109.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	Host name for Ganglia server	239.2.11.71	String
port	Port for Ganglia server	8649	int

109.2.2. Query Parameters (13 parameters):

Name	Description	Default	Type
dmax (producer)	Minumum time in seconds before Ganglia will purge the metric value if it expires. Set to 0 and the value will remain in Ganglia indefinitely until a gmond agent restart.	0	int
groupName (producer)	The group that the metric belongs to.	java	String
metricName (producer)	The name to use for the metric.	metric	String
mode (producer)	Send the UDP metric packets using MULTICAST or UNICAST	MULTICAST	UDPAddressingMode
prefix (producer)	Prefix the metric name with this string and an underscore.		String
slope (producer)	The slope	BOTH	GMetricSlope
spoofHostname (producer)	Spoofing information IP:hostname		String
tmax (producer)	Maximum time in seconds that the value can be considered current. After this, Ganglia considers the value to have expired.	60	int
ttl (producer)	If using multicast, set the TTL of the packets	5	int
type (producer)	The type of value	STRING	GMetricType

Name	Description	Default	Type
units (producer)	Any unit of measurement that qualifies the metric, e.g. widgets, litres, bytes. Do not include a prefix such as k (kilo) or m (milli), other tools may scale the units later. The value should be unscaled.		String
wireFormat31x (producer)	Use the wire format of Ganglia 3.1.0 and later versions. Set this to false to use Ganglia 3.0.x or earlier.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

109.3. MESSAGE BODY

Any value (such as a string or numeric type) in the body is sent to the Ganglia system.

109.4. RETURN VALUE / RESPONSE

Ganglia sends metrics using unidirectional UDP or multicast. There is no response or change to the message body.

109.5. EXAMPLES

109.5.1. Sending a String metric

The message body will be converted to a String and sent as a metric value. Unlike numeric metrics, String values can't be charted but Ganglia makes them available for reporting. The os_version string at the top of every Ganglia host page is an example of a String metric.

```
from("direct:string.for.ganglia")
    .setHeader(GangliaConstants.METRIC_NAME, simple("my_string_metric"))
    .setHeader(GangliaConstants.METRIC_TYPE, GMetricType.STRING)
    .to("direct:ganglia.tx");

from("direct:ganglia.tx")
    .to("ganglia:239.2.11.71:8649?mode=MULTICAST&prefix=test");
```

109.5.2. Sending a numeric metric

```
from("direct:value.for.ganglia")
    .setHeader(GangliaConstants.METRIC_NAME, simple("widgets_in_stock"))
    .setHeader(GangliaConstants.METRIC_TYPE, GMetricType.UINT32)
    .setHeader(GangliaConstants.METRIC_UNITS, simple("widgets"))
    .to("direct:ganglia.tx");
```

```
from("direct:ganglia.tx")
    .to("ganglia:239.2.11.71:8649?mode=MULTICAST&prefix=test");
```

CHAPTER 110. GEOCODER COMPONENT

Available as of Camel version 2.12

The **geocoder:** component is used for looking up geocodes (latitude and longitude) for a given address, or reverse lookup. The component uses the Java API for Google Geocoder library.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-geocoder</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

110.1. URI FORMAT

```
geocoder:address:name[?options]
geocoder:latlng:latitude,longitude[?options]
```

110.2. OPTIONS

The Geocoder component has no options.

The Geocoder endpoint is configured using URI syntax:

```
geocoder:address:latlng
```

with the following path and query parameters:

110.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
address	The geo address which should be prefixed with address:		String
latlng	The geo latitude and longitude which should be prefixed with latlng:		String

110.2.2. Query Parameters (14 parameters):

Name	Description	Default	Type
clientId (producer)	To use google premium with this client id		String

Name	Description	Default	Type
clientKey (producer)	To use google premium with this client key		String
headersOnly (producer)	Whether to only enrich the Exchange with headers, and leave the body as-is.	false	boolean
language (producer)	The language to use.	en	String
httpClientConfigurer (advanced)	Register a custom configuration strategy for new HttpClient instances created by producers or consumers such as to configure authentication mechanisms etc		HttpClientConfigurer
httpConnectionManager (advanced)	To use a custom HttpConnectionManager to manage connections		HttpConnectionManager
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
proxyAuthDomain (proxy)	Domain for proxy NTML authentication		String
proxyAuthHost (proxy)	Optional host for proxy NTML authentication		String
proxyAuthMethod (proxy)	Authentication method for proxy, either as Basic, Digest or NTLM.		String
proxyAuthPassword (proxy)	Password for proxy authentication		String
proxyAuthUsername (proxy)	Username for proxy authentication		String
proxyHost (proxy)	The proxy host name		String
proxyPort (proxy)	The proxy port number		Integer

110.3. EXCHANGE DATA FORMAT

Camel will deliver the body as a **com.google.code.geocoder.model.GeocodeResponse** type.
And if the address is **"current"** then the response is a String type with a JSON representation of the current location.

If the option **headersOnly** is set to **true** then the message body is left as-is, and only headers will be added to the Exchange.

110.4. MESSAGE HEADERS

Header	Description
CamelGeoCoderStatus	Mandatory. Status code from the geocoder library. If status is **GeocoderStatus.OK** then additional headers is enriched
CamelGeoCoderAddress	The formatted address
CamelGeoCoderLat	The latitude of the location.
CamelGeoCoderLng	The longitude of the location.
CamelGeoCoderLatlng	The latitude and longitude of the location. Separated by comma.
CamelGeoCoderCity	The city long name.
CamelGeoCoderRegionCode	The region code.
CamelGeoCoderRegionName	The region name.
CamelGeoCoderCountryLong	The country long name.
CamelGeoCoderCountryShort	The country short name.

Notice not all headers may be provided depending on available data and mode in use (address vs latlng).

110.5. SAMPLES

In the example below we get the latitude and longitude for Paris, France

```
from("direct:start")
  .to("geocoder:address:Paris, France")
```

If you provide a header with the **CamelGeoCoderAddress** then that overrides the endpoint configuration, so to get the location of Copenhagen, Denmark we can send a message with a headers as shown:

```
template.sendBodyAndHeader("direct:start", "Hello", GeoCoderConstants.ADDRESS, "Copenhagen, Denmark");
```

To get the address for a latitude and longitude we can do:

```
  from("direct:start")
    .to("geocoder:latlng:40.714224,-73.961452")
    .log("Location ${header.CamelGeocoderAddress} is at lat/lng: ${header.CamelGeocoderLatlng}
and in country ${header.CamelGeoCoderCountryShort}")
```

Which will log

```
Location 285 Bedford Avenue, Brooklyn, NY 11211, USA is at lat/lng: 40.71412890,-73.96140740
and in country US
```

To get the current location you can use "current" as the address as shown:

```
  from("direct:start")
    .to("geocoder:address:current")
```

CHAPTER 111. GIT COMPONENT

Available as of Camel version 2.16

The **git:** component allows you to work with a generic Git repository.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-git</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI Format

```
git://localRepositoryPath[?options]
```

111.1. URI OPTIONS

The producer allows to do operations on a specific repository.
The consumer allows consuming commits, tags and branches on a specific repository.

The Git component has no options.

The Git endpoint is configured using URI syntax:

```
git:localPath
```

with the following path and query parameters:

111.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
localPath	**Required** Local repository path		String

111.1.2. Query Parameters (13 parameters):

Name	Description	Default	Type
branchName (common)	The branch name to work on		String
password (common)	Remote repository password		String

Name	Description	Default	Type
remoteName (common)	The remote repository name to use in particular operation like pull		String
remotePath (common)	The remote repository path		String
tagName (common)	The tag name to work on		String
username (common)	Remote repository username		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
type (consumer)	The consumer type		GitType
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
allowEmpty (producer)	The flag to manage empty git commits	true	boolean
operation (producer)	The operation to do on the repository		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

111.2. MESSAGE HEADERS

Name	Default Value	Type	Context	Description
CamelGitOperation	**null**	String	Producer	The operation to do on a repository, if not specified as endpoint option
CamelGitFilename	**null**	String	Producer	The file name in an add operation
CamelGitCommitMessage	**null**	String	Producer	The commit message related in a commit operation
CamelGitCommitUsername	**null**	String	Producer	The commit username in a commit operation
CamelGitCommitEmail	**null**	String	Producer	The commit email in a commit operation
CamelGitCommitId	**null**	String	Producer	The commit id
CamelGitAllowEmpty	**null**	Boolean	Producer	The flag to manage empty git commits

111.3. PRODUCER EXAMPLE

Below is an example route of a producer that add a file test.java to a local repository, commit it with a specific message on master branch and then push it to remote repository.

```
from("direct:start")
        .setHeader(GitConstants.GIT_FILE_NAME, constant("test.java"))
        .to("git:///tmp/testRepo?operation=add")
        .setHeader(GitConstants.GIT_COMMIT_MESSAGE, constant("first commit"))
        .to("git:///tmp/testRepo?operation=commit")
        .to("git:///tmp/testRepo?
operation=push&remotePath=https://foo.com/test/test.git&username=xxx&password=xxx")
```

111.4. CONSUMER EXAMPLE

Below is an example route of a consumer that consumes commit:

```
from("git:///tmp/testRepo?type=commit")
                        .to(....)
```

CHAPTER 112. GITHUB COMPONENT

Available as of Camel version 2.15

The GitHub component interacts with the GitHub API by encapsulating egit-github. It currently provides polling for new pull requests, pull request comments, tags, and commits. It is also able to produce comments on pull requests, as well as close the pull request entirely.

Rather than webhooks, this endpoint relies on simple polling. Reasons include:

- Concern for reliability/stability
- The types of payloads we're polling aren't typically large (plus, paging is available in the API)
- The need to support apps running somewhere not publicly accessible where a webhook would fail

Note that the GitHub API is fairly expansive. Therefore, this component could be easily expanded to provide additional interactions.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-github</artifactId>
  <version>${camel-version}</version>
</dependency>
```

112.1. URI FORMAT

```
github://endpoint[?options]
```

112.2. MANDATORY OPTIONS:

Note that these can be configured directly through the endpoint.

The GitHub component has no options.

The GitHub endpoint is configured using URI syntax:

```
github:type/branchName
```

with the following path and query parameters:

112.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
type	**Required** What git operation to execute		GitHubType

Name	Description	Default	Type
branchName	Name of branch		String

112.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type
oauthToken (common)	GitHub OAuth token, required unless username & password are provided		String
password (common)	GitHub password, required unless oauthToken is provided		String
repoName (common)	**Required** GitHub repository name		String
repoOwner (common)	**Required** GitHub repository owner (organization)		String
username (common)	GitHub username, required unless oauthToken is provided		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
encoding (producer)	To use the given encoding when getting a git commit file		String
state (producer)	To set git commit status state		String

Name	Description	Default	Type
targetUrl (producer)	To set git commit status target url		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

112.3. CONSUMER ENDPOINTS:

Endpoint	Context	Body Type
pullRequest	polling	org.eclipse.egit.github.core.PullRequest
pullRequestComment	polling	org.eclipse.egit.github.core.Comment (comment on the general pull request discussion) or org.eclipse.egit.github.core.CommitComment (inline comment on a pull request diff)
tag	polling	org.eclipse.egit.github.core.RepositoryTag
commit	polling	org.eclipse.egit.github.core.RepositoryCommit

112.4. PRODUCER ENDPOINTS:

Endpoint	Body	Message Headers
pullRequestComment	String (comment text)	- GitHubPullRequest (integer) (REQUIRED): Pull request number. - GitHubInResponseTo (integer): Required if responding to another inline comment on the pull request diff. If left off, a general comment on the pull request discussion is assumed.
closePullRequest	none	- GitHubPullRequest (integer) (REQUIRED): Pull request number.
createIssue (From Camel 2.18)	String (issue body text)	- GitHubIssueTitle (String) (REQUIRED): Issue Title.

CHAPTER 113. GZIP DATAFORMAT

Available as of Camel version 2.0

The GZip Data Format is a message compression and de-compression format. It uses the same deflate algorithm that is used in Zip DataFormat, although some additional headers are provided. This format is produced by popular **gzip**/**gunzip** tool. Messages marshalled using GZip compression can be unmarshalled using GZip decompression just prior to being consumed at the endpoint. The compression capability is quite useful when you deal with large XML and Text based payloads or when you read messages previously comressed using **gzip** tool.

113.1. OPTIONS

The GZip dataformat supports 1 options which are listed below.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

113.2. MARSHAL

In this example we marshal a regular text/XML payload to a compressed payload employing gzip compression format and send it an ActiveMQ queue called MY_QUEUE.

```
from("direct:start").marshal().gzip().to("activemq:queue:MY_QUEUE");
```

113.3. UNMARSHAL

In this example we unmarshal a gzipped payload from an ActiveMQ queue called MY_QUEUE to its original format, and forward it for processing to the **UnGZippedMessageProcessor**.

```
from("activemq:queue:MY_QUEUE").unmarshal().gzip().process(new
UnGZippedMessageProcessor());
```

113.4. DEPENDENCIES

This data format is provided in **camel-core** so no additional dependencies is needed.

CHAPTER 114. GOOGLE BIGQUERY COMPONENT

Available as of Camel version 2.20

114.1. COMPONENT DESCRIPTION

The Google Bigquery component provides access to Cloud BigQuery Infrastructure via the Google Client Services API.

The current implementation does not use gRPC.

The current implementation does not support querying BigQuery i.e. is a producer only.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-google-bigquery</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

114.2. AUTHENTICATION CONFIGURATION

Google BigQuery component authentication is targeted for use with the GCP Service Accounts. For more information please refer to Google Cloud Platform Auth Guide

Google security credentials can be set explicitly via one of the two options:

- Service Account Email and Service Account Key (PEM format)
- GCP credentials file location

If both are set, the Service Account Email/Key will take precedence.

Or implicitly, where the connection factory falls back on Application Default Credentials.

OBS! The location of the default credentials file is configurable - via GOOGLE_APPLICATION_CREDENTIALS environment variable.

Service Account Email and Service Account Key can be found in the GCP JSON credentials file as client_email and private_key respectively.

114.3. URI FORMAT

```
google-bigquery://project-id:datasetId[:tableId]?[options]
```

114.4. OPTIONS

The Google BigQuery component supports 4 options which are listed below.

Name	Description	Default	Type
projectId (producer)	Google Cloud Project Id		String
datasetId (producer)	BigQuery Dataset Id		String
connectionFactory (producer)	ConnectionFactory to obtain connection to Bigquery Service. If non provided the default one will be used		GoogleBigQueryConnectionFactory
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Google BigQuery endpoint is configured using URI syntax:

```
google-bigquery:projectId:datasetId:tableName
```

with the following path and query parameters:

114.4.1. Path Parameters (3 parameters):

Name	Description	Default	Type
projectId	**Required** Google Cloud Project Id		String
datasetId	**Required** BigQuery Dataset Id		String
tableId	BigQuery table id		String

114.4.2. Query Parameters (3 parameters):

Name	Description	Default	Type
connectionFactory (producer)	ConnectionFactory to obtain connection to Bigquery Service. If non provided the default will be used.		GoogleBigQueryConnectionFactory
useAsInsertId (producer)	Field name to use as insert id		String

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

114.5. MESSAGE HEADERS

Name	Type	Description
CamelGoogleBigQuery.TableSuffix	**String**	Table suffix to use when inserting data
CamelGoogleBigQuery.InsertId	**String**	InsertId to use when inserting data
CamelGoogleBigQuery.PartitionDecorator	**String**	Partition decorator to indicate partition to use when inserting data
CamelGoogleBigQuery.TableId	**String**	Table id where data will be submitted. If specified will override endpoint configuration

114.6. PRODUCER ENDPOINTS

Producer endpoints can accept and deliver to BigQuery individual and grouped exchanges alike. Grouped exchanges have **Exchange.GROUPED_EXCHANGE** property set.

Goole BigQuery producer will send a grouped exchange in a single api call unless different table suffix or partition decorators are specified in which case it will break it down to ensure data is written with the correct suffix or partition decorator.

Google BigQuery endpoint expects the payload to be either a map or list of maps. A payload containing a map will insert a single row and a payload containing a list of map's will insert a row for each entry in the list.

114.7. TEMPLATE TABLES

Reference: https://cloud.google.com/bigquery/streaming-data-into-bigquery#template-tables

Templated tables can be specified using the **GoogleBigQueryConstants.TABLE_SUFFIX** header.

I.e. the following route will create tables and insert records sharded on a per day basis:

```
from("direct:start")
.header(GoogleBigQueryConstants.TABLE_SUFFIX, "_${date:now:yyyyMMdd}")
.to("google-bigquery:sampleDataset:sampleTable")
```

Note it is recommended to use partitioning for this use case.

114.8. PARTITIONING

Reference: https://cloud.google.com/bigquery/docs/creating-partitioned-tables

Partitioning is specified when creating a table and if set data will be automatically partitioned into separate tables. When inserting data a specific partition can be specified by setting the **GoogleBigQueryConstants.PARTITION_DECORATOR** header on the exchange.

114.9. ENSURING DATA CONSISTENCY

Reference: https://cloud.google.com/bigquery/streaming-data-into-bigquery#dataconsistency

A insert id can be set on the exchange with the header **GoogleBigQueryConstants.INSERT_ID** or by specifying query parameter **useAsInsertId**. As an insert id need to be specified per row inserted the exchange header can't be used when the payload is a list - if the payload is a list the **GoogleBigQueryConstants.INSERT_ID** will be ignored. In that case use the query parameter **useAsInsertId**.

CHAPTER 115. GOOGLE CALENDAR COMPONENT

Available as of Camel version 2.15

The Google Calendar component provides access to Google Calendar via the Google Calendar Web APIs.

Google Calendar uses the OAuth 2.0 protocol for authenticating a Google account and authorizing access to user data. Before you can use this component, you will need to create an account and generate OAuth credentials. Credentials comprise of a clientId, clientSecret, and a refreshToken. A handy resource for generating a long-lived refreshToken is the OAuth playground.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-google-calendar</artifactId>
    <version>2.15.0</version>
</dependency>
```

115.1. 1. GOOGLE CALENDAR OPTIONS

The Google Calendar component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		GoogleCalendarConfiguration
clientFactory (advanced)	To use the GoogleCalendarClientFactory as factory for creating the client. Will by default use BatchGoogleCalendarClientFactory		GoogleCalendarClient Factory
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Google Calendar endpoint is configured using URI syntax:

```
google-calendar:apiName/methodName
```

with the following path and query parameters:

115.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		GoogleCalendarApiName
methodName	**Required** What sub operation to use for the selected operation		String

115.1.2. Query Parameters (14 parameters):

Name	Description	Default	Type
accessToken (common)	OAuth 2 access token. This typically expires after an hour so refreshToken is recommended for long term usage.		String
applicationName (common)	Google calendar application name. Example would be camel-google-calendar/1.0		String
clientId (common)	Client ID of the calendar application		String
clientSecret (common)	Client secret of the calendar application		String
emailAddress (common)	The emailAddress of the Google Service Account.		String
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
p12FileName (common)	The name of the p12 file which has the private key to use with the Google Service Account.		String
refreshToken (common)	OAuth 2 refresh token. Using this, the Google Calendar component can obtain a new accessToken whenever the current one expires - a necessity if the application is long-lived.		String
scopes (common)	Specifies the level of permissions you want a calendar application to have to a user account. You can separate multiple scopes by comma. See https://developers.google.com/google-apps/calendar/auth for more info.	https://www.googleapis.com/auth/calendar	String

Name	Description	Default	Type
user (common)	The email address of the user the application is trying to impersonate in the service account flow		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

115.2. URI FORMAT

The GoogleCalendar Component uses the following URI format:

```
google-calendar://endpoint-prefix/endpoint?[options]
```

Endpoint prefix can be one of:

- acl
- calendars
- channels
- colors
- events
- freebusy
- list
- settings

115.3. PRODUCER ENDPOINTS

Producer endpoints can use endpoint prefixes followed by endpoint names and associated options described next. A shorthand alias can be used for some endpoints. The endpoint URI MUST contain a prefix.

Endpoint options that are not mandatory are denoted by []. When there are no mandatory options for an endpoint, one of the set of [] options MUST be provided. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelGoogleCalendar.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelGoogleCalendar.option** header.

115.4. CONSUMER ENDPOINTS

Any of the producer endpoints can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. Consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange.

115.5. MESSAGE HEADERS

Any URI option can be provided in a message header for producer endpoints with a **CamelGoogleCalendar.** prefix.

115.6. MESSAGE BODY

All result message bodies utilize objects provided by the underlying APIs used by the GoogleCalendarComponent. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint URI parameter. For endpoints that return an array or collection, a consumer endpoint will map every element to distinct messages.

CHAPTER 116. GOOGLE DRIVE COMPONENT

Available as of Camel version 2.14

The Google Drive component provides access to the Google Drive file storage service via the Google Drive Web APIs.

Google Drive uses the OAuth 2.0 protocol for authenticating a Google account and authorizing access to user data. Before you can use this component, you will need to create an account and generate OAuth credentials. Credentials comprise of a clientId, clientSecret, and a refreshToken. A handy resource for generating a long-lived refreshToken is the OAuth playground.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-google-drive</artifactId>
        <version>2.14-SNAPSHOT</version>
</dependency>
```

116.1. URI FORMAT

The GoogleDrive Component uses the following URI format:

```
google-drive://endpoint-prefix/endpoint?[options]
```

Endpoint prefix can be one of:

- drive-about
- drive-apps
- drive-changes
- drive-channels
- drive-children
- drive-comments
- drive-files
- drive-parents
- drive-permissions
- drive-properties
- drive-realtime
- drive-replies
- drive-revisions

116.2. GOOGLEDRIVECOMPONENT

The Google Drive component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		GoogleDrive Configuration
clientFactory (advanced)	To use the GoogleCalendarClientFactory as factory for creating the client. Will by default use BatchGoogleDriveClientFactory		GoogleDriveClient Factory
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Google Drive endpoint is configured using URI syntax:

```
google-drive:apiName/methodName
```

with the following path and query parameters:

116.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		GoogleDriveApiName
methodName	**Required** What sub operation to use for the selected operation		String

116.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type
accessToken (common)	OAuth 2 access token. This typically expires after an hour so refreshToken is recommended for long term usage.		String
applicationName (common)	Google drive application name. Example would be camel-google-drive/1.0		String

Name	Description	Default	Type
clientFactory (common)	To use the GoogleCalendarClientFactory as factory for creating the client. Will by default use BatchGoogleDriveClientFactory		GoogleDriveClientFactory
clientId (common)	Client ID of the drive application		String
clientSecret (common)	Client secret of the drive application		String
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
refreshToken (common)	OAuth 2 refresh token. Using this, the Google Calendar component can obtain a new accessToken whenever the current one expires - a necessity if the application is long-lived.		String
scopes (common)	Specifies the level of permissions you want a drive application to have to a user account. See https://developers.google.com/drive/web/scopes for more info.		List
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

116.3. PRODUCER ENDPOINTS

Producer endpoints can use endpoint prefixes followed by endpoint names and associated options described next. A shorthand alias can be used for some endpoints. The endpoint URI MUST contain a prefix.

Endpoint options that are not mandatory are denoted by []. When there are no mandatory options for an endpoint, one of the set of [] options MUST be provided. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelGoogleDrive.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelGoogleDrive.option** header.

For more information on the endpoints and options see API documentation at: https://developers.google.com/drive/v2/reference/

116.4. CONSUMER ENDPOINTS

Any of the producer endpoints can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. Consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange.

116.5. MESSAGE HEADERS

Any URI option can be provided in a message header for producer endpoints with a **CamelGoogleDrive.** prefix.

116.6. MESSAGE BODY

All result message bodies utilize objects provided by the underlying APIs used by the GoogleDriveComponent. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint URI parameter. For endpoints that return an array or collection, a consumer endpoint will map every element to distinct messages.

CHAPTER 117. GOOGLE MAIL COMPONENT

Available as of Camel version 2.15

The Google Mail component provides access to Gmail via the Google Mail Web APIs.

Google Mail uses the OAuth 2.0 protocol for authenticating a Google account and authorizing access to user data. Before you can use this component, you will need to create an account and generate OAuth credentials. Credentials comprise of a clientId, clientSecret, and a refreshToken. A handy resource for generating a long-lived refreshToken is the OAuth playground.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-google-mail</artifactId>
        <version>2.15-SNAPSHOT</version>
</dependency>
```

117.1. URI FORMAT

The GoogleMail Component uses the following URI format:

```
google-mail://endpoint-prefix/endpoint?[options]
```

Endpoint prefix can be one of:

- attachments
- drafts
- history
- labels
- messages
- threads
- users

117.2. GOOGLEMAILCOMPONENT

The Google Mail component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		GoogleMailConfiguration

Name	Description	Default	Type
clientFactory (advanced)	To use the GoogleCalendarClientFactory as factory for creating the client. Will by default use BatchGoogleMailClientFactory		GoogleMailClientFactory
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Google Mail endpoint is configured using URI syntax:

```
google-mail:apiName/methodName
```

with the following path and query parameters:

117.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		GoogleMailApiName
methodName	**Required** What sub operation to use for the selected operation		String

117.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
accessToken (common)	OAuth 2 access token. This typically expires after an hour so refreshToken is recommended for long term usage.		String
applicationName (common)	Google mail application name. Example would be camel-google-mail/1.0		String
clientId (common)	Client ID of the mail application		String
clientSecret (common)	Client secret of the mail application		String

Name	Description	Default	Type
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
refreshToken (common)	OAuth 2 refresh token. Using this, the Google Calendar component can obtain a new accessToken whenever the current one expires - a necessity if the application is long-lived.		String
scopes (common)	Specifies the level of permissions you want a mail application to have to a user account. See https://developers.google.com/gmail/api/auth/scopes for more info.		List
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

117.3. PRODUCER ENDPOINTS

Producer endpoints can use endpoint prefixes followed by endpoint names and associated options described next. A shorthand alias can be used for some endpoints. The endpoint URI MUST contain a prefix.

Endpoint options that are not mandatory are denoted by []. When there are no mandatory options for an endpoint, one of the set of [] options MUST be provided. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelGoogleMail.<option>**. Note that the

inBody option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelGoogleMail.option** header.

For more information on the endpoints and options see API documentation at: https://developers.google.com/gmail/api/v1/reference/

117.4. CONSUMER ENDPOINTS

Any of the producer endpoints can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. Consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange.

117.5. MESSAGE HEADERS

Any URI option can be provided in a message header for producer endpoints with a **CamelGoogleMail.** prefix.

117.6. MESSAGE BODY

All result message bodies utilize objects provided by the underlying APIs used by the GoogleMailComponent. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint URI parameter. For endpoints that return an array or collection, a consumer endpoint will map every element to distinct messages.

CHAPTER 118. GOOGLE PUBSUB COMPONENT

Available as of Camel version 2.19

The Google Pubsub component provides access to Cloud Pub/Sub Infrastructure via the Google Client Services API.

The current implementation does not use gRPC.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-google-pubsub</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

118.1. URI FORMAT

The GoogleMail Component uses the following URI format:

```
google-pubsub://project-id:destinationName?[options]
```

Destination Name can be either a topic or a subscription name.

118.2. OPTIONS

The Google Pubsub component supports 2 options which are listed below.

Name	Description	Default	Type
connectionFactory (common)	Sets the connection factory to use: provides the ability to explicitly manage connection credentials: - the path to the key file - the Service Account Key / Email pair		GooglePubsubConnection Factory
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Google Pubsub endpoint is configured using URI syntax:

```
google-pubsub:projectId:destinationName
```

with the following path and query parameters:

118.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
projectId	**Required** Project Id		String
destinationName	**Required** Destination Name		String

118.2.2. Query Parameters (9 parameters):

Name	Description	Default	Type
ackMode (common)	AUTO = exchange gets ack'ed/nack'ed on completion. NONE = downstream process has to ack/nack explicitly	AUTO	AckMode
concurrentConsumers (common)	The number of parallel streams consuming from the subscription	1	Integer
connectionFactory (common)	ConnectionFactory to obtain connection to PubSub Service. If non provided the default will be used.		GooglePubsubConnection Factory
loggerId (common)	Logger ID to use when a match to the parent route required		String
maxMessagesPerPoll (common)	The max number of messages to receive from the server in a single API call	1	Integer
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

118.3. PRODUCER ENDPOINTS

Producer endpoints can accept and deliver to PubSub individual and grouped exchanges alike. Grouped exchanges have **Exchange.GROUPED_EXCHANGE** property set.

Google PubSub expects the payload to be byte[] array, Producer endpoints will send:

- String body as byte[] encoded as UTF-8
- byte[] body as is
- Everything else will be serialised into byte[] array

A Map set as message header **GooglePubsubConstants.ATTRIBUTES** will be sent as PubSub attributes. Once exchange has been delivered to PubSub the PubSub Message ID will be assigned to the header **GooglePubsubConstants.MESSAGE_ID**.

118.4. CONSUMER ENDPOINTS

Google PubSub will redeliver the message if it has not been acknowledged within the time period set as a configuration option on the subscription.

The component will acknowledge the message once exchange processing has been completed.

If the route throws an exception, the exchange is marked as failed and the component will NACK the message - it will be redelivered immediately.

To ack/nack the message the component uses Acknowledgement ID stored as header **GooglePubsubConstants.ACK_ID**. If the header is removed or tampered with, the ack will fail and the message will be redelivered again after the ack deadline.

118.5. MESSAGE HEADERS

Headers set by the consumer endpoints:

- GooglePubsubConstants.MESSAGE_ID
- GooglePubsubConstants.ATTRIBUTES
- GooglePubsubConstants.PUBLISH_TIME
- GooglePubsubConstants.ACK_ID

118.6. MESSAGE BODY

The consumer endpoint returns the content of the message as byte[] - exactly as the underlying system sends it. It is up for the route to convert/unmarshall the contents.

118.7. AUTHENTICATION CONFIGURATION

Google Pubsub component authentication is targeted for use with the GCP Service Accounts. For more information please refer to Google Cloud Platform Auth Guide

Google security credentials can be set explicitly via one of the two options:

- Service Account Email and Service Account Key (PEM format)
- GCP credentials file location

If both are set, the Service Account Email/Key will take precedence.

Or implicitly, where the connection factory falls back on Application Default Credentials.

OBS! The location of the default credentials file is configurable - via GOOGLE_APPLICATION_CREDENTIALS environment variable.

Service Account Email and Service Account Key can be found in the GCP JSON credentials file as client_email and private_key respectively.

118.8. ROLLBACK AND REDELIVERY

The rollback for Google PubSub relies on the idea of the Acknowledgement Deadline - the time period where Google PubSub expects to receive the acknowledgement. If the acknowledgement has not been received, the message is redelivered.

Google provides an API to extend the deadline for a message.

More information in Google PubSub Documentation

So, rollback is effectively a deadline extension API call with zero value - i.e. deadline is reached now and message can be redelivered to the next consumer.

It is possible to delay the message redelivery by setting the acknowledgement deadline explicitly for the rollback by setting the message header **GooglePubsubConstants.ACK_DEADLINE** to the value in seconds.

CHAPTER 119. GROOVY LANGUAGE

Available as of Camel version 1.3

Camel supports Groovy among other Scripting Languages to allow an Expression or Predicate to be used in the DSL or Xml Configuration.

To use a Groovy expression use the following Java code

```
... groovy("someGroovyExpression") ...
```

For example you could use the **groovy** function to create an Predicate in a Message Filter or as an Expression for a Recipient List

119.1. GROOVY OPTIONS

The Groovy language supports 1 options which are listed below.

Name	Default	Java Type	Description
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

119.2. CUSTOMIZING GROOVY SHELL

Sometimes you may need to use custom **GroovyShell** instance in your Groovy expressions. To provide custom **GroovyShell**, add implementation of the **org.apache.camel.language.groovy.GroovyShellFactory** SPI interface to your Camel registry. For example after adding the following bean to your Spring context...

```
public class CustomGroovyShellFactory implements GroovyShellFactory {

  public GroovyShell createGroovyShell(Exchange exchange) {
    ImportCustomizer importCustomizer = new ImportCustomizer();
    importCustomizer.addStaticStars("com.example.Utils");
    CompilerConfiguration configuration = new CompilerConfiguration();
    configuration.addCompilationCustomizers(importCustomizer);
    return new GroovyShell(configuration);
  }

}
```

...Camel will use your custom GroovyShell instance (containing your custom static imports), instead of the default one.

119.3. EXAMPLE

```
// lets route if a line item is over $100
from("queue:foo").filter(groovy("request.lineItems.any { i -> i.value > 100 }")).to("queue:bar")
```

And the Spring DSL:

```
<route>
  <from uri="queue:foo"/>
  <filter>
    <groovy>request.lineItems.any { i -> i.value > 100 }</groovy>
    <to uri="queue:bar"/>
  </filter>
</route>
```

119.4. SCRIPTCONTEXT

The JSR-223 scripting languages ScriptContext is pre configured with the following attributes all set at **ENGINE_SCOPE**:

Attribute	Type	Value
context	**org.apache.camel.CamelContext**	The Camel Context (It cannot be used in groovy)
camelContext	**org.apache.camel.CamelContext**	The Camel Context
exchange	**org.apache.camel.Exchange**	The current Exchange
request	**org.apache.camel.Message**	The message (IN message)
response	**org.apache.camel.Message**	**Deprecated**: The OUT message. The OUT message if null by default. Use IN message instead.

Attribute	Type	Value
properties	**org.apache.camel.builder.script.PropertiesFunction**	**Camel 2.9:** Function with a **resolve** method to make it easier to use Camels Properties component from scripts. See further below for example.

See Scripting Languages for the list of languages with explicit DSL support.

119.5. ADDITIONAL ARGUMENTS TO SCRIPTINGENGINE

Available as of Camel 2.8

You can provide additional arguments to the **ScriptingEngine** using a header on the Camel message with the key **CamelScriptArguments**.
See this example:

119.6. USING PROPERTIES FUNCTION

Available as of Camel 2.9

If you need to use the Properties component from a script to lookup property placeholders, then its a bit cumbersome to do so. For example to set a header name myHeader with a value from a property placeholder, which key is provided in a header named "foo".

```
.setHeader("myHeader").groovy(""context.resolvePropertyPlaceholders( + '{{' +
request.headers.get(&#39;foo&#39;) + '}}' + ")")
```

From Camel 2.9 onwards you can now use the properties function and the same example is simpler:

```
.setHeader("myHeader").groovy("properties.resolve(request.headers.get(&#39;foo&#39;))")
```

119.7. LOADING SCRIPT FROM EXTERNAL RESOURCE

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as **"classpath:"**, **"file:"**, or **"http:"**.
This is done using the following syntax: **"resource:scheme:location"**, eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").groovy("resource:classpath:mygroovy.groovy")
```

119.8. HOW TO GET THE RESULT FROM MULTIPLE STATEMENTS SCRIPT

Available as of Camel 2.14

As the scripteengine evale method just return a Null if it runs a multiple statments script. Camel now look up the value of script result by using the key of "result" from the value set. If you have multiple statements script, you need to make sure you set the value of result variable as the script return value.

```
bar = "baz";
# some other statements ...
# camel take the result value as the script evaluation result
result = body * 2 + 1
```

119.9. DEPENDENCIES

To use scripting languages in your camel routes you need to add a dependency on **camel-groovy**.

If you use Maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-groovy</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 120. GRPC COMPONENT

Available as of Camel version 2.19

The gRPC component allows you to call or expose Remote Procedure Call (RPC) services using Protocol Buffers (protobuf) exchange format over HTTP/2 transport.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-grpc</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

120.1. URI FORMAT

```
grpc://service[?options]
```

120.2. ENDPOINT OPTIONS

The gRPC component has no options.

The gRPC endpoint is configured using URI syntax:

```
grpc:host:port/service
```

with the following path and query parameters:

120.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	**Required** The gRPC server host name. This is localhost or 0.0.0.0 when being a consumer or remote server host name when using producer.		String
port	**Required** The gRPC local or remote server port		int
service	**Required** Fully qualified service name from the protocol buffer descriptor file (package dot service definition name)		String

120.2.2. Query Parameters (25 parameters):

Name	Description	Default	Type
flowControlWindow (common)	The HTTP/2 flow control window size (MiB)	1048576	int
maxMessageSize (common)	The maximum message size allowed to be received/sent (MiB)	4194304	int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
consumerStrategy (consumer)	This option specifies the top-level strategy for processing service requests and responses in streaming mode. If an aggregation strategy is selected, all requests will be accumulated in the list, then transferred to the flow, and the accumulated responses will be sent to the sender. If a propagation strategy is selected, request is sent to the stream, and the response will be immediately sent back to the sender.	PROPAGATION	GrpcConsumerStrategy
forwardOnCompleted (consumer)	Determines if onCompleted events should be pushed to the Camel route.	false	boolean
forwardOnError (consumer)	Determines if onError events should be pushed to the Camel route. Exceptions will be set as message body.	false	boolean
maxConcurrentCallsPerConnection (consumer)	The maximum number of concurrent calls permitted for each incoming server connection	2147483647	int
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
method (producer)	gRPC method name		String

Name	Description	Default	Type
producerStrategy (producer)	The mode used to communicate with a remote gRPC server. In SIMPLE mode a single exchange is translated into a remote procedure call. In STREAMING mode all exchanges will be sent within the same request (input and output of the recipient gRPC service must be of type 'stream').	SIMPLE	GrpcProducerStrategy
streamRepliesTo (producer)	When using STREAMING client mode, it indicates the endpoint where responses should be forwarded.		String
userAgent (producer)	The user agent header passed to the server		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
authenticationType (security)	Authentication method type in advance to the SSL/TLS negotiation	NONE	GrpcAuthType
jwtAlgorithm (security)	JSON Web Token sign algorithm	HMAC256	JwtAlgorithm
jwtIssuer (security)	JSON Web Token issuer		String
jwtSecret (security)	JSON Web Token secret		String
jwtSubject (security)	JSON Web Token subject		String
keyCertChainResource (security)	The X.509 certificate chain file resource in PEM format link		String
keyPassword (security)	The PKCS8 private key file password		String
keyResource (security)	The PKCS8 private key file resource in PEM format link		String
negotiationType (security)	Identifies the security negotiation type used for HTTP/2 communication	PLAINTEXT	NegotiationType
serviceAccountResource (security)	Service Account key file in JSON format resource link supported by the Google Cloud SDK		String

Name	Description	Default	Type
trustCertCollectionResource (security)	The trusted certificates collection file resource in PEM format for verifying the remote endpoint's certificate		String

120.3. TRANSPORT SECURITY AND AUTHENTICATION SUPPORT (AVAILABLE FROM CAMEL 2.20)

The following authentication mechanisms are built-in to gRPC and available in this component:

- **SSL/TLS:** gRPC has SSL/TLS integration and promotes the use of SSL/TLS to authenticate the server, and to encrypt all the data exchanged between the client and the server. Optional mechanisms are available for clients to provide certificates for mutual authentication.
- **Token-based authentication with Google:**gRPC provides a generic mechanism to attach metadata based credentials to requests and responses. Additional support for acquiring access tokens while accessing Google APIs through gRPC is provided. In general this mechanism must be used as well as SSL/TLS on the channel.

To enable these features the following component properties combinations must be configured:

Num.	Option	Parameter	Value	Required/Optional
1	**SSL/TLS**	negotiationType	TLS	Required
		keyCertChainResource		Required
		keyResource		Required
		keyPassword		Optional
		trustCertCollectionResource		Optional
2	**Token-based authentication with Google API**	authenticationType	GOOGLE	Required
		negotiationType	TLS	Required
		serviceAccountResource		Required
3	**Custom JSON Web Token implementation authentication**	authenticationType	JWT	Required

Num.	Option	Parameter	Value	Required/Optional
		negotiationType	NONE or TLS	Optional. The TLS/SSL not checking for this type, but strongly recommended.
		jwtAlgorithm	HMAC256(default) or (HMAC384, HMAC512)	Optional
		jwtSecret		Required
		jwtIssuer		Optional
		jwtSubject		Optional

TLS with OpenSSL is currently the recommended approach for using gRPC over TLS component. Using the JDK for ALPN is generally much slower and may not support the necessary ciphers for HTTP2. This function is not implemented in the component.

120.4. GRPC PRODUCER RESOURCE TYPE MAPPING

The table below shows the types of objects in the message body, depending on the types (simple or stream) of incoming and outgoing parameters, as well as the invocation style (synchronous or asynchronous). Please note, that invocation of the procedures with incoming stream parameter in asynchronous style are not allowed.

Invocation style	Request type	Response type	Request Body Type	Result Body Type
synchronous	simple	simple	Object	Object
synchronous	simple	stream	Object	List<Object>
synchronous	stream	simple	not allowed	not allowed
synchronous	stream	stream	not allowed	not allowed
asynchronous	simple	simple	Object	List<Object>
asynchronous	simple	stream	Object	List<Object>
asynchronous	stream	simple	Object or List<Object>	List<Object>
asynchronous	stream	stream	Object or List<Object>	List<Object>

120.5. GRPC CONSUMER HEADERS (WILL BE INSTALLED AFTER THE CONSUMER INVOCATION)

Header name	Description	Possible values
CamelGrpcMethodName	Method name handled by the consumer service	
CamelGrpcEventType	Received event type from the sent request	onNext, onCompleted or onError
CamelGrpcUserAgent	If provided, the given agent will prepend the gRPC library's user agent information	

120.6. EXAMPLES

Below is a simple synchronous method invoke with host and port parameters

```
from("direct:grpc-sync")
.to("grpc://remotehost:1101/org.apache.camel.component.grpc.PingPong?
method=sendPing&synchronous=true");
```

```
<route>
    <from uri="direct:grpc-sync" />
    <to uri="grpc://remotehost:1101/org.apache.camel.component.grpc.PingPong?
method=sendPing&synchronous=true"/>
</route>
```

An asynchronous method invoke

```
from("direct:grpc-async")
.to("grpc://remotehost:1101/org.apache.camel.component.grpc.PingPong?
method=pingAsyncResponse");
```

gRPC service consumer with propagation consumer strategy

```
from("grpc://localhost:1101/org.apache.camel.component.grpc.PingPong?
consumerStrategy=PROPAGATION")
.to("direct:grpc-service");
```

gRPC service producer with streaming producer strategy (requires a service that uses "stream" mode as input and output)

```
from("direct:grpc-request-stream")
.to("grpc://remotehost:1101/org.apache.camel.component.grpc.PingPong?
method=PingAsyncAsync&producerStrategy=STREAMING&streamRepliesTo=direct:grpc-response-
stream");

from("direct:grpc-response-stream")
.log("Response received: ${body}");
```

gRPC service consumer TLS/SLL security negotiation enable

```
from("grpc://localhost:1101/org.apache.camel.component.grpc.PingPong?
consumerStrategy=PROPAGATION&negotiationType=TLS&keyCertChainResource=file:src/test/resour
ces/certs/server.pem&keyResource=file:src/test/resources/certs/server.key&trustCertCollectionResource
=file:src/test/resources/certs/ca.pem")
.to("direct:tls-enable")
```

gRPC service producer with custom JSON Web Token implementation authentication

```
from("direct:grpc-jwt")
.to("grpc://localhost:1101/org.apache.camel.component.grpc.PingPong?
method=pingSyncSync&synchronous=true&authenticationType=JWT&jwtSecret=supersecuredsecret"
);
```

120.7. CONFIGURATION

It's it is recommended to use Maven Protocol Buffers Plugin which calls Protocol Buffer Compiler (protoc) tool to generate Java source files from .proto (protocol buffer definition) files for the custom project. This plugin will generate procedures request and response classes, their builders and gRPC procedures stubs classes as well.

Following steps are required:

Insert operating system and CPU architecture detection extension inside **<build>** tag of the project pom.xml or set ${os.detected.classifier} parameter manually

```
<extensions>
  <extension>
    <groupId>kr.motd.maven</groupId>
    <artifactId>os-maven-plugin</artifactId>
    <version>1.4.1.Final</version>
  </extension>
</extensions>
```

Insert gRPC and protobuf Java code generator plugin **<plugins>** tag of the project pom.xml

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.5.0</version>
  <configuration>
    <protocArtifact>com.google.protobuf:protoc:${protobuf-
version}:exe:${os.detected.classifier}</protocArtifact>
    <pluginId>grpc-java</pluginId>
    <pluginArtifact>io.grpc:protoc-gen-grpc-java:${grpc-
version}:exe:${os.detected.classifier}</pluginArtifact>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>compile-custom</goal>
        <goal>test-compile</goal>
```

```
        <goal>test-compile-custom</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

120.8. FOR MORE INFORMATION, SEE THESE RESOURCES

gRPC project site

Maven Protocol Buffers Plugin

120.9. SEE ALSO

- Getting Started
- Configuring Camel
- Component
- Endpoint
- Protocol Buffers Data Format

CHAPTER 121. GUAVA EVENTBUS COMPONENT

Available as of Camel version 2.10

The Google Guava EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). The **guava-eventbus:** component provides integration bridge between Camel and Google Guava EventBus infrastructure. With the latter component, messages exchanged with the Guava **EventBus** can be transparently forwarded to the Camel routes. EventBus component allows also to route body of Camel exchanges to the Guava **EventBus**.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-guava-eventbus</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

121.1. URI FORMAT

```
guava-eventbus:busName[?options]
```

Where **busName** represents the name of the **com.google.common.eventbus.EventBus** instance located in the Camel registry.

121.2. OPTIONS

The Guava EventBus component supports 3 options which are listed below.

Name	Description	Default	Type
eventBus (common)	To use the given Guava EventBus instance		EventBus
listenerInterface (common)	The interface with method(s) marked with the Subscribe annotation. Dynamic proxy will be created over the interface so it could be registered as the EventBus listener. Particularly useful when creating multi-event listeners and for handling DeadEvent properly. This option cannot be used together with eventClass option.		Class<?>
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Guava EventBus endpoint is configured using URI syntax:

```
guava-eventbus:eventBusRef
```

with the following path and query parameters:

121.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
eventBusRef	To lookup the Guava EventBus from the registry with the given name		String

121.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
eventClass (common)	If used on the consumer side of the route, will filter events received from the EventBus to the instances of the class and superclasses of eventClass. Null value of this option is equal to setting it to the java.lang.Object i.e. the consumer will capture all messages incoming to the event bus. This option cannot be used together with listenerInterface option.		Class<?>
listenerInterface (common)	The interface with method(s) marked with the Subscribe annotation. Dynamic proxy will be created over the interface so it could be registered as the EventBus listener. Particularly useful when creating multi-event listeners and for handling DeadEvent properly. This option cannot be used together with eventClass option.		Class<?>
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

121.3. USAGE

Using **guava-eventbus** component on the consumer side of the route will capture messages sent to the Guava **EventBus** and forward them to the Camel route. Guava EventBus consumer processes incoming messages asynchronously.

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);

from("guava-eventbus:busName").to("seda:queue");

eventBus.post("Send me to the SEDA queue.");
```

Using **guava-eventbus** component on the producer side of the route will forward body of the Camel exchanges to the Guava **EventBus** instance.

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);

from("direct:start").to("guava-eventbus:busName");

ProducerTemplate producerTemplate = camel.createProducerTemplate();
producer.sendBody("direct:start", "Send me to the Guava EventBus.");

eventBus.register(new Object(){
  @Subscribe
  public void messageHander(String message) {
    System.out.println("Message received from the Camel: " + message);
  }
});
```

121.4. DEADEVENT CONSIDERATIONS

Keep in mind that due to the limitations caused by the design of the Guava EventBus, you cannot specify event class to be received by the listener without creating class annotated with **@Subscribe** method. This limitation implies that endpoint with **eventClass** option specified actually listens to all

possible events (**java.lang.Object**) and filter appropriate messages programmatically at runtime. The snipped below demonstrates an appropriate excerpt from the Camel code base.

```
@Subscribe
public void eventReceived(Object event) {
  if (eventClass == null || eventClass.isAssignableFrom(event.getClass())) {
    doEventReceived(event);
...
```

This drawback of this approach is that **EventBus** instance used by Camel will never generate **com.google.common.eventbus.DeadEvent** notifications. If you want Camel to listen only to the precisely specified event (and therefore enable **DeadEvent** support), use **listenerInterface** endpoint option. Camel will create dynamic proxy over the interface you specify with the latter option and listen only to messages specified by the interface handler methods. The example of the listener interface with single method handling only **SpecificEvent** instances is demonstrated below.

```
package com.example;

public interface CustomListener {

  @Subscribe
  void eventReceived(SpecificEvent event);

}
```

The listener presented above could be used in the endpoint definition as follows.

```
from("guava-eventbus:busName?listenerInterface=com.example.CustomListener").to("seda:queue");
```

121.5. CONSUMING MULTIPLE TYPE OF EVENTS

In order to define multiple type of events to be consumed by Guava EventBus consumer use **listenerInterface** endpoint option, as listener interface could provide multiple methods marked with the **@Subscribe** annotation.

```
package com.example;

public interface MultipleEventsListener {

  @Subscribe
  void someEventReceived(SomeEvent event);

  @Subscribe
  void anotherEventReceived(AnotherEvent event);

}
```

The listener presented above could be used in the endpoint definition as follows.

```
from("guava-eventbus:busName?
listenerInterface=com.example.MultipleEventsListener").to("seda:queue");
```

121.6. HAWTDB

Available as of Camel 2.3

HawtDB is a very lightweight and embedable key value database. It allows together with Camel to provide persistent support for various Camel features such as Aggregator.

Deprecated

The HawtDB project is being deprecated and replaced by leveldb as the lightweight and embedable key value database. To make using leveldb easy there is a leveldbjni project for that. The Apache ActiveMQ project is planning on using leveldb as their primary file based message store in the future, to replace kahadb.

There is a camel-leveldb component we recommend to use instead of this.

Issue with HawtDB 1.4 or older

There is a bug in HawtDB 1.4 or older which means the filestore will not free unused space. That means the file keeps growing. This has been fixed in HawtDB 1.5 which is shipped with Camel 2.5 onwards.

Current features it provides:

- HawtDBAggregationRepository

121.6.1. Using HawtDBAggregationRepository

HawtDBAggregationRepository is an **AggregationRepository** which on the fly persists the aggregated messages. This ensures that you will not loose messages, as the default aggregator will use an in memory only **AggregationRepository**.

It has the following options:

Option	Type	Description
repositoryName	String	A mandatory repository name. Allows you to use a shared **HawtDBFile** for multiple repositories.
persistentFileName	String	Filename for the persistent storage. If no file exists on startup a new file is created.
bufferSize	int	The size of the memory segment buffer which is mapped to the file store. By default its 8mb. The value is in bytes.
sync	boolean	Whether or not the **HawtDBFile** should sync on write or not. Default is **true**. By sync on write ensures that its always waiting for all writes to be spooled to disk and thus will not loose updates. If you disable this option, then HawtDB will auto sync when it has batched up a number of writes.

Option	Type	Description
pageSize	short	The size of memory pages. By default its 512 bytes. The value is in bytes.
hawtDBFile	HawtDBFile	Use an existing configured **org.apache.camel.component.hawtdb.HawtDBFile** instance.
returnOldExchange	boolean	Whether the get operation should return the old existing Exchange if any existed. By default this option is **false** to optimize as we do not need the old exchange when aggregating.
useRecovery	boolean	Whether or not recovery is enabled. This option is by default **true**. When enabled the Camel Aggregator automatic recover failed aggregated exchange and have them resubmitted.
recoveryInterval	long	If recovery is enabled then a background task is run every x'th time to scan for failed exchanges to recover and resubmit. By default this interval is 5000 millis.
maximumRedeliveries	int	Allows you to limit the maximum number of redelivery attempts for a recovered exchange. If enabled then the Exchange will be moved to the dead letter channel if all redelivery attempts failed. By default this option is disabled. If this option is used then the **deadLetterUri** option must also be provided.
deadLetterUri	String	An endpoint uri for a Dead Letter Channel where exhausted recovered Exchanges will be moved. If this option is used then the **maximumRedeliveries** option must also be provided.
optimisticLocking	**false**	**Camel 2.12:** To turn on optimistic locking, which often would be needed in clustered environments where multiple Camel applications shared the same HawtDB based aggregation repository.

The **repositoryName** option must be provided. Then either the **persistentFileName** or the **hawtDBFile** must be provided.

121.6.2. What is preserved when persisting

HawtDBAggregationRepository will only preserve any **Serializable** compatible data types. If a data type is not such a type its dropped and a **WARN** is logged. And it only persists the **Message** body and the **Message** headers. The **Exchange** properties are **not** persisted.

121.6.3. Recovery

The **HawtDBAggregationRepository** will by default recover any failed Exchange. It does this by having a background tasks that scans for failed Exchanges in the persistent store. You can use the **checkInterval** option to set how often this task runs. The recovery works as transactional which ensures

that Camel will try to recover and redeliver the failed Exchange. Any Exchange which was found to be recovered will be restored from the persistent store and resubmitted and send out again.

The following headers is set when an Exchange is being recovered/redelivered:

Header	Type	Description
Exchange.REDELIVERED	Boolean	Is set to true to indicate the Exchange is being redelivered.
Exchange.REDELIVERY_COUNTER	Integer	The redelivery attempt, starting from 1.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the **confirm** method is invoked on the **AggregationRepository**. This means if the same Exchange fails again it will be kept retried until it success.

You can use option **maximumRedeliveries** to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the **deadLetterUri** option so Camel knows where to send the Exchange when the **maximumRedeliveries** was hit.

You can see some examples in the unit tests of camel-hawtdb, for example this test.

121.6.3.1. Using HawtDBAggregationRepository in Java DSL

In this example we want to persist aggregated messages in the **target/data/hawtdb.dat** file.

121.6.3.2. Using HawtDBAggregationRepository in Spring XML

The same example but using Spring XML instead:

121.6.4. Dependencies

To use HawtDB in your camel routes you need to add the a dependency on **camel-hawtdb**.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hawtdb</artifactId>
  <version>2.3.0</version>
</dependency>
```

121.6.5. See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Aggregator
- Components

CHAPTER 122. HAZELCAST COMPONENT

Available as of Camel version 2.7

The **hazelcast-** component allows you to work with the Hazelcast distributed data grid / cache. Hazelcast is a in memory data grid, entirely written in Java (single jar). It offers a great palette of different data stores like map, multi map (same key, n values), queue, list and atomic number. The main reason to use Hazelcast is its simple cluster support. If you have enabled multicast on your network you can run a cluster with hundred nodes with no extra configuration. Hazelcast can simply configured to add additional features like n copies between nodes (default is 1), cache persistence, network configuration (if needed), near cache, enviction and so on. For more information consult the Hazelcast documentation on http://www.hazelcast.com/docs.jsp.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hazelcast</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

122.1. HAZELCAST COMPONENTS

See followings for each component usage: * map * multimap * queue * topic * list * seda * set * atomic number * cluster support (instance) * replicatedmap * ringbuffer

122.2. USING HAZELCAST REFERENCE

122.2.1. By its name

```
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
      factory-bean="hazelcastInstance" factory-method="getLifecycleService"
      destroy-method="shutdown" />

<bean id="config" class="com.hazelcast.config.Config">
  <constructor-arg type="java.lang.String" value="HZ.INSTANCE" />
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-
method="newHazelcastInstance">
  <constructor-arg type="com.hazelcast.config.Config" ref="config"/>
</bean>
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route id="testHazelcastInstanceBeanRefPut">
    <from uri="direct:testHazelcastInstanceBeanRefPut"/>
    <setHeader headerName="CamelHazelcastOperationType">
      <constant>put</constant>
    </setHeader>
    <to uri="hazelcast-map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
  </route>

  <route id="testHazelcastInstanceBeanRefGet">
```

```
        <from uri="direct:testHazelcastInstanceBeanRefGet" />
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>get</constant>
        </setHeader>
        <to uri="hazelcast-map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
        <to uri="seda:out" />
    </route>
</camelContext>
```

122.2.2. By instance

```
<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast"
      factory-method="newHazelcastInstance" />
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
      factory-bean="hazelcastInstance" factory-method="getLifecycleService"
      destroy-method="shutdown" />

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="testHazelcastInstanceBeanRefPut">
        <from uri="direct:testHazelcastInstanceBeanRefPut"/>
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>put</constant>
        </setHeader>
        <to uri="hazelcast-map:testmap?hazelcastInstance=#hazelcastInstance"/>
    </route>

    <route id="testHazelcastInstanceBeanRefGet">
        <from uri="direct:testHazelcastInstanceBeanRefGet" />
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>get</constant>
        </setHeader>
        <to uri="hazelcast-map:testmap?hazelcastInstance=#hazelcastInstance"/>
        <to uri="seda:out" />
    </route>
</camelContext>
```

122.3. PUBLISHING HAZELCAST INSTANCE AS AN OSGI SERVICE

If operating in an OSGI container and you would want to use one instance of hazelcast across all bundles in the same container. You can publish the instance as an OSGI service and bundles using the cache al need is to reference the service in the hazelcast endpoint.

122.3.1. Bundle A create an instance and publishes it as an OSGI service

```
<bean id="config" class="com.hazelcast.config.FileSystemXmlConfig">
    <argument type="java.lang.String" value="${hazelcast.config}"/>
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-
method="newHazelcastInstance">
    <argument type="com.hazelcast.config.Config" ref="config"/>
</bean>
```

```
<!-- publishing the hazelcastInstance as a service -->
<service ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />
```

122.3.2. Bundle B uses the instance

```
<!-- referencing the hazelcastInstance as a service -->
<reference ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />

<camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route id="testHazelcastInstanceBeanRefPut">
        <from uri="direct:testHazelcastInstanceBeanRefPut"/>
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>put</constant>
        </setHeader>
        <to uri="hazelcast-map:testmap?hazelcastInstance=#hazelcastInstance"/>
    </route>

    <route id="testHazelcastInstanceBeanRefGet">
        <from uri="direct:testHazelcastInstanceBeanRefGet" />
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>get</constant>
        </setHeader>
        <to uri="hazelcast-map:testmap?hazelcastInstance=#hazelcastInstance"/>
        <to uri="seda:out" />
    </route>
</camelContext>
```

CHAPTER 123. HAZELCAST ATOMIC NUMBER COMPONENT

Available as of Camel version 2.7

The Hazelcast atomic number component is one of Camel Hazelcast Components which allows you to access Hazelcast atomic number. An atomic number is an object that simply provides a grid wide number (long).

There is no consumer for this endpoint!

123.1. OPTIONS

The Hazelcast Atomic Number component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Atomic Number endpoint is configured using URI syntax:

```
hazelcast-atomicvalue:cacheName
```

with the following path and query parameters:

123.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

123.1.2. Query Parameters (10 parameters):

Name	Description	Default	Type
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
defaultOperation (producer)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation
hazelcastInstance (producer)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (producer)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

123.2. ATOMIC NUMBER PRODUCER - TO("HAZELCAST-ATOMICVALUE:FOO")

The operations for this producer are: * setvalue (set the number with a given value) * get * increase (+1) * decrease (-1) * destroy

Header Variables for the request message:

Name	Type	Description
CamelHazelcastOperationType	**String**	valid values are: setvalue, get, increase, decrease, destroy

123.2.1. Sample for set:

Java DSL:

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.SET_VALUE))
.toF("hazelcast-%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:set" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>setvalue</constant>
    </setHeader>
    <to uri="hazelcast-atomicvalue:foo" />
</route>
```

Provide the value to set inside the message body (here the value is 10):
template.sendBody("direct:set", 10);

123.2.2. Sample for get:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.GET))
.toF("hazelcast-%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:get" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>get</constant>
    </setHeader>
    <to uri="hazelcast-atomicvalue:foo" />
</route>
```

You can get the number with **long body = template.requestBody("direct:get", null, Long.class);**.

123.2.3. Sample for increment:

Java DSL:

```
from("direct:increment")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.INCREMENT))
.toF("hazelcast-%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:increment" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>increment</constant>
    </setHeader>
    <to uri="hazelcast-atomicvalue:foo" />
</route>
```

The actual value (after increment) will be provided inside the message body.

123.2.4. Sample for decrement:

Java DSL:

```
from("direct:decrement")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.DECREMENT))
.toF("hazelcast-%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:decrement" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>decrement</constant>
    </setHeader>
    <to uri="hazelcast-atomicvalue:foo" />
</route>
```

The actual value (after decrement) will be provided inside the message body.

123.2.5. Sample for destroy

Java DSL:

```
from("direct:destroy")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.DESTROY))
.toF("hazelcast-%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
```

```
    <from uri="direct:destroy" />
      <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
      <constant>destroy</constant>
    </setHeader>
    <to uri="hazelcast-atomicvalue:foo" />
</route>
```

CHAPTER 124. HAZELCAST INSTANCE COMPONENT

Available as of Camel version 2.7

The Hazelcast instance component is one of Camel Hazelcast Components which allows you to consume join/leave events of the cache instance in the cluster. Hazelcast makes sense in one single "server node", but it's extremly powerful in a clustered environment.

This endpoint provides no producer!

124.1. OPTIONS

The Hazelcast Instance component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Instance endpoint is configured using URI syntax:

```
hazelcast-instance:cacheName
```

with the following path and query parameters:

124.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

124.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
defaultOperation (consumer)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation
hazelcastInstance (consumer)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (consumer)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

124.2. INSTANCE CONSUMER - FROM("HAZELCAST-INSTANCE:FOO")

> The instance consumer fires if a new cache instance will join or leave the cluster.

Here's a sample:

```
fromF("hazelcast-%sfoo", HazelcastConstants.INSTANCE_PREFIX)
.log("instance...")
.choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
        .log("...added")
        .to("mock:added")
    .otherwise()
        .log("...removed")
        .to("mock:removed");
```

Each event provides the following information inside the message header:

Header Variables inside the response message:

Name	Type	Description
CamelHazelcastListenerTime	**Long**	time of the event in millis
CamelHazelcastListenerType	**String**	the map consumer sets here "instancelistener"
CamelHazelcastListenerAction	**String**	type of event - here **added** or **removed**.
CamelHazelcastInstanceHost	**String**	host name of the instance
CamelHazelcastInstancePort	**Integer**	port number of the instance

CHAPTER 125. HAZELCAST LIST COMPONENT

Available as of Camel version 2.7

The Hazelcast List component is one of Camel Hazelcast Components which allows you to access Hazelcast distributed list.

125.1. OPTIONS

The Hazelcast List component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast List endpoint is configured using URI syntax:

```
hazelcast-list:cacheName
```

with the following path and query parameters:

125.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

125.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation

Name	Description	Default	Type
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int

Name	Description	Default	Type
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

125.2. LIST PRODUCER – TO("HAZELCAST-LIST:FOO")

The list producer provides 7 operations: * add * addAll * set * get * removevalue * removeAll * clear

125.2.1. Sample for add:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.ADD))
.toF("hazelcast-%sbar", HazelcastConstants.LIST_PREFIX);
```

125.2.2. Sample for get:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.GET))
.toF("hazelcast-%sbar", HazelcastConstants.LIST_PREFIX)
.to("seda:out");
```

125.2.3. Sample for setvalue:

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.SET_VALUE))
.toF("hazelcast-%sbar", HazelcastConstants.LIST_PREFIX);
```

125.2.4. Sample for removevalue:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.REMOVE_VALUE))
.toF("hazelcast-%sbar", HazelcastConstants.LIST_PREFIX);
```

Note that **CamelHazelcastObjectIndex** header is used for indexing purpose.

125.3. LIST CONSUMER – FROM("HAZELCAST-LIST:FOO")

The list consumer provides 2 operations: * add * remove

```
fromF("hazelcast-%smm", HazelcastConstants.LIST_PREFIX)
    .log("object...")
    .choice()

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
            .log("...added")
                    .to("mock:added")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))

            .log("...removed")
                    .to("mock:removed")
               .otherwise()
                    .log("fail!");
```

CHAPTER 126. HAZELCAST MAP COMPONENT

Available as of Camel version 2.7

The Hazelcast Map component is one of Camel Hazelcast Components which allows you to access Hazelcast distributed map.

126.1. OPTIONS

The Hazelcast Map component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Map endpoint is configured using URI syntax:

```
hazelcast-map:cacheName
```

with the following path and query parameters:

126.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

126.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation

Name	Description	Default	Type
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int

Name	Description	Default	Type
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

126.2. MAP CACHE PRODUCER - TO("HAZELCAST-MAP:FOO")

If you want to store a value in a map you can use the map cache producer.

The map cache producer provides follow operations specified by **CamelHazelcastOperationType** header:

- put
- putIfAbsent
- get
- getAll
- keySet
- containsKey
- containsValue
- delete
- update
- query
- clear
- evict
- evictAll

All operations are provide the inside the "hazelcast.operation.type" header variable. In Java DSL you can use the constants from **org.apache.camel.component.hazelcast.HazelcastOperation**.

Header Variables for the request message:

Name	Type	Description
CamelHazelcastOperationType	**String**	as already described.
CamelHazelcastObjectId	**String**	the object id to store / find your object inside the cache (not needed for the query operation)

put and **putIfAbsent** operations provide an eviction mechanism:

Name	Type	Description
CamelHazelcastObjectTtlValue	**Integer**	value of TTL.
CamelHazelcastObjectTtlUnit	**java.util.concurrent.TimeUnit**	value of time unit (DAYS / HOURS / MINUTES / ....

You can call the samples with:

```
template.sendBodyAndHeader("direct:[put|get|update|delete|query|evict]", "my-foo",
HazelcastConstants.OBJECT_ID, "4711");
```

126.2.1. Sample for put:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.PUT))
.toF("hazelcast-%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
```

```
    <from uri="direct:put" />
      <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
      <constant>put</constant>
    </setHeader>
    <to uri="hazelcast-map:foo" />
</route>
```

Sample for **put** with eviction:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.PUT))
.setHeader(HazelcastConstants.TTL_VALUE, constant(Long.valueOf(1)))
.setHeader(HazelcastConstants.TTL_UNIT, constant(TimeUnit.MINUTES))
.toF("hazelcast-%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:put" />
      <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
      <constant>put</constant>
    </setHeader>
    <setHeader headerName="HazelcastConstants.TTL_VALUE">
      <simple resultType="java.lang.Long">1</simple>
    </setHeader>
    <setHeader headerName="HazelcastConstants.TTL_UNIT">
      <simple resultType="java.util.concurrent.TimeUnit">TimeUnit.MINUTES</simple>
    </setHeader>
    <to uri="hazelcast-map:foo" />
</route>
```

126.2.2. Sample for get:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.GET))
.toF("hazelcast-%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
    <from uri="direct:get" />
      <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
      <constant>get</constant>
    </setHeader>
```

```
    <to uri="hazelcast-map:foo" />
    <to uri="seda:out" />
</route>
```

126.2.3. Sample for update:

Java DSL:

```
from("direct:update")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.UPDATE))
.toF("hazelcast-%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:update" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>update</constant>
    </setHeader>
    <to uri="hazelcast-map:foo" />
</route>
```

126.2.4. Sample for delete:

Java DSL:

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.DELETE))
.toF("hazelcast-%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:delete" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>delete</constant>
    </setHeader>
    <to uri="hazelcast-map:foo" />
</route>
```

126.2.5. Sample for query

Java DSL:

```
from("direct:query")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.QUERY))
.toF("hazelcast-%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
    <from uri="direct:query" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>query</constant>
    </setHeader>
    <to uri="hazelcast-map:foo" />
    <to uri="seda:out" />
</route>
```

For the query operation Hazelcast offers a SQL like syntax to query your distributed map.

```
String q1 = "bar > 1000";
template.sendBodyAndHeader("direct:query", null, HazelcastConstants.QUERY, q1);
```

126.3. MAP CACHE CONSUMER - FROM("HAZELCAST-MAP:FOO")

Hazelcast provides event listeners on their data grid. If you want to be notified if a cache will be manipulated, you can use the map consumer. There're 4 events: **put**, **update**, **delete** and **envict**. The event type will be stored in the "**hazelcast.listener.action**" header variable. The map consumer provides some additional information inside these variables:

Header Variables inside the response message:

Name	Type	Description
CamelHazelcastListenerTime	**Long**	time of the event in millis
CamelHazelcastListenerType	**String**	the map consumer sets here "cachelistener"
CamelHazelcastListenerAction	**String**	type of event - here **added**, **updated**, **envicted** and **removed**.
CamelHazelcastObjectId	**String**	the oid of the object

Name	Type	Description
CamelHazelcastCacheName	**String**	the name of the cache - e.g. "foo"
CamelHazelcastCacheType	**String**	the type of the cache - here map

The object value will be stored within **put** and **update** actions inside the message body.

Here's a sample:

```
fromF("hazelcast-%sfoo", HazelcastConstants.MAP_PREFIX)
.log("object...")
.choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
        .log("...added")
        .to("mock:added")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))

        .log("...envicted")
        .to("mock:envicted")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.UPDATED))
        .log("...updated")
        .to("mock:updated")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))

        .log("...removed")
        .to("mock:removed")
    .otherwise()
        .log("fail!");
```

CHAPTER 127. HAZELCAST MULTIMAP COMPONENT

Available as of Camel version 2.7

The Hazelcast Multimap component is one of Camel Hazelcast Components which allows you to access Hazelcast distributed multimap.

127.1. OPTIONS

The Hazelcast Multimap component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Multimap endpoint is configured using URI syntax:

```
hazelcast-multimap:cacheName
```

with the following path and query parameters:

127.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

127.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation

Name	Description	Default	Type
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int

Name	Description	Default	Type
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

127.2. MULTIMAP CACHE PRODUCER - TO("HAZELCAST-MULTIMAP:FOO")

A multimap is a cache where you can store n values to one key. The multimap producer provides 4 operations (put, get, removevalue, delete).

Header Variables for the request message:

Name	Type	Description
CamelHazelcastOperationType	**String**	valid values are: put, get, removevalue, delete **From Camel 2.16:** clear.
CamelHazelcastObjectId	**String**	the object id to store / find your object inside the cache

127.2.1. Sample for put:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.PUT))
.to(String.format("hazelcast-%sbar", HazelcastConstants.MULTIMAP_PREFIX));
```

Spring DSL:

```
<route>
    <from uri="direct:put" />
    <log message="put.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>put</constant>
    </setHeader>
    <to uri="hazelcast-multimap:foo" />
</route>
```

127.2.2. Sample for removevalue:

Java DSL:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.REMOVE_VALUE))
.toF("hazelcast-%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:removevalue" />
    <log message="removevalue..."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>removevalue</constant>
    </setHeader>
    <to uri="hazelcast-multimap:foo" />
</route>
```

To remove a value you have to provide the value you want to remove inside the message body. If you have a multimap object \{**key: "4711" values: { "my-foo", "my-bar"**}} you have to put "my-foo" inside the message body to remove the "my-foo" value.

127.2.3. Sample for get:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.GET))
.toF("hazelcast-%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
    <from uri="direct:get" />
    <log message="get.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>get</constant>
```

```
    </setHeader>
    <to uri="hazelcast-multimap:foo" />
    <to uri="seda:out" />
</route>
```

127.2.4. Sample for delete:

Java DSL:

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.DELETE))
.toF("hazelcast-%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:delete" />
    <log message="delete.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>delete</constant>
    </setHeader>
    <to uri="hazelcast-multimap:foo" />
</route>
```

you can call them in your test class with:

```
template.sendBodyAndHeader("direct:[put|get|removevalue|delete]", "my-foo",
HazelcastConstants.OBJECT_ID, "4711");
```

127.3. MULTIMAP CACHE CONSUMER - FROM("HAZELCAST-MULTIMAP:FOO")

For the multimap cache this component provides the same listeners / variables as for the map cache consumer (except the update and enviction listener). The only difference is the **multimap** prefix inside the URI. Here is a sample:

```
fromF("hazelcast-%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.log("object...")
.choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
        .log("...added")
            .to("mock:added")

//.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))

        //        .log("...envicted")
        //        .to("mock:envicted")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))

            .log("...removed")
```

```
            .to("mock:removed")
        .otherwise()
            .log("fail!");
```

Header Variables inside the response message:

Name	Type	Description
CamelHazelcastListenerTime	**Long**	time of the event in millis
CamelHazelcastListenerType	**String**	the map consumer sets here "cachelistener"
CamelHazelcastListenerAction	**String**	type of event - here **added** and **removed** (and soon **envicted**)
CamelHazelcastObjectId	**String**	the oid of the object
CamelHazelcastCacheName	**String**	the name of the cache - e.g. "foo"
CamelHazelcastCacheType	**String**	the type of the cache - here multimap

CHAPTER 128. HAZELCAST QUEUE COMPONENT

Available as of Camel version 2.7

The Hazelcast Queue component is one of Camel Hazelcast Components which allows you to access Hazelcast distributed queue.

128.1. OPTIONS

The Hazelcast Queue component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Queue endpoint is configured using URI syntax:

```
hazelcast-queue:cacheName
```

with the following path and query parameters:

128.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

128.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation

Name	Description	Default	Type
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int

Name	Description	Default	Type
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

128.2. QUEUE PRODUCER – TO("HAZELCAST-QUEUE:FOO")

The queue producer provides 10 operations: * add * put * poll * peek * offer * remove value * remaining capacity * remove all * remove if * drain to * take * retain all

128.2.1. Sample for add:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.ADD))
.toF("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX);
```

128.2.2. Sample for put:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.PUT))
.toF("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX);
```

128.2.3. Sample for poll:

```
from("direct:poll")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.POLL))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

128.2.4. Sample for peek:

```
from("direct:peek")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.PEEK))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

128.2.5. Sample for offer:

```
from("direct:offer")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.OFFER))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

128.2.6. Sample for removevalue:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.REMOVE_VALUE))
.toF("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX);
```

128.2.7. Sample for remaining capacity:

```
from("direct:remaining-capacity").setHeader(HazelcastConstants.OPERATION,
constant(HazelcastOperation.REMAINING_CAPACITY)).to(
String.format("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX));
```

128.2.8. Sample for remove all:

```
from("direct:removeAll").setHeader(HazelcastConstants.OPERATION,
constant(HazelcastOperation.REMOVE_ALL)).to(
String.format("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX));
```

128.2.9. Sample for remove if:

```
from("direct:removeIf").setHeader(HazelcastConstants.OPERATION,
constant(HazelcastOperation.REMOVE_IF)).to(
String.format("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX));
```

128.2.10. Sample for drain to:

```
from("direct:drainTo").setHeader(HazelcastConstants.OPERATION,
constant(HazelcastOperation.DRAIN_TO)).to(
String.format("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX));
```

128.2.11. Sample for take:

```
from("direct:take").setHeader(HazelcastConstants.OPERATION,
constant(HazelcastOperation.TAKE)).to(
String.format("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX));
```

128.2.12. Sample for retain all:

-

```
from("direct:retainAll").setHeader(HazelcastConstants.OPERATION,
constant(HazelcastOperation.RETAIN_ALL)).to(
String.format("hazelcast-%sbar", HazelcastConstants.QUEUE_PREFIX));
```

128.3. QUEUE CONSUMER – FROM("HAZELCAST-QUEUE:FOO")

The queue consumer provides two different modes:

- Poll
- Listen

Sample for **Poll** mode

```
fromF("hazelcast-%sfoo?queueConsumerMode=Poll",
HazelcastConstants.QUEUE_PREFIX)).to("mock:result");
```

In this way the consumer will poll the queue and return the head of the queue or null after a timeout.

In Listen mode instead the consumer will listen for events on queue.

The queue consumer in Listen mode provides 2 operations: * add * remove

Sample for **Listen** mode

```
fromF("hazelcast-%smm", HazelcastConstants.QUEUE_PREFIX)
   .log("object...")
   .choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
           .log("...added")
       .to("mock:added")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))

       .log("...removed")
       .to("mock:removed")
    .otherwise()
       .log("fail!");
```

CHAPTER 129. HAZELCAST REPLICATED MAP COMPONENT

Available as of Camel version 2.16

The Hazelcast instance component is one of Camel Hazelcast Components which allows you to consume join/leave events of the cache instance in the cluster. A replicated map is a weakly consistent, distributed key-value data structure with no data partition.

129.1. OPTIONS

The Hazelcast Replicated Map component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Replicated Map endpoint is configured using URI syntax:

```
hazelcast-replicatedmap:cacheName
```

with the following path and query parameters:

129.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

129.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int

Name	Description	Default	Type
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

129.2. REPLICATEDMAP CACHE PRODUCER

The replicatedmap producer provides 4 operations: * put * get * delete * clear

Header Variables for the request message:

Name	Type	Description
CamelHazelcastOperationType	**String**	valid values are: put, get, removevalue, delete
CamelHazelcastObjectId	**String**	the object id to store / find your object inside the cache

129.2.1. Sample for put:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.PUT))
.to(String.format("hazelcast-%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX));
```

Spring DSL:

```
<route>
    <from uri="direct:put" />
    <log message="put.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>put</constant>
    </setHeader>
    <to uri="hazelcast-replicatedmap:foo" />
</route>
```

129.2.2. Sample for get:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.GET))
.toF("hazelcast-%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
    <from uri="direct:get" />
    <log message="get.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>get</constant>
    </setHeader>
    <to uri="hazelcast-replicatedmap:foo" />
    <to uri="seda:out" />
</route>
```

129.2.3. Sample for delete:

Java DSL:

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.DELETE))
.toF("hazelcast-%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX);
```

Spring DSL:

```
<route>
    <from uri="direct:delete" />
    <log message="delete.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>delete</constant>
    </setHeader>
    <to uri="hazelcast-replicatedmap:foo" />
</route>
```

you can call them in your test class with:

```
template.sendBodyAndHeader("direct:[put|get|delete|clear]", "my-foo",
HazelcastConstants.OBJECT_ID, "4711");
```

129.3. REPLICATEDMAP CACHE CONSUMER

For the multimap cache this component provides the same listeners / variables as for the map cache consumer (except the update and enviction listener). The only difference is the **multimap** prefix inside the URI. Here is a sample:

```
fromF("hazelcast-%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.log("object...")
.choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
        .log("...added")
            .to("mock:added")

//.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))

        //      .log("...envicted")
        //      .to("mock:envicted")

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))

            .log("...removed")
            .to("mock:removed")
        .otherwise()
            .log("fail!");
```

Header Variables inside the response message:

Name	Type	Description
CamelHazelcastListenerTime	**Long**	time of the event in millis
CamelHazelcastListenerType	**String**	the map consumer sets here "cachelistener"
CamelHazelcastListenerAction	**String**	type of event - here **added** and **removed** (and soon **envicted**)

Name	Type	Description
CamelHazelcastObjectId	**String**	the oid of the object
CamelHazelcastCacheName	**String**	the name of the cache - e.g. "foo"
CamelHazelcastCacheType	**String**	the type of the cache - here replicatedmap

CHAPTER 130. HAZELCAST RINGBUFFER COMPONENT

Available as of Camel version 2.16

Avalaible from Camel 2.16

The Hazelcast ringbuffer component is one of Camel Hazelcast Components which allows you to access Hazelcast ringbuffer. Ringbuffer is a distributed data structure where the data is stored in a ring-like structure. You can think of it as a circular array with a certain capacity.

130.1. OPTIONS

The Hazelcast Ringbuffer component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Ringbuffer endpoint is configured using URI syntax:

```
hazelcast-ringbuffer:cacheName
```

with the following path and query parameters:

130.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

130.1.2. Query Parameters (10 parameters):

Name	Description	Default	Type
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
defaultOperation (producer)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation
hazelcastInstance (producer)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (producer)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

130.2. RINGBUFFER CACHE PRODUCER

The ringbuffer producer provides 5 operations: * add * readonceHead * readonceTail * remainingCapacity * capacity

Header Variables for the request message:

Name	Type	Description
CamelHazelcastOperationType	**String**	valid values are: put, get, removevalue, delete
CamelHazelcastObjectId	**String**	the object id to store / find your object inside the cache

130.2.1. Sample for put:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.ADD))
.to(String.format("hazelcast-%sbar", HazelcastConstants.RINGBUFFER_PREFIX));
```

Spring DSL:

```
<route>
    <from uri="direct:put" />
    <log message="put.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
    <setHeader headerName="hazelcast.operation.type">
        <constant>add</constant>
    </setHeader>
    <to uri="hazelcast-ringbuffer:foo" />
</route>
```

130.2.2. Sample for readonce from head:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.READ_ONCE_HEAD))
.toF("hazelcast-%sbar", HazelcastConstants.RINGBUFFER_PREFIX)
.to("seda:out");
```

CHAPTER 131. HAZELCAST SEDA COMPONENT

Available as of Camel version 2.7

The Hazelcast SEDA component is one of Camel Hazelcast Components which allows you to access Hazelcast BlockingQueue. SEDA component differs from the rest components provided. It implements a work-queue in order to support asynchronous SEDA architectures, similar to the core "SEDA" component.

131.1. OPTIONS

The Hazelcast SEDA component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast SEDA endpoint is configured using URI syntax:

```
hazelcast-seda:cacheName
```

with the following path and query parameters:

131.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

131.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

131.2. SEDA PRODUCER – TO(“HAZELCAST-SEDA:FOO”)

The SEDA producer provides no operations. You only send data to the specified queue.

Java DSL :

```
from("direct:foo")
.to("hazelcast-seda:foo");
```

Spring DSL :

```
<route>
  <from uri="direct:start" />
  <to uri="hazelcast-seda:foo" />
</route>
```

131.3. SEDA CONSUMER – FROM(“HAZELCAST-SEDA:FOO”)

The SEDA consumer provides no operations. You only retrieve data from the specified queue.

Java DSL :

```
from("hazelcast-seda:foo")
.to("mock:result");
```

Spring DSL:

```
<route>
  <from uri="hazelcast-seda:foo" />
  <to uri="mock:result" />
</route>
```

CHAPTER 132. HAZELCAST SET COMPONENT

Available as of Camel version 2.7

The Hazelcast Set component is one of Camel Hazelcast Components which allows you to access Hazelcast distributed set.

132.1. OPTIONS

The Hazelcast Set component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Set endpoint is configured using URI syntax:

```
hazelcast-set:cacheName
```

with the following path and query parameters:

132.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

132.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation

Name	Description	Default	Type
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int

Name	Description	Default	Type
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

CHAPTER 133. HAZELCAST TOPIC COMPONENT

Available as of Camel version 2.15

The Hazelcast Topic component is one of Camel Hazelcast Components which allows you to access Hazelcast distributed topic.

133.1. OPTIONS

The Hazelcast Topic component supports 3 options which are listed below.

Name	Description	Default	Type
hazelcastInstance (advanced)	The hazelcast instance reference which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		HazelcastInstance
hazelcastMode (advanced)	The hazelcast mode reference which kind of instance should be used. If you don't specify the mode, then the node mode will be the default.	node	String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Hazelcast Topic endpoint is configured using URI syntax:

```
hazelcast-topic:cacheName
```

with the following path and query parameters:

133.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

133.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
defaultOperation (common)	To specify a default operation to use, if no operation header has been provided.		HazelcastOperation

Name	Description	Default	Type
hazelcastInstance (common)	The hazelcast instance reference which can be used for hazelcast endpoint.		HazelcastInstance
hazelcastInstanceName (common)	The hazelcast instance reference name which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.		String
reliable (common)	Define if the endpoint will use a reliable Topic struct or not.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollingTimeout (consumer)	Define the polling timeout of the Queue consumer in Poll mode	10000	long
poolSize (consumer)	Define the Pool size for Queue Consumer Executor	1	int
queueConsumerMode (consumer)	Define the Queue Consumer mode: Listen or Poll	Listen	HazelcastQueueConsumer Mode
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
concurrentConsumers (seda)	To use concurrent consumers polling from the SEDA queue.	1	int

Name	Description	Default	Type
onErrorDelay (seda)	Milliseconds before consumer continues polling after an error has occurred.	1000	int
pollTimeout (seda)	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
transacted (seda)	If set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.	false	boolean
transferExchange (seda)	If set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.	false	boolean

133.2. TOPIC PRODUCER – TO("HAZELCAST-TOPIC:FOO")

The topic producer provides only one operation (publish).

133.2.1. Sample for publish:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastOperation.PUBLISH))
.toF("hazelcast-%sbar", HazelcastConstants.PUBLISH_OPERATION);
```

133.3. TOPIC CONSUMER – FROM("HAZELCAST-TOPIC:FOO")

The topic consumer provides only one operation (received). This component is supposed to support multiple consumption as it's expected when it comes to topics so you are free to have as much consumers as you need on the same hazelcast topic.

```
fromF("hazelcast-%sfoo", HazelcastConstants.TOPIC_PREFIX)
  .choice()

.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.RECEIVED))

      .log("...message received")
    .otherwise()
      .log("...this should never have happened")
```

CHAPTER 134. HBASE COMPONENT

Available as of Camel version 2.10

This component provides an idemptotent repository, producers and consumers for Apache HBase.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hbase</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

134.1. APACHE HBASE OVERVIEW

HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable: A Distributed Storage System for Structured Data. You can use HBase when you need random, realtime read/write access to your Big Data. More information at Apache HBase.

134.2. CAMEL AND HBASE

When using a datasotre inside a camel route, there is always the chalenge of specifying how the camel message will stored to the datastore. In document based stores things are more easy as the message body can be directly mapped to a document. In relational databases an ORM solution can be used to map properties to columns etc. In column based stores things are more challenging as there is no standard way to perform that kind of mapping.

HBase adds two additional challenges:

- HBase groups columns into families, so just mapping a property to a column using a name convention is just not enough.
- HBase doesn't have the notion of type, which means that it stores everything as byte[] and doesn't know if the byte[] represents a String, a Number, a serialized Java object or just binary data.

To overcome these challenges, camel-hbase makes use of the message headers to specify the mapping of the message to HBase columns. It also provides the ability to use some camel-hbase provided classes that model HBase data and can be easily convert to and from xml/json etc.
Finally it provides the ability to the user to implement and use his own mapping strategy.

Regardless of the mapping strategy camel-hbase will convert a message into an org.apache.camel.component.hbase.model.HBaseData object and use that object for its internal operations.

134.3. CONFIGURING THE COMPONENT

The HBase component can be provided a custom HBaseConfiguration object as a property or it can create an HBase configuration object on its own based on the HBase related resources that are found on classpath.

```
<bean id="hbase" class="org.apache.camel.component.hbase.HBaseComponent">
   <property name="configuration" ref="config"/>
</bean>
```

If no configuration object is provided to the component, the component will create one. The created configuration will search the class path for an hbase-site.xml file, from which it will draw the configuration. You can find more information about how to configure HBase clients at: HBase client configuration and dependencies

134.4. HBASE PRODUCER

As mentioned above camel provides produers endpoints for HBase. This allows you to store, delete, retrieve or query data from HBase using your camel routes.

```
hbase://table[?options]
```

where **table** is the table name.

The supported operations are:

- Put
- Get
- Delete
- Scan

134.4.1. Supported URI options

The HBase component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared configuration		Configuration
poolMaxSize (common)	Maximum number of references to keep for each table in the HTable pool. The default value is 10.	10	int
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The HBase endpoint is configured using URI syntax:

```
hbase:tableName
```

with the following path and query parameters:

134.4.2. Path Parameters (1 parameters):

Name	Description	Default	Type
tableName	**Required** The name of the table		String

134.4.3. Query Parameters (16 parameters):

Name	Description	Default	Type
cellMappingStrategyFactory (common)	To use a custom CellMappingStrategyFactory that is responsible for mapping cells.		CellMappingStrategy Factory
filters (common)	A list of filters to use.		List
mappingStrategyClassName (common)	The class name of a custom mapping strategy implementation.		String
mappingStrategyName (common)	The strategy to use for mapping Camel messages to HBase columns. Supported values: header, or body.		String
rowMapping (common)	To map the key/values from the Map to a HBaseRow. The following keys is supported: rowId - The id of the row. This has limited use as the row usually changes per Exchange. rowType - The type to covert row id to. Supported operations: CamelHBaseScan. family - The column family. Supports a number suffix for referring to more than one columns. qualifier - The column qualifier. Supports a number suffix for referring to more than one columns. value - The value. Supports a number suffix for referring to more than one columns valueType - The value type. Supports a number suffix for referring to more than one columns. Supported operations: CamelHBaseGet, and CamelHBaseScan.		Map
rowModel (common)	An instance of org.apache.camel.component.hbase.model.HBaseRow which describes how each row should be modeled		HBaseRow
userGroupInformation (common)	Defines privileges to communicate with HBase such as using kerberos.		UserGroupInformation

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
maxMessagesPerPoll (consumer)	Gets the maximum number of messages as a limit to poll at each polling. Is default unlimited, but use 0 or negative number to disable it as unlimited.		int
operation (consumer)	The HBase operation to perform		String
remove (consumer)	If the option is true, Camel HBase Consumer will remove the rows which it processes.	true	boolean
removeHandler (consumer)	To use a custom HBaseRemoveHandler that is executed when a row is to be removed.		HBaseRemoveHandler
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
maxResults (producer)	The maximum number of rows to scan.	100	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

134.4.4. Put Operations.

HBase is a column based store, which allows you to store data into a specific column of a specific row. Columns are grouped into families, so in order to specify a column you need to specify the column family and the qualifier of that column. To store data into a specific column you need to specify both the column and the row.

The simplest scenario for storing data into HBase from a camel route, would be to store part of the message body to specified HBase column.

```
        <route>
            <from uri="direct:in"/>
            <!-- Set the HBase Row -->
            <setHeader headerName="CamelHBaseRowId">
                <el>${in.body.id}</el>
            </setHeader>
            <!-- Set the HBase Value -->
            <setHeader headerName="CamelHBaseValue">
                <el>${in.body.value}</el>
            </setHeader>
            <to uri="hbase:mytable?
operation=CamelHBasePut&amp;family=myfamily&amp;qualifier=myqualifier"/>
        </route>
```

The route above assumes that the message body contains an object that has an id and value property and will store the content of value in the HBase column myfamily:myqualifier in the row specified by id. If we needed to specify more than one column/value pairs we could just specify additional column mappings. Notice that you must use numbers from the 2nd header onwards, eg RowId2, RowId3, RowId4, etc. Only the 1st header does not have the number 1.

```
        <route>
            <from uri="direct:in"/>
            <!-- Set the HBase Row 1st column -->
            <setHeader headerName="CamelHBaseRowId">
                <el>${in.body.id}</el>
            </setHeader>
            <!-- Set the HBase Row 2nd column -->
            <setHeader headerName="CamelHBaseRowId2">
                <el>${in.body.id}</el>
            </setHeader>
            <!-- Set the HBase Value for 1st column -->
            <setHeader headerName="CamelHBaseValue">
                <el>${in.body.value}</el>
            </setHeader>
            <!-- Set the HBase Value for 2nd column -->
            <setHeader headerName="CamelHBaseValue2">
                <el>${in.body.othervalue}</el>
            </setHeader>
            <to uri="hbase:mytable?
operation=CamelHBasePut&amp;family=myfamily&amp;qualifier=myqualifier&amp;family2=myfamily&a
p;qualifier2=myqualifier2"/>
        </route>
```

It is important to remember that you can use uri options, message headers or a combination of both. It is recommended to specify constants as part of the uri and dynamic values as headers. If something is defined both as header and as part of the uri, the header will be used.

134.4.5. Get Operations.

A Get Operation is an operation that is used to retrieve one or more values from a specified HBase row. To specify what are the values that you want to retrieve you can just specify them as part of the uri or as message headers.

```
        <route>
            <from uri="direct:in"/>
```

```
        <!-- Set the HBase Row of the Get -->
        <setHeader headerName="CamelHBaseRowId">
          <el>${in.body.id}</el>
        </setHeader>
        <to uri="hbase:mytable?
operation=CamelHBaseGet&amp;family=myfamily&amp;qualifier=myqualifier&amp;valueType=java.lang
Long"/>
        <to uri="log:out"/>
      </route>
```

In the example above the result of the get operation will be stored as a header with name CamelHBaseValue.

134.4.6. Delete Operations.

You can also you camel-hbase to perform HBase delete operation. The delete operation will remove an entire row. All that needs to be specified is one or more rows as part of the message headers.

```
      <route>
        <from uri="direct:in"/>
        <!-- Set the HBase Row of the Get -->
        <setHeader headerName="CamelHBaseRowId">
          <el>${in.body.id}</el>
        </setHeader>
        <to uri="hbase:mytable?operation=CamelHBaseDelete"/>
      </route>
```

134.4.7. Scan Operations.

A scan operation is the equivalent of a query in HBase. You can use the scan operation to retrieve multiple rows. To specify what columns should be part of the result and also specify how the values will be converted to objects you can use either uri options or headers.

```
      <route>
        <from uri="direct:in"/>
        <to uri="hbase:mytable?
operation=CamelHBaseScan&amp;family=myfamily&amp;qualifier=myqualifier&amp;valueType=java.lan
g.Long&amp;rowType=java.lang.String"/>
        <to uri="log:out"/>
      </route>
```

In this case its probable that you also also need to specify a list of filters for limiting the results. You can specify a list of filters as part of the uri and camel will return only the rows that satisfy **ALL** the filters.
To have a filter that will be aware of the information that is part of the message, camel defines the ModelAwareFilter. This will allow your filter to take into consideration the model that is defined by the message and the mapping strategy.
When using a ModelAwareFilter camel-hbase will apply the selected mapping strategy to the in message, will create an object that models the mapping and will pass that object to the Filter.

For example to perform scan using as criteria the message headers, you can make use of the ModelAwareColumnMatchingFilter as shown below.

```
      <route>
        <from uri="direct:scan"/>
```

```
        <!-- Set the Criteria -->
        <setHeader headerName="CamelHBaseFamily">
          <constant>name</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseQualifier">
          <constant>first</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseValue">
          <el>in.body.firstName</el>
        </setHeader>
        <setHeader headerName="CamelHBaseFamily2">
          <constant>name</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseQualifier2">
          <constant>last</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseValue2">
          <el>in.body.lastName</el>
        </setHeader>
        <!-- Set additional fields that you want to be return by skipping value -->
        <setHeader headerName="CamelHBaseFamily3">
          <constant>address</constant>
        </setHeader>
        <setHeader headerName="CamelHBaseQualifier3">
          <constant>country</constant>
        </setHeader>
        <to uri="hbase:mytable?operation=CamelHBaseScan&amp;filters=#myFilterList"/>
      </route>

      <bean id="myFilters" class="java.util.ArrayList">
        <constructor-arg>
          <list>
            <bean
class="org.apache.camel.component.hbase.filters.ModelAwareColumnMatchingFilter"/>
          </list>
        </constructor-arg>
      </bean>
```

The route above assumes that a pojo is with properties firstName and lastName is passed as the message body, it takes those properties and adds them as part of the message headers. The default mapping strategy will create a model object that will map the headers to HBase columns and will pass that model the the ModelAwareColumnMatchingFilter. The filter will filter out any rows, that do not contain columns that match the model. It is like query by example.

134.5. HBASE CONSUMER

The Camel HBase Consumer, will perform repeated scan on the specified HBase table and will return the scan results as part of the message. You can either specify header mapping (default) or body mapping. The later will just add the org.apache.camel.component.hbase.model.HBaseData as part of the message body.

```
hbase://table[?options]
```

You can specify the columns that you want to be return and their types as part of the uri options:

```
hbase:mutable?
family=name&qualifer=first&valueType=java.lang.String&family=address&qualifer=number&valueType2=
ava.lang.Integer&rowType=java.lang.Long
```

The example above will create a model object that is consisted of the specified fields and the scan results will populate the model object with values. Finally the mapping strategy will be used to map this model to the camel message.

134.6. HBASE IDEMPOTENT REPOSITORY

The camel-hbase component also provides an idempotent repository which can be used when you want to make sure that each message is processed only once. The HBase idempotent repository is configured with a table, a column family and a column qualifier and will create to that table a row per message.

```
HBaseConfiguration configuration = HBaseConfiguration.create();
HBaseIdempotentRepository repository = new HBaseIdempotentRepository(configuration,
tableName, family, qualifier);

from("direct:in")
  .idempotentConsumer(header("messageId"), repository)
  .to("log:out);
```

134.7. HBASE MAPPING

It was mentioned above that you the default mapping strategies are **header** and **body** mapping. Below you can find some detailed examples of how each mapping strategy works.

134.7.1. HBase Header mapping Examples

The header mapping is the default mapping. To put the value "myvalue" into HBase row "myrow" and column "myfamily:mycolum" the message should contain the following headers:

Header	Value
Camel HBase RowId	myrow
Camel HBase Family	myfamily
Camel HBase Qualifier	myqualifier
Camel HBase Value	myvalue

To put more values for different columns and / or different rows you can specify additional headers suffixed with the index of the headers, e.g:

Header	Value
CamelHBaseRowId	myrow
CamelHBaseFamily	myfamily
CamelHBaseQualifier	myqualifier
CamelHBaseValue	myvalue
CamelHBaseRowId2	myrow2
CamelHBaseFamily2	myfamily
CamelHBaseQualifier2	myqualifier
CamelHBaseValue2	myvalue2

In the case of retrieval operations such as get or scan you can also specify for each column the type that you want the data to be converted to. For exampe:

Header	Value
CamelHBaseFamily	myfamily

Header	Value
Camel HBase Qualifier	myqualifier
Camel HBase ValueType	Long

Please note that in order to avoid boilerplate headers that are considered constant for all messages, you can also specify them as part of the endpoint uri, as you will see below.

134.7.2. Body mapping Examples

In order to use the body mapping strategy you will have to specify the option mappingStrategy as part of the uri, for example:

```
hbase:mytable?mappingStrategyName=body
```

To use the body mapping strategy the body needs to contain an instance of org.apache.camel.component.hbase.model.HBaseData. You can construct t

```
HBaseData data = new HBaseData();
HBaseRow row = new HBaseRow();
row.setId("myRowId");
HBaseCell cell = new HBaseCell();
cell.setFamily("myfamily");
cell.setQualifier("myqualifier");
cell.setValue("myValue");
row.getCells().add(cell);
data.addRows().add(row);
```

The object above can be used for example in a put operation and will result in creating or updating the row with id myRowId and add the value myvalue to the column myfamily:myqualifier.
The body mapping strategy might not seem very appealing at first. The advantage it has over the header mapping strategy is that the HBaseData object can be easily converted to or from xml/json.

134.8. SEE ALSO

- Polling Consumer
- Apache HBase

CHAPTER 135. HDFS COMPONENT (DEPRECATED)

Available as of Camel version 2.8

The **hdfs** component enables you to read and write messages from/to an HDFS file system. HDFS is the distributed file system at the heart of Hadoop.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hdfs</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

135.1. URI FORMAT

```
hdfs://hostname[:port][/path][?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…** The path is treated in the following way:

1. as a consumer, if it's a file, it just reads the file, otherwise if it represents a directory it scans all the file under the path satisfying the configured pattern. All the files under that directory must be of the same type.

2. as a producer, if at least one split strategy is defined, the path is considered a directory and under that directory the producer creates a different file per split named using the configured UuidGenerator.

Note

When consuming from hdfs then in normal mode, a file is split into chunks, producing a message per chunk. You can configure the size of the chunk using the chunkSize option. If you want to read from hdfs and write to a regular file using the file component, then you can use the fileMode=Append to append each of the chunks together.

135.2. OPTIONS

The HDFS component supports 2 options which are listed below.

Name	Description	Default	Type
jAASConfiguration (common)	To use the given configuration for security with JAAS.		Configuration

Name	Description	Default	Type
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The HDFS endpoint is configured using URI syntax:

```
hdfs:hostName:port/path
```

with the following path and query parameters:

135.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
hostName	**Required** HDFS host to use		String
port	HDFS port to use	8020	int
path	**Required** The directory path to use		String

135.2.2. Query Parameters (38 parameters):

Name	Description	Default	Type
connectOnStartup (common)	Whether to connect to the HDFS file system on starting the producer/consumer. If false then the connection is created on-demand. Notice that HDFS may take up till 15 minutes to establish a connection, as it has hardcoded 45 x 20 sec redelivery. By setting this option to false allows your application to startup, and not block for up till 15 minutes.	true	boolean
fileSystemType (common)	Set to LOCAL to not use HDFS but local java.io.File instead.	HDFS	HdfsFileSystemType
fileType (common)	The file type to use. For more details see Hadoop HDFS documentation about the various files types.	NORMAL_FILE	HdfsFileType
keyType (common)	The type for the key in case of sequence or map files.	NULL	WritableType

Name	Description	Default	Type
owner (common)	The file owner must match this owner for the consumer to pickup the file. Otherwise the file is skipped.		String
valueType (common)	The type for the key in case of sequence or map files	BYTES	WritableType
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delay (consumer)	The interval (milliseconds) between the directory scans.	1000	long
initialDelay (consumer)	For the consumer, how much to wait (milliseconds) before to start scanning the directory.		long
pattern (consumer)	The pattern used for scanning the directory	*	String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
append (producer)	Append to existing file. Notice that not all HDFS file systems support the append option.	false	boolean

Name	Description	Default	Type
overwrite (producer)	Whether to overwrite existing files with the same name	true	boolean
blockSize (advanced)	The size of the HDFS blocks	67108864	long
bufferSize (advanced)	The buffer size used by HDFS	4096	int
checkIdleInterval (advanced)	How often (time in millis) in to run the idle checker background task. This option is only in use if the splitter strategy is IDLE.	500	int
chunkSize (advanced)	When reading a normal file, this is split into chunks producing a message per chunk.	4096	int
compressionCodec (advanced)	The compression codec to use	DEFAULT	HdfsCompressionCodec
compressionType (advanced)	The compression type to use (is default not in use)	NONE	CompressionType
openedSuffix (advanced)	When a file is opened for reading/writing the file is renamed with this suffix to avoid to read it during the writing phase.	opened	String
readSuffix (advanced)	Once the file has been read is renamed with this suffix to avoid to read it again.	read	String
replication (advanced)	The HDFS replication factor	3	short

Name	Description	Default	Type
splitStrategy (advanced)	In the current version of Hadoop opening a file in append mode is disabled since it's not very reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way: If the split strategy option has been defined, the hdfs path will be used as a directory and files will be created using the configured UuidGenerator. Every time a splitting condition is met, a new file is created. The splitStrategy option is defined as a string with the following syntax: splitStrategy=ST:value,ST:value,... where ST can be: BYTES a new file is created, and the old is closed when the number of written bytes is more than value MESSAGES a new file is created, and the old is closed when the number of written messages is more than value IDLE a new file is created, and the old is closed when no writing happened in the last value milliseconds		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel

Name	Description	Default	Type
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

135.2.3. KeyType and ValueType

- NULL it means that the key or the value is absent
- BYTE for writing a byte, the java Byte class is mapped into a BYTE
- BYTES for writing a sequence of bytes. It maps the java ByteBuffer class
- INT for writing java integer
- FLOAT for writing java float
- LONG for writing java long
- DOUBLE for writing java double
- TEXT for writing java strings

BYTES is also used with everything else, for example, in Camel a file is sent around as an InputStream, int this case is written in a sequence file or a map file as a sequence of bytes.

135.3. SPLITTING STRATEGY

In the current version of Hadoop opening a file in append mode is disabled since it's not very reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way:

- If the split strategy option has been defined, the hdfs path will be used as a directory and files will be created using the configured UuidGenerator
- Every time a splitting condition is met, a new file is created.
  The splitStrategy option is defined as a string with the following syntax:
  splitStrategy=<ST>:<value>,<ST>:<value>,*

where <ST> can be:

- BYTES a new file is created, and the old is closed when the number of written bytes is more than <value>
- MESSAGES a new file is created, and the old is closed when the number of written messages is more than <value>
- IDLE a new file is created, and the old is closed when no writing happened in the last <value> milliseconds

Note

note that this strategy currently requires either setting an IDLE value or setting the HdfsConstants.HDFS_CLOSE header to false to use the BYTES/MESSAGES configuration...otherwise, the file will be closed with each message

for example:

```
hdfs://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5
```

it means: a new file is created either when it has been idle for more than 1 second or if more than 5 bytes have been written. So, running **hadoop fs -ls /tmp/simple-file** you'll see that multiple files have been created.

135.4. MESSAGE HEADERS

The following headers are supported by this component:

135.4.1. Producer only

Header	Description
CamelFileName	**Camel 2.13:** Specifies the name of the file to write (relative to the endpoint path). The name can be a **String** or an Expression object. Only relevant when not using a split strategy.

135.5. CONTROLLING TO CLOSE FILE STREAM

Available as of Camel 2.10.4

When using the HDFS producer **without** a split strategy, then the file output stream is by default closed after the write. However you may want to keep the stream open, and only explicitly close the stream later. For that you can use the header **HdfsConstants.HDFS_CLOSE** (value = **"CamelHdfsClose"**) to control this. Setting this value to a boolean allows you to explicit control whether the stream should be closed or not.

Notice this does not apply if you use a split strategy, as there are various strategies that can control when the stream is closed.

135.6. USING THIS COMPONENT IN OSGI

This component is fully functional in an OSGi environment, however, it requires some actions from the user. Hadoop uses the thread context class loader in order to load resources. Usually, the thread context classloader will be the bundle class loader of the bundle that contains the routes. So, the default configuration files need to be visible from the bundle class loader. A typical way to deal with it is to keep a copy of core-default.xml in your bundle root. That file can be found in the hadoop-common.jar.

CHAPTER 136. HDFS2 COMPONENT

Available as of Camel version 2.14

The **hdfs2** component enables you to read and write messages from/to an HDFS file system using Hadoop 2.x. HDFS is the distributed file system at the heart of Hadoop.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hdfs2</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

136.1. URI FORMAT

```
hdfs2://hostname[:port][/path][?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…** The path is treated in the following way:

1. as a consumer, if it's a file, it just reads the file, otherwise if it represents a directory it scans all the file under the path satisfying the configured pattern. All the files under that directory must be of the same type.

2. as a producer, if at least one split strategy is defined, the path is considered a directory and under that directory the producer creates a different file per split named using the configured UuidGenerator.

When consuming from hdfs2 then in normal mode, a file is split into chunks, producing a message per chunk. You can configure the size of the chunk using the chunkSize option. If you want to read from hdfs and write to a regular file using the file component, then you can use the fileMode=Append to append each of the chunks together.

136.2. OPTIONS

The HDFS2 component supports 2 options which are listed below.

Name	Description	Default	Type
jAASConfiguration (common)	To use the given configuration for security with JAAS.		Configuration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The HDFS2 endpoint is configured using URI syntax:

```
hdfs2:hostName:port/path
```

with the following path and query parameters:

136.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
hostName	**Required** HDFS host to use		String
port	HDFS port to use	8020	int
path	**Required** The directory path to use		String

136.2.2. Query Parameters (38 parameters):

Name	Description	Default	Type
connectOnStartup (common)	Whether to connect to the HDFS file system on starting the producer/consumer. If false then the connection is created on-demand. Notice that HDFS may take up till 15 minutes to establish a connection, as it has hardcoded 45 x 20 sec redelivery. By setting this option to false allows your application to startup, and not block for up till 15 minutes.	true	boolean
fileSystemType (common)	Set to LOCAL to not use HDFS but local java.io.File instead.	HDFS	HdfsFileSystemType
fileType (common)	The file type to use. For more details see Hadoop HDFS documentation about the various files types.	NORMAL_FILE	HdfsFileType
keyType (common)	The type for the key in case of sequence or map files.	NULL	WritableType
owner (common)	The file owner must match this owner for the consumer to pickup the file. Otherwise the file is skipped.		String
valueType (common)	The type for the key in case of sequence or map files	BYTES	WritableType

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pattern (consumer)	The pattern used for scanning the directory	*	String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
append (producer)	Append to existing file. Notice that not all HDFS file systems support the append option.	false	boolean
overwrite (producer)	Whether to overwrite existing files with the same name	true	boolean
blockSize (advanced)	The size of the HDFS blocks	67108864	long
bufferSize (advanced)	The buffer size used by HDFS	4096	int
checkIdleInterval (advanced)	How often (time in millis) in to run the idle checker background task. This option is only in use if the splitter strategy is IDLE.	500	int

Name	Description	Default	Type
chunkSize (advanced)	When reading a normal file, this is split into chunks producing a message per chunk.	4096	int
compressionCodec (advanced)	The compression codec to use	DEFAULT	HdfsCompressionCodec
compressionType (advanced)	The compression type to use (is default not in use)	NONE	CompressionType
openedSuffix (advanced)	When a file is opened for reading/writing the file is renamed with this suffix to avoid to read it during the writing phase.	opened	String
readSuffix (advanced)	Once the file has been read is renamed with this suffix to avoid to read it again.	read	String
replication (advanced)	The HDFS replication factor	3	short
splitStrategy (advanced)	In the current version of Hadoop opening a file in append mode is disabled since it's not very reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way: If the split strategy option has been defined, the hdfs path will be used as a directory and files will be created using the configured UuidGenerator. Every time a splitting condition is met, a new file is created. The splitStrategy option is defined as a string with the following syntax: splitStrategy=ST:value,ST:value,... where ST can be: BYTES a new file is created, and the old is closed when the number of written bytes is more than value MESSAGES a new file is created, and the old is closed when the number of written messages is more than value IDLE a new file is created, and the old is closed when no writing happened in the last value milliseconds		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

136.2.3. KeyType and ValueType

- NULL it means that the key or the value is absent
- BYTE for writing a byte, the java Byte class is mapped into a BYTE
- BYTES for writing a sequence of bytes. It maps the java ByteBuffer class
- INT for writing java integer
- FLOAT for writing java float
- LONG for writing java long
- DOUBLE for writing java double
- TEXT for writing java strings

BYTES is also used with everything else, for example, in Camel a file is sent around as an InputStream, int this case is written in a sequence file or a map file as a sequence of bytes.

136.3. SPLITTING STRATEGY

In the current version of Hadoop opening a file in append mode is disabled since it's not very reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way:

- If the split strategy option has been defined, the hdfs path will be used as a directory and files will be created using the configured UuidGenerator
- Every time a splitting condition is met, a new file is created.
  The splitStrategy option is defined as a string with the following syntax: splitStrategy=<ST>:<value>,<ST>:<value>,*

where <ST> can be:

- BYTES a new file is created, and the old is closed when the number of written bytes is more than <value>
- MESSAGES a new file is created, and the old is closed when the number of written messages is more than <value>
- IDLE a new file is created, and the old is closed when no writing happened in the last <value> milliseconds

note that this strategy currently requires either setting an IDLE value or setting the HdfsConstants.HDFS_CLOSE header to false to use the BYTES/MESSAGES configuration...otherwise, the file will be closed with each message

for example:

```
hdfs2://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5
```

it means: a new file is created either when it has been idle for more than 1 second or if more than 5 bytes have been written. So, running **hadoop fs -ls /tmp/simple-file** you'll see that multiple files have been created.

136.4. MESSAGE HEADERS

The following headers are supported by this component:

136.4.1. Producer only

Header	Description
CamelFileName	**Camel 2.13:** Specifies the name of the file to write (relative to the endpoint path). The name can be a **String** or an Expression object. Only relevant when not using a split strategy.

136.5. CONTROLLING TO CLOSE FILE STREAM

When using the HDFS2 producer **without** a split strategy, then the file output stream is by default closed after the write. However you may want to keep the stream open, and only explicitly close the stream later. For that you can use the header **HdfsConstants.HDFS_CLOSE** (value = **"CamelHdfsClose"**) to control this. Setting this value to a boolean allows you to explicit control whether the stream should be closed or not.

Notice this does not apply if you use a split strategy, as there are various strategies that can control when the stream is closed.

136.6. USING THIS COMPONENT IN OSGI

There are some quirks when running this component in an OSGi environment related to the mechanism Hadoop 2.x uses to discover different **org.apache.hadoop.fs.FileSystem** implementations. Hadoop 2.x uses **java.util.ServiceLoader** which looks for **/META-INF/services/org.apache.hadoop.fs.FileSystem** files defining available filesystem types and implementations. These resources are not available when running inside OSGi.

As with **camel-hdfs** component, the default configuration files need to be visible from the bundle class loader. A typical way to deal with it is to keep a copy of **core-default.xml** (and e.g., **hdfs-default.xml**) in your bundle root.

136.6.1. Using this component with manually defined routes

There are two options:

1. Package **/META-INF/services/org.apache.hadoop.fs.FileSystem** resource with bundle that defines the routes. This resource should list all the required Hadoop 2.x filesystem implementations.

2. Provide boilerplate initialization code which populates internal, static cache inside **org.apache.hadoop.fs.FileSystem** class:

```
org.apache.hadoop.conf.Configuration conf = new org.apache.hadoop.conf.Configuration();
conf.setClass("fs.file.impl", org.apache.hadoop.fs.LocalFileSystem.class, FileSystem.class);
conf.setClass("fs.hdfs.impl", org.apache.hadoop.hdfs.DistributedFileSystem.class, FileSystem.class);
...
```

```
FileSystem.get("file:///", conf);
FileSystem.get("hdfs://localhost:9000/", conf);
...
```

136.6.2. Using this component with Blueprint container

Two options:

1. Package **/META-INF/services/org.apache.hadoop.fs.FileSystem** resource with bundle that contains blueprint definition.

2. Add the following to the blueprint definition file:

```
<bean id="hdfsOsgiHelper" class="org.apache.camel.component.hdfs2.HdfsOsgiHelper">
  <argument>
    <map>
      <entry key="file:///" value="org.apache.hadoop.fs.LocalFileSystem"  />
      <entry key="hdfs://localhost:9000/" value="org.apache.hadoop.hdfs.DistributedFileSystem" />
      ...
    </map>
  </argument>
</bean>

<bean id="hdfs2" class="org.apache.camel.component.hdfs2.HdfsComponent" depends-
on="hdfsOsgiHelper" />
```

This way Hadoop 2.x will have correct mapping of URI schemes to filesystem implementations.

CHAPTER 137. HEADERSMAP

Available as of Camel 2.20

The camel-headersmap is a faster implementation of a case-insenstive map which can be plugged in and used by Camel at runtime to have slight faster performance in the Camel Message headers.

137.1. AUTO DETECTION FROM CLASSPATH

To use this implementation all you need to do is to add the **camel-headersmap** dependency to the classpath, and Camel should auto-detect this on startup and log as follows:

```
Detected and using custom HeadersMapFactory:
org.apache.camel.component.headersmap.FastHeadersMapFactory@71e9ebae
```

For spring-boot there is a **camel-headersmap-starter** dependency you should use.

137.2. MANUAL ENABLING

If you use OSGi or the implementation is not added to the classpath, you need to enable this explict such .Title

```
CamelContext camel = ...

camel.setHeadersMapFactory(new FastHeadersMapFactory());
```

Or in XML DSL (spring or blueprint XML file) you can declare the factory as a **<bean>**

```
<bean id="fastMapFactory"
class="org.apache.camel.component.headersmap.FastHeadersMapFactory"/>
```

and then Camel should detect the bean and use the factory, which is logged:

CHAPTER 138. HESSIAN DATAFORMAT (DEPRECATED)

Available as of Camel version 2.17

Hessian is Data Format for marshalling and unmarshalling messages using Caucho's Hessian format.

If you want to use Hessian Data Format from Maven, add the following dependency to your **pom.xml**:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hessian</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

138.1. OPTIONS

The Hessian dataformat supports 4 options which are listed below.

Name	Default	Java Type	Description
whitelistEnabled	**true**	**Boolean**	Define if Whitelist feature is enabled or not
allowedUnmarshallObjects		**String**	Define the allowed objects to be unmarshalled
deniedUnmarshallObjects		**String**	Define the denied objects to be unmarshalled
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

138.2. USING THE HESSIAN DATA FORMAT IN JAVA DSL

```
from("direct:in")
    .marshal().hessian();
```

138.3. USING THE HESSIAN DATA FORMAT IN SPRING DSL

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:in"/>
    <marshal ref="hessian"/>
  </route>
</camelContext>
```

-

CHAPTER 139. HIPCHAT COMPONENT

Available as of Camel version 2.15

The Hipchat component supports producing and consuming messages from/to Hipchat service.

Prerequisites

You must have a valid Hipchat user account and get a personal access token that you can use to produce/consume messages.

139.1. URI FORMAT

```
hipchat://[host][:port]?options
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

139.2. URI OPTIONS

The Hipchat component has no options.

The Hipchat endpoint is configured using URI syntax:

```
hipchat:protocol:host:port
```

with the following path and query parameters:

139.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol for the hipchat server, such as http.		String
host	**Required** The host for the hipchat server, such as api.hipchat.com		String
port	The port for the hipchat server. Is by default 80.	80	Integer

139.2.2. Query Parameters (22 parameters):

Name	Description	Default	Type
authToken (common)	OAuth 2 auth token		String

Name	Description	Default	Type
consumeUsers (common)	Username(s) when consuming messages from the hiptchat server. Multiple user names can be separated by comma.		String
httpClient (common)	The CloseableHttpClient reference from registry to be used during API HTTP requests.	CloseableHttpClient default from HttpClient library	CloseableHttpClient
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean

Name	Description	Default	Type
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

139.3. SCHEDULED POLL CONSUMER

This component implements the ScheduledPollConsumer. Only the last message from the provided 'consumeUsers' are retrieved and sent as Exchange body. If you do not want the same message to be retrieved again when there are no new messages on next poll then you can add the idempotent consumer as shown below. All the options on the ScheduledPollConsumer can also be used for more control on the consumer.

```
@Override
public void configure() throws Exception {
 String hipchatEndpointUri = "hipchat://?authToken=XXXX&consumeUsers=@Joe,@John";
 from(hipchatEndpointUri)
  .idempotentConsumer(
    simple("${in.header.HipchatMessageDate} ${in.header.HipchatFromUser}"),
    MemoryIdempotentRepository.memoryIdempotentRepository(200)
  )
  .to("mock:result");
}
```

139.3.1. Message headers set by the Hipchat consumer

Header	Constant	Type	Description
HipchatFromUser	HipchatConstants.FROM_USER	*String*	The body has the message that was sent from this user to the owner of authToken
HipchatMessageDate	HipchatConstants.MESSAGE_DATE	*String*	The date message was sent. The format is ISO-8601 as present in the Hipchat response.

139.4. HIPCHAT PRODUCER

Producer can send messages to both Room's and User's simultaneously. The body of the exchange is sent as message. Sample usage is shown below. Appropriate headers needs to be set.

```
@Override
 public void configure() throws Exception {
  String hipchatEndpointUri = "hipchat://?authToken=XXXX";
  from("direct:start")
   .to(hipchatEndpointUri)
   .to("mock:result");
 }
```

139.4.1. Message headers evaluated by the Hipchat producer

Header	Constant	Type	Description
HipchatToUser	HipchatConstants.TO_USER	*String*	The Hipchat user to which the message needs to be sent.
HipchatToRoom	HipchatConstants.TO_ROOM	*String*	The Hipchat room to which the message needs to be sent.
HipchatMessageFormat	HipchatConstants.MESSAGE_FORMAT	String	Valid formats are 'text' or 'html'. **Default: 'text'**
HipchatMessageBackgroundColor	HipchatConstants.MESSAGE_BACKGROUND_COLOR	*String*	Valid color values are 'yellow', 'green', 'red', 'purple', 'gray', 'random'. **Default: 'yellow' (Room Only)**
HipchatTriggerNotification	HipchatConstants.TRIGGER_NOTIFY	*String*	Valid values are 'true' or 'false'. Whether this message should trigger a user notification (change the tab color, play a sound, notify mobile phones, etc). **Default: 'false' (Room Only)**

139.4.2. Message headers set by the Hipchat producer

Header	Constant	Type	Description
HipchatToUserResponseStatus	HipchatConstants.TO_USER_RESPONSE_STATUS	*StatusLine* The status of the API response received when message sent to the user.	HipchatFromUserResponseStatus

139.4.3. Configuring Http Client

The HipChat component allow your own **HttpClient** configuration. This can be done by defining a reference for **CloseableHttpClient** in the registry (e.g. Spring Context) and then, set the parameter during the Endpoint definition, for example: **hipchat:http://api.hipchat.com?httpClient=#myHttpClient**.

```
CloseableHttpClient httpclient = HttpClients.custom()
    .setConnectionManager(connManager)
    .setDefaultCookieStore(cookieStore)
    .setDefaultCredentialsProvider(credentialsProvider)
    .setProxy(new HttpHost("myproxy", 8080))
    .setDefaultRequestConfig(defaultRequestConfig)
    .build();
```

To see more information about Http Client configuration, please check the official documentation.

139.4.4. Dependencies

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hipchat</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.15.0 or higher)

CHAPTER 140. HL7 DATAFORMAT

Available as of Camel version 2.0

The **HL7** component is used for working with the HL7 MLLP protocol and HL7 v2 messages using the HAPI library.

This component supports the following:

- HL7 MLLP codec for Mina
- HL7 MLLP codec for Netty4 from **Camel 2.15** onwards
- Type Converter from/to HAPI and String
- HL7 DataFormat using the HAPI library
- Even more ease-of-use as it's integrated well with the camel-mina2 component.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hl7</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

140.1. HL7 MLLP PROTOCOL

HL7 is often used with the HL7 MLLP protocol, which is a text based TCP socket based protocol. This component ships with a Mina and Netty4 Codec that conforms to the MLLP protocol so you can easily expose an HL7 listener accepting HL7 requests over the TCP transport layer. To expose a HL7 listener service, the camel-mina2 or camel-netty4 component is used with the **HL7MLLPCodec** (mina2) or **HL7MLLPNettyDecoder**/**HL7MLLPNettyEncoder** (Netty4).

HL7 MLLP codec can be configured as follows:

Name	Default Value	Description
startByte	**0x0b**	The start byte spanning the HL7 payload.
endByte1	**0x1c**	The first end byte spanning the HL7 payload.
endByte2	**0x0d**	The 2nd end byte spanning the HL7 payload.
charset	JVM Default	The encoding (a charset name) to use for the codec. If not provided, Camel will use the JVM default Charset.

Name	Default Value	Description
produceString	**true**	**(as of Camel 2.14.1)** If true, the codec creates a string using the defined charset. If false, the codec sends a plain byte array into the route, so that the HL7 Data Format can determine the actual charset from the HL7 message content.
convertLFtoCR	**false**	Will convert **\n** to **\r** (**0x0d**, 13 decimal) as HL7 stipulates **\r** as segment terminators. The HAPI library requires the use of **\r**.

140.1.1. Exposing an HL7 listener using Mina

In the Spring XML file, we configure a mina2 endpoint to listen for HL7 requests using TCP on port **8888**:

```
<endpoint id="hl7MinaListener" uri="mina2:tcp://localhost:8888?sync=true&amp;codec=#hl7codec"/>
```

sync=true indicates that this listener is synchronous and therefore will return a HL7 response to the caller. The HL7 codec is setup with **codec=#hl7codec**. Note that **hl7codec** is just a Spring bean ID, so it could be named **mygreatcodecforhl7** or whatever. The codec is also set up in the Spring XML file:

```
<bean id="hl7codec" class="org.apache.camel.component.hl7.HL7MLLPCodec">
    <property name="charset" value="iso-8859-1"/>
</bean>
```

The endpoint **hl7MinaLlistener** can then be used in a route as a consumer, as this Java DSL example illustrates:

```
from("hl7MinaListener")
  .bean("patientLookupService");
```

This is a very simple route that will listen for HL7 and route it to a service named **patientLookupService**. This is also Spring bean ID, configured in the Spring XML as:

```
<bean id="patientLookupService"
class="com.mycompany.healthcare.service.PatientLookupService"/>
```

The business logic can be implemented in POJO classes that do not depend on Camel, as shown here:

```
import ca.uhn.hl7v2.HL7Exception;
import ca.uhn.hl7v2.model.Message;
import ca.uhn.hl7v2.model.v24.segment.QRD;

public class PatientLookupService {
    public Message lookupPatient(Message input) throws HL7Exception {
        QRD qrd = (QRD)input.get("QRD");
        String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();

        // find patient data based on the patient id and create a HL7 model object with the response
```

```
        Message response = ... create and set response data
        return response
    }
```

140.1.2. Exposing an HL7 listener using Netty (available from Camel 2.15 onwards)

In the Spring XML file, we configure a netty4 endpoint to listen for HL7 requests using TCP on port **8888**:

```
<endpoint id="hl7NettyListener" uri="netty4:tcp://localhost:8888?
sync=true&amp;encoder=#hl7encoder&amp;decoder=#hl7decoder"/>
```

sync=true indicates that this listener is synchronous and therefore will return a HL7 response to the caller. The HL7 codec is setup with **encoder=#hl7encoder*and*decoder=#hl7decoder**. Note that **hl7encoder** and **hl7decoder** are just bean IDs, so they could be named differently. The beans can be set in the Spring XML file:

```
<bean id="hl7decoder" class="org.apache.camel.component.hl7.HL7MLLPNettyDecoderFactory"/>
<bean id="hl7encoder" class="org.apache.camel.component.hl7.HL7MLLPNettyEncoderFactory"/>
```

The endpoint **hl7NettyListener** can then be used in a route as a consumer, as this Java DSL example illustrates:

```
from("hl7NettyListener")
  .bean("patientLookupService");
```

140.2. HL7 MODEL USING JAVA.LANG.STRING OR BYTE[]

The HL7 MLLP codec uses plain String as its data format. Camel uses its Type Converter to convert to/from strings to the HAPI HL7 model objects, but you can use the plain String objects if you prefer, for instance if you wish to parse the data yourself.

As of Camel 2.14.1 you can also let both the Mina and Netty codecs use a plain **byte[]** as its data format by setting the **produceString** property to false. The Type Converter is also capable of converting the **byte[]** to/from HAPI HL7 model objects.

140.3. HL7V2 MODEL USING HAPI

The HL7v2 model uses Java objects from the HAPI library. Using this library, you can encode and decode from the EDI format (ER7) that is mostly used with HL7v2.

The sample below is a request to lookup a patient with the patient ID **0101701234**.

```
MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|1234|P|2.4
QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||
```

Using the HL7 model you can work with a **ca.uhn.hl7v2.model.Message** object, e.g. to retrieve a patient ID:

```
Message msg = exchange.getIn().getBody(Message.class);
QRD qrd = (QRD)msg.get("QRD");
String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();  // 0101701234
```

This is powerful when combined with the HL7 listener, because you don't have to work with **byte[]**, **String** or any other simple object formats. You can just use the HAPI HL7v2 model objects. If you know the message type in advance, you can be more type-safe:

```
QRY_A19 msg = exchange.getIn().getBody(QRY_A19.class);
String patientId = msg.getQRD().getWhoSubjectFilter(0).getIDNumber().getValue();
```

140.4. HL7 DATAFORMAT

The HL7 component ships with a HL7 data format that can be used to marshal or unmarshal HL7 model objects.

The HL7 dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
validate	**true**	**Boolean**	Whether to validate the HL7 message Is by default true.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

- **marshal** = from Message to byte stream (can be used when responding using the HL7 MLLP codec)
- **unmarshal** = from byte stream to Message (can be used when receiving streamed data from the HL7 MLLP

To use the data format, simply instantiate an instance and invoke the marshal or unmarshal operation in the route builder:

```
DataFormat hl7 = new HL7DataFormat();

from("direct:hl7in")
  .marshal(hl7)
  .to("jms:queue:hl7out");
```

In the sample above, the HL7 is marshalled from a HAPI Message object to a byte stream and put on a JMS queue.
The next example is the opposite:

```
DataFormat hl7 = new HL7DataFormat();

from("jms:queue:hl7out")
  .unmarshal(hl7)
  .to("patientLookupService");
```

Here we unmarshal the byte stream into a HAPI Message object that is passed to our patient lookup service.

140.4.1. Serializable messages

As of HAPI 2.0 (used by **Camel 2.11**), the HL7v2 model classes are fully serializable. So you can put HL7v2 messages directly into a JMS queue (i.e. without calling **marshal()** and read them again directly from the queue (i.e. without calling **unmarshal()**.

140.4.2. Segment separators

As of **Camel 2.11**, **unmarshal** does not automatically fix segment separators anymore by converting **\n** to **\r**. If you
need this conversion, **org.apache.camel.component.hl7.HL7#convertLFToCR** provides a handy **Expression** for this purpose.

140.4.3. Charset

As of **Camel 2.14.1**, both **marshal and unmarshal** evaluate the charset provided in the field **MSH-18**. If this field is empty, by default the charset contained in the corresponding Camel charset property/header is assumed. You can even change this default behavior by overriding the **guessCharsetName** method when inheriting from the **HL7DataFormat** class.

There is a shorthand syntax in Camel for well-known data formats that are commonly used. Then you don't need to create an instance of the **HL7DataFormat** object:

```
from("direct:hl7in")
  .marshal().hl7()
  .to("jms:queue:hl7out");

from("jms:queue:hl7out")
  .unmarshal().hl7()
  .to("patientLookupService");
```

140.5. MESSAGE HEADERS

The unmarshal operation adds these fields from the MSH segment as headers on the Camel message:

Key	MSH field	Example
CamelHL7SendingApplication	**MSH-3**	**MYSERVER**

Key	MSH field	Example
CamelHL7SendingFacility	**MSH-4**	**MYSERVERAPP**
CamelHL7ReceivingApplication	**MSH-5**	**MYCLIENT**
CamelHL7ReceivingFacility	**MSH-6**	**MYCLIENTAPP**
CamelHL7Timestamp	**MSH-7**	**20071231235900**
CamelHL7Security	**MSH-8**	**null**
CamelHL7MessageType	**MSH-9-1**	**ADT**
CamelHL7TriggerEvent	**MSH-9-2**	**A01**
CamelHL7MessageControl	**MSH-10**	**1234**

Key	MSH field	Example
CamelHL7ProcessingId	**MSH-11**	**P**
CamelHL7VersionId	**MSH-12**	**2.4**
`CamelHL7Context	``	` **(Camel 2.14)** contains the HapiContext that was used to parse the message
CamelHL7Charset	**MSH-18**	(Camel 2.14.1) **UNICODE UTF-8**

All headers except **CamelHL7Context `are `String** types. If a header value is missing, its value is **null**.

140.6. OPTIONS

The HL7 Data Format supports the following options:

Option	Default	Description
validate	true	Whether the HAPI Parser should validate the message using the default validation rules. It is recommended to use the **parser** or **hapiContext** option and initialize it with the desired HAPI **ValidationContext**
parser	**ca.uhn.hl7v2.parser.GenericParser**	Custom parser to be used. Must be of type **ca.uhn.hl7v2.parser.Parser**. Note that **GenericParser** also allows to parse XML-encoded HL7v2 messages
hapiContext	**ca.uhn.hl7v2.DefaultHapiContext**	**Camel 2.14:** Custom HAPI context that can define a custom parser, custom ValidationContext etc. This gives you full control over the HL7 parsing and rendering process.

140.7. DEPENDENCIES

To use HL7 in your Camel routes you'll need to add a dependency on **camel-hl7** listed above, which implements this data format.

The HAPI library is split into a base library and several structure libraries, one for each HL7v2 message version:

- v2.1 structures library
- v2.2 structures library
- v2.3 structures library
- v2.3.1 structures library
- v2.4 structures library
- v2.5 structures library
- v2.5.1 structures library
- v2.6 structures library

By default **camel-hl7** only references the HAPI base library. Applications are responsible for including structure libraries themselves. For example, if an application works with HL7v2 message versions 2.4 and 2.5 then the following dependencies must be added:

```
<dependency>
  <groupId>ca.uhn.hapi</groupId>
  <artifactId>hapi-structures-v24</artifactId>
  <version>2.2</version>
  <!-- use the same version as your hapi-base version -->
</dependency>
<dependency>
  <groupId>ca.uhn.hapi</groupId>
  <artifactId>hapi-structures-v25</artifactId>
  <version>2.2</version>
  <!-- use the same version as your hapi-base version -->
</dependency>
```

Alternatively, an OSGi bundle containing the base library, all structures libraries and required dependencies (on the bundle classpath) can be downloaded from the central Maven repository.

```
<dependency>
  <groupId>ca.uhn.hapi</groupId>
  <artifactId>hapi-osgi-base</artifactId>
  <version>2.2</version>
</dependency>
```

140.8. TERSER LANGUAGE

HAPI provides a Terser class that provides access to fields using a commonly used terse location specification syntax. The Terser language allows to use this syntax to extract values from messages and to use them as expressions and predicates for filtering, content-based routing etc.

Sample:

```
import static org.apache.camel.component.hl7.HL7.terser;

  // extract patient ID from field QRD-8 in the QRY_A19 message above and put into message
header
  from("direct:test1")
    .setHeader("PATIENT_ID",terser("QRD-8(0)-1"))
    .to("mock:test1");

  // continue processing if extracted field equals a message header
  from("direct:test2")
    .filter(terser("QRD-8(0)-1").isEqualTo(header("PATIENT_ID"))
    .to("mock:test2");
```

140.9. HL7 VALIDATION PREDICATE

Often it is preferable to first parse a HL7v2 message and in a separate step validate it against a HAPI ValidationContext.

Sample:

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import ca.uhn.hl7v2.validation.impl.DefaultValidation;

  // Use standard or define your own validation rules
  ValidationContext defaultContext = new DefaultValidation();

  // Throws PredicateValidationException if message does not validate
  from("direct:test1")
    .validate(messageConformsTo(defaultContext))
    .to("mock:test1");
```

140.10. HL7 VALIDATION PREDICATE USING THE HAPICONTEXT (CAMEL 2.14)

The HAPI Context is always configured with a ValidationContext (or a ValidationRuleBuilder), so you can access the validation rules indirectly. Furthermore, when unmarshalling the HL7DataFormat forwards the configured HAPI context in the **CamelHL7Context** header, and the validation rules of this context can be easily reused:

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import static org.apache.camel.component.hl7.HL7.messageConforms

  HapiContext hapiContext = new DefaultHapiContext();
  hapiContext.getParserConfiguration().setValidating(false); // don't validate during parsing

  // customize HapiContext some more ... e.g. enforce that PID-8 in ADT_A01 messages of version
2.4 is not empty
  ValidationRuleBuilder builder = new ValidationRuleBuilder() {
    @Override
    protected void configure() {
      forVersion(Version.V24)
          .message("ADT", "A01")
          .terser("PID-8", not(empty()));
```

```
        }
    };
hapiContext.setValidationRuleBuilder(builder);

HL7DataFormat hl7 = new HL7DataFormat();
hl7.setHapiContext(hapiContext);

from("direct:test1")
  .unmarshal(hl7)              // uses the GenericParser returned from the HapiContext
  .validate(messageConforms())   // uses the validation rules returned from the HapiContext
                          // equivalent with .validate(messageConformsTo(hapiContext))
  // route continues from here
```

140.11. HL7 ACKNOWLEDGEMENT EXPRESSION

A common task in HL7v2 processing is to generate an acknowledgement message as response to an incoming HL7v2 message, e.g. based on a validation result. The **ack** expression lets us accomplish this very elegantly:

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import static org.apache.camel.component.hl7.HL7.ack;
import ca.uhn.hl7v2.validation.impl.DefaultValidation;

 // Use standard or define your own validation rules
  ValidationContext defaultContext = new DefaultValidation();

  from("direct:test1")
    .onException(Exception.class)
      .handled(true)
      .transform(ack()) // auto-generates negative ack because of exception in Exchange
      .end()
    .validate(messageConformsTo(defaultContext))
    // do something meaningful here

    // acknowledgement
    .transform(ack())
```

140.12. MORE SAMPLES

In the following example, a plain **String** HL7 request is sent to an HL7 listener that sends back a response:

In the next sample, HL7 requests from the HL7 listener are routed to the business logic, which is implemented as plain POJO registered in the registry as **hl7service**.

Then the Camel routes using the **RouteBuilder** may look as follows:

Note that by using the HL7 DataFormat the Camel message headers are populated with the fields from the MSH segment. The headers are particularly useful for filtering or content-based routing as shown in the example above.

CHAPTER 141. HTTP COMPONENT (DEPRECATED)

Available as of Camel version 1.0

The **http:** component provides HTTP based endpoints for consuming external HTTP resources (as a client to call external servers using HTTP).

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-http</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

141.1. URI FORMAT

```
http:hostname[:port][/resourceUri][?param1=value1][&param2=value2]
```

Will by default use port 80 for HTTP and 443 for HTTPS.

camel-http vs camel-jetty

You can only produce to endpoints generated by the HTTP component. Therefore it should never be used as input into your camel Routes. To bind/expose an HTTP endpoint via a HTTP server as input to a camel route, you can use the Jetty Component or the Servlet Component

141.2. EXAMPLES

Call the url with the body using POST and return response as out message. If body is null call URL using GET and return response as out message

Java DSL

Spring DSL

```
from("direct:start")
  .to("http://myhost/mypath");
```

```
<from uri="direct:start"/>
<to uri="http://oldhost"/>
```

You can override the HTTP endpoint URI by adding a header. Camel will call the http://newhost. This is very handy for e.g. REST urls.

Java DSL

```
from("direct:start")
  .setHeader(Exchange.HTTP_URI, simple("http://myserver/orders/${header.orderId}"))
  .to("http://dummyhost");
```

URI parameters can either be set directly on the endpoint URI or as a header

Java DSL

```
from("direct:start")
  .to("http://oldhost?order=123&detail=short");
from("direct:start")
  .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
  .to("http://oldhost");
```

Set the HTTP request method to POST

Java DSL

Spring DSL

```
from("direct:start")
  .setHeader(Exchange.HTTP_METHOD, constant("POST"))
  .to("http://www.google.com");
```

```
<from uri="direct:start"/>
<setHeader headerName="CamelHttpMethod">
  <constant>POST</constant>
</setHeader>
<to uri="http://www.google.com"/>
<to uri="mock:results"/>
```

141.3. HTTP OPTIONS

The HTTP component supports 8 options which are listed below.

Name	Description	Default	Type
httpClientConfigurer (advanced)	To use the custom HttpClientConfigurer to perform configuration of the HttpClient that will be used.		HttpClientConfigurer
httpConnectionManager (advanced)	To use a custom HttpConnectionManager to manage connections		HttpConnectionManager
httpBinding (producer)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
httpConfiguration (producer)	To use the shared HttpConfiguration as base configuration.		HttpConfiguration
allowJavaSerializedObject (producer)	Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The HTTP endpoint is configured using URI syntax:

```
http:httpUri
```

with the following path and query parameters:

141.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
httpUri	**Required** The url of the HTTP endpoint to call.		URI

141.3.2. Query Parameters (38 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
disableStreamCache (common)	Determines whether or not the raw input stream from Servlet is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Servlet input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times. If you use Servlet to bridge/proxy an endpoint then consider enabling this option to improve performance, in case you do not need to read the message payload multiple times. The http/http4 producer will by default cache the response body stream. If setting this option to true, then the producers will not cache the response body stream but use the response stream as-is as the message body.	false	boolean
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
httpBinding (common)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
bridgeEndpoint (producer)	If the option is true, HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the option throwExceptionOnFailure to be false to let the HttpProducer send all the fault response back.	false	boolean
chunked (producer)	If this option is false the Servlet will disable the HTTP streaming and set the content-length header on the response	true	boolean
connectionClose (producer)	Specifies whether a Connection Close header must be added to HTTP Request. By default connectionClose is false.	false	boolean
copyHeaders (producer)	If this option is true then IN exchange headers will be copied to OUT exchange headers according to copy strategy. Setting this to false, allows to only include the headers from the HTTP response (not propagating IN headers).	true	boolean

Name	Description	Default	Type
httpMethod (producer)	Configure the HTTP method to use. The HttpMethod header cannot override this option if set.		HttpMethods
ignoreResponseBody (producer)	If this option is true, The http producer won't read response body and cache the input stream	false	boolean
preserveHostHeader (producer)	If the option is true, HttpProducer will set the Host header to the value contained in the current exchange Host header, useful in reverse proxy applications where you want the Host header received by the downstream server to reflect the URL called by the upstream client, this allows applications which use the Host header to generate accurate URL's for a proxied service	false	boolean
throwExceptionOnFailure (producer)	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
transferException (producer)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
okStatusCodeRange (producer)	The status codes which are considered a success response. The values are inclusive. Multiple ranges can be defined, separated by comma, e.g. 200-204,209,301-304. Each range must be a single number or from-to with the dash included.	200-299	String
urlRewrite (producer)	**Deprecated** Refers to a custom org.apache.camel.component.http.UrlRewrite which allows you to rewrite urls when you bridge/proxy endpoints. See more details at http://camel.apache.org/urlrewrite.html		UrlRewrite

Name	Description	Default	Type
httpClientConfigurer (advanced)	Register a custom configuration strategy for new HttpClient instances created by producers or consumers such as to configure authentication mechanisms etc		HttpClientConfigurer
httpClientOptions (advanced)	To configure the HttpClient using the key/values from the Map.		Map
httpConnectionManager (advanced)	To use a custom HttpConnectionManager to manage connections		HttpConnectionManager
httpConnectionManager Options (advanced)	To configure the HttpConnectionManager using the key/values from the Map.		Map
mapHttpMessageBody (advanced)	If this option is true then IN exchange Body of the exchange will be mapped to HTTP body. Setting this to false will avoid the HTTP mapping.	true	boolean
mapHttpMessageFormUrlEncodedBody (advanced)	If this option is true then IN exchange Form Encoded body of the exchange will be mapped to HTTP. Setting this to false will avoid the HTTP Form Encoded body mapping.	true	boolean
mapHttpMessageHeaders (advanced)	If this option is true then IN exchange Headers of the exchange will be mapped to HTTP headers. Setting this to false will avoid the HTTP Headers mapping.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
proxyAuthDomain (proxy)	Proxy authentication domain to use with NTML		String
proxyAuthHost (proxy)	Proxy authentication host		String
proxyAuthMethod (proxy)	Proxy authentication method to use		String
proxyAuthPassword (proxy)	Proxy authentication password		String

Name	Description	Default	Type
proxyAuthPort (proxy)	Proxy authentication port		int
proxyAuthScheme (proxy)	Proxy authentication scheme to use		String
proxyAuthUsername (proxy)	Proxy authentication username		String
proxyHost (proxy)	Proxy hostname to use		String
proxyPort (proxy)	Proxy port to use		int
authDomain (security)	Authentication domain to use with NTML		String
authHost (security)	Authentication host to use with NTML		String
authMethod (security)	Authentication methods allowed to use as a comma separated list of values Basic, Digest or NTLM.		String
authMethodPriority (security)	Which authentication method to prioritize to use, either as Basic, Digest or NTLM.		String
authPassword (security)	Authentication password		String
authUsername (security)	Authentication username		String

141.4. MESSAGE HEADERS

Name	Type	Description
Exchange.HTTP_URI	**String**	URI to call. Will override existing URI set directly on the endpoint. This uri is the uri of the http server to call. Its not the same as the Camel endpoint uri, where you can configure endpoint options such as security etc. This header does not support that, its only the uri of the http server.
Exchange.HTTP_METHOD	**String**	HTTP Method / Verb to use (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE)

Name	Type	Description
Exchange.HTTP_PATH	**String**	Request URI's path, the header will be used to build the request URI with the HTTP_URI. **Camel 2.3.0:** If the path is start with "/", http producer will try to find the relative path based on the Exchange.HTTP_BASE_URI header or the exchange.getFromEndpoint().getEndpointUri();
Exchange.HTTP_QUERY	**String**	URI parameters. Will override existing URI parameters set directly on the endpoint.
Exchange.HTTP_RESPONSE_CODE	**int**	The HTTP response code from the external server. Is 200 for OK.
Exchange.HTTP_CHARACTER_ENCODING	**String**	Character encoding.
Exchange.CONTENT_TYPE	**String**	The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as **text/html**.
Exchange.CONTENT_ENCODING	**String**	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as **gzip**.
Exchange.HTTP_SERVLET_REQUEST	**HttpServletRequest**	The **HttpServletRequest** object.

Name	Type	Description
Exchange.HTTP_SERVLET_RESPONSE	**HttpServletResponse**	The **HttpServletResponse** object.
Exchange.HTTP_PROTOCOL_VERSION	**String**	**Camel 2.5:** You can set the http protocol version with this header, eg. "HTTP/1.0". If you didn't specify the header, HttpProducer will use the default value "HTTP/1.1"

The header name above are constants. For the spring DSL you have to use the value of the constant instead of the name.

141.5. MESSAGE BODY

Camel will store the HTTP response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so headers are preserved during routing. Additionally Camel will add the HTTP response headers as well to the OUT message headers.

141.6. RESPONSE CODE

Camel will handle according to the HTTP response code:

- Response code is in the range 100..299, Camel regards it as a success response.
- Response code is in the range 300..399, Camel regards it as a redirection response and will throw a **HttpOperationFailedException** with the information.
- Response code is 400+, Camel regards it as an external server failure and will throw a **HttpOperationFailedException** with the information.

throwExceptionOnFailure

The option, **throwExceptionOnFailure**, can be set to **false** to prevent the **HttpOperationFailedException** from being thrown for failed response codes. This allows you to get any response from the remote server.
There is a sample below demonstrating this.

141.7. HTTPOPERATIONFAILEDEXCEPTION

This exception contains the following information:

- The HTTP status code

- The HTTP status line (text of the status code)
- Redirect location, if server returned a redirect
- Response body as a **java.lang.String**, if server provided a body as response

141.8. WHICH HTTP METHOD WILL BE USED

The following algorithm is used to determine what HTTP method should be used:
1. Use method provided as endpoint configuration (**httpMethod**).
2. Use method provided in header (**Exchange.HTTP_METHOD**).
3. **GET** if query string is provided in header.
4. **GET** if endpoint is configured with a query string.
5. **POST** if there is data to send (body is not **null**).
6. **GET** otherwise.

141.9. HOW TO GET ACCESS TO HTTPSERVLETREQUEST AND HTTPSERVLETRESPONSE

You can get access to these two using the Camel type converter system using

```
HttpServletRequest request = exchange.getIn().getBody(HttpServletRequest.class);
HttpServletRequest response = exchange.getIn().getBody(HttpServletResponse.class);
```

141.10. USING CLIENT TIMEOUT - SO_TIMEOUT

See the unit test in this link

141.11. MORE EXAMPLES

141.11.1. Configuring a Proxy

Java DSL

```
from("direct:start")
  .to("http://oldhost?proxyHost=www.myproxy.com&proxyPort=80");
```

There is also support for proxy authentication via the **proxyUsername** and **proxyPassword** options.

141.11.2. Using proxy settings outside of URI

Java DSL

Spring DSL

```
context.getProperties().put("http.proxyHost", "172.168.18.9");
context.getProperties().put("http.proxyPort" "8080");
```

```
<camelContext>
  <properties>
    <property key="http.proxyHost" value="172.168.18.9"/>
```

```
        <property key="http.proxyPort" value="8080"/>
    </properties>
</camelContext>
```

Options on Endpoint will override options on the context.

141.12. CONFIGURING CHARSET

If you are using **POST** to send data you can configure the **charset**

```
setProperty(Exchange.CHARSET_NAME, "iso-8859-1");
```

141.13. SAMPLE WITH SCHEDULED POLL

The sample polls the Google homepage every 10 seconds and write the page to the file **message.html**:

```
from("timer://foo?fixedRate=true&delay=0&period=10000")
    .to("http://www.google.com")
    .setHeader(FileComponent.HEADER_FILE_NAME, "message.html").to("file:target/google");
```

141.14. GETTING THE RESPONSE CODE

You can get the HTTP response code from the HTTP component by getting the value from the Out message header with **Exchange.HTTP_RESPONSE_CODE**.

```
Exchange exchange = template.send("http://www.google.com/search", new Processor() {
        public void process(Exchange exchange) throws Exception {
            exchange.getIn().setHeader(Exchange.HTTP_QUERY, constant("hl=en&q=activemq"));
        }
});
Message out = exchange.getOut();
int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

141.15. USING THROWEXCEPTIONONFAILURE=FALSE TO GET ANY RESPONSE BACK

In the route below we want to route a message that we enrich with data returned from a remote HTTP call. As we want any response from the remote server, we set the **throwExceptionOnFailure** option to **false** so we get any response in the **AggregationStrategy**. As the code is based on a unit test that simulates a HTTP status code 404, there is some assertion code etc.

141.16. DISABLING COOKIES

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option: **httpClient.cookiePolicy=ignoreCookies**

141.17. ADVANCED USAGE

If you need more control over the HTTP producer you should use the **HttpComponent** where you can set various classes to give you custom behavior.

141.17.1. Setting MaxConnectionsPerHost

The HTTP Component has a **org.apache.commons.httpclient.HttpConnectionManager** where you can configure various global configuration for the given component.
By global, we mean that any endpoint the component creates has the same shared **HttpConnectionManager**. So, if we want to set a different value for the max connection per host, we need to define it on the HTTP component and **not** on the endpoint URI that we usually use. So here comes:

First, we define the **http** component in Spring XML. Yes, we use the same scheme name, **http**, because otherwise Camel will auto-discover and create the component with default settings. What we need is to overrule this so we can set our options. In the sample below we set the max connection to 5 instead of the default of 2.

And then we can just use it as we normally do in our routes:

141.17.2. Using preemptive authentication

An end user reported that he had problem with authenticating with HTTPS. The problem was eventually resolved when he discovered the HTTPS server did not return a HTTP code 401 Authorization Required. The solution was to set the following URI option: **httpClient.authenticationPreemptive=true**

141.17.3. Accepting self signed certificates from remote server

See this link from a mailing list discussion with some code to outline how to do this with the Apache Commons HTTP API.

141.17.4. Setting up SSL for HTTP Client

Using the JSSE Configuration Utility

As of Camel 2.8, the HTTP4 component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the HTTP4 component.

The version of the Apache HTTP client used in this component resolves SSL/TLS information from a global "protocol" registry. This component provides an implementation, **org.apache.camel.component.http.SSLContextParametersSecureProtocolSocketFactory**, of the HTTP client's protocol socket factory in order to support the use of the Camel JSSE Configuration utility. The following example demonstrates how to configure the protocol registry and use the registered protocol information in a route.

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

ProtocolSocketFactory factory =
```

```
    new SSLContextParametersSecureProtocolSocketFactory(scp);

Protocol.registerProtocol("https",
        new Protocol(
        "https",
        factory,
        443));

from("direct:start")
        .to("https://mail.google.com/mail/").to("mock:results");
```

Configuring Apache HTTP Client Directly

Basically camel-http component is built on the top of Apache HTTP client, and you can implement a custom **org.apache.camel.component.http.HttpClientConfigurer** to do some configuration on the http client if you need full control of it.

However if you *just* want to specify the keystore and truststore you can do this with Apache HTTP **HttpClientConfigurer**, for example:

```
Protocol authhttps = new Protocol("https", new AuthSSLProtocolSocketFactory(
  new URL("file:my.keystore"), "mypassword",
  new URL("file:my.truststore"), "mypassword"), 443);

Protocol.registerProtocol("https", authhttps);
```

And then you need to create a class that implements **HttpClientConfigurer**, and registers https protocol providing a keystore or truststore per example above. Then, from your camel route builder class you can hook it up like so:

```
HttpComponent httpComponent = getContext().getComponent("http", HttpComponent.class);
httpComponent.setHttpClientConfigurer(new MyHttpClientConfigurer());
```

If you are doing this using the Spring DSL, you can specify your **HttpClientConfigurer** using the URI. For example:

```
<bean id="myHttpClientConfigurer"
 class="my.https.HttpClientConfigurer">
</bean>

<to uri="https://myhostname.com:443/myURL?httpClientConfigurerRef=myHttpClientConfigurer"/>
```

As long as you implement the HttpClientConfigurer and configure your keystore and truststore as described above, it will work fine.

141.18. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

- Jetty

CHAPTER 142. HTTP4 COMPONENT

Available as of Camel version 2.3

The **http4:** component provides HTTP based endpoints for calling external HTTP resources (as a client to call external servers using HTTP).

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-http4</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

camel-http4 vs camel-http

Camel-http4 uses Apache HttpClient 4.x while camel-http uses Apache HttpClient 3.x.

142.1. URI FORMAT

```
http4:hostname[:port][/resourceUri][?options]
```

Will by default use port 80 for HTTP and 443 for HTTPS.

You can append query options to the URI in the following format, **?option=value&option=value&...**

camel-http4 vs camel-jetty

You can only produce to endpoints generated by the HTTP4 component. Therefore it should never be used as input into your Camel Routes. To bind/expose an HTTP endpoint via a HTTP server as input to a Camel route, use the Jetty Component instead.

142.2. HTTP4 COMPONENT OPTIONS

The HTTP4 component supports 18 options which are listed below.

Name	Description	Default	Type
httpClientConfigurer (advanced)	To use the custom HttpClientConfigurer to perform configuration of the HttpClient that will be used.		HttpClientConfigurer
clientConnectionManager (advanced)	To use a custom and shared HttpClientConnectionManager to manage connections. If this has been configured then this is always used for all endpoints created by this component.		HttpClientConnection Manager

Name	Description	Default	Type
httpContext (advanced)	To use a custom org.apache.http.protocol.HttpContext when executing requests.		HttpContext
sslContextParameters (security)	To configure security using SSLContextParameters. Important: Only one instance of org.apache.camel.util.jsse.SSLContextParameters is supported per HttpComponent. If you need to use 2 or more different instances, you need to define a new HttpComponent per instance you need.		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
x509HostnameVerifier (security)	To use a custom X509HostnameVerifier such as DefaultHostnameVerifier or org.apache.http.conn.ssl.NoopHostnameVerifier.		HostnameVerifier
maxTotalConnections (advanced)	The maximum number of connections.	200	int
connectionsPerRoute (advanced)	The maximum number of connections per route.	20	int
connectionTimeToLive (advanced)	The time for connection to live, the time unit is millisecond, the default value is always keep alive.		long
cookieStore (producer)	To use a custom org.apache.http.client.CookieStore. By default the org.apache.http.impl.client.BasicCookieStore is used which is an in-memory only cookie store. Notice if bridgeEndpoint=true then the cookie store is forced to be a noop cookie store as cookie shouldn't be stored as we are just bridging (eg acting as a proxy).		CookieStore
connectionRequest Timeout (timeout)	The timeout in milliseconds used when requesting a connection from the connection manager. A timeout value of zero is interpreted as an infinite timeout. A timeout value of zero is interpreted as an infinite timeout. A negative value is interpreted as undefined (system default). Default: code -1	-1	int

Name	Description	Default	Type
connectTimeout (timeout)	Determines the timeout in milliseconds until a connection is established. A timeout value of zero is interpreted as an infinite timeout. A timeout value of zero is interpreted as an infinite timeout. A negative value is interpreted as undefined (system default). Default: code -1	-1	int
socketTimeout (timeout)	Defines the socket timeout (SO_TIMEOUT) in milliseconds, which is the timeout for waiting for data or, put differently, a maximum period inactivity between two consecutive data packets). A timeout value of zero is interpreted as an infinite timeout. A negative value is interpreted as undefined (system default). Default: code -1	-1	int
httpBinding (advanced)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
httpConfiguration (advanced)	To use the shared HttpConfiguration as base configuration.		HttpConfiguration
allowJavaSerialized Object (advanced)	Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The HTTP4 endpoint is configured using URI syntax:

```
http4:httpUri
```

with the following path and query parameters:

142.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
httpUri	**Required** The url of the HTTP endpoint to call.		URI

142.2.2. Query Parameters (48 parameters):

Name	Description	Default	Type
disableStreamCache (common)	Determines whether or not the raw input stream from Servlet is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Servlet input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times. If you use Servlet to bridge/proxy an endpoint then consider enabling this option to improve performance, in case you do not need to read the message payload multiple times. The http/http4 producer will by default cache the response body stream. If setting this option to true, then the producers will not cache the response body stream but use the response stream as-is as the message body.	false	boolean
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
httpBinding (common)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
authenticationPreemptive (producer)	If this option is true, camel-http4 sends preemptive basic authentication to the server.	false	boolean
bridgeEndpoint (producer)	If the option is true, HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the option throwExceptionOnFailure to be false to let the HttpProducer send all the fault response back.	false	boolean
chunked (producer)	If this option is false the Servlet will disable the HTTP streaming and set the content-length header on the response	true	boolean

Name	Description	Default	Type
clearExpiredCookies (producer)	Whether to clear expired cookies before sending the HTTP request. This ensures the cookies store does not keep growing by adding new cookies which is newer removed when they are expired.	true	boolean
connectionClose (producer)	Specifies whether a Connection Close header must be added to HTTP Request. By default connectionClose is false.	false	boolean
cookieStore (producer)	To use a custom CookieStore. By default the BasicCookieStore is used which is an in-memory only cookie store. Notice if bridgeEndpoint=true then the cookie store is forced to be a noop cookie store as cookie shouldn't be stored as we are just bridging (eg acting as a proxy). If a cookieHandler is set then the cookie store is also forced to be a noop cookie store as cookie handling is then performed by the cookieHandler.		CookieStore
copyHeaders (producer)	If this option is true then IN exchange headers will be copied to OUT exchange headers according to copy strategy. Setting this to false, allows to only include the headers from the HTTP response (not propagating IN headers).	true	boolean
deleteWithBody (producer)	Whether the HTTP DELETE should include the message body or not. By default HTTP DELETE do not include any HTTP message. However in some rare cases users may need to be able to include the message body.	false	boolean
httpMethod (producer)	Configure the HTTP method to use. The HttpMethod header cannot override this option if set.		HttpMethods
ignoreResponseBody (producer)	If this option is true, The http producer won't read response body and cache the input stream	false	boolean
preserveHostHeader (producer)	If the option is true, HttpProducer will set the Host header to the value contained in the current exchange Host header, useful in reverse proxy applications where you want the Host header received by the downstream server to reflect the URL called by the upstream client, this allows applications which use the Host header to generate accurate URL's for a proxied service	false	boolean

Name	Description	Default	Type
throwExceptionOnFailure (producer)	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
transferException (producer)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
okStatusCodeRange (producer)	The status codes which are considered a success response. The values are inclusive. Multiple ranges can be defined, separated by comma, e.g. 200-204,209,301-304. Each range must be a single number or from-to with the dash included.	200-299	String
urlRewrite (producer)	**Deprecated** Refers to a custom org.apache.camel.component.http.UrlRewrite which allows you to rewrite urls when you bridge/proxy endpoints. See more details at http://camel.apache.org/urlrewrite.html		UrlRewrite
clientBuilder (advanced)	Provide access to the http client request parameters used on new RequestConfig instances used by producers or consumers of this endpoint.		HttpClientBuilder
clientConnectionManager (advanced)	To use a custom HttpClientConnectionManager to manage connections		HttpClientConnection Manager
connectionsPerRoute (advanced)	The maximum number of connections per route.	20	int
httpClient (advanced)	Sets a custom HttpClient to be used by the producer		HttpClient

Name	Description	Default	Type
httpClientConfigurer (advanced)	Register a custom configuration strategy for new HttpClient instances created by producers or consumers such as to configure authentication mechanisms etc		HttpClientConfigurer
httpClientOptions (advanced)	To configure the HttpClient using the key/values from the Map.		Map
httpContext (advanced)	To use a custom HttpContext instance		HttpContext
mapHttpMessageBody (advanced)	If this option is true then IN exchange Body of the exchange will be mapped to HTTP body. Setting this to false will avoid the HTTP mapping.	true	boolean
mapHttpMessageFormUrlEncodedBody (advanced)	If this option is true then IN exchange Form Encoded body of the exchange will be mapped to HTTP. Setting this to false will avoid the HTTP Form Encoded body mapping.	true	boolean
mapHttpMessageHeaders (advanced)	If this option is true then IN exchange Headers of the exchange will be mapped to HTTP headers. Setting this to false will avoid the HTTP Headers mapping.	true	boolean
maxTotalConnections (advanced)	The maximum number of connections.	200	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
useSystemProperties (advanced)	To use System Properties as fallback for configuration	false	boolean
proxyAuthDomain (proxy)	Proxy authentication domain to use with NTML		String
proxyAuthHost (proxy)	Proxy authentication host		String
proxyAuthMethod (proxy)	Proxy authentication method to use		String

Name	Description	Default	Type
proxyAuthPassword (proxy)	Proxy authentication password		String
proxyAuthPort (proxy)	Proxy authentication port		int
proxyAuthScheme (proxy)	Proxy authentication scheme to use		String
proxyAuthUsername (proxy)	Proxy authentication username		String
proxyHost (proxy)	Proxy hostname to use		String
proxyPort (proxy)	Proxy port to use		int
authDomain (security)	Authentication domain to use with NTML		String
authHost (security)	Authentication host to use with NTML		String
authMethod (security)	Authentication methods allowed to use as a comma separated list of values Basic, Digest or NTLM.		String
authMethodPriority (security)	Which authentication method to prioritize to use, either as Basic, Digest or NTLM.		String
authPassword (security)	Authentication password		String
authUsername (security)	Authentication username		String
x509HostnameVerifier (security)	To use a custom X509HostnameVerifier such as DefaultHostnameVerifier or org.apache.http.conn.ssl.NoopHostnameVerifier.		HostnameVerifier

142.3. MESSAGE HEADERS

Name	Type	Description
Exchange.HTTP_URI	**String**	URI to call. Will override existing URI set directly on the endpoint. This uri is the uri of the http server to call. Its not the same as the Camel endpoint uri, where you can configure endpoint options such as security etc. This header does not support that, its only the uri of the http server.
Exchange.HTTP_PATH	**String**	Request URI's path, the header will be used to build the request URI with the HTTP_URI.
Exchange.HTTP_QUERY	**String**	URI parameters. Will override existing URI parameters set directly on the endpoint.
Exchange.HTTP_RESPONSE_CODE	**int**	The HTTP response code from the external server. Is 200 for OK.
Exchange.HTTP_RESPONSE_TEXT	**String**	The HTTP response text from the external server.
Exchange.HTTP_CHARACTER_ENCODING	**String**	Character encoding.
Exchange.CONTENT_TYPE	**String**	The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as **text/html**.

Name	Type	Description
Exchange.CONTENT_ENCODING	**String**	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as **gzip**.

142.4. MESSAGE BODY

Camel will store the HTTP response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so headers are preserved during routing. Additionally Camel will add the HTTP response headers as well to the OUT message headers.

142.5. USING SYSTEM PROPERTIES

When setting useSystemProperties to true, the HTTP Client will look for the following System Properties and it will use it:

- ssl.TrustManagerFactory.algorithm
- javax.net.ssl.trustStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStoreProvider
- javax.net.ssl.trustStorePassword
- java.home
- ssl.KeyManagerFactory.algorithm
- javax.net.ssl.keyStoreType
- javax.net.ssl.keyStore
- javax.net.ssl.keyStoreProvider
- javax.net.ssl.keyStorePassword
- http.proxyHost
- http.proxyPort
- http.nonProxyHosts
- http.keepAlive
- http.maxConnections

142.6. RESPONSE CODE

Camel will handle according to the HTTP response code:

- Response code is in the range 100..299, Camel regards it as a success response.
- Response code is in the range 300..399, Camel regards it as a redirection response and will throw a **HttpOperationFailedException** with the information.
- Response code is 400+, Camel regards it as an external server failure and will throw a **HttpOperationFailedException** with the information.

throwExceptionOnFailure The option, **throwExceptionOnFailure**, can be set to **false** to prevent the **HttpOperationFailedException** from being thrown for failed response codes. This allows you to get any response from the remote server.
There is a sample below demonstrating this.

142.7. HTTPOPERATIONFAILEDEXCEPTION

This exception contains the following information:

- The HTTP status code
- The HTTP status line (text of the status code)
- Redirect location, if server returned a redirect
- Response body as a **java.lang.String**, if server provided a body as response

142.8. WHICH HTTP METHOD WILL BE USED

The following algorithm is used to determine what HTTP method should be used:
1. Use method provided as endpoint configuration (**httpMethod**).
2. Use method provided in header (**Exchange.HTTP_METHOD**).
3. **GET** if query string is provided in header.
4. **GET** if endpoint is configured with a query string.
5. **POST** if there is data to send (body is not **null**).
6. **GET** otherwise.

142.9. HOW TO GET ACCESS TO HTTPSERVLETREQUEST AND HTTPSERVLETRESPONSE

You can get access to these two using the Camel type converter system using
NOTE You can get the request and response not just from the processor after the camel-jetty or camel-cxf endpoint.

```
HttpServletRequest request = exchange.getIn().getBody(HttpServletRequest.class);
HttpServletRequest response = exchange.getIn().getBody(HttpServletResponse.class);
```

142.10. CONFIGURING URI TO CALL

You can set the HTTP producer's URI directly form the endpoint URI. In the route below, Camel will call out to the external server, **oldhost**, using HTTP.

```
from("direct:start")
        .to("http4://oldhost");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="http4://oldhost"/>
  </route>
</camelContext>
```

You can override the HTTP endpoint URI by adding a header with the key, **Exchange.HTTP_URI**, on the message.

```
from("direct:start")
  .setHeader(Exchange.HTTP_URI, constant("http://newhost"))
  .to("http4://oldhost");
```

In the sample above Camel will call the http://newhost despite the endpoint is configured with http4://oldhost.
If the http4 endpoint is working in bridge mode, it will ignore the message header of **Exchange.HTTP_URI**.

142.11. CONFIGURING URI PARAMETERS

The **http** producer supports URI parameters to be sent to the HTTP server. The URI parameters can either be set directly on the endpoint URI or as a header with the key **Exchange.HTTP_QUERY** on the message.

```
from("direct:start")
  .to("http4://oldhost?order=123&detail=short");
```

Or options provided in a header:

```
from("direct:start")
  .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
  .to("http4://oldhost");
```

142.12. HOW TO SET THE HTTP METHOD (GET/PATCH/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) TO THE HTTP PRODUCER

Using the http PATCH method

The http PATCH method is supported starting with Camel 2.11.3 / 2.12.1.

The HTTP4 component provides a way to set the HTTP request method by setting the message header. Here is an example:

```
from("direct:start")
  .setHeader(Exchange.HTTP_METHOD,
```

```
constant(org.apache.camel.component.http4.HttpMethods.POST))
  .to("http4://www.google.com")
  .to("mock:results");
```

The method can be written a bit shorter using the string constants:

```
.setHeader("CamelHttpMethod", constant("POST"))
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <setHeader headerName="CamelHttpMethod">
        <constant>POST</constant>
    </setHeader>
    <to uri="http4://www.google.com"/>
    <to uri="mock:results"/>
  </route>
</camelContext>
```

142.13. USING CLIENT TIMEOUT - SO_TIMEOUT

See the HttpSOTimeoutTest unit test.

Since Camel 2.13.0: See the updated HttpSOTimeoutTest unit test.

142.14. CONFIGURING A PROXY

The HTTP4 component provides a way to configure a proxy.

```
from("direct:start")
  .to("http4://oldhost?proxyAuthHost=www.myproxy.com&proxyAuthPort=80");
```

There is also support for proxy authentication via the **proxyAuthUsername** and **proxyAuthPassword** options.

142.14.1. Using proxy settings outside of URI

To avoid System properties conflicts, you can set proxy configuration only from the CamelContext or URI.
Java DSL :

```
context.getProperties().put("http.proxyHost", "172.168.18.9");
context.getProperties().put("http.proxyPort" "8080");
```

Spring XML

```
<camelContext>
   <properties>
      <property key="http.proxyHost" value="172.168.18.9"/>
```

```
        <property key="http.proxyPort" value="8080"/>
    </properties>
</camelContext>
```

Camel will first set the settings from Java System or CamelContext Properties and then the endpoint proxy options if provided.
So you can override the system properties with the endpoint options.

Notice in **Camel 2.8** there is also a **http.proxyScheme** property you can set to explicit configure the scheme to use.

142.15. CONFIGURING CHARSET

If you are using **POST** to send data you can configure the **charset** using the **Exchange** property:

```
exchange.setProperty(Exchange.CHARSET_NAME, "ISO-8859-1");
```

142.15.1. Sample with scheduled poll

This sample polls the Google homepage every 10 seconds and write the page to the file **message.html**:

```
from("timer://foo?fixedRate=true&delay=0&period=10000")
  .to("http4://www.google.com")
  .setHeader(FileComponent.HEADER_FILE_NAME, "message.html")
  .to("file:target/google");
```

142.15.2. URI Parameters from the endpoint URI

In this sample we have the complete URI endpoint that is just what you would have typed in a web browser. Multiple URI parameters can of course be set using the **&** character as separator, just as you would in the web browser. Camel does no tricks here.

```
// we query for Camel at the Google page
template.sendBody("http4://www.google.com/search?q=Camel", null);
```

142.15.3. URI Parameters from the Message

```
Map headers = new HashMap();
headers.put(Exchange.HTTP_QUERY, "q=Camel&lr=lang_en");
// we query for Camel and English language at Google
template.sendBody("http4://www.google.com/search", null, headers);
```

In the header value above notice that it should **not** be prefixed with **?** and you can separate parameters as usual with the **&** char.

142.15.4. Getting the Response Code

You can get the HTTP response code from the HTTP4 component by getting the value from the Out message header with **Exchange.HTTP_RESPONSE_CODE**.

```
Exchange exchange = template.send("http4://www.google.com/search", new Processor() {
```

```
  public void process(Exchange exchange) throws Exception {
    exchange.getIn().setHeader(Exchange.HTTP_QUERY, constant("hl=en&q=activemq"));
  }
});
Message out = exchange.getOut();
int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

142.16. DISABLING COOKIES

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option:
httpClient.cookieSpec=ignoreCookies

142.17. ADVANCED USAGE

If you need more control over the HTTP producer you should use the **HttpComponent** where you can set various classes to give you custom behavior.

142.17.1. Setting up SSL for HTTP Client

Using the JSSE Configuration Utility

As of Camel 2.8, the HTTP4 component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the HTTP4 component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

HttpComponent httpComponent = getContext().getComponent("https4", HttpComponent.class);
httpComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:keyManagers
        keyPassword="keyPassword">
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
```

```
  </camel:sslContextParameters>...
...
  <to uri="https4://127.0.0.1/mail/?sslContextParameters=#sslContextParameters"/>...
```

Configuring Apache HTTP Client Directly

Basically camel-http4 component is built on the top of Apache HttpClient. Please refer to SSL/TLS customization for details or have a look into the **org.apache.camel.component.http4.HttpsServerTestSupport** unit test base class.
You can also implement a custom **org.apache.camel.component.http4.HttpClientConfigurer** to do some configuration on the http client if you need full control of it.

However if you *just* want to specify the keystore and truststore you can do this with Apache HTTP **HttpClientConfigurer**, for example:

```
KeyStore keystore = ...;
KeyStore truststore = ...;

SchemeRegistry registry = new SchemeRegistry();
registry.register(new Scheme("https", 443, new SSLSocketFactory(keystore, "mypassword",
truststore)));
```

And then you need to create a class that implements **HttpClientConfigurer**, and registers https protocol providing a keystore or truststore per example above. Then, from your camel route builder class you can hook it up like so:

```
HttpComponent httpComponent = getContext().getComponent("http4", HttpComponent.class);
httpComponent.setHttpClientConfigurer(new MyHttpClientConfigurer());
```

If you are doing this using the Spring DSL, you can specify your **HttpClientConfigurer** using the URI. For example:

```
<bean id="myHttpClientConfigurer"
 class="my.https.HttpClientConfigurer">
</bean>

<to uri="https4://myhostname.com:443/myURL?httpClientConfigurer=myHttpClientConfigurer"/>
```

As long as you implement the HttpClientConfigurer and configure your keystore and truststore as described above, it will work fine.

Using HTTPS to authenticate gotchas

An end user reported that he had problem with authenticating with HTTPS. The problem was eventually resolved by providing a custom configured **org.apache.http.protocol.HttpContext**:

- 1. Create a (Spring) factory for HttpContexts:

```
public class HttpContextFactory {

  private String httpHost = "localhost";
  private String httpPort = 9001;

  private BasicHttpContext httpContext = new BasicHttpContext();
  private BasicAuthCache authCache = new BasicAuthCache();
```

```
  private BasicScheme basicAuth = new BasicScheme();

  public HttpContext getObject() {
    authCache.put(new HttpHost(httpHost, httpPort), basicAuth);

    httpContext.setAttribute(ClientContext.AUTH_CACHE, authCache);

    return httpContext;
  }

  // getter and setter
}
```

- 2. Declare an HttpContext in the Spring application context file:

```
<bean id="myHttpContext" factory-bean="httpContextFactory" factory-method="getObject"/>
```

- 3. Reference the context in the http4 URL:

```
<to uri="https4://myhostname.com:443/myURL?httpContext=myHttpContext"/>
```

Using different SSLContextParameters

The HTTP4 component only support one instance of **org.apache.camel.util.jsse.SSLContextParameters** per component. If you need to use 2 or more different instances, then you need to setup multiple HTTP4 components as shown below. Where we have 2 components, each using their own instance of **sslContextParameters** property.

```
<bean id="http4-foo" class="org.apache.camel.component.http4.HttpComponent">
  <property name="sslContextParameters" ref="sslContextParams1"/>
  <property name="x509HostnameVerifier" ref="hostnameVerifier"/>
</bean>

<bean id="http4-bar" class="org.apache.camel.component.http4.HttpComponent">
  <property name="sslContextParameters" ref="sslContextParams2"/>
  <property name="x509HostnameVerifier" ref="hostnameVerifier"/>
</bean>
```

CHAPTER 143. HYSTRIX COMPONENT

Available as of Camel version 2.18

The hystrix component integrates Netflix Hystrix circuit breaker in Camel routes.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hystrix</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

For more information see the Hystrix EIP

CHAPTER 144. ICAL DATAFORMAT

Available as of Camel version 2.12

The **ICal** dataformat is used for working with iCalendar messages.

A typical iCalendar message looks like:

```
BEGIN:VCALENDAR
VERSION:2.0
PRODID:-//Events Calendar//iCal4j 1.0//EN
CALSCALE:GREGORIAN
BEGIN:VEVENT
DTSTAMP:20130324T180000Z
DTSTART:20130401T170000
DTEND:20130401T210000
SUMMARY:Progress Meeting
TZID:America/New_York
UID:00000000
ATTENDEE;ROLE=REQ-PARTICIPANT;CN=Developer 1:mailto:dev1@mycompany.com
ATTENDEE;ROLE=OPT-PARTICIPANT;CN=Developer 2:mailto:dev2@mycompany.com
END:VEVENT
END:VCALENDAR
```

144.1. OPTIONS

The iCal dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
validating	**false**	**Boolean**	Whether to validate.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

144.2. BASIC USAGE

To unmarshal and marshal the message shown above, your route will look like the following:

```
from("direct:ical-unmarshal")
  .unmarshal("ical")
  .to("mock:unmarshaled")
  .marshal("ical")
  .to("mock:marshaled");
```

Maven users will need to add the following dependency to their **pom.xml** for this component:

-

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ical</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

144.3. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 145. IEC 60870 CLIENT COMPONENT

Available as of Camel version 2.20

The **IEC 60870-5-104 Client** component provides access to IEC 60870 servers using the Eclipse NeoSCADA™ implementation.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-iec60870</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

The IEC 60870 Client component supports 2 options which are listed below.

Name	Description	Default	Type
defaultConnection Options (common)	Default connection options		ClientOptions
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

145.1. URI FORMAT

The URI syntax of the endpoint is:

```
iec60870-client:host:port/00-01-02-03-04
```

The information object address is encoded in the path in the syntax shows above. Please note that always the full, 5 octet address format is being used. Unused octets have to be filled with zero.

145.2. URI OPTIONS

The IEC 60870 Client endpoint is configured using URI syntax:

```
iec60870-client:uriPath
```

with the following path and query parameters:

145.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
uriPath	**Required** The object information address		ObjectAddress

145.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
dataModuleOptions (common)	Data module options		DataModuleOptions
protocolOptions (common)	Protocol options		ProtocolOptions
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
acknowledgeWindow (connection)	Parameter W - Acknowledgment window.	10	short
adsuAddressType (connection)	The common ASDU address size. May be either SIZE_1 or SIZE_2.		ASDUAddressType
causeOfTransmissionType (connection)	The cause of transmission type. May be either SIZE_1 or SIZE_2.		CauseOfTransmission Type

Name	Description	Default	Type
informationObjectAddress Type (connection)	The information address size. May be either SIZE_1, SIZE_2 or SIZE_3.		InformationObjectAddressType
maxUnacknowledged (connection)	Parameter K - Maximum number of un-acknowledged messages.	15	short
timeout1 (connection)	Timeout T1 in milliseconds.	15000	int
timeout2 (connection)	Timeout T2 in milliseconds.	10000	int
timeout3 (connection)	Timeout T3 in milliseconds.	20000	int
ignoreBackgroundScan (data)	Whether background scan transmissions should be ignored.	true	boolean
ignoreDaylightSavingTime (data)	Whether to ignore or respect DST	false	boolean
timeZone (data)	The timezone to use. May be any Java time zone string	UTC	TimeZone
connectionId (id)	An identifier grouping connection instances		String

A connection instance if identified by the host and port part of the URI, plus all parameters in the "id" group. If a new connection id is encountered the connection options will be evaluated and the connection instance is created with those options.

NOTE

If two URIs specify the same connection (host, port, ...) but different connection options, then it is undefined which of those connection options will be used.

The final connection options will be evaluated in the following order:

- If present, the connectionOptions parameter will be used
- Otherwise the defaultConnectionOptions instance is copied and customized in the following steps
- Apply protocolOptions if present
- Apply dataModuleOptions if present

- Apply all explicit connection parameters (e.g. timeZone)

CHAPTER 146. IEC 60870 SERVER COMPONENT

Available as of Camel version 2.20

The **IEC 60870-5-104 Server** component provides access to IEC 60870 servers using the Eclipse NeoSCADA™ implementation.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-iec60870</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

The IEC 60870 Server component supports 2 options which are listed below.

Name	Description	Default	Type
defaultConnection Options (common)	Default connection options		ServerOptions
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

146.1. URI FORMAT

The URI syntax of the endpoint is:

```
iec60870-server:host:port/00-01-02-03-04
```

The information object address is encoded in the path in the syntax shows above. Please note that always the full, 5 octet address format is being used. Unused octets have to be filled with zero.

146.2. URI OPTIONS

The IEC 60870 Server endpoint is configured using URI syntax:

```
iec60870-server:uriPath
```

with the following path and query parameters:

146.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
uriPath	**Required** The object information address		ObjectAddress

146.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
dataModuleOptions (common)	Data module options		DataModuleOptions
filterNonExecute (common)	Filter out all requests which don't have the execute bit set	true	boolean
protocolOptions (common)	Protocol options		ProtocolOptions
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
acknowledgeWindow (connection)	Parameter W - Acknowledgment window.	10	short
adsuAddressType (connection)	The common ASDU address size. May be either SIZE_1 or SIZE_2.		ASDUAddressType

Name	Description	Default	Type
causeOfTransmissionType (connection)	The cause of transmission type. May be either SIZE_1 or SIZE_2.		CauseOfTransmission Type
informationObjectAddress Type (connection)	The information address size. May be either SIZE_1, SIZE_2 or SIZE_3.		InformationObjectAddressType
maxUnacknowledged (connection)	Parameter K - Maximum number of un-acknowledged messages.	15	short
timeout1 (connection)	Timeout T1 in milliseconds.	15000	int
timeout2 (connection)	Timeout T2 in milliseconds.	10000	int
timeout3 (connection)	Timeout T3 in milliseconds.	20000	int
ignoreBackgroundScan (data)	Whether background scan transmissions should be ignored.	true	boolean
ignoreDaylightSavingTime (data)	Whether to ignore or respect DST	false	boolean
timeZone (data)	The timezone to use. May be any Java time zone string	UTC	TimeZone
connectionId (id)	An identifier grouping connection instances		String

CHAPTER 147. IGNITE CACHE COMPONENT

Available as of Camel version 2.17

The Ignite Cache endpoint is one of camel-ignite endpoints which allows you to interact with an Ignite Cache. This offers both a Producer (to invoke cache operations on an Ignite cache) and a Consumer (to consume changes from a continuous query).

The cache value is always the body of the message, whereas the cache key is always stored in the **IgniteConstants.IGNITE_CACHE_KEY** message header.

Even if you configure a fixed operation in the endpoint URI, you can vary it per-exchange by setting the **IgniteConstants.IGNITE_CACHE_OPERATION** message header.

147.1. OPTIONS

The Ignite Cache component supports 4 options which are listed below.

Name	Description	Default	Type
ignite (common)	Sets the Ignite instance.		Ignite
configurationResource (common)	Sets the resource from where to load the configuration. It can be a: URI, String (URI) or an InputStream.		Object
igniteConfiguration (common)	Allows the user to set a programmatic IgniteConfiguration.		IgniteConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ignite Cache endpoint is configured using URI syntax:

```
ignite-cache:cacheName
```

with the following path and query parameters:

147.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The cache name.		String

147.1.2. Query Parameters (16 parameters):

Name	Description	Default	Type
propagateIncomingBodyIfNoReturnValue (common)	Sets whether to propagate the incoming body if the return type of the underlying Ignite operation is void.	true	boolean
treatCollectionsAsCache Objects (common)	Sets whether to treat Collections as cache objects or as Collections of items to insert/update/compute, etc.	false	boolean
autoUnsubscribe (consumer)	Whether auto unsubscribe is enabled in the Continuous Query Consumer.	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
fireExistingQueryResults (consumer)	Whether to process existing results that match the query. Used on initialization of the Continuous Query Consumer.	false	boolean
oneExchangePerUpdate (consumer)	Whether to pack each update in an individual Exchange, even if multiple updates are received in one batch. Only used by the Continuous Query Consumer.	true	boolean
pageSize (consumer)	The page size. Only used by the Continuous Query Consumer.	1	int
query (consumer)	The Query to execute, only needed for operations that require it, and for the Continuous Query Consumer.		Object>>
remoteFilter (consumer)	The remote filter, only used by the Continuous Query Consumer.		Object>
timeInterval (consumer)	The time interval for the Continuous Query Consumer.	0	long

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
cachePeekMode (producer)	The CachePeekMode, only needed for operations that require it (link IgniteCacheOperationSIZE).	ALL	CachePeekMode
failIfInexistentCache (producer)	Whether to fail the initialization if the cache doesn't exist.	false	boolean
operation (producer)	The cache operation to invoke. Possible values: GET, PUT, REMOVE, SIZE, REBALANCE, QUERY, CLEAR.		IgniteCacheOperation
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

147.1.3. Headers used

This endpoint uses the following headers:

Header name	Constant	Expected type	Description
CamelIgniteCacheKey	IgniteConstants.IGNITE_CACHE_KEY	String	The cache key for the entry value in the message body.
CamelIgniteCacheQuery	IgniteConstants.IGNITE_CACHE_QUERY	Query	The query to run (producer) when invoking the QUERY operation.
CamelIgniteCacheOperation	IgniteConstants.IGNITE_CACHE_OPERATION	IgniteCacheOperation enum	Allows you to dynamically change the cache operation to execute (producer).

Header name	Constant	Expected type	Description
CamelIgniteCachePeekMode	IgniteConstants.IGNITE_CACHE_PEEK_MODE	CachePeekMode enum	Allows you to dynamically change the cache peek mode when running the SIZE operation.
CamelIgniteCacheEventType	IgniteConstants.IGNITE_CACHE_EVENT_TYPE	int (EventType constants)	This header carries the received event type when using the continuous query consumer.
CamelIgniteCacheName	IgniteConstants.IGNITE_CACHE_NAME	String	This header carries the cache name for which a continuous query event was received (consumer). It does not allow you to dynamically change the cache against which a producer operation is performed. Use EIPs for that (e.g. recipient list, dynamic router).
CamelIgniteCacheOldValue	IgniteConstants.IGNITE_CACHE_OLD_VALUE	Object	This header carries the old cache value when passed in the incoming cache event (consumer).

CHAPTER 148. IGNITE COMPUTE COMPONENT

Available as of Camel version 2.17

The Ignite Compute endpoint is one of camel-ignite endpoints which allows you to run compute operations on the cluster by passing in an IgniteCallable, an IgniteRunnable, an IgniteClosure, or collections of them, along with their parameters if necessary.

This endpoint only supports producers.

The host part of the endpoint URI is a symbolic endpoint ID, it is not used for any purposes.

The endpoint tries to run the object passed in the body of the IN message as the compute job. It expects different payload types depending on the execution type.

148.1. OPTIONS

The Ignite Compute component supports 4 options which are listed below.

Name	Description	Default	Type
ignite (producer)	Sets the Ignite instance.		Ignite
configurationResource (producer)	Sets the resource from where to load the configuration. It can be a: URI, String (URI) or an InputStream.		Object
igniteConfiguration (producer)	Allows the user to set a programmatic IgniteConfiguration.		IgniteConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ignite Compute endpoint is configured using URI syntax:

```
ignite-compute:endpointId
```

with the following path and query parameters:

148.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
endpointId	**Required** The endpoint ID (not used).		String

148.1.2. Query Parameters (8 parameters):

Name	Description	Default	Type
clusterGroupExpression (producer)	An expression that returns the Cluster Group for the IgniteCompute instance.		ClusterGroupExpression
computeName (producer)	The name of the compute job, which will be set via link IgniteComputewithName(String).		String
executionType (producer)	**Required** The compute operation to perform. Possible values: CALL, BROADCAST, APPLY, EXECUTE, RUN, AFFINITY_CALL, AFFINITY_RUN. The component expects different payload types depending on the operation.		IgniteComputeExecution Type
propagateIncomingBodyIfNoReturnValue (producer)	Sets whether to propagate the incoming body if the return type of the underlying Ignite operation is void.	true	boolean
taskName (producer)	The task name, only applicable if using the link IgniteComputeExecutionTypeEXECUTE execution type.		String
timeoutMillis (producer)	The timeout interval for triggered jobs, in milliseconds, which will be set via link IgniteComputewithTimeout(long).		Long
treatCollectionsAsCache Objects (producer)	Sets whether to treat Collections as cache objects or as Collections of items to insert/update/compute, etc.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

148.1.3. Expected payload types

Each operation expects the indicated types:

Operation	Expected payloads
CALL	Collection of IgniteCallable, or a single IgniteCallable.
BROADCAST	IgniteCallable, IgniteRunnable, IgniteClosure.
APPLY	IgniteClosure.

Operation	Expected payloads
EXECUTE	ComputeTask, Class<? extends ComputeTask> or an object representing parameters if the taskName option is not null.
RUN	A Collection of IgniteRunnables, or a single IgniteRunnable.
AFFINITY_CALL	IgniteCallable.
AFFINITY_RUN	IgniteRunnable.

148.1.4. Headers used

This endpoint uses the following headers:

Header name	Constant	Expected type	Description
CamelIgniteComputeExecutionType	IgniteConstants.IGNITE_COMPUTE_EXECUTION_TYPE	IgniteComputeExecutionType enum	Allows you to dynamically change the compute operation to perform.
CamelIgniteComputeParameters	IgniteConstants.IGNITE_COMPUTE_PARAMS	Any object or Collection of objects.	Parameters for APPLY, BROADCAST and EXECUTE operations.
CamelIgniteComputeReducer	IgniteConstants.IGNITE_COMPUTE_REDUCER	IgniteReducer	Reducer for the APPLY and CALL operations.
CamelIgniteComputeAffinityCacheName	IgniteConstants.IGNITE_COMPUTE_AFFINITY_CACHE_NAME	String	Affinity cache name for the AFFINITY_CALL and AFFINITY_RUN operations.
CamelIgniteComputeAffinityKey	IgniteConstants.IGNITE_COMPUTE_AFFINITY_KEY	Object	Affinity key for the AFFINITY_CALL and AFFINITY_RUN operations.

CHAPTER 149. IGNITE EVENTS COMPONENT

Available as of Camel version 2.17

The Ignite Events endpoint is one of camel-ignite endpoints which allows you to receive events from the Ignite cluster by creating a local event listener.

This endpoint only supports consumers. The Exchanges created by this consumer put the received Event object into the body of the IN message.

149.1. OPTIONS

The Ignite Events component supports 4 options which are listed below.

Name	Description	Default	Type
ignite (consumer)	Sets the Ignite instance.		Ignite
configurationResource (consumer)	Sets the resource from where to load the configuration. It can be a: URI, String (URI) or an InputStream.		Object
igniteConfiguration (consumer)	Allows the user to set a programmatic IgniteConfiguration.		IgniteConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ignite Events endpoint is configured using URI syntax:

```
ignite-events:endpointId
```

with the following path and query parameters:

149.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
endpointId	The endpoint ID (not used).		String

149.1.2. Query Parameters (8 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
clusterGroupExpression (consumer)	The cluster group expression.		ClusterGroupExpression
events (consumer)	The event IDs to subscribe to as a Set directly where the IDs are the different constants in org.apache.ignite.events.EventType.	EventType.EVTS_ALL	Set<Integer>OrString
propagateIncomingBodyIfNoReturnValue (consumer)	Sets whether to propagate the incoming body if the return type of the underlying Ignite operation is void.	true	boolean
treatCollectionsAsCache Objects (consumer)	Sets whether to treat Collections as cache objects or as Collections of items to insert/update/compute, etc.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 150. IGNITE ID GENERATOR COMPONENT

Available as of Camel version 2.17

The Ignite ID Generator endpoint is one of camel-ignite endpoints which allows you to interact with Ignite Atomic Sequences and ID Generators.

This endpoint only supports producers.

150.1. OPTIONS

The Ignite ID Generator component supports 4 options which are listed below.

Name	Description	Default	Type
ignite (producer)	Sets the Ignite instance.		Ignite
configurationResource (producer)	Sets the resource from where to load the configuration. It can be a: URI, String (URI) or an InputStream.		Object
igniteConfiguration (producer)	Allows the user to set a programmatic IgniteConfiguration.		IgniteConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ignite ID Generator endpoint is configured using URI syntax:

```
ignite-idgen:name
```

with the following path and query parameters:

150.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** The sequence name.		String

150.1.2. Query Parameters (6 parameters):

Name	Description	Default	Type
batchSize (producer)	The batch size.		Integer
initialValue (producer)	The initial value.	0	Long
operation (producer)	The operation to invoke on the Ignite ID Generator. Superseded by the IgniteConstants.IGNITE_IDGEN_OPERATION header in the IN message. Possible values: ADD_AND_GET, GET, GET_AND_ADD, GET_AND_INCREMENT, INCREMENT_AND_GET.		IgniteIdGenOperation
propagateIncomingBodyIfNoReturnValue (producer)	Sets whether to propagate the incoming body if the return type of the underlying Ignite operation is void.	true	boolean
treatCollectionsAsCache Objects (producer)	Sets whether to treat Collections as cache objects or as Collections of items to insert/update/compute, etc.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

CHAPTER 151. IGNITE MESSAGING COMPONENT

Available as of Camel version 2.17

The Ignite Messaging endpoint is one of camel-ignite endpoints which allows you to send and consume messages from an Ignite topic.

This endpoint supports producers (to send messages) and consumers (to receive messages).

151.1. OPTIONS

The Ignite Messaging component supports 4 options which are listed below.

Name	Description	Default	Type
ignite (common)	Sets the Ignite instance.		Ignite
configurationResource (common)	Sets the resource from where to load the configuration. It can be a: URI, String (URI) or an InputStream.		Object
igniteConfiguration (common)	Allows the user to set a programmatic IgniteConfiguration.		IgniteConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ignite Messaging endpoint is configured using URI syntax:

```
ignite-messaging:topic
```

with the following path and query parameters:

151.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
topic	**Required** The topic name.		String

151.1.2. Query Parameters (9 parameters):

Name	Description	Default	Type
propagateIncomingBodyIfNoReturnValue (common)	Sets whether to propagate the incoming body if the return type of the underlying Ignite operation is void.	true	boolean
treatCollectionsAsCache Objects (common)	Sets whether to treat Collections as cache objects or as Collections of items to insert/update/compute, etc.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
clusterGroupExpression (producer)	The cluster group expression.		ClusterGroupExpression
sendMode (producer)	The send mode to use. Possible values: UNORDERED, ORDERED.	UNORDERED	IgniteMessagingSend Mode
timeout (producer)	The timeout for the send operation when using ordered messages.		Long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

151.1.3. Headers used

This endpoint uses the following headers:

Header name	Constant	Expected type	Description
CamelIgniteMessagingTopic	IgniteConstants.IGNITE_MESSAGING_TOPIC	String	Allows you to dynamically change the topic to send messages to (producer). It also carries the topic on which a message was received (consumer).
CamelIgniteMessagingUUID	IgniteConstants.IGNITE_MESSAGING_UUID	UUID	This header is filled in with the UUID of the subscription when a message arrives (consumer).

CHAPTER 152. IGNITE QUEUES COMPONENT

Available as of Camel version 2.17

The Ignite Queue endpoint is one of camel-ignite endpoints which allows you to interact with Ignite Queue data structures.

This endpoint only supports producers.

152.1. OPTIONS

The Ignite Queues component supports 4 options which are listed below.

Name	Description	Default	Type
ignite (producer)	Sets the Ignite instance.		Ignite
configurationResource (producer)	Sets the resource from where to load the configuration. It can be a: URI, String (URI) or an InputStream.		Object
igniteConfiguration (producer)	Allows the user to set a programmatic IgniteConfiguration.		IgniteConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ignite Queues endpoint is configured using URI syntax:

```
ignite-queue:name
```

with the following path and query parameters:

152.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** The queue name.		String

152.1.2. Query Parameters (7 parameters):

Name	Description	Default	Type
capacity (producer)	The queue capacity. Default: non-bounded.		int
configuration (producer)	The collection configuration. Default: empty configuration. You can also conveniently set inner properties by using configuration.xyz=123 options.		CollectionConfiguration
operation (producer)	The operation to invoke on the Ignite Queue. Superseded by the IgniteConstants.IGNITE_QUEUE_OPERATION header in the IN message. Possible values: CONTAINS, ADD, SIZE, REMOVE, ITERATOR, CLEAR, RETAIN_ALL, ARRAY, DRAIN, ELEMENT, PEEK, OFFER, POLL, TAKE, PUT.		IgniteQueueOperation
propagateIncomingBodyIfNoReturnValue (producer)	Sets whether to propagate the incoming body if the return type of the underlying Ignite operation is void.	true	boolean
timeoutMillis (producer)	The queue timeout in milliseconds. Default: no timeout.		Long
treatCollectionsAsCache Objects (producer)	Sets whether to treat Collections as cache objects or as Collections of items to insert/update/compute, etc.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

152.1.3. Headers used

This endpoint uses the following headers:

Header name	Constant	Expected type	Description
CamelIgniteQueueOperation	IgniteConstants.IGNITE_QUEUE_OPERATION	IgniteQueueOperation enum	Allows you to dynamically change the queue operation.

Header name	Constant	Expected type	Description
CamelIgniteQueueMaxElements	IgniteConstants.IGNITE_QUEUE_MAX_ELEMENTS	Integer or int	When invoking the DRAIN operation, the amount of items to drain.
CamelIgniteQueueTransferredCount	IgniteConstants.IGNITE_QUEUE_TRANSFERRED_COUNT	Integer or int	The amount of items transferred as the result of the DRAIN operation.
CamelIgniteQueueTimeoutMillis	IgniteConstants.IGNITE_QUEUE_TIMEOUT_MILLIS	Long or long	Dynamically sets the timeout in milliseconds to use when invoking the OFFER or POLL operations.

CHAPTER 153. IGNITE SETS COMPONENT

Available as of Camel version 2.17

The Ignite Sets endpoint is one of camel-ignite endpoints which allows you to interact with Ignite Set data structures.

This endpoint only supports producers.

153.1. OPTIONS

The Ignite Sets component supports 4 options which are listed below.

Name	Description	Default	Type
ignite (producer)	Sets the Ignite instance.		Ignite
configurationResource (producer)	Sets the resource from where to load the configuration. It can be a: URI, String (URI) or an InputStream.		Object
igniteConfiguration (producer)	Allows the user to set a programmatic IgniteConfiguration.		IgniteConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Ignite Sets endpoint is configured using URI syntax:

```
ignite-set:name
```

with the following path and query parameters:

153.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** The set name.		String

153.1.2. Query Parameters (5 parameters):

Name	Description	Default	Type
configuration (producer)	The collection configuration. Default: empty configuration. You can also conveniently set inner properties by using configuration.xyz=123 options.		CollectionConfiguration
operation (producer)	The operation to invoke on the Ignite Set. Superseded by the IgniteConstants.IGNITE_SETS_OPERATION header in the IN message. Possible values: CONTAINS, ADD, SIZE, REMOVE, ITERATOR, CLEAR, RETAIN_ALL, ARRAY.The set operation to perform.		IgniteSetOperation
propagateIncomingBodyIfNoReturnValue (producer)	Sets whether to propagate the incoming body if the return type of the underlying Ignite operation is void.	true	boolean
treatCollectionsAsCache Objects (producer)	Sets whether to treat Collections as cache objects or as Collections of items to insert/update/compute, etc.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

153.1.3. Headers used

This endpoint uses the following headers:

Header name	Constant	Expected type	Description
CamelIgniteSetsOperation	IgniteConstants.IGNITE_SETS_OPERATION	IgniteSetOperation enum	Allows you to dynamically change the set operation.

CHAPTER 154. INFINISPAN COMPONENT

Available as of Camel version 2.13

This component allows you to interact with Infinispan distributed data grid / cache. Infinispan is an extremely scalable, highly available key/value data store and data grid platform written in Java.

From **Camel 2.17** onwards Infinispan requires Java 8.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-infinispan</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

154.1. URI FORMAT

```
infinispan://cacheName?[options]
```

154.2. URI OPTIONS

The producer allows sending messages to a local infinispan cache configured in the registry, or to a remote cache using the HotRod protocol. The consumer allows listening for events from local infinispan cache accessible from the registry.

The Infinispan component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (common)	The default configuration shared among endpoints.		InfinispanConfiguration
cacheContainer (common)	The default cache container.		BasicCacheContainer
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Infinispan endpoint is configured using URI syntax:

```
infinispan:cacheName
```

with the following path and query parameters:

154.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The cache to use		String

154.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
hosts (common)	Specifies the host of the cache on Infinispan instance		String
queryBuilder (common)	Specifies the query builder.		InfinispanQueryBuilder
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
clusteredListener (consumer)	If true, the listener will be installed for the entire cluster	false	boolean
command (consumer)	**Deprecated** The operation to perform.	PUT	String
customListener (consumer)	Returns the custom listener in use, if provided		InfinispanCustomListener

Name	Description	Default	Type
eventTypes (consumer)	Specifies the set of event types to register by the consumer. Multiple event can be separated by comma. The possible event types are: CACHE_ENTRY_ACTIVATED, CACHE_ENTRY_PASSIVATED, CACHE_ENTRY_VISITED, CACHE_ENTRY_LOADED, CACHE_ENTRY_EVICTED, CACHE_ENTRY_CREATED, CACHE_ENTRY_REMOVED, CACHE_ENTRY_MODIFIED, TRANSACTION_COMPLETED, TRANSACTION_REGISTERED, CACHE_ENTRY_INVALIDATED, DATA_REHASHED, TOPOLOGY_CHANGED, PARTITION_STATUS_CHANGED		String
sync (consumer)	If true, the consumer will receive notifications synchronously	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	The operation to perform.	PUT	InfinispanOperation
cacheContainer (advanced)	Specifies the cache Container to connect		BasicCacheContainer
cacheContainerConfiguration (advanced)	The CacheContainer configuration		Object
configurationProperties (advanced)	Implementation specific properties for the CacheManager		Map
configurationUri (advanced)	An implementation specific URI for the CacheManager		String

Name	Description	Default	Type
flags (advanced)	A comma separated list of Flag to be applied by default on each cache invocation, not applicable to remote caches.		String
resultHeader (advanced)	Store the operation result in a header instead of the message body. By default, resultHeader == null and the query result is stored in the message body, any existing content in the message body is discarded. If resultHeader is set, the value is used as the name of the header to store the query result and the original message body is preserved. This value can be overridden by an in message header named: CamelInfinispanOperationResultHeader		Object
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

154.3. MESSAGE HEADERS

Name	Default Value	Type	Context	Description
CamelInfinispanCacheName	**null**	String	Shared	The cache participating in the operation or event.
CamelInfinispanOperation	**PUT**	InfinispanOperation	Producer	The operation to perform.
CamelInfinispanMap	**null**	Map	Producer	A Map to use in case of CamelInfinispanOperationPutAll operation
CamelInfinispanKey	**null**	Object	Shared	The key to perform the operation to or the key generating the event.
CamelInfinispanValue	**null**	Object	Producer	The value to use for the operation.

Name	Default Value	Type	Context	Description
CamelInfinispanEventType	**null**	String	Consumer	The type of the received event. Possible values defined here org.infinispan.notifications.cachelistener.event.Event.Type
CamelInfinispanIsPre	**null**	Boolean	Consumer	Infinispan fires two events for each operation: one before and one after the operation.
CamelInfinispanLifespanTime	**null**	long	Producer	The Lifespan time of a value inside the cache. Negative values are interpreted as infinity.
CamelInfinispanTimeUnit	**null**	String	Producer	The Time Unit of an entry Lifespan Time.
CamelInfinispanMaxIdleTime	**null**	long	Producer	The maximum amount of time an entry is allowed to be idle for before it is considered as expired.
CamelInfinispanMaxIdleTimeUnit	**null**	String	Producer	The Time Unit of an entry Max Idle Time.
CamelInfinispanQueryBuilder	null	InfinispanQueryBuilder	Producer	**From Camel 2.17:** The QueryBuilde to use for QUERY command, if not present the command defaults to InifinispanConfiguration's one
CamelInfinispanIgnoreReturnValues	null	Boolean	Producer	**From Camel 2.17:** If this header is set, the return value for cache operation returning something is ignored by the client application

Name	Default Value	Type	Context	Description
CamelInfinispanOperationResultHeader	null	String	Producer	**From Camel 2.20:** Store the operation result in a header instead of the message body

154.4. EXAMPLES

- Retrieve a specific key from the default cache using a custom cache container:

```
from("direct:start")
    .setHeader(InfinispanConstants.OPERATION).constant(InfinispanOperation.GET)
    .setHeader(InfinispanConstants.KEY).constant("123")
    .to("infinispan?cacheContainer=#cacheContainer");
```

- Retrieve a specific key from a named cache:

```
from("direct:start")
    .setHeader(InfinispanConstants.OPERATION).constant(InfinispanOperation.PUT)
    .setHeader(InfinispanConstants.KEY).constant("123")
    .to("infinispan:myCacheName");
```

- Put a value with lifespan

```
from("direct:start")
    .setHeader(InfinispanConstants.OPERATION).constant(InfinispanOperation.GET)
    .setHeader(InfinispanConstants.KEY).constant("123")
    .setHeader(InfinispanConstants.LIFESPAN_TIME).constant(100L)

.setHeader(InfinispanConstants.LIFESPAN_TIME_UNIT.constant(TimeUnit.MILLISECONDS.t
oString())
    .to("infinispan:myCacheName");
```

154.5. USING THE INFINISPAN BASED IDEMPOTENT REPOSITORY

In this section we will use the Infinispan based idempotent repository.

First, we need to create a cacheManager and then configure our

```
org.apache.camel.component.infinispan.processor.idempotent.InfinispanIdempotentRepository:
```

```
<!-- set up the cache manager -->
<bean id="cacheManager"
      class="org.infinispan.manager.DefaultCacheManager"
      init-method="start"
      destroy-method="stop"/>
```

```
<!-- set up the repository -->
<bean id="infinispanRepo"

class="org.apache.camel.component.infinispan.processor.idempotent.InfinispanIdempotentRepository"

      factory-method="infinispanIdempotentRepository">
    <argument ref="cacheManager"/>
    <argument value="idempotent"/>
</bean>
```

Then we can create our Infinispan idempotent repository in the spring XML file as well:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="JpaMessageIdRepositoryTest">
        <from uri="direct:start" />
        <idempotentConsumer messageIdRepositoryRef="infinispanStore">
            <header>messageId</header>
            <to uri="mock:result" />
        </idempotentConsumer>
    </route>
</camelContext>
```

154.6. USING THE INFINISPAN BASED ROUTE POLICY

154.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 155. INFLUXDB COMPONENT

Available as of Camel version 2.18

This component allows you to interact with InfluxDB https://influxdata.com/time-series-platform/influxdb/ a time series database. The native body type for this component is Point (the native influxdb class), but it can also accept Map<String, Object> as message body and it will get converted to Point.class, please note that the map must contain an element with InfluxDbConstants.MEASUREMENT_NAME as key.

Aditionally of course you may register your own Converters to your data type to Point, or use the (un)marshalling tools provided by camel.

From **Camel 2.18** onwards Influxdb requires Java 8.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-influxdb</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

155.1. URI FORMAT

```
influxdb://beanName?[options]
```

155.2. URI OPTIONS

The producer allows sending messages to a influxdb configured in the registry, using the native java driver.

The InfluxDB component has no options.

The InfluxDB endpoint is configured using URI syntax:

```
influxdb:connectionBean
```

with the following path and query parameters:

155.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
connectionBean	**Required** Connection to the influx database, of class InfluxDB.class		String

155.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
batch (producer)	Define if this operation is a batch operation or not	false	boolean
databaseName (producer)	The name of the database where the time series will be stored		String
operation (producer)	Define if this operation is an insert or a query	insert	String
query (producer)	Define the query in case of operation query		String
retentionPolicy (producer)	The string that defines the retention policy to the data created by the endpoint	default	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

155.3. MESSAGE HEADERS

Name	Default Value	Type	Context	Description

155.4. EXAMPLE

Below is an example route that stores a point into the db (taking the db name from the URI) specific key:

```
from("direct:start")
        .setHeader(InfluxDbConstants.DBNAME_HEADER, constant("myTimeSeriesDB"))
        .to("influxdb://connectionBean);
```

```
from("direct:start")
        .to("influxdb://connectionBean?databaseName=myTimeSeriesDB");
```

For more information, see these resources...

155.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint

- Getting Started

CHAPTER 156. IRC COMPONENT

Available as of Camel version 1.1

The **irc** component implements an IRC (Internet Relay Chat) transport.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-irc</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

156.1. URI FORMAT

```
irc:nick@host[:port]/#room[?options]
irc:nick@host[:port]?channels=#channel1,#channel2,#channel3[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

156.2. OPTIONS

The IRC component supports 2 options which are listed below.

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The IRC endpoint is configured using URI syntax:

```
irc:hostname:port
```

with the following path and query parameters:

156.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
hostname	**Required** Hostname for the IRC chat server		String
port	Port number for the IRC chat server. If no port is configured then a default port of either 6667, 6668 or 6669 is used.		int

156.2.2. Query Parameters (24 parameters):

Name	Description	Default	Type
autoRejoin (common)	Whether to auto re-join when being kicked	true	boolean
namesOnJoin (common)	Sends NAMES command to channel after joining it. link onReply has to be true in order to process the result which will have the header value irc.num = '353'.	false	boolean
nickname (common)	The nickname used in chat.		String
persistent (common)	**Deprecated** Use persistent messages.	true	boolean
realname (common)	The IRC user's actual name.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
colors (advanced)	Whether or not the server supports color codes.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
onJoin (filter)	Handle user join events.	true	boolean
onKick (filter)	Handle kick events.	true	boolean
onMode (filter)	Handle mode change events.	true	boolean
onNick (filter)	Handle nickname change events.	true	boolean
onPart (filter)	Handle user part events.	true	boolean
onPrivmsg (filter)	Handle private message events.	true	boolean
onQuit (filter)	Handle user quit events.	true	boolean
onReply (filter)	Whether or not to handle general responses to commands or informational messages.	false	boolean
onTopic (filter)	Handle topic change events.	true	boolean
nickPassword (security)	Your IRC server nickname password.		String
password (security)	The IRC server password.		String
sslContextParameters (security)	Used for configuring security using SSL. Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry. This reference overrides any configured SSLContextParameters at the component level. Note that this setting overrides the trustManager option.		SSLContextParameters
trustManager (security)	The trust manager used to verify the SSL server's certificate.		SSLTrustManager
username (security)	The IRC server user name.		String

156.3. SSL SUPPORT

156.3.1. Using the JSSE Configuration Utility

As of Camel 2.9, the IRC component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the IRC component.

Programmatic configuration of the endpoint

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/truststore.jks");
ksp.setPassword("keystorePassword");

TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);

SSLContextParameters scp = new SSLContextParameters();
scp.setTrustManagers(tmp);

Registry registry = ...
registry.bind("sslContextParameters", scp);

...

from(...)
    .to("ircs://camel-prd-user@server:6669/#camel-test?nickname=camel-
prd&password=password&sslContextParameters=#sslContextParameters");
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
     id="sslContextParameters">
   <camel:trustManagers>
     <camel:keyStore
        resource="/users/home/server/truststore.jks"
        password="keystorePassword"/>
   </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="ircs://camel-prd-user@server:6669/#camel-test?nickname=camel-
prd&password=password&sslContextParameters=#sslContextParameters"/>...
```

156.3.2. Using the legacy basic configuration options

You can also connect to an SSL enabled IRC server, as follows:

```
ircs:host[:port]/#room?username=user&password=pass
```

By default, the IRC transport uses SSLDefaultTrustManager. If you need to provide your own custom trust manager, use the **trustManager** parameter as follows:

```
ircs:host[:port]/#room?
username=user&password=pass&trustManager=#referenceToMyTrustManagerBean
```

156.4. USING KEYS

Available as of Camel 2.2

Some irc rooms requires you to provide a key to be able to join that channel. The key is just a secret word.

For example we join 3 channels where as only channel 1 and 3 uses a key.

```
irc:nick@irc.server.org?channels=#chan1,#chan2,#chan3&keys=chan1Key,,chan3key
```

156.5. GETTING A LIST OF USERS OF THE CHANNEL

Using the **namesOnJoin** option one can invoke the IRC- **NAMES** command after the component has joined a channel. The server will reply with **irc.num = 353**. So in order to process the result the property **onReply** has to be **true**. Furthermore one has to filter the **onReply** exchanges in order to get the names.

For example we want to get all exchanges that contain the usernames of the channel:

```
from("ircs:nick@myserver:1234/#mychannelname?namesOnJoin=true&onReply=true")
 .choice()
  .when(header("irc.messageType").isEqualToIgnoreCase("REPLY"))
   .filter(header("irc.num").isEqualTo("353"))
   .to("mock:result").stop();
```

156.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 157. JACKSONXML DATAFORMAT

Available as of Camel version 2.16

Jackson XML is a Data Format which uses the Jackson library with the XMLMapper extension to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

INFO:If you are familiar with Jackson, this XML data format behaves in the same way as its JSON counterpart, and thus can be used with classes annotated for JSON serialization/deserialization.

This extension also mimics JAXB's "Code first" approach.

This data format relies on Woodstox (especially for features like pretty printing), a fast and efficient XML processor.

```
from("activemq:My.Queue").
  unmarshal().jacksonxml().
  to("mqseries:Another.Queue");
```

157.1. JACKSONXML OPTIONS

The JacksonXML dataformat supports 15 options which are listed below.

Name	Default	Java Type	Description
xmlMapper		**String**	Lookup and use the existing XmlMapper with the given id.
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
jsonView		**Class<?>**	When marshalling a POJO to JSON you might want to exclude certain fields from the JSON output. With Jackson you can use JSON views to accomplish this. This option is to refer to the class which has JsonView annotations
include		**String**	If you want to marshal a pojo to JSON, and the pojo has some fields with null values. And you want to skip these null values, you can set this option to NOT_NULL
allowJmsType	**false**	**Boolean**	Used for JMS users to allow the JMSType header from the JMS spec to specify a FQN classname to use to unmarshal to.
collectionTypeName		**String**	Refers to a custom collection type to lookup in the registry to use. This option should rarely be used, but allows to use different collection types than java.util.Collection based as default.

Name	Default	Java Type	Description
useList	**false**	**Boolean**	To unarmshal to a List of Map or a List of Pojo.
enableJaxbAnnotationModule	**false**	**Boolean**	Whether to enable the JAXB annotations module when using jackson. When enabled then JAXB annotations can be used by Jackson.
moduleClassNames		**String**	To use custom Jackson modules com.fasterxml.jackson.databind.Module specified as a String with FQN class names. Multiple classes can be separated by comma.
moduleRefs		**String**	To use custom Jackson modules referred from the Camel registry. Multiple modules can be separated by comma.
enableFeatures		**String**	Set of features to enable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
disableFeatures		**String**	Set of features to disable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
allowUnmarshallType	**false**	**Boolean**	If enabled then Jackson is allowed to attempt to use the CamelJacksonUnmarshalType header during the unmarshalling. This should only be enabled when desired to be used.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

157.1.1. Using Jackson XML in Spring DSL

When using Data Format in Spring DSL you need to declare the data formats first. This is done in the **DataFormats** XML tag.

```
<dataFormats>
  <!-- here we define a Xml data format with the id jack and that it should use the TestPojo as
```

```
the class type when
            doing unmarshal. The unmarshalTypeName is optional, if not provided Camel will use a
Map as the type -->
        <jacksonxml id="jack"
unmarshalTypeName="org.apache.camel.component.jacksonxml.TestPojo"/>
      </dataFormats>
```

And then you can refer to this id in the route:

```
    <route>
      <from uri="direct:back"/>
      <unmarshal ref="jack"/>
      <to uri="mock:reverse"/>
    </route>
```

157.2. EXCLUDING POJO FIELDS FROM MARSHALLING

When marshalling a POJO to XML you might want to exclude certain fields from the XML output. With Jackson you can use JSON views to accomplish this. First create one or more marker classes.

Use the marker classes with the **@JsonView** annotation to include/exclude certain fields. The annotation also works on getters.

Finally use the Camel **JacksonXMLDataFormat** to marshall the above POJO to XML.

Note that the weight field is missing in the resulting XML:

```
<pojo age="30" weight="70"/>
```

157.3. INCLUDE/EXCLUDE FIELDS USING THE JSONVIEW ATTRIBUTE WITH `JACKSONXML`DATAFORMAT

As an example of using this attribute you can instead of:

```
JacksonXMLDataFormat ageViewFormat = new JacksonXMLDataFormat(TestPojoView.class,
Views.Age.class);
from("direct:inPojoAgeView").
  marshal(ageViewFormat);
```

Directly specify your JSON view inside the Java DSL as:

```
from("direct:inPojoAgeView").
  marshal().jacksonxml(TestPojoView.class, Views.Age.class);
```

And the same in XML DSL:

```
<from uri="direct:inPojoAgeView"/>
  <marshal>
    <jacksonxml unmarshalTypeName="org.apache.camel.component.jacksonxml.TestPojoView"
jsonView="org.apache.camel.component.jacksonxml.Views$Age"/>
  </marshal>
```

157.4. SETTING SERIALIZATION INCLUDE OPTION

If you want to marshal a pojo to XML, and the pojo has some fields with null values. And you want to skip these null values, then you need to set either an annotation on the pojo,

```
@JsonInclude(Include.NON_NULL)
public class MyPojo {
   ...
}
```

But this requires you to include that annotation in your pojo source code. You can also configure the Camel JacksonXMLDataFormat to set the include option, as shown below:

```
JacksonXMLDataFormat format = new JacksonXMLDataFormat();
format.setInclude("NON_NULL");
```

Or from XML DSL you configure this as

```
<dataFormats>
  <jacksonxml id="jacksonxml" include="NOT_NULL"/>
</dataFormats>
```

157.5. UNMARSHALLING FROM XML TO POJO WITH DYNAMIC CLASS NAME

If you use jackson to unmarshal XML to POJO, then you can now specify a header in the message that indicate which class name to unmarshal to.
The header has key **CamelJacksonUnmarshalType** if that header is present in the message, then Jackson will use that as FQN for the POJO class to unmarshal the XML payload as.

For JMS end users there is the JMSType header from the JMS spec that indicates that also. To enable support for JMSType you would need to turn that on, on the jackson data format as shown:

```
JacksonDataFormat format = new JacksonDataFormat();
format.setAllowJmsType(true);
```

Or from XML DSL you configure this as

```
<dataFormats>
  <jacksonxml id="jacksonxml" allowJmsType="true"/>
</dataFormats>
```

157.6. UNMARSHALLING FROM XML TO LIST<MAP> OR LIST<POJO>

If you are using Jackson to unmarshal XML to a list of map/pojo, you can now specify this by setting **useList="true"** or use the **org.apache.camel.component.jacksonxml.ListJacksonXMLDataFormat**. For example with Java you can do as shown below:

```
JacksonXMLDataFormat format = new ListJacksonXMLDataFormat();
// or
JacksonXMLDataFormat format = new JacksonXMLDataFormat();
```

```
format.useList();
// and you can specify the pojo class type also
format.setUnmarshalType(MyPojo.class);
```

And if you use XML DSL then you configure to use list using **useList** attribute as shown below:

```
<dataFormats>
  <jacksonxml id="jack" useList="true"/>
</dataFormats>
```

And you can specify the pojo type also

```
<dataFormats>
  <jacksonxml id="jack" useList="true" unmarshalTypeName="com.foo.MyPojo"/>
</dataFormats>
```

157.7. USING CUSTOM JACKSON MODULES

You can use custom Jackson modules by specifying the class names of those using the moduleClassNames option as shown below.

```
<dataFormats>
  <jacksonxml id="jack" useList="true" unmarshalTypeName="com.foo.MyPojo"
moduleClassNames="com.foo.MyModule,com.foo.MyOtherModule"/>
</dataFormats>
```

When using moduleClassNames then the custom jackson modules are not configured, by created using default constructor and used as-is. If a custom module needs any custom configuration, then an instance of the module can be created and configured, and then use modulesRefs to refer to the module as shown below:

```
<bean id="myJacksonModule" class="com.foo.MyModule">
  ... // configure the module as you want
</bean>

<dataFormats>
  <jacksonxml id="jacksonxml" useList="true" unmarshalTypeName="com.foo.MyPojo"
moduleRefs="myJacksonModule"/>
</dataFormats>
```

Multiple modules can be specified separated by comma, such as moduleRefs="myJacksonModule,myOtherModule"

157.8. ENABLING OR DISABLE FEATURES USING JACKSON

Jackson has a number of features you can enable or disable, which its ObjectMapper uses. For example to disable failing on unknown properties when marshalling, you can configure this using the disableFeatures:

```
<dataFormats>
  <jacksonxml id="jacksonxml" unmarshalTypeName="com.foo.MyPojo"
disableFeatures="FAIL_ON_UNKNOWN_PROPERTIES"/>
</dataFormats>
```

-

You can disable multiple features by separating the values using comma. The values for the features must be the name of the enums from Jackson from the following enum classes

- com.fasterxml.jackson.databind.SerializationFeature
- com.fasterxml.jackson.databind.DeserializationFeature
- com.fasterxml.jackson.databind.MapperFeature

To enable a feature use the enableFeatures options instead.

From Java code you can use the type safe methods from camel-jackson module:

```
JacksonDataFormat df = new JacksonDataFormat(MyPojo.class);
df.disableFeature(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
df.disableFeature(DeserializationFeature.FAIL_ON_NULL_FOR_PRIMITIVES);
```

157.9. CONVERTING MAPS TO POJO USING JACKSON

Jackson **ObjectMapper** can be used to convert maps to POJO objects. Jackson component comes with the data converter that can be used to convert **java.util.Map** instance to non-String, non-primitive and non-Number objects.

```
Map<String, Object> invoiceData = new HashMap<String, Object>();
invoiceData.put("netValue", 500);
producerTemplate.sendBody("direct:mapToInvoice", invoiceData);
...
// Later in the processor
Invoice invoice = exchange.getIn().getBody(Invoice.class);
```

If there is a single **ObjectMapper** instance available in the Camel registry, it will used by the converter to perform the conversion. Otherwise the default mapper will be used.

157.10. FORMATTED XML MARSHALLING (PRETTY-PRINTING)

Using the **prettyPrint** option one can output a well formatted XML while marshalling:

```
<dataFormats>
    <jacksonxml id="jack" prettyPrint="true"/>
</dataFormats>
```

And in Java DSL:

```
from("direct:inPretty").marshal().jacksonxml(true);
```

Please note that there are 5 different overloaded **jacksonxml()** DSL methods which support the **prettyPrint** option in combination with other settings for **unmarshalType**, **jsonView** etc.

157.11. DEPENDENCIES

To use Jackson XML in your camel routes you need to add the dependency on **camel-jacksonxml** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jacksonxml</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 158. JASYPT COMPONENT

Available as of Camel 2.5

Jasypt is a simplified encryption library which makes encryption and decryption easy. Camel integrates with Jasypt to allow sensitive information in Properties files to be encrypted. By dropping **camel-jasypt** on the classpath those encrypted values will automatically be decrypted on-the-fly by Camel. This ensures that human eyes can't easily spot sensitive information such as usernames and passwords.

If you are using Maven, you need to add the following dependency to your **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jasypt</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

If you are using an Apache Karaf container, you need to add the following dependency to your **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.karaf.jaas</groupId>
  <artifactId>org.apache.karaf.jaas.jasypt</artifactId>
  <version>x.x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

158.1. TOOLING

The Jasypt component provides a little command line tooling to encrypt or decrypt values.

The console output the syntax and which options it provides:

```
Apache Camel Jasypt takes the following options

  -h or -help = Displays the help screen
  -c or -command <command> = Command either encrypt or decrypt
  -p or -password <password> = Password to use
  -i or -input <input> = Text to encrypt or decrypt
  -a or -algorithm <algorithm> = Optional algorithm to use
```

For example to encrypt the value **tiger** you run with the following parameters. In the apache camel kit, you cd into the lib folder and run the following java cmd, where *<CAMEL_HOME>* is where you have downloaded and extract the Camel distribution.

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c encrypt -p secret -i tiger
```

Which outputs the following result

```
Encrypted text: qaEEacuW7BUti8LcMgyjKw==
```

This means the encrypted representation **qaEEacuW7BUti8LcMgyjKw==** can be decrypted back to **tiger** if you know the master password which was **secret**.
If you run the tool again then the encrypted value will return a different result. But decrypting the value will always return the correct original value.

So you can test it by running the tooling using the following parameters:

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c decrypt -p secret -i qaEEacuW7BUti8LcMgyjKw==
```

Which outputs the following result:

```
Decrypted text: tiger
```

The idea is then to use those encrypted values in your Properties files. Notice how the password value is encrypted and the value has the tokens surrounding **ENC(value here)**

TIP

When running jasypt tooling, if you come across **java.lang.NoClassDefFoundError: org/jasypt/encryption/pbe/StandardPBEStringEncryptor** this means you have to include jasypt7.3.jar in your classpath. Example of adding jar to classpath may be copying jasypt7.3.jar to $JAVA_HOME\jre\lib\ext if you are going to run as **java -jar ...**. The latter may be adding jasypt7.3.jar to classpath using **-cp**, in that case you should provide main class to execute as eg: **java -cp jasypt-1.9.2.jar:camel-jasypt-2.18.2.jar org.apache.camel.component.jasypt.Main -c encrypt -p secret -i tiger**

158.2. URI OPTIONS

The options below are exclusive for the Jasypt component.

Name	Default Value	Type	Description
password	**null**	**String**	Specifies the master password to use for decrypting. This option is mandatory. See below for more details.
algorithm	**null**	**String**	Name of an optional algorithm to use.

158.3. PROTECTING THE MASTER PASSWORD

The master password used by Jasypt must be provided, so that it's capable of decrypting the values. However having this master password out in the open may not be an ideal solution. Therefore you could for example provide it as a JVM system property or as a OS environment setting. If you decide to do so then the **password** option supports prefixes which dictates this. **sysenv:** means to lookup the OS system environment with the given key. **sys:** means to lookup a JVM system property.

For example you could provided the password before you start the application

```
$ export CAMEL_ENCRYPTION_PASSWORD=secret
```

Then start the application, such as running the start script.

When the application is up and running you can unset the environment

```
$ unset CAMEL_ENCRYPTION_PASSWORD
```

The **password** option is then a matter of defining as follows:
password=sysenv:CAMEL_ENCRYPTION_PASSWORD.

158.4. EXAMPLE WITH JAVA DSL

In Java DSL you need to configure Jasypt as a **JasyptPropertiesParser** instance and set it on the Properties component as show below:

The properties file **myproperties.properties** then contain the encrypted value, such as shown below. Notice how the password value is encrypted and the value has the tokens surrounding **ENC(value here)**

158.5. EXAMPLE WITH SPRING XML

In Spring XML you need to configure the **JasyptPropertiesParser** which is shown below. Then the Camel Properties component is told to use **jasypt** as the properties parser, which means Jasypt has its chance to decrypt values looked up in the properties.

```
<!-- define the jasypt properties parser with the given password to be used -->
<bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <property name="password" value="secret"/>
</bean>

<!-- define the camel properties component -->
<bean id="properties" class="org.apache.camel.component.properties.PropertiesComponent">
    <!-- the properties file is in the classpath -->
    <property name="location"
value="classpath:org/apache/camel/component/jasypt/myproperties.properties"/>
    <!-- and let it leverage the jasypt parser -->
    <property name="propertiesParser" ref="jasypt"/>
</bean>
```

The Properties component can also be inlined inside the **<camelContext>** tag which is shown below. Notice how we use the **propertiesParserRef** attribute to refer to Jasypt.

```
<!-- define the jasypt properties parser with the given password to be used -->
<bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <!-- password is mandatory, you can prefix it with sysenv: or sys: to indicate it should use
         an OS environment or JVM system property value, so you dont have the master password
defined here -->
    <property name="password" value="secret"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define the camel properties placeholder, and let it leverage jasypt -->
    <propertyPlaceholder id="properties"
                         location="classpath:org/apache/camel/component/jasypt/myproperties.properties"
                         propertiesParserRef="jasypt"/>
    <route>
```

```
        <from uri="direct:start"/>
        <to uri="{{cool.result}}"/>
    </route>
</camelContext>
```

158.6. EXAMPLE WITH BLUEPRINT XML

In Blueprint XML you need to configure the **JasyptPropertiesParser** which is shown below. Then the Camel Properties component is told to use **jasypt** as the properties parser, which means Jasypt has its chance to decrypt values looked up in the properties.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
           xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

  <cm:property-placeholder id="myblue" persistent-id="mypersistent">
    <!-- list some properties for this test -->
    <cm:default-properties>
      <cm:property name="cool.result" value="mock:{{cool.password}}"/>
      <cm:property name="cool.password" value="ENC(bsW9uV37gQ0QHFu7KO03Ww==)"/>
    </cm:default-properties>
  </cm:property-placeholder>

  <!-- define the jasypt properties parser with the given password to be used -->
  <bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <property name="password" value="secret"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <!-- define the camel properties placeholder, and let it leverage jasypt -->
    <propertyPlaceholder id="properties"
                         location="blueprint:myblue"
                         propertiesParserRef="jasypt"/>
    <route>
      <from uri="direct:start"/>
      <to uri="{{cool.result}}"/>
    </route>
  </camelContext>

</blueprint>
```

The Properties component can also be inlined inside the **<camelContext>** tag which is shown below. Notice how we use the **propertiesParserRef** attribute to refer to Jasypt.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
           xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

  <!-- define the jasypt properties parser with the given password to be used -->
```

```
    <bean id="jasypt" class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
      <property name="password" value="secret"/>
    </bean>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
      <!-- define the camel properties placeholder, and let it leverage jasypt -->
      <propertyPlaceholder id="properties"
                           location="classpath:org/apache/camel/component/jasypt/myproperties.properties"
                           propertiesParserRef="jasypt"/>
      <route>
          <from uri="direct:start"/>
          <to uri="{{cool.result}}"/>
      </route>
    </camelContext>

</blueprint>
```

158.7. SEE ALSO

- Security
- Properties
- Encrypted passwords in ActiveMQ - ActiveMQ has a similar feature as this **camel-jasypt** component

CHAPTER 159. JAXB DATAFORMAT

Available as of Camel version 1.0

JAXB is a Data Format which uses the JAXB2 XML marshalling standard which is included in Java 6 to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

159.1. OPTIONS

The JAXB dataformat supports 18 options which are listed below.

Name	Default	Java Type	Description
contextPath		**String**	Package name where your JAXB classes are located.
schema		**String**	To validate against an existing schema. Your can use the prefix classpath:, file: or http: to specify how the resource should by resolved. You can separate multiple schema files by using the ',' character.
schemaSeverityLevel	**0**	**Integer**	Sets the schema severity level to use when validating against a schema. This level determines the minimum severity error that triggers JAXB to stop continue parsing. The default value of 0 (warning) means that any error (warning, error or fatal error) will trigger JAXB to stop. There are the following three levels: 0=warning, 1=error, 2=fatal error.
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
objectFactory	**false**	**Boolean**	Whether to allow using ObjectFactory classes to create the POJO classes during marshalling. This only applies to POJO classes that has not been annotated with JAXB and providing jaxb.index descriptor files.
ignoreJAXBElement	**false**	**Boolean**	Whether to ignore JAXBElement elements - only needed to be set to false in very special use-cases.
mustBeJAXBElement	**false**	**Boolean**	Whether marhsalling must be java objects with JAXB annotations. And if not then it fails. This option can be set to false to relax that, such as when the data is already in XML format.
filterNonXmlChars	**false**	**Boolean**	To ignore non xml characheters and replace them with an empty space.
encoding		**String**	To overrule and use a specific encoding

Name	Default	Java Type	Description
fragment	**false**	**Boolean**	To turn on marshalling XML fragment trees. By default JAXB looks for XmlRootElement annotation on given class to operate on whole XML tree. This is useful but not always - sometimes generated code does not have XmlRootElement annotation, sometimes you need unmarshall only part of tree. In that case you can use partial unmarshalling. To enable this behaviours you need set property partClass. Camel will pass this class to JAXB's unmarshaler.
partClass		**String**	Name of class used for fragment parsing. See more details at the fragment option.
partNamespace		**String**	XML namespace to use for fragment parsing. See more details at the fragment option.
namespacePrefixRef		**String**	When marshalling using JAXB or SOAP then the JAXB implementation will automatic assign namespace prefixes, such as ns2, ns3, ns4 etc. To control this mapping, Camel allows you to refer to a map which contains the desired mapping.
xmlStreamWriterWrapper		**String**	To use a custom xml stream writer.
schemaLocation		**String**	To define the location of the schema
noNamespaceSchemaLocation		**String**	To define the location of the namespaceless schema
jaxbProviderProperties		**String**	Refers to a custom java.util.Map to lookup in the registry containing custom JAXB provider properties to be used with the JAXB marshaller.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

159.2. USING THE JAVA DSL

For example the following uses a named DataFormat of *jaxb* which is configured with a number of Java package names to initialize the JAXBContext.

```
DataFormat jaxb = new JaxbDataFormat("com.acme.model");

from("activemq:My.Queue").
  unmarshal(jaxb).
  to("mqseries:Another.Queue");
```

You can if you prefer use a named reference to a data format which can then be defined in your Registry such as via your Spring XML file. e.g.

```
from("activemq:My.Queue").
  unmarshal("myJaxbDataType").
  to("mqseries:Another.Queue");
```

159.3. USING SPRING XML

The following example shows how to use JAXB to unmarshal using Spring configuring the jaxb data type

This example shows how to configure the data type just once and reuse it on multiple routes.

Multiple context paths

It is possible to use this data format with more than one context path. You can specify context path using **:** as separator, for example **com.mycompany:com.mycompany2**. Note that this is handled by JAXB implementation and might change if you use different vendor than RI.

159.4. PARTIAL MARSHALLING/UNMARSHALLING

This feature is new to Camel 2.2.0.
JAXB 2 supports marshalling and unmarshalling XML tree fragments. By default JAXB looks for **@XmlRootElement** annotation on given class to operate on whole XML tree. This is useful but not always - sometimes generated code does not have @XmlRootElement annotation, sometimes you need unmarshall only part of tree.
In that case you can use partial unmarshalling. To enable this behaviours you need set property **partClass**. Camel will pass this class to JAXB's unmarshaler. If **JaxbConstants.JAXB_PART_CLASS** is set as one of headers, (even if partClass property is set on DataFormat), the property on DataFormat is surpassed and the one set in the headers is used.

For marshalling you have to add **partNamespace** attribute with QName of destination namespace. Example of Spring DSL you can find above. If **JaxbConstants.JAXB_PART_NAMESPACE** is set as one of headers, (even if partNamespace property is set on DataFormat), the property on DataFormat is surpassed and the one set in the headers is used. While setting **partNamespace** through **JaxbConstants.JAXB_PART_NAMESPACE**, please note that you need to specify its value {[namespaceUri]}[localPart]

```
...
.setHeader(JaxbConstants.JAXB_PART_NAMESPACE, simple("
{http://www.camel.apache.org/jaxb/example/address/1}address"));
...
```

159.5. FRAGMENT

This feature is new to Camel 2.8.0.
JaxbDataFormat has new property fragment which can set the the **Marshaller.JAXB_FRAGMENT** encoding property on the JAXB Marshaller. If you don't want the JAXB Marshaller to generate the XML declaration, you can set this option to be true. The default value of this property is false.

159.6. IGNORING THE NONXML CHARACTER

This feature is new to Camel 2.2.0.
JaxbDataFromat supports to ignore the NonXML Character, you just need to set the filterNonXmlChars property to be true, JaxbDataFormat will replace the NonXML character with " " when it is marshaling or unmarshaling the message. You can also do it by setting the Exchange property **Exchange.FILTER_NON_XML_CHARS**.

	JDK 1.5	JDK 1.6+
Filtering in use	StAX API and implementation	No
Filtering not in use	StAX API only	No

This feature has been tested with Woodstox 3.2.9 and Sun JDK 1.6 StAX implementation.

New for Camel 2.12.1
JaxbDataFormat now allows you to customize the XMLStreamWriter used to marshal the stream to XML. Using this configuration, you can add your own stream writer to completely remove, escape, or replace non-xml characters.

```
JaxbDataFormat customWriterFormat = new JaxbDataFormat("org.apache.camel.foo.bar");
customWriterFormat.setXmlStreamWriterWrapper(new TestXmlStreamWriter());
```

The following example shows using the Spring DSL and also enabling Camel's NonXML filtering:

```
<bean id="testXmlStreamWriterWrapper" class="org.apache.camel.jaxb.TestXmlStreamWriter"/>
<jaxb filterNonXmlChars="true"  contextPath="org.apache.camel.foo.bar"
xmlStreamWriterWrapper="#testXmlStreamWriterWrapper" />
```

159.7. WORKING WITH THE OBJECTFACTORY

If you use XJC to create the java class from the schema, you will get an ObjectFactory for you JAXB context. Since the ObjectFactory uses JAXBElement to hold the reference of the schema and element instance value, jaxbDataformat will ignore the JAXBElement by default and you will get the element instance value instead of the JAXBElement object form the unmarshaled message body.
If you want to get the JAXBElement object form the unmarshaled message body, you need to set the JaxbDataFormat object's ignoreJAXBElement property to be false.

159.8. SETTING ENCODING

You can set the **encoding** option to use when marshalling. Its the **Marshaller.JAXB_ENCODING** encoding property on the JAXB Marshaller.
You can setup which encoding to use when you declare the JAXB data format. You can also provide the

encoding in the Exchange property **Exchange.CHARSET_NAME**. This property will overrule the encoding set on the JAXB data format.

In this Spring DSL we have defined to use **iso-8859-1** as the encoding:

159.9. CONTROLLING NAMESPACE PREFIX MAPPING

Available as of Camel 2.11

When marshalling using JAXB or SOAP then the JAXB implementation will automatic assign namespace prefixes, such as ns2, ns3, ns4 etc. To control this mapping, Camel allows you to refer to a map which contains the desired mapping.

Notice this requires having JAXB-RI 2.1 or better (from SUN) on the classpath, as the mapping functionality is dependent on the implementation of JAXB, whether its supported.

For example in Spring XML we can define a Map with the mapping. In the mapping file below, we map SOAP to use soap as prefix. While our custom namespace "http://www.mycompany.com/foo/2" is not using any prefix.

```
<util:map id="myMap">
  <entry key="http://www.w3.org/2003/05/soap-envelope" value="soap"/>
  <!-- we dont want any prefix for our namespace -->
  <entry key="http://www.mycompany.com/foo/2" value=""/>
</util:map>
```

To use this in JAXB or SOAP you refer to this map, using the **namespacePrefixRef** attribute as shown below. Then Camel will lookup in the Registry a **java.util.Map** with the id "myMap", which was what we defined above.

```
<marshal>
  <soapjaxb version="1.2" contextPath="com.mycompany.foo" namespacePrefixRef="myMap"/>
</marshal>
```

159.10. SCHEMA VALIDATION

Available as of Camel 2.11

The JAXB Data Format supports validation by marshalling and unmarshalling from/to XML. Your can use the prefix **classpath:**, **file:** or **http:** to specify how the resource should by resolved. You can separate multiple schema files by using the **','** character.

Known issue

Camel 2.11.0 and 2.11.1 has a known issue by validation multiple `Exchange's in parallel. See CAMEL-6630. This is fixed with Camel 2.11.2/2.12.0.

Using the Java DSL, you can configure it in the following way:

```
JaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
jaxbDataFormat.setContextPath(Person.class.getPackage().getName());
jaxbDataFormat.setSchema("classpath:person.xsd,classpath:address.xsd");
```

You can do the same using the XML DSL:

```
<marshal>
  <jaxb id="jaxb" schema="classpath:person.xsd,classpath:address.xsd"/>
</marshal>
```

Camel will create and pool the underling **SchemaFactory** instances on the fly, because the **SchemaFactory** shipped with the JDK is not thread safe.
However, if you have a **SchemaFactory** implementation which is thread safe, you can configure the JAXB data format to use this one:

```
JaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
jaxbDataFormat.setSchemaFactory(thradSafeSchemaFactory);
```

159.11. SCHEMA LOCATION

Available as of Camel 2.14

The JAXB Data Format supports to specify the SchemaLocation when marshaling the XML.

Using the Java DSL, you can configure it in the following way:

```
JaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
jaxbDataFormat.setContextPath(Person.class.getPackage().getName());
jaxbDataFormat.setSchemaLocation("schema/person.xsd");
```

You can do the same using the XML DSL:

```
<marshal>
  <jaxb id="jaxb" schemaLocation="schema/person.xsd"/>
</marshal>
```

159.12. MARSHAL DATA THAT IS ALREADY XML

Available as of Camel 2.14.1

The JAXB marshaller requires that the message body is JAXB compatible, eg its a JAXBElement, eg a java instance that has JAXB annotations, or extend JAXBElement. There can be situations where the message body is already in XML, eg from a String type. There is a new option **mustBeJAXBElement** you can set to false, to relax this check, so the JAXB marshaller only attempts to marshal JAXBElements (javax.xml.bind.JAXBIntrospector#isElement returns true). And in those situations the marshaller fallbacks to marshal the message body as-is.

159.13. DEPENDENCIES

To use JAXB in your camel routes you need to add the a dependency on **camel-jaxb** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jaxb</artifactId>
```

```
  <version>x.x.x</version>
</dependency>
```

CHAPTER 160. JBOSS DATA GRID COMPONENT

160.1. RED HAT JBOSS DATA GRID COMPONENT WITH APACHE CAMEL

Using Camel with Red Hat JBoss Data Grid and Red Hat JBoss Fuse simplifies integration in large enterprise applications by providing a wide variety of transports and APIs that add connectivity.

JBoss Data Grid provides support for caching on Camel routes in JBoss Fuse, partially replacing Ehcache. JBoss Data Grid is supported as an embedded cache (local or clustered) or as a remote cache in a Camel route.

For more information about running the Red Hat JBoss Data Grid with Apache Camel, see Introducing Red Hat JBoss Data Grid.

CHAPTER 161. JCACHE COMPONENT

Available as of Camel version 2.17

The jcache component enables you to perform caching operations using JSR107/JCache as cache implementation.

161.1. URI FORMAT

```
jcache:cacheName[?options]
```

161.2. URI OPTIONS

The JCache endpoint is configured using URI syntax:

```
jcache:cacheName
```

with the following path and query parameters:

161.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
cacheName	**Required** The name of the cache		String

161.2.2. Query Parameters (22 parameters):

Name	Description	Default	Type
cacheConfiguration (common)	A Configuration for the Cache		Configuration
cacheConfigurationProperties (common)	The Properties for the javax.cache.spi.CachingProvider to create the CacheManager		Properties
cachingProvider (common)	The fully qualified class name of the javax.cache.spi.CachingProvider		String
configurationUri (common)	An implementation specific URI for the CacheManager		String
managementEnabled (common)	Whether management gathering is enabled	false	boolean

Name	Description	Default	Type
readThrough (common)	If read-through caching should be used	false	boolean
statisticsEnabled (common)	Whether statistics gathering is enabled	false	boolean
storeByValue (common)	If cache should use store-by-value or store-by-reference semantics	true	boolean
writeThrough (common)	If write-through caching should be used	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
filteredEvents (consumer)	Events a consumer should filter. If using filteredEvents option, then eventFilters one will be ignored		List
oldValueRequired (consumer)	if the old value is required for events	false	boolean
synchronous (consumer)	if the the event listener should block the thread causing the event	false	boolean
eventFilters (consumer)	The CacheEntryEventFilter. If using eventFilters option, then filteredEvents one will be ignored		List
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
action (producer)	To configure using a cache operation by default. If an operation in the message header, then the operation from the header takes precedence.		String

Name	Description	Default	Type
cacheLoaderFactory (advanced)	The CacheLoader factory		CacheLoader>
cacheWriterFactory (advanced)	The CacheWriter factory		CacheWriter>
createCacheIfNotExists (advanced)	Configure if a cache need to be created if it does exist or can't be pre-configured.	true	boolean
expiryPolicyFactory (advanced)	The ExpiryPolicy factory		ExpiryPolicy>
lookupProviders (advanced)	Configure if a camel-cache should try to find implementations of jcache api in runtimes like OSGi.	false	boolean

The JCache component supports 5 options which are listed below.

Name	Description	Default	Type
cachingProvider (common)	The fully qualified class name of the javax.cache.spi.CachingProvider		String
cacheConfiguration (common)	A Configuration for the Cache		Configuration
cacheConfiguration Properties (common)	The Properties for the javax.cache.spi.CachingProvider to create the CacheManager		Properties
configurationUri (common)	An implementation specific URI for the CacheManager		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

CHAPTER 162. JCLOUDS COMPONENT

Available as of Camel version 2.9

This component allows interaction with cloud provider key-value engines (blobstores) and compute services. The component uses jclouds which is
a library that provides abstractions for blobstores and compute services.

ComputeService simplifies the task of managing machines in the cloud. For example, you can use ComputeService to start 5 machines and install your software on them.
BlobStore simplifies dealing with key-value providers such as Amazon S3. For example, BlobStore can give you a simple Map view of a container.

The camel jclouds component allows you to use both abstractions, as it specifes two types of endpoint the JcloudsBlobStoreEndpoint and the JcloudsComputeEndpoint. You can have both producers and consumers on a blobstore endpoint but you can only have producers on compute endpoints.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jclouds</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

162.1. CONFIGURING THE COMPONENT

The camel jclouds component will make use of multiple jclouds blobstores and compute services as long as they are passed to the component during initialization. The component accepts a list blobstores and compute services. Here is how it can be configured.

```
  <bean id="jclouds" class="org.apache.camel.component.jclouds.JcloudsComponent">
    <property name="computeServices">
      <list>
        <ref bean="computeService"/>
      </list>
    </property>
    <property name="blobStores">
      <list>
        <ref bean="blobStore"/>
      </list>
    </property>
  </bean>

  <!-- Creating a blobstore from spring / blueprint xml -->
  <bean id="blobStoreContextFactory" class="org.jclouds.blobstore.BlobStoreContextFactory"/>

  <bean id="blobStoreContext" factory-bean="blobStoreContextFactory" factory-
method="createContext">
    <constructor-arg name="provider" value="PROVIDER_NAME"/>
    <constructor-arg name="identity"  value="IDENTITY"/>
    <constructor-arg name="credential" value="CREDENTIAL"/>
  </bean>
```

```
    <bean id="blobStore" factory-bean="blobStoreContext" factory-method="getBlobStore"/>

    <!-- Creating a compute service from spring / blueprint xml -->
    <bean id="computeServiceContextFactory"
class="org.jclouds.compute.ComputeServiceContextFactory"/>

    <bean id="computeServiceContext" factory-bean="computeServiceContextFactory" factory-
method="createContext">
        <constructor-arg name="provider" value="PROVIDER_NAME"/>
        <constructor-arg name="identity"  value="IDENTITY"/>
        <constructor-arg name="credential" value="CREDENTIAL"/>
    </bean>

    <bean id="computeService" factory-bean="computeServiceContext" factory-
method="getComputeService"/>
```

As you can see the component is capable of handling multiple blobstores and compute services. The actual implementation that will be used by each endpoint is specified by passing the provider inside the URI.

162.2. JCLOUDS OPTIONS

```
jclouds:blobstore:[provider id][?options]
jclouds:compute:[provider id][?options]
```

The **provider id** is the name of the cloud provider that provides the target service (*e.g. aws-s3 or aws_ec2*).

You can append query options to the URI in the following format, **?option=value&option=value&...**

162.3. BLOBSTORE URI OPTIONS

The JClouds component supports 3 options which are listed below.

Name	Description	Default	Type
blobStores (common)	To use the given BlobStore which must be configured when using blobstore.		List
computeServices (common)	To use the given ComputeService which must be configured when use compute.		List
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The JClouds endpoint is configured using URI syntax:

```
jclouds:command:providerId
```

with the following path and query parameters:

162.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
command	**Required** What command to execute such as blobstore or compute.		JcloudsCommand
providerId	**Required** The name of the cloud provider that provides the target service (e.g. aws-s3 or aws_ec2).		String

162.3.2. Query Parameters (15 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
blobName (blobstore)	The name of the blob.		String
container (blobstore)	The name of the blob container.		String
directory (blobstore)	An optional directory name to use		String

Name	Description	Default	Type
group (compute)	The group that will be assigned to the newly created node. Values depend on the actual cloud provider.		String
hardwareId (compute)	The hardware that will be used for creating a node. Values depend on the actual cloud provider.		String
imageId (compute)	The imageId that will be used for creating a node. Values depend on the actual cloud provider.		String
locationId (compute)	The location that will be used for creating a node. Values depend on the actual cloud provider.		String
nodeId (compute)	The id of the node that will run the script or destroyed.		String
nodeState (compute)	To filter by node status to only select running nodes etc.		String
operation (compute)	Specifies the type of operation that will be performed to the blobstore.		String
user (compute)	The user on the target node that will run the script.		String

You can have as many of these options as you like.

```
jclouds:blobstore:aws-s3?
operation=CamelJcloudsGet&container=mycontainer&blobName=someblob
```

For producer endpoint you can override all of the above URI options by passing the appropriate headers to the message.

162.3.3. Message Headers for blobstore

Header	Description
CamelJcloudsOperation	The operation to be performed on the blob. The valid options are * PUT * GET
CamelJcloudsContainer	The name of the blob container.

Header	Description
CamelJcloudsBlobName	The name of the blob.

162.4. BLOBSTORE USAGE SAMPLES

162.4.1. Example 1: Putting to the blob

This example will show you how you can store any message inside a blob using the jclouds component.

```
from("direct:start")
    .to("jclouds:blobstore:aws-s3" +
       "?operation=PUT" +
       "&container=mycontainer" +
       "&blobName=myblob");
```

In the above example you can override any of the URI parameters with headers on the message. Here is how the above example would look like using xml to define our route.

```
<route>
   <from uri="direct:start"/>
   <to uri="jclouds:blobstore:aws-s3?operation=PUT&container=mycontainer&blobName=myblob"/>
</route>
```

162.4.2. Example 2: Getting/Reading from a blob

This example will show you how you can read the contnet of a blob using the jclouds component.

```
from("direct:start")
    .to("jclouds:blobstore:aws-s3" +
       "?operation=GET" +
       "&container=mycontainer" +
       "&blobName=myblob");
```

In the above example you can override any of the URI parameters with headers on the message. Here is how the above example would look like using xml to define our route.

```
<route>
   <from uri="direct:start"/>
   <to uri="jclouds:blobstore:aws-s3?operation=PUT&container=mycontainer&blobName=myblob"/>
</route>
```

162.4.3. Example 3: Consuming a blob

This example will consume all blob that are under the specified container. The generated exchange will contain the payload of the blob as body.

```
    from("jclouds:blobstore:aws-s3" +
        "?container=mycontainer")
        .to("direct:next");
```

You can achieve the same goal by using xml, as you can see below.

```
<route>
    <from uri="jclouds:blobstore:aws-s3?
operation=GET&container=mycontainer&blobName=myblob"/>
    <to uri="direct:next"/>
</route>
```

```
jclouds:compute:aws-ec2?
operation=CamelJcloudsCreateNode&imageId=AMI_XXXXX&locationId=eu-west-1&group=mygroup
```

162.5. COMPUTE USAGE SAMPLES

Below are some examples that demonstrate the use of jclouds compute producer in java dsl and spring/blueprint xml.

162.5.1. Example 1: Listing the available images.

```
    from("jclouds:compute:aws-ec2" +
        "&operation=CamelJCloudsListImages")
        .to("direct:next");
```

This will create a message that will contain the list of images inside its body. You can also do the same using xml.

```
<route>
    <from uri="jclouds:compute:aws-ec2?operation=CamelJCloudsListImages"/>
    <to uri="direct:next"/>
</route>
```

162.5.2. Example 2: Create a new node.

```
    from("direct:start").
    to("jclouds:compute:aws-ec2" +
        "?operation=CamelJcloudsCreateNode" +
        "&imageId=AMI_XXXXX" +
        "&locationId=XXXXX" +
        "&group=myGroup");
```

This will create a new node on the cloud provider. The out message in this case will be a set of metadata that contains information about the newly created node (e.g. the ip, hostname etc). Here is the same using spring xml.

```
<route>
    <from uri="direct:start"/>
```

```
    <to uri="jclouds:compute:aws-ec2?
operation=CamelJcloudsCreateNode&imageId=AMI_XXXXX&locationId=XXXXX&group=myGroup"/>
</route>
```

162.5.3. Example 3: Run a shell script on running node.

```
    from("direct:start").
    to("jclouds:compute:aws-ec2" +
       "?operation=CamelJcloudsRunScript" +
       "?nodeId=10" +
       "&user=ubuntu");
```

The sample above will retrieve the body of the in message, which is expected to contain the shell script to be executed. Once the script is retrieved, it will be sent to the node for execution under the specified user (*in order case ubuntu*). The target node is specified using its nodeId. The nodeId can be retrieved either upon the creation of the node, it will be part of the resulting metadata or by a executing a LIST_NODES operation.

Note This will require that the compute service that will be passed to the component, to be initialized with the appropriate jclouds ssh capable module (*e.g. jsch or sshj*).

Here is the same using spring xml.

```
<route>
    <from uri="direct:start"/>
    <to uri="jclouds:compute:aws-ec2?operation=CamelJcloudsListNodes&?
nodeId=10&user=ubuntu"/>
</route>
```

162.5.4. See also

If you want to find out more about jclouds here is list of interesting resources

Jclouds Blobstore wiki

Jclouds Compute wiki

CHAPTER 163. JCR COMPONENT

Available as of Camel version 1.3

The **jcr** component allows you to add/read nodes to/from a JCR compliant content repository (for example, Apache Jackrabbit) with its producer, or register an EventListener with the consumer.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jcr</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

163.1. URI FORMAT

```
jcr://user:password@repository/path/to/node
```

Consumer added

From **Camel 2.10** onwards you can use consumer as an EventListener in JCR or a producer to read a node by identifier.

163.2. USAGE

The **repository** element of the URI is used to look up the JCR **Repository** object in the Camel context registry.

163.2.1. JCR Options

The JCR component has no options.

The JCR endpoint is configured using URI syntax:

```
jcr:host/base
```

with the following path and query parameters:

163.2.2. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** Name of the javax.jcr.Repository to lookup from the Camel registry to be used.		String
base	Get the base node when accessing the repository		String

163.2.3. Query Parameters (14 parameters):

Name	Description	Default	Type
deep (common)	When isDeep is true, events whose associated parent node is at absPath or within its subgraph are received.	false	boolean
eventTypes (common)	eventTypes (a combination of one or more event types encoded as a bit mask value such as javax.jcr.observation.Event.NODE_ADDED, javax.jcr.observation.Event.NODE_REMOVED, etc.).		int
nodeTypeNames (common)	When a comma separated nodeTypeName list string is set, only events whose associated parent node has one of the node types (or a subtype of one of the node types) in this list will be received.		String
noLocal (common)	If noLocal is true, then events generated by the session through which the listener was registered are ignored. Otherwise, they are not ignored.	false	boolean
password (common)	Password for login		String
sessionLiveCheckInterval (common)	Interval in milliseconds to wait before each session live checking The default value is 60000 ms.	60000	long
sessionLiveCheckIntervalOnStart (common)	Interval in milliseconds to wait before the first session live checking. The default value is 3000 ms.	3000	long
username (common)	Username for login		String
uuids (common)	When a comma separated uuid list string is set, only events whose associated parent node has one of the identifiers in the comma separated uuid list will be received.		String
workspaceName (common)	The workspace to access. If it's not specified then the default one will be used		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Please note that the JCR Producer used message properties instead of message headers in Camel versions earlier than 2.12.3. See https://issues.apache.org/jira/browse/CAMEL-7067 for more details.

163.3. EXAMPLE

The snippet below creates a node named **node** under the **/home/test** node in the content repository. One additional property is added to the node as well: **my.contents.property** which will contain the body of the message being sent.

```
from("direct:a").setHeader(JcrConstants.JCR_NODE_NAME, constant("node"))
    .setHeader("my.contents.property", body())
    .to("jcr://user:pass@repository/home/test");
```

The following code will register an EventListener under the path import-application/inbox for Event.NODE_ADDED and Event.NODE_REMOVED events (event types 1 and 2, both masked as 3) and listening deep for all the children.

```
<route>
    <from uri="jcr://user:pass@repository/import-application/inbox?eventTypes=3&deep=true" />
    <to uri="direct:execute-import-application" />
</route>
```

163.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 164. JDBC COMPONENT

Available as of Camel version 1.2

The **jdbc** component enables you to access databases through JDBC, where SQL queries (SELECT) and operations (INSERT, UPDATE, etc) are sent in the message body. This component uses the standard JDBC API, unlike the SQL Component component, which uses spring-jdbc.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jdbc</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

This component can only be used to define producer endpoints, which means that you cannot use the JDBC component in a **from()** statement.

164.1. URI FORMAT

```
jdbc:dataSourceName[?options]
```

This component only supports producer endpoints.

You can append query options to the URI in the following format, **?option=value&option=value&…**

164.2. OPTIONS

The JDBC component supports 2 options which are listed below.

Name	Description	Default	Type
dataSource (producer)	To use the DataSource instance instead of looking up the data source by name from the registry.		DataSource
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The JDBC endpoint is configured using URI syntax:

```
jdbc:dataSourceName
```

with the following path and query parameters:

164.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
dataSourceName	**Required** Name of DataSource to lookup in the Registry.		String

164.2.2. Query Parameters (13 parameters):

Name	Description	Default	Type
allowNamedParameters (producer)	Whether to allow using named parameters in the queries.	true	boolean
outputClass (producer)	Specify the full package and class name to use as conversion when outputType=SelectOne or SelectList.		String
outputType (producer)	Determines the output the producer should use.	SelectList	JdbcOutputType
parameters (producer)	Optional parameters to the java.sql.Statement. For example to set maxRows, fetchSize etc.		Map
readSize (producer)	The default maximum number of rows that can be read by a polling query. The default value is 0.		int
resetAutoCommit (producer)	Camel will set the autoCommit on the JDBC connection to be false, commit the change after executed the statement and reset the autoCommit flag of the connection at the end, if the resetAutoCommit is true. If the JDBC connection doesn't support to reset the autoCommit flag, you can set the resetAutoCommit flag to be false, and Camel will not try to reset the autoCommit flag. When used with XA transactions you most likely need to set it to false so that the transaction manager is in charge of committing this tx.	true	boolean
transacted (producer)	Whether transactions are in use.	false	boolean
useGetBytesForBlob (producer)	To read BLOB columns as bytes instead of string data. This may be needed for certain databases such as Oracle where you must read BLOB columns as bytes.	false	boolean

Name	Description	Default	Type
useHeadersAsParameters (producer)	Set this option to true to use the prepareStatementStrategy with named parameters. This allows to define queries with named placeholders, and use headers with the dynamic values for the query placeholders.	false	boolean
useJDBC4ColumnNameAndLabelSemantics (producer)	Sets whether to use JDBC 4 or JDBC 3.0 or older semantic when retrieving column name. JDBC 4.0 uses columnLabel to get the column name where as JDBC 3.0 uses both columnName or columnLabel. Unfortunately JDBC drivers behave differently so you can use this option to work out issues around your JDBC driver if you get problem using this component This option is default true.	true	boolean
beanRowMapper (advanced)	To use a custom org.apache.camel.component.jdbc.BeanRowMapper when using outputClass. The default implementation will lower case the row names and skip underscores, and dashes. For example CUST_ID is mapped as custId.		BeanRowMapper
prepareStatementStrategy (advanced)	Allows to plugin to use a custom org.apache.camel.component.jdbc.JdbcPrepareStatementStrategy to control preparation of the query and prepared statement.		JdbcPrepareStatement Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

164.3. RESULT

By default the result is returned in the OUT body as an **ArrayList<HashMap<String, Object>>**. The **List** object contains the list of rows and the **Map** objects contain each row with the **String** key as the column name. You can use the option **outputType** to control the result.

Note: This component fetches **ResultSetMetaData** to be able to return the column name as the key in the **Map**.

164.3.1. Message Headers

Header	Description
CamelJdbcRowCount	If the query is a **SELECT**, query the row count is returned in this OUT header.
CamelJdbcUpdateCount	If the query is an **UPDATE**, query the update count is returned in this OUT header.
CamelGeneratedKeysRows	**Camel 2.10:** Rows that contains the generated kets.
CamelGeneratedKeysRowCount	**Camel 2.10:** The number of rows in the header that contains generated keys.
CamelJdbcColumnNames	**Camel 2.11.1:** The column names from the ResultSet as a**java.util.Set** type.
CamelJdbcParametes	**Camel 2.12:** A **java.util.Map** which has the headers to be used if**useHeadersAsParameters** has been enabled.

164.4. GENERATED KEYS

Available as of Camel 2.10

If you insert data using SQL INSERT, then the RDBMS may support auto generated keys. You can instruct the JDBC producer to return the generated keys in headers.
To do that set the header **CamelRetrieveGeneratedKeys=true**. Then the generated keys will be provided as headers with the keys listed in the table above.

You can see more details in this unit test.

Using generated keys does not work with together with named parameters.

164.5. USING NAMED PARAMETERS

Available as of Camel 2.12

In the given route below, we want to get all the projects from the projects table. Notice the SQL query has 2 named parameters, :?lic and :?min.
Camel will then lookup these parameters from the message headers. Notice in the example above we set two headers with constant value
for the named parameters:

```
from("direct:projects")
  .setHeader("lic", constant("ASF"))
  .setHeader("min", constant(123))
  .setBody("select * from projects where license = :?lic and id > :?min order by id")
  .to("jdbc:myDataSource?useHeadersAsParameters=true")
```

You can also store the header values in a **java.util.Map** and store the map on the headers with the key **CamelJdbcParameters**.

164.6. SAMPLES

In the following example, we fetch the rows from the customer table.

First we register our datasource in the Camel registry as **testdb**:

Then we configure a route that routes to the JDBC component, so the SQL will be executed. Note how we refer to the **testdb** datasource that was bound in the previous step:

Or you can create a **DataSource** in Spring like this:

We create an endpoint, add the SQL query to the body of the IN message, and then send the exchange. The result of the query is returned in the OUT body:

If you want to work on the rows one by one instead of the entire ResultSet at once you need to use the Splitter EIP such as:

In Camel 2.13.x or older

In Camel 2.14.x or newer

```
from("direct:hello")
// here we split the data from the testdb into new messages one by one
// so the mock endpoint will receive a message per row in the table
// the StreamList option allows to stream the result of the query without creating a List of rows
// and notice we also enable streaming mode on the splitter
.to("jdbc:testdb?outputType=StreamList")
  .split(body()).streaming()
  .to("mock:result");
```

164.7. SAMPLE - POLLING THE DATABASE EVERY MINUTE

If we want to poll a database using the JDBC component, we need to combine it with a polling scheduler such as the Timer or Quartz etc. In the following example, we retrieve data from the database every 60 seconds:

```
from("timer://foo?period=60000").setBody(constant("select * from
customer")).to("jdbc:testdb").to("activemq:queue:customers");
```

164.8. SAMPLE - MOVE DATA BETWEEN DATA SOURCES

A common use case is to query for data, process it and move it to another data source (ETL operations). In the following example, we retrieve new customer records from the source table every hour, filter/transform them and move them to a destination table:

```
from("timer://MoveNewCustomersEveryHour?period=3600000")
    .setBody(constant("select * from customer where create_time > (sysdate-1/24)"))
    .to("jdbc:testdb")
    .split(body())
        .process(new MyCustomerProcessor()) //filter/transform results as needed
        .setBody(simple("insert into processed_customer values('${body[ID]}','${body[NAME]}')"))
        .to("jdbc:testdb");
```

164.9. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- SQL

CHAPTER 165. JETTY 9 COMPONENT

Available as of Camel version 1.2

WARNING

The producer is deprecated - do not use. We only recommend using jetty as consumer (eg from jetty)

The **jetty** component provides HTTP-based endpoints for consuming and producing HTTP requests. That is, the Jetty component behaves as a simple Web server.
Jetty can also be used as a http client which mean you can also use it with Camel as a producer.

Stream

The **assert** call appears in this example, because the code is part of an unit test.Jetty is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**.
If you find a situation where the message body appears to be empty or you need to access the Exchange.HTTP_RESPONSE_CODE data multiple times (e.g.: doing multicasting, or redelivery error handling), you should use Stream caching or convert the message body to a **String** which is safe to be re-read multiple times.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jetty</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

165.1. URI FORMAT

```
jetty:http://hostname[:port][/resourceUri][?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

165.2. OPTIONS

The Jetty 9 component supports 33 options which are listed below.

Name	Description	Default	Type

Name	Description	Default	Type
sslKeyPassword (security)	The key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the keystore command's -keypass option).		String
sslPassword (security)	The ssl password, which is required to access the keystore file (this is the same password that is supplied to the keystore command's -storepass option).		String
keystore (security)	Specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a key entry.		String
errorHandler (advanced)	This option is used to set the ErrorHandler that Jetty server uses.		ErrorHandler
sslSocketConnectors (security)	A map which contains per port number specific SSL connectors.		Map
socketConnectors (security)	A map which contains per port number specific HTTP connectors. Uses the same principle as sslSocketConnectors.		Map
httpClientMinThreads (producer)	To set a value for minimum number of threads in HttpClient thread pool. Notice that both a min and max size must be configured.		Integer
httpClientMaxThreads (producer)	To set a value for maximum number of threads in HttpClient thread pool. Notice that both a min and max size must be configured.		Integer
minThreads (consumer)	To set a value for minimum number of threads in server thread pool. Notice that both a min and max size must be configured.		Integer
maxThreads (consumer)	To set a value for maximum number of threads in server thread pool. Notice that both a min and max size must be configured.		Integer
threadPool (consumer)	To use a custom thread pool for the server. This option should only be used in special circumstances.		ThreadPool
enableJmx (common)	If this option is true, Jetty JMX support will be enabled for this endpoint.	false	boolean

Name	Description	Default	Type
jettyHttpBinding (advanced)	To use a custom org.apache.camel.component.jetty.JettyHttpBinding, which are used to customize how a response should be written for the producer.		JettyHttpBinding
httpBinding (advanced)	Not to be used - use JettyHttpBinding instead.		HttpBinding
httpConfiguration (advanced)	Jetty component does not use HttpConfiguration.		HttpConfiguration
mbContainer (advanced)	To use a existing configured org.eclipse.jetty.jmx.MBeanContainer if JMX is enabled that Jetty uses for registering mbeans.		MBeanContainer
sslSocketConnectorProperties (security)	A map which contains general SSL connector properties.		Map
socketConnectorProperties (security)	A map which contains general HTTP connector properties. Uses the same principle as sslSocketConnectorProperties.		Map
continuationTimeout (consumer)	Allows to set a timeout in millis when using Jetty as consumer (server). By default Jetty uses 30000. You can use a value of = 0 to never expire. If a timeout occurs then the request will be expired and Jetty will return back a http error 503 to the client. This option is only in use when using Jetty with the Asynchronous Routing Engine.	30000	Long
useContinuation (consumer)	Whether or not to use Jetty continuations for the Jetty Server.	true	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters	false	boolean
responseBufferSize (common)	Allows to configure a custom value of the response buffer size on the Jetty connectors.		Integer
requestBufferSize (common)	Allows to configure a custom value of the request buffer size on the Jetty connectors.		Integer

Name	Description	Default	Type
requestHeaderSize (common)	Allows to configure a custom value of the request header size on the Jetty connectors.		Integer
responseHeaderSize (common)	Allows to configure a custom value of the response header size on the Jetty connectors.		Integer
proxyHost (proxy)	To use a http proxy to configure the hostname.		String
proxyPort (proxy)	To use a http proxy to configure the port number.		Integer
useXForwardedFor Header (common)	To use the X-Forwarded-For header in HttpServletRequest.getRemoteAddr.	false	boolean
sendServerVersion (consumer)	If the option is true, jetty server will send the date header to the client which sends the request. NOTE please make sure there is no any other camel-jetty endpoint is share the same port, otherwise this option may not work as expected.	true	boolean
allowJavaSerialized Object (advanced)	Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Jetty 9 endpoint is configured using URI syntax:

```
jetty:httpUri
```

with the following path and query parameters:

165.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
httpUri	**Required** The url of the HTTP endpoint to call.		URI

165.2.2. Query Parameters (54 parameters):

Name	Description	Default	Type
chunked (common)	If this option is false the Servlet will disable the HTTP streaming and set the content-length header on the response	true	boolean
disableStreamCache (common)	Determines whether or not the raw input stream from Servlet is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Servlet input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times. If you use Servlet to bridge/proxy an endpoint then consider enabling this option to improve performance, in case you do not need to read the message payload multiple times. The http/http4 producer will by default cache the response body stream. If setting this option to true, then the producers will not cache the response body stream but use the response stream as-is as the message body.	false	boolean
enableMultipartFilter (common)	Whether Jetty org.eclipse.jetty.servlets.MultiPartFilter is enabled or not. You should set this value to false when bridging endpoints, to ensure multipart requests is proxied/bridged as well.	false	boolean
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy

Name	Description	Default	Type
transferException (common)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
httpBinding (common)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
async (consumer)	Configure the consumer to work in async mode	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
continuationTimeout (consumer)	Allows to set a timeout in millis when using Jetty as consumer (server). By default Jetty uses 30000. You can use a value of = 0 to never expire. If a timeout occurs then the request will be expired and Jetty will return back a http error 503 to the client. This option is only in use when using Jetty with the Asynchronous Routing Engine.	30000	Long
enableCORS (consumer)	If the option is true, Jetty server will setup the CrossOriginFilter which supports the CORS out of box.	false	boolean
enableJmx (consumer)	If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.	false	boolean
httpMethodRestrict (consumer)	Used to only allow consuming if the HttpMethod matches, such as GET/POST/PUT etc. Multiple methods can be specified separated by comma.		String

Name	Description	Default	Type
matchOnUriPrefix (consumer)	Whether or not the consumer should try to find a target consumer by matching the URI prefix if no exact match is found.	false	boolean
responseBufferSize (consumer)	To use a custom buffer size on the javax.servlet.ServletResponse.		Integer
sendDateHeader (consumer)	If the option is true, jetty server will send the date header to the client which sends the request. NOTE please make sure there is no any other camel-jetty endpoint is share the same port, otherwise this option may not work as expected.	false	boolean
sendServerVersion (consumer)	If the option is true, jetty will send the server header with the jetty version information to the client which sends the request. NOTE please make sure there is no any other camel-jetty endpoint is share the same port, otherwise this option may not work as expected.	true	boolean
sessionSupport (consumer)	Specifies whether to enable the session manager on the server side of Jetty.	false	boolean
useContinuation (consumer)	Whether or not to use Jetty continuations for the Jetty Server.		Boolean
eagerCheckContentAvailable (consumer)	Whether to eager check whether the HTTP requests has content if the content-length header is 0 or not present. This can be turned on in case HTTP clients do not send streamed data.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
filterInitParameters (consumer)	Configuration of the filter init parameters. These parameters will be applied to the filter list before starting the jetty server.		Map
filtersRef (consumer)	Allows using a custom filters which is putted into a list and can be find in the Registry. Multiple values can be separated by comma.		String

Name	Description	Default	Type
handlers (consumer)	Specifies a comma-delimited set of Handler instances to lookup in your Registry. These handlers are added to the Jetty servlet context (for example, to add security). Important: You can not use different handlers with different Jetty endpoints using the same port number. The handlers is associated to the port number. If you need different handlers, then use different port numbers.		String
httpBindingRef (consumer)	**Deprecated** Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.		String
multipartFilter (consumer)	Allows using a custom multipart filter. Note: setting multipartFilterRef forces the value of enableMultipartFilter to true.		Filter
multipartFilterRef (consumer)	**Deprecated** Allows using a custom multipart filter. Note: setting multipartFilterRef forces the value of enableMultipartFilter to true.		String
optionsEnabled (consumer)	Specifies whether to enable HTTP OPTIONS for this Servlet consumer. By default OPTIONS is turned off.	false	boolean
traceEnabled (consumer)	Specifies whether to enable HTTP TRACE for this Servlet consumer. By default TRACE is turned off.	false	boolean
bridgeEndpoint (producer)	If the option is true, HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the option throwExceptionOnFailure to be false to let the HttpProducer send all the fault response back.	false	boolean
connectionClose (producer)	Specifies whether a Connection Close header must be added to HTTP Request. By default connectionClose is false.	false	boolean
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
copyHeaders (producer)	If this option is true then IN exchange headers will be copied to OUT exchange headers according to copy strategy. Setting this to false, allows to only include the headers from the HTTP response (not propagating IN headers).	true	boolean

Name	Description	Default	Type
httpClientMaxThreads (producer)	To set a value for maximum number of threads in HttpClient thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured. If not set it default to max 254 threads used in Jettys thread pool.	254	Integer
httpClientMinThreads (producer)	To set a value for minimum number of threads in HttpClient thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured. If not set it default to min 8 threads used in Jettys thread pool.	8	Integer
httpMethod (producer)	Configure the HTTP method to use. The HttpMethod header cannot override this option if set.		HttpMethods
ignoreResponseBody (producer)	If this option is true, The http producer won't read response body and cache the input stream	false	boolean
preserveHostHeader (producer)	If the option is true, HttpProducer will set the Host header to the value contained in the current exchange Host header, useful in reverse proxy applications where you want the Host header received by the downstream server to reflect the URL called by the upstream client, this allows applications which use the Host header to generate accurate URL's for a proxied service	false	boolean
throwExceptionOnFailure (producer)	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
httpClient (producer)	Sets a shared HttpClient to use for all producers created by this endpoint. By default each producer will use a new http client, and not share. Important: Make sure to handle the lifecycle of the shared client, such as stopping the client, when it is no longer in use. Camel will call the start method on the client to ensure its started when this endpoint creates a producer. This options should only be used in special circumstances.		HttpClient

Name	Description	Default	Type
httpClientParameters (producer)	Configuration of Jetty's HttpClient. For example, setting httpClient.idleTimeout=30000 sets the idle timeout to 30 seconds. And httpClient.timeout=30000 sets the request timeout to 30 seconds, in case you want to timeout sooner if you have long running request/response calls.		Map
jettyBinding (producer)	To use a custom JettyHttpBinding which be used to customize how a response should be written for the producer.		JettyHttpBinding
jettyBindingRef (producer)	**Deprecated** To use a custom JettyHttpBinding which be used to customize how a response should be written for the producer.		String
okStatusCodeRange (producer)	The status codes which are considered a success response. The values are inclusive. Multiple ranges can be defined, separated by comma, e.g. 200-204,209,301-304. Each range must be a single number or from-to with the dash included.	200-299	String
urlRewrite (producer)	**Deprecated** Refers to a custom org.apache.camel.component.http.UrlRewrite which allows you to rewrite urls when you bridge/proxy endpoints. See more details at http://camel.apache.org/urlrewrite.html		UrlRewrite
mapHttpMessageBody (advanced)	If this option is true then IN exchange Body of the exchange will be mapped to HTTP body. Setting this to false will avoid the HTTP mapping.	true	boolean
mapHttpMessageFormUrlEncodedBody (advanced)	If this option is true then IN exchange Form Encoded body of the exchange will be mapped to HTTP. Setting this to false will avoid the HTTP Form Encoded body mapping.	true	boolean
mapHttpMessageHeaders (advanced)	If this option is true then IN exchange Headers of the exchange will be mapped to HTTP headers. Setting this to false will avoid the HTTP Headers mapping.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
proxyAuthScheme (proxy)	Proxy authentication scheme to use		String

Name	Description	Default	Type
proxyHost (proxy)	Proxy hostname to use		String
proxyPort (proxy)	Proxy port to use		int
authHost (security)	Authentication host to use with NTML		String
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters

165.3. MESSAGE HEADERS

Camel uses the same message headers as the HTTP component. From Camel 2.2, it also uses (Exchange.HTTP_CHUNKED,CamelHttpChunked) header to turn on or turn off the chuched encoding on the camel-jetty consumer.

Camel also populates **all** request.parameter and request.headers. For example, given a client request with the URL, **http://myserver/myserver?orderid=123**, the exchange will contain a header named **orderid** with the value 123.

Starting with Camel 2.2.0, you can get the request.parameter from the message header not only from Get Method, but also other HTTP method.

165.4. USAGE

The Jetty component supports both consumer and producer endpoints. Another option for producing to other HTTP endpoints, is to use the HTTP Component

165.5. PRODUCER EXAMPLE

WARNING

The producer is deprecated - do not use. We only recommend using jetty as consumer (eg from jetty)

The following is a basic example of how to send an HTTP request to an existing HTTP endpoint.

in Java DSL

```
from("direct:start").to("jetty://http://www.google.com");
```

or in Spring XML

```
<route>
  <from uri="direct:start"/>
  <to uri="jetty://http://www.google.com"/>
<route>
```

165.6. CONSUMER EXAMPLE

In this sample we define a route that exposes a HTTP service at **http://localhost:8080/myapp/myservice**:

Usage of localhost

When you specify **localhost** in a URL, Camel exposes the endpoint only on the local TCP/IP network interface, so it cannot be accessed from outside the machine it operates on.

If you need to expose a Jetty endpoint on a specific network interface, the numerical IP address of this interface should be used as the host. If you need to expose a Jetty endpoint on all network interfaces, the **0.0.0.0** address should be used.

To listen across an entire URI prefix, see How do I let Jetty match wildcards .

If you actually want to expose routes by HTTP and already have a Servlet, you should instead refer to the Servlet Transport.

Our business logic is implemented in the **MyBookService** class, which accesses the HTTP request contents and then returns a response.
Note: The **assert** call appears in this example, because the code is part of an unit test.

The following sample shows a content-based route that routes all requests containing the URI parameter, **one**, to the endpoint, **mock:one**, and all others to **mock:other**.

So if a client sends the HTTP request, **http://serverUri?one=hello**, the Jetty component will copy the HTTP request parameter, **one** to the exchange's **in.header**. We can then use the **simple** language to route exchanges that contain this header to a specific endpoint and all others to another. If we used a language more powerful than Simple (such as OGNL) we could also test for the parameter value and do routing based on the header value as well.

165.7. SESSION SUPPORT

The session support option, **sessionSupport**, can be used to enable a **HttpSession** object and access the session object while processing the exchange. For example, the following route enables sessions:

```
<route>
  <from uri="jetty:http://0.0.0.0/myapp/myservice/?sessionSupport=true"/>
  <processRef ref="myCode"/>
<route>
```

The **myCode** Processor can be instantiated by a Spring **bean** element:

```
<bean id="myCode"class="com.mycompany.MyCodeProcessor"/>
```

Where the processor implementation can access the **HttpSession** as follows:

```
public void process(Exchange exchange) throws Exception {
```

```
    HttpSession session = exchange.getIn(HttpMessage.class).getRequest().getSession();
    ...
}
```

165.8. SSL SUPPORT (HTTPS)

Using the JSSE Configuration Utility

As of Camel 2.8, the Jetty component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Jetty component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

JettyComponent jettyComponent = getContext().getComponent("jetty", JettyComponent.class);
jettyComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
     id="sslContextParameters">
    <camel:keyManagers
       keyPassword="keyPassword">
      <camel:keyStore
         resource="/users/home/server/keystore.jks"
         password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="jetty:https://127.0.0.1/mail/?sslContextParameters=#sslContextParameters"/>
...
```

Configuring Jetty Directly

Jetty provides SSL support out of the box. To enable Jetty to run in SSL mode, simply format the URI with the **https:**// prefix---for example:

```
<from uri="jetty:https://0.0.0.0/myapp/myservice/"/>
```

Jetty also needs to know where to load your keystore from and what passwords to use in order to load the correct SSL certificate. Set the following JVM System Properties:

until Camel 2.2

- **jetty.ssl.keystore** specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a *key entry*. A key entry stores the X.509 certificate (effectively, the *public key*) and also its associated private key.
- **jetty.ssl.password** the store password, which is required to access the keystore file (this is the same password that is supplied to the **keystore** command's **-storepass** option).
- **jetty.ssl.keypassword** the key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the **keystore** command's **-keypass** option).

from Camel 2.3 onwards

- **org.eclipse.jetty.ssl.keystore** specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a *key entry*. A key entry stores the X.509 certificate (effectively, the *public key*) and also its associated private key.
- **org.eclipse.jetty.ssl.password** the store password, which is required to access the keystore file (this is the same password that is supplied to the **keystore** command's **-storepass** option).
- **org.eclipse.jetty.ssl.keypassword** the key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the **keystore** command's **-keypass** option).

For details of how to configure SSL on a Jetty endpoint, read the following documentation at the Jetty Site: http://docs.codehaus.org/display/JETTY/How+to+configure+SSL

Some SSL properties aren't exposed directly by Camel, however Camel does expose the underlying SslSocketConnector, which will allow you to set properties like needClientAuth for mutual authentication requiring a client certificate or wantClientAuth for mutual authentication where a client doesn't need a certificate but can have one. There's a slight difference between the various Camel versions:

Up to Camel 2.2

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="sslSocketConnectors">
    <map>
      <entry key="8043">
        <bean class="org.mortbay.jetty.security.SslSocketConnector">
          <property name="password"value="..."/>
          <property name="keyPassword"value="..."/>
          <property name="keystore"value="..."/>
          <property name="needClientAuth"value="..."/>
          <property name="truststore"value="..."/>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

Camel 2.3, 2.4

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="sslSocketConnectors">
```

```
        <map>
          <entry key="8043">
            <bean class="org.eclipse.jetty.server.ssl.SslSocketConnector">
              <property name="password"value="..."/>
              <property name="keyPassword"value="..."/>
              <property name="keystore"value="..."/>
              <property name="needClientAuth"value="..."/>
              <property name="truststore"value="..."/>
            </bean>
          </entry>
        </map>
      </property>
    </bean>
```

*From Camel 2.5 we switch to use SslSelectChannelConnector *

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="sslSocketConnectors">
    <map>
      <entry key="8043">
        <bean class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
          <property name="password"value="..."/>
          <property name="keyPassword"value="..."/>
          <property name="keystore"value="..."/>
          <property name="needClientAuth"value="..."/>
          <property name="truststore"value="..."/>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

The value you use as keys in the above map is the port you configure Jetty to listen on.

165.8.1. Configuring general SSL properties

Available as of Camel 2.5

Instead of a per port number specific SSL socket connector (as shown above) you can now configure general properties which applies for all SSL socket connectors (which is not explicit configured as above with the port number as entry).

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="sslSocketConnectorProperties">
    <map>
      <entry key="password"value="..."/>
      <entry key="keyPassword"value="..."/>
      <entry key="keystore"value="..."/>
      <entry key="needClientAuth"value="..."/>
      <entry key="truststore"value="..."/>
    </map>
  </property>
</bean>
```

165.8.2. How to obtain reference to the X509Certificate

Jetty stores a reference to the certificate in the HttpServletRequest which you can access from code as follows:

```
HttpServletRequest req = exchange.getIn().getBody(HttpServletRequest.class);
X509Certificate cert = (X509Certificate) req.getAttribute("javax.servlet.request.X509Certificate")
```

165.8.3. Configuring general HTTP properties

Available as of Camel 2.5

Instead of a per port number specific HTTP socket connector (as shown above) you can now configure general properties which applies for all HTTP socket connectors (which is not explicit configured as above with the port number as entry).

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="socketConnectorProperties">
    <map>
      <entry key="acceptors" value="4"/>
      <entry key="maxIdleTime" value="300000"/>
    </map>
  </property>
</bean>
```

165.8.4. Obtaining X-Forwarded-For header with HttpServletRequest.getRemoteAddr()

If the HTTP requests are handled by an Apache server and forwarded to jetty with mod_proxy, the original client IP address is in the X-Forwarded-For header and the HttpServletRequest.getRemoteAddr() will return the address of the Apache proxy.

Jetty has a forwarded property which takes the value from X-Forwarded-For and places it in the HttpServletRequest remoteAddr property. This property is not available directly through the endpoint configuration but it can be easily added using the socketConnectors property:

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="socketConnectors">
    <map>
      <entry key="8080">
        <bean class="org.eclipse.jetty.server.nio.SelectChannelConnector">
          <property name="forwarded" value="true"/>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

This is particularly useful when an existing Apache server handles TLS connections for a domain and proxies them to application servers internally.

165.9. DEFAULT BEHAVIOR FOR RETURNING HTTP STATUS CODES

The default behavior of HTTP status codes is defined by the **org.apache.camel.component.http.DefaultHttpBinding** class, which handles how a response is written and also sets the HTTP status code.

If the exchange was processed successfully, the 200 HTTP status code is returned.
If the exchange failed with an exception, the 500 HTTP status code is returned, and the stacktrace is returned in the body. If you want to specify which HTTP status code to return, set the code in the **Exchange.HTTP_RESPONSE_CODE** header of the OUT message.

165.10. CUSTOMIZING HTTPBINDING

By default, Camel uses the **org.apache.camel.component.http.DefaultHttpBinding** to handle how a response is written. If you like, you can customize this behavior either by implementing your own **HttpBinding** class or by extending **DefaultHttpBinding** and overriding the appropriate methods.

The following example shows how to customize the **DefaultHttpBinding** in order to change how exceptions are returned:

We can then create an instance of our binding and register it in the Spring registry as follows:

```
<bean id="mybinding"class="com.mycompany.MyHttpBinding"/>
```

And then we can reference this binding when we define the route:

```
<route><from uri="jetty:http://0.0.0.0:8080/myapp/myservice?httpBindingRef=mybinding"/><to
uri="bean:doSomething"/></route>
```

165.11. JETTY HANDLERS AND SECURITY CONFIGURATION

You can configure a list of Jetty handlers on the endpoint, which can be useful for enabling advanced Jetty security features. These handlers are configured in Spring XML as follows:

```
<-- Jetty Security handling -->
<bean id="userRealm" class="org.mortbay.jetty.plus.jaas.JAASUserRealm">
   <property name="name" value="tracker-users"/>
   <property name="loginModuleName" value="ldaploginmodule"/>
</bean>

<bean id="constraint" class="org.mortbay.jetty.security.Constraint">
   <property name="name" value="BASIC"/>
   <property name="roles" value="tracker-users"/>
   <property name="authenticate" value="true"/>
</bean>

<bean id="constraintMapping" class="org.mortbay.jetty.security.ConstraintMapping">
   <property name="constraint" ref="constraint"/>
   <property name="pathSpec" value="/*"/>
</bean>

<bean id="securityHandler" class="org.mortbay.jetty.security.SecurityHandler">
   <property name="userRealm" ref="userRealm"/>
   <property name="constraintMappings" ref="constraintMapping"/>
</bean>
```

And from Camel 2.3 onwards you can configure a list of Jetty handlers as follows:

```
<-- Jetty Security handling -->
<bean id="constraint" class="org.eclipse.jetty.http.security.Constraint">
   <property name="name" value="BASIC"/>
   <property name="roles" value="tracker-users"/>
   <property name="authenticate" value="true"/>
</bean>

<bean id="constraintMapping" class="org.eclipse.jetty.security.ConstraintMapping">
   <property name="constraint" ref="constraint"/>
   <property name="pathSpec" value="/*"/>
</bean>

<bean id="securityHandler" class="org.eclipse.jetty.security.ConstraintSecurityHandler">
   <property name="authenticator">
      <bean class="org.eclipse.jetty.security.authentication.BasicAuthenticator"/>
   </property>
   <property name="constraintMappings">
      <list>
         <ref bean="constraintMapping"/>
      </list>
   </property>
</bean>
```

You can then define the endpoint as:

```
from("jetty:http://0.0.0.0:9080/myservice?handlers=securityHandler")
```

If you need more handlers, set the **handlers** option equal to a comma-separated list of bean IDs.

165.12. HOW TO RETURN A CUSTOM HTTP 500 REPLY MESSAGE

You may want to return a custom reply message when something goes wrong, instead of the default reply message Camel Jetty replies with.
You could use a custom **HttpBinding** to be in control of the message mapping, but often it may be easier to use Camel's Exception Clause to construct the custom reply message. For example as show here, where we return **Dude something went wrong** with HTTP error code 500:

165.13. MULTI-PART FORM SUPPORT

From Camel 2.3.0, camel-jetty support to multipart form post out of box. The submitted form-data are mapped into the message header. Camel-jetty creates an attachment for each uploaded file. The file name is mapped to the name of the attachment. The content type is set as the content type of the attachment file name. You can find the example here.

Note: getName() functions as shown below in versions 2.5 and higher. In earlier versions you receive the temporary file name for the attachment instead

165.14. JETTY JMX SUPPORT

From Camel 2.3.0, camel-jetty supports the enabling of Jetty's JMX capabilities at the component and endpoint level with the endpoint configuration taking priority. Note that JMX must be enabled within the Camel context in order to enable JMX support in this component as the component provides Jetty with

a reference to the MBeanServer registered with the Camel context. Because the camel-jetty component caches and reuses Jetty resources for a given protocol/host/port pairing, this configuration option will only be evaluated during the creation of the first endpoint to use a protocol/host/port pairing. For example, given two routes created from the following XML fragments, JMX support would remain enabled for all endpoints listening on "https://0.0.0.0".

```
<from uri="jetty:https://0.0.0.0/myapp/myservice1/?enableJmx=true"/>
```

```
<from uri="jetty:https://0.0.0.0/myapp/myservice2/?enableJmx=false"/>
```

The camel-jetty component also provides for direct configuration of the Jetty MBeanContainer. Jetty creates MBean names dynamically. If you are running another instance of Jetty outside of the Camel context and sharing the same MBeanServer between the instances, you can provide both instances with a reference to the same MBeanContainer in order to avoid name collisions when registering Jetty MBeans.

165.15. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- HTTP

CHAPTER 166. JGROUPS COMPONENT

Available as of Camel version 2.13

JGroups is a toolkit for reliable multicast communication. The **jgroups:** component provides exchange of messages between Camel infrastructure and JGroups clusters.

Maven users will need to add the following dependency to their **pom.xml** for this component.

```
<dependency>
  <groupId>org.apache-extras.camel-extra</groupId>
  <artifactId>camel-jgroups</artifactId>
  <!-- use the same version as your Camel core version -->
  <version>x.y.z</version>
</dependency>
```

Starting from the Camel **2.13.0**, JGroups component has been moved from Camel Extra under the umbrella of the Apache Camel. If you are using Camel **2.13.0** or higher, please use the following POM entry instead.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jgroups</artifactId>
  <!-- use the same version as your Camel core version -->
  <version>x.y.z</version>
</dependency>
```

166.1. URI FORMAT

```
jgroups:clusterName[?options]
```

Where **clusterName** represents the name of the JGroups cluster the component should connect to.

166.2. OPTIONS

The JGroups component supports 4 options which are listed below.

Name	Description	Default	Type
channel (common)	Channel to use		JChannel
channelProperties (common)	Specifies configuration properties of the JChannel used by the endpoint.		String
enableViewMessages (consumer)	If set to true, the consumer endpoint will receive org.jgroups.View messages as well (not only org.jgroups.Message instances). By default only regular messages are consumed by the endpoint.	false	boolean

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The JGroups endpoint is configured using URI syntax:

```
jgroups:clusterName
```

with the following path and query parameters:

166.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
clusterName	**Required** The name of the JGroups cluster the component should connect to.		String

166.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
channelProperties (common)	Specifies configuration properties of the JChannel used by the endpoint.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
enableViewMessages (consumer)	If set to true, the consumer endpoint will receive org.jgroups.View messages as well (not only org.jgroups.Message instances). By default only regular messages are consumed by the endpoint.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

166.3. HEADERS

Header	Constant	Since version	Description
JGROUPS_ORIGINAL_MESSAGE	**JGroupsEndpoint.HEADER_JGROUPS_ORIGINAL_MESSAGE**	2.13.0	The original **org.jgroups.Message** instance from which the body of the consumed message has been extracted.
JGROUPS_SRC	`JGroupsEndpoint.`HEADER_JGROUPS_SRC	2.10.0	**Consumer** : The **org.jgroups.Address** instance extracted by **org.jgroups.Message**.getSrc() method of the consumed message. **Producer**: The custom source **org.jgroups.Address** of the message to be sent.
JGROUPS_DEST	`JGroupsEndpoint.`HEADER_JGROUPS_DEST	2.10.0	**Consumer**: The **org.jgroups.Address** instance extracted by **org.jgroups.Message**.getDest() method of the consumed message. **Producer**: The custom destination **org.jgroups.Address** of the message to be sent.

Header	Constant	Since version	Description
JGROUPS_CHANNEL_ADDRESS	`JGroupsEndpoint.`HEADER_JGROUPS_CHANNEL_ADDRESS	2.13.0	Address (**org.jgroups.Address**) of the channel associated with the endpoint.

Usage

Using **jgroups** component on the consumer side of the route will capture messages received by the **JChannel** associated with the endpoint and forward them to the Camel route. JGroups consumer processes incoming messages asynchronously.

```
// Capture messages from cluster named
// 'clusterName' and send them to Camel route.
from("jgroups:clusterName").to("seda:queue");
```

Using **jgroups** component on the producer side of the route will forward body of the Camel exchanges to the **JChannel** instance managed by the endpoint.

```
// Send message to the cluster named 'clusterName'
from("direct:start").to("jgroups:clusterName");
```

166.4. PREDEFINED FILTERS

Starting from version **2.13.0** of Camel, JGroups component comes with predefined filters factory class named **JGroupsFilters.**

If you would like to consume only view changes notifications sent to coordinator of the cluster (and ignore these sent to the "slave" nodes), use the **JGroupsFilters.dropNonCoordinatorViews()** filter. This filter is particularly useful when you want a single Camel node to become the master in the cluster, because messages passing this filter notifies you when given node has become a coordinator of the cluster. The snippet below demonstrates how to collect only messages received by the master node.

```
import static org.apache.camel.component.jgroups.JGroupsFilters.dropNonCoordinatorViews;
...
from("jgroups:clusterName?enableViewMessages=true").
  filter(dropNonCoordinatorViews()).
  to("seda:masterNodeEventsQueue");
```

166.5. PREDEFINED EXPRESSIONS

Starting from version **2.13.0** of Camel, JGroups component comes with predefined expressions factory class named **JGroupsExpressions.**

If you would like to create delayer that would affect the route only if the Camel context has not been started yet, use the **JGroupsExpressions.delayIfContextNotStarted(long delay)** factory method. The expression created by this factory method will return given delay value only if the Camel context is in the state different than **started**. This expression is particularly useful if you would like to use JGroups component for keeping singleton (master) route within the cluster. Control Bus **start** command won't initialize the singleton route if the Camel Context hasn't been yet started. So you need to delay a startup of the master route, to be sure that it has been initialized after the Camel Context startup. Because such scenario can happen only during the initialization of the cluster, we don't want to delay startup of the slave node becoming the new master - that's why we need a conditional delay expression.

The snippet below demonstrates how to use conditional delaying with the JGroups component to delay the initial startup of master node in the cluster.

```
import static java.util.concurrent.TimeUnit.SECONDS;
import static org.apache.camel.component.jgroups.JGroupsExpressions.delayIfContextNotStarted;
import static org.apache.camel.component.jgroups.JGroupsFilters.dropNonCoordinatorViews;
...
from("jgroups:clusterName?enableViewMessages=true").
  filter(dropNonCoordinatorViews()).
  threads().delay(delayIfContextNotStarted(SECONDS.toMillis(5))). // run in separated and delayed
thread. Delay only if the context hasn't been started already.
  to("controlbus:route?routeId=masterRoute&action=start&async=true");

from("timer://master?repeatCount=1").routeId("masterRoute").autoStartup(false).to(masterMockUri);
```

166.6. EXAMPLES

166.6.1. Sending (receiving) messages to (from) the JGroups cluster

In order to send message to the JGroups cluster use producer endpoint, just as demonstrated on the snippet below.

```
from("direct:start").to("jgroups:myCluster");
...
producerTemplate.sendBody("direct:start", "msg")
```

To receive the message from the snippet above (on the same or the other physical machine) listen on the messages coming from the given cluster, just as demonstrated on the code fragment below.

```
mockEndpoint.setExpectedMessageCount(1);
mockEndpoint.message(0).body().isEqualTo("msg");
...
from("jgroups:myCluster").to("mock:messagesFromTheCluster");
...
mockEndpoint.assertIsSatisfied();
```

166.6.2. Receive cluster view change notifications

The snippet below demonstrates how to create the consumer endpoint listening to the notifications regarding cluster membership changes. By default only regular messages are consumed by the endpoint.

```
mockEndpoint.setExpectedMessageCount(1);
```

```
mockEndpoint.message(0).body().isInstanceOf(org.jgroups.View.class);
...
from("jgroups:clusterName?enableViewMessages=true").to(mockEndpoint);
...
mockEndpoint.assertIsSatisfied();
```

166.6.3. Keeping singleton route within the cluster

The snippet below demonstrates how to keep the singleton consumer route in the cluster of Camel Contexts. As soon as the master node dies, one of the slaves will be elected as a new master and started. In this particular example we want to keep singleton jetty instance listening for the requests on address` http://localhost:8080/orders`.

```
import static java.util.concurrent.TimeUnit.SECONDS;
import static org.apache.camel.component.jgroups.JGroupsExpressions.delayIfContextNotStarted;
import static org.apache.camel.component.jgroups.JGroupsFilters.dropNonCoordinatorViews;
...
from("jgroups:clusterName?enableViewMessages=true").
  filter(dropNonCoordinatorViews()).
  threads().delay(delayIfContextNotStarted(SECONDS.toMillis(5))). // run in separated and delayed
thread. Delay only if the context hasn't been started already.
  to("controlbus:route?routeId=masterRoute&action=start&async=true");

from("jetty:http://localhost:8080/orders").routeId("masterRoute").autoStartup(false).to("jms:orders");
```

CHAPTER 167. JIBX DATAFORMAT

Available as of Camel version 2.6

JiBX is a Data Format which uses the JiBX library to marshal and unmarshal Java objects to and from XML.

```
// lets turn Object messages into XML then send to MQSeries
from("activemq:My.Queue").
  marshal().jibx().
  to("mqseries:Another.Queue");
```

Please note that marshaling process can recognize the message type at the runtime. However while unmarshaling message from XML we need to specify target class explicitly.

```
// lets turn XML into PurchaseOrder message
from("mqseries:Another.Queue").
  unmarshal().jibx(PurchaseOrder.class).
  to("activemq:My.Queue");
```

167.1. OPTIONS

The JiBX dataformat supports 3 options which are listed below.

Name	Default	Java Type	Description
unmarshallClass		**String**	Class name to use when unmarshalling from XML to Java.
bindingName		**String**	To use a custom binding factory
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

167.2. JIBX SPRING DSL

JiBX data format is also supported by Camel Spring DSL.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

  <!-- Define data formats -->
  <dataFormats>
    <jibx id="jibx" unmarshallClass="org.apache.camel.dataformat.jibx.PurchaseOrder"/>
  </dataFormats>

  <!-- Marshal message to XML -->
  <route>
    <from uri="direct:marshal"/>
```

```
    <marshal ref="jibx"/>
    <to uri="mock:result"/>
  </route>

  <!-- Unmarshal message from XML -->
  <route>
    <from uri="direct:unmarshal"/>
    <unmarshal ref="jibx"/>
    <to uri="mock:result"/>
  </route>

</camelContext>
```

167.3. DEPENDENCIES

To use JiBX in your camel routes you need to add the a dependency on **camel-jibx** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jibx</artifactId>
  <version>2.6.0</version>
</dependency>
```

CHAPTER 168. JING COMPONENT

Available as of Camel version 1.1

The Jing component uses the Jing Library to perform XML validation of the message body using either

- RelaxNG XML Syntax
- RelaxNG Compact Syntax

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jing</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Note that the MSV component can also support RelaxNG XML syntax.

168.1. URI FORMAT CAMEL 2.16

```
jing:someLocalOrRemoteResource
```

From Camel 2.16 the component use jing as name, and you can use the option compactSyntax to turn on either RNG or RNC mode.

168.2. OPTIONS

The Jing component has no options.

The Jing endpoint is configured using URI syntax:

```
jing:resourceUri
```

with the following path and query parameters:

168.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** URL to a local resource on the classpath or a full URL to a remote resource or resource on the file system which contains the schema to validate against.		String

168.2.2. Query Parameters (2 parameters):

Name	Description	Default	Type
compactSyntax (producer)	Whether to validate using RelaxNG compact syntax or not. By default this is false for using RelaxNG XML Syntax (rng) And true is for using RelaxNG Compact Syntax (rnc)	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

168.3. EXAMPLE

The following example shows how to configure a route from the endpoint **direct:start** which then goes to one of two endpoints, either **mock:valid** or **mock:invalid** based on whether or not the XML matches the given RelaxNG Compact Syntax schema (which is supplied on the classpath).

168.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 169. JIRA COMPONENT

Available as of Camel version 2.15

The JIRA component interacts with the JIRA API by encapsulating Atlassian's REST Java Client for JIRA. It currently provides polling for new issues and new comments. It is also able to create new issues.

Rather than webhooks, this endpoint relies on simple polling. Reasons include:

- Concern for reliability/stability
- The types of payloads we're polling aren't typically large (plus, paging is available in the API)
- The need to support apps running somewhere not publicly accessible where a webhook would fail

Note that the JIRA API is fairly expansive. Therefore, this component could be easily expanded to provide additional interactions.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jira</artifactId>
  <version>${camel-version}</version>
</dependency>
```

169.1. URI FORMAT

```
jira://endpoint[?options]
```

169.2. JIRA OPTIONS

The JIRA component has no options.

The JIRA endpoint is configured using URI syntax:

```
jira:type
```

with the following path and query parameters:

169.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
type	**Required** Operation to perform such as create a new issue or a new comment		JIRAType

169.2.2. Query Parameters (9 parameters):

Name	Description	Default	Type
password (common)	Password for login		String
serverUrl (common)	**Required** URL to the JIRA server		String
username (common)	Username for login		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delay (consumer)	Delay in seconds when querying JIRA using the consumer.	6000	int
jql (consumer)	JQL is the query language from JIRA which allows you to retrieve the data you want. For example jql=project=MyProject Where MyProject is the product key in Jira.		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

169.3. JQL:

The JQL URI option is used by both consumer endpoints. Theoretically, items like "project key", etc. could be URI options themselves. However, by requiring the use of JQL, the consumers become much more flexible and powerful.

At the bare minimum, the consumers will require the following:

```
jira://[endpoint]?[required options]&jql=project=[project key]
```

One important thing to note is that the newIssue consumer will automatically append "ORDER BY key desc" to your JQL. This is in order to optimize startup processing, rather than having to index every single issue in the project.

Another note is that, similarly, the newComment consumer will have to index every single issue **and** comment in the project. Therefore, for large projects, it's **vital** to optimize the JQL expression as much as possible. For example, the JIRA Toolkit Plugin includes a "Number of comments" custom field – use '"Number of comments" > 0' in your query. Also try to minimize based on state (status=Open), increase the polling delay, etc. Example:

```
jira://[endpoint]?[required options]&jql=RAW(project=[project key] AND status in (Open, \"Coding In Progress\") AND \"Number of comments\">0)"
```

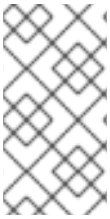
CHAPTER 170. JMS COMPONENT

170.1. JMS COMPONENT

TIP

Using ActiveMQ

If you are using Apache ActiveMQ, you should prefer the ActiveMQ component as it has been optimized for ActiveMQ. All of the options and samples on this page are also valid for the ActiveMQ component.



NOTE

Transacted and caching

See section *Transactions and Cache Levels* below if you are using transactions with JMS as it can impact performance.

NOTE

Request/Reply over JMS

Make sure to read the section *Request-reply over JMS* further below on this page for important notes about request/reply, as Camel offers a number of options to configure for performance, and clustered environments.

This component allows messages to be sent to (or consumed from) a JMS Queue or Topic. It uses Spring's JMS support for declarative transactions, including Spring's **JmsTemplate** for sending and a **MessageListenerContainer** for consuming.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

170.2. URI FORMAT

```
jms:[queue:|topic:]destinationName[?options]
```

Where **destinationName** is a JMS queue or topic name. By default, the **destinationName** is interpreted as a queue name. For example, to connect to the queue, **FOO.BAR** use:

```
jms:FOO.BAR
```

You can include the optional **queue:** prefix, if you prefer:

```
jms:queue:FOO.BAR
```

To connect to a topic, you *must* include the **topic:** prefix. For example, to connect to the topic, **Stocks.Prices**, use:

```
jms:topic:Stocks.Prices
```

You append query options to the URI using the following format, **?option=value&option=value&...**

170.3. NOTES

170.3.1. Using ActiveMQ

The JMS component reuses Spring 2's **JmsTemplate** for sending messages. This is not ideal for use in a non-J2EE container and typically requires some caching in the JMS provider to avoid poor performance.

If you intend to use Apache ActiveMQ as your Message Broker - which is a good choice as ActiveMQ rocks, then we recommend that you either:

- Use the ActiveMQ component, which is already optimized to use ActiveMQ efficiently
- Use the **PoolingConnectionFactory** in ActiveMQ.

170.3.2. Transactions and Cache Levels

If you are consuming messages and using transactions (**transacted=true**) then the default settings for cache level can impact performance.

If you are using XA transactions then you cannot cache as it can cause the XA transaction to not work properly.

If you are **not** using XA, then you should consider caching as it speeds up performance, such as setting **cacheLevelName=CACHE_CONSUMER**.

Through Camel 2.7.x, the default setting for **cacheLevelName** is **CACHE_CONSUMER**. You will need to explicitly set **cacheLevelName=CACHE_NONE**.

In Camel 2.8 onwards, the default setting for **cacheLevelName** is **CACHE_AUTO**. This default auto detects the mode and sets the cache level accordingly to:

- **CACHE_CONSUMER** if **transacted=false**
- **CACHE_NONE** if **transacted=true**

So you can say the default setting is conservative. Consider using **cacheLevelName=CACHE_CONSUMER** if you are using non-XA transactions.

170.3.3. Durable Subscriptions

If you wish to use durable topic subscriptions, you need to specify both **clientId** and **durableSubscriptionName**. The value of the **clientId** must be unique and can only be used by a single JMS connection instance in your entire network. You may prefer to use Virtual Topics instead to avoid this limitation. More background on durable messaging here.

170.3.4. Message Header Mapping

When using message headers, the JMS specification states that header names must be valid Java identifiers. So try to name your headers to be valid Java identifiers. One benefit of doing this is that you can then use your headers inside a JMS Selector (whose SQL92 syntax mandates Java identifier syntax for headers).

A simple strategy for mapping header names is used by default. The strategy is to replace any dots and hyphens in the header name as shown below and to reverse the replacement when the header name is restored from a JMS message sent over the wire. What does this mean? No more losing method names to invoke on a bean component, no more losing the filename header for the File Component, and so on.

The current header name strategy for accepting header names in Camel is as follows:

- Dots are replaced by ***DOT*** and the replacement is reversed when Camel consume the message
- Hyphen is replaced by ***HYPHEN*** and the replacement is reversed when Camel consumes the message

170.4. OPTIONS

You can configure many different properties on the JMS endpoint which map to properties on the JMSConfiguration POJO.

WARNING

Mapping to Spring JMS

Many of these properties map to properties on Spring JMS, which Camel uses for sending and receiving messages. So you can get more information about these properties by consulting the relevant Spring documentation.

170.4.1. Component options

The JMS component supports 80 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use a shared JMS configuration		JmsConfiguration

Name	Description	Default	Type
acceptMessagesWhileStopping (consumer)	Specifies whether the consumer accept messages while it is stopping. You may consider enabling this option, if you start and stop JMS routes at runtime, while there are still messages enqueued on the queue. If this option is false, and you stop the JMS route, then messages may be rejected, and the JMS broker would have to attempt redeliveries, which yet again may be rejected, and eventually the message may be moved at a dead letter queue on the JMS broker. To avoid this its recommended to enable this option.	false	boolean
allowReplyManagerQuickStop (consumer)	Whether the DefaultMessageListenerContainer used in the reply managers for request-reply messaging allow the DefaultMessageListenerContainer.runningAllowed flag to quick stop in case JmsConfigurationisAcceptMessagesWhileStopping is enabled, and org.apache.camel.CamelContext is currently being stopped. This quick stop ability is enabled by default in the regular JMS consumers but to enable for reply managers you must enable this flag.	false	boolean
acknowledgementMode (consumer)	The JMS acknowledgement mode defined as an Integer. Allows you to set vendor-specific extensions to the acknowledgment mode. For the regular modes, it is preferable to use the acknowledgementModeName instead.		int
eagerLoadingOfProperties (consumer)	Enables eager loading of JMS properties as soon as a message is loaded which generally is inefficient as the JMS properties may not be required but sometimes can catch early any issues with the underlying JMS provider and the use of JMS properties	false	boolean
acknowledgementModeName (consumer)	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE	AUTO_ACKNOWLEDGE	String
autoStartup (consumer)	Specifies whether the consumer container should auto-startup.	true	boolean
cacheLevel (consumer)	Sets the cache level by ID for the underlying JMS resources. See cacheLevelName option for more details.		int

Name	Description	Default	Type
cacheLevelName (consumer)	Sets the cache level by name for the underlying JMS resources. Possible values are: CACHE_AUTO, CACHE_CONNECTION, CACHE_CONSUMER, CACHE_NONE, and CACHE_SESSION. The default setting is CACHE_AUTO. See the Spring documentation and Transactions Cache Levels for more information.	CACHE_AUTO	String
replyToCacheLevelName (producer)	Sets the cache level by name for the reply consumer when doing request/reply over JMS. This option only applies when using fixed reply queues (not temporary). Camel will by default use: CACHE_CONSUMER for exclusive or shared w/ replyToSelectorName. And CACHE_SESSION for shared without replyToSelectorName. Some JMS brokers such as IBM WebSphere may require to set the replyToCacheLevelName=CACHE_NONE to work. Note: If using temporary queues then CACHE_NONE is not allowed, and you must use a higher value such as CACHE_CONSUMER or CACHE_SESSION.		String
clientId (common)	Sets the JMS client ID to use. Note that this value, if specified, must be unique and can only be used by a single JMS connection instance. It is typically only required for durable topic subscriptions. If using Apache ActiveMQ you may prefer to use Virtual Topics instead.		String
concurrentConsumers (consumer)	Specifies the default number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.	1	int
replyToConcurrent Consumers (producer)	Specifies the default number of concurrent consumers when doing request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.	1	int
connectionFactory (common)	The connection factory to be use. A connection factory must be configured either on the component or endpoint.		ConnectionFactory

Name	Description	Default	Type
username (security)	Username to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String
password (security)	Password to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String
deliveryPersistent (producer)	Specifies whether persistent delivery is used by default.	true	boolean
deliveryMode (producer)	Specifies the delivery mode to be used. Possible values are Possibles values are those defined by javax.jms.DeliveryMode. NON_PERSISTENT = 1 and PERSISTENT = 2.		Integer
durableSubscriptionName (common)	The durable subscriber name for specifying durable topic subscriptions. The clientId option must be configured as well.		String
exceptionListener (advanced)	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.		ExceptionListener
errorHandler (advanced)	Specifies a org.springframework.util.ErrorHandler to be invoked in case of any uncaught exceptions thrown while processing a Message. By default these exceptions will be logged at the WARN level, if no errorHandler has been configured. You can configure logging level and whether stack traces should be logged using errorHandlerLoggingLevel and errorHandlerLogStackTrace options. This makes it much easier to configure, than having to code a custom errorHandler.		ErrorHandler
errorHandlerLoggingLevel (logging)	Allows to configure the default errorHandler logging level for logging uncaught exceptions.	WARN	LoggingLevel
errorHandlerLogStackTrace (logging)	Allows to control whether stacktraces should be logged or not, by the default errorHandler.	true	boolean

Name	Description	Default	Type
explicitQosEnabled (producer)	Set if the deliveryMode, priority or timeToLive qualities of service should be used when sending messages. This option is based on Spring's JmsTemplate. The deliveryMode, priority and timeToLive options are applied to the current endpoint. This contrasts with the preserveMessageQos option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.	false	boolean
exposeListenerSession (consumer)	Specifies whether the listener session should be exposed when consuming messages.	false	boolean
idleTaskExecutionLimit (advanced)	Specifies the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the maxConcurrentConsumers setting). There is additional doc available from Spring.	1	int
idleConsumerLimit (advanced)	Specify the limit for the number of consumers that are allowed to be idle at any given time.	1	int
maxConcurrentConsumers (consumer)	Specifies the maximum number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToMaxConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.		int
replyToMaxConcurrent Consumers (producer)	Specifies the maximum number of concurrent consumers when using request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.		int
replyOnTimeoutToMax ConcurrentConsumers (producer)	Specifies the maximum number of concurrent consumers for continue routing when timeout occurred when using request/reply over JMS.	1	int
maxMessagesPerTask (advanced)	The number of messages per task. -1 is unlimited. If you use a range for concurrent consumers (eg min max), then this option can be used to set a value to eg 100 to control how fast the consumers will shrink when less work is required.	-1	int

Name	Description	Default	Type
messageConverter (advanced)	To use a custom Spring org.springframework.jms.support.converter.MessageConverter so you can be in control how to map to/from a javax.jms.Message.		MessageConverter
mapJmsMessage (advanced)	Specifies whether Camel should auto map the received JMS message to a suited payload type, such as javax.jms.TextMessage to a String etc. See section about how mapping works below for more details.	true	boolean
messageIdEnabled (advanced)	When sending, specifies whether message IDs should be added. This is just an hint to the JMS Broker. If the JMS provider accepts this hint, these messages must have the message ID set to null; if the provider ignores the hint, the message ID must be set to its normal unique value	true	boolean
messageTimestampEnabled (advanced)	Specifies whether timestamps should be enabled by default on sending messages.	true	boolean
alwaysCopyMessage (producer)	If true, Camel will always make a JMS message copy of the message when it is passed to the producer for sending. Copying the message is needed in some situations, such as when a replyToDestinationSelectorName is set (incidentally, Camel will set the alwaysCopyMessage option to true, if a replyToDestinationSelectorName is set)	false	boolean
useMessageIDAsCorrelationID (advanced)	Specifies whether JMSMessageID should always be used as JMSCorrelationID for InOut messages.	false	boolean
priority (producer)	Values greater than 1 specify the message priority when sending (where 0 is the lowest priority and 9 is the highest). The explicitQosEnabled option must also be enabled in order for this option to have any effect.	4	int
pubSubNoLocal (advanced)	Specifies whether to inhibit the delivery of messages published by its own connection.	false	boolean
receiveTimeout (advanced)	The timeout for receiving messages (in milliseconds).	1000	long
recoveryInterval (advanced)	Specifies the interval between recovery attempts, i.e. when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.	5000	long

Name	Description	Default	Type
taskExecutor (consumer)	Allows you to specify a custom task executor for consuming messages.		TaskExecutor
timeToLive (producer)	When sending messages, specifies the time-to-live of the message (in milliseconds).	-1	long
transacted (transaction)	Specifies whether to use transacted mode	false	boolean
lazyCreateTransaction Manager (transaction)	If true, Camel will create a JmsTransactionManager, if there is no transactionManager injected when option transacted=true.	true	boolean
transactionManager (transaction)	The Spring transaction manager to use.		PlatformTransaction Manager
transactionName (transaction)	The name of the transaction to use.		String
transactionTimeout (transaction)	The timeout value of the transaction (in seconds), if using transacted mode.	-1	int
testConnectionOn Startup (common)	Specifies whether to test the connection on startup. This ensures that when Camel starts that all the JMS consumers have a valid connection to the JMS broker. If a connection cannot be granted then Camel throws an exception on startup. This ensures that Camel is not started with failed connections. The JMS producers is tested as well.	false	boolean
asyncStartListener (advanced)	Whether to startup the JmsConsumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean
asyncStopListener (advanced)	Whether to stop the JmsConsumer message listener asynchronously, when stopping a route.	false	boolean

Name	Description	Default	Type
forceSendOriginal Message (producer)	When using mapJmsMessage=false Camel will create a new JMS message to send to a new JMS destination if you touch the headers (get or set) during the route. Set this option to true to force Camel to send the original JMS message that was received.	false	boolean
requestTimeout (producer)	The timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds). The default is 20 seconds. You can include the header CamelJmsRequestTimeout to override this endpoint configured timeout value, and thus have per message individual timeout values. See also the requestTimeoutCheckerInterval option.	20000	long
requestTimeoutChecker Interval (advanced)	Configures how often Camel should check for timed out Exchanges when doing request/reply over JMS. By default Camel checks once per second. But if you must react faster when a timeout occurs, then you can lower this interval, to check more frequently. The timeout is determined by the option requestTimeout.	1000	long
transferExchange (advanced)	You can transfer the exchange over the wire instead of just the body and headers. The following fields are transferred: In body, Out body, Fault body, In headers, Out headers, Fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level. You must enable this option on both the producer and consumer side, so Camel knows the payloads is an Exchange and not a regular payload.	false	boolean
transferException (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed on the consumer side, then the caused Exception will be send back in response as a javax.jms.ObjectMessage. If the client is Camel, the returned Exception is rethrown. This allows you to use Camel JMS as a bridge in your routing - for example, using persistent queues to enable robust routing. Notice that if you also have transferExchange enabled, this option takes precedence. The caught exception is required to be serializable. The original Exception on the consumer side can be wrapped in an outer exception such as org.apache.camel.RuntimeCamelException when returned to the producer.	false	boolean

Name	Description	Default	Type
transferFault (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed with a SOAP fault (not exception) on the consumer side, then the fault flag on link org.apache.camel.MessageisFault() will be send back in the response as a JMS header with the key link JmsConstantsJMS_TRANSFER_FAULT. If the client is Camel, the returned fault flag will be set on the link org.apache.camel.MessagesetFault(boolean). You may want to enable this when using Camel components that support faults such as SOAP based such as cxf or spring-ws.	false	boolean
jmsOperations (advanced)	Allows you to use your own implementation of the org.springframework.jms.core.JmsOperations interface. Camel uses JmsTemplate as default. Can be used for testing purpose, but not used much as stated in the spring API docs.		JmsOperations
destinationResolver (advanced)	A pluggable org.springframework.jms.support.destination.DestinationResolver that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).		DestinationResolver
replyToType (producer)	Allows for explicitly specifying which kind of strategy to use for replyTo queues when doing request/reply over JMS. Possible values are: Temporary, Shared, or Exclusive. By default Camel will use temporary queues. However if replyTo has been configured, then Shared is used by default. This option allows you to use exclusive queues instead of shared ones. See Camel JMS documentation for more details, and especially the notes about the implications if running in a clustered environment, and the fact that Shared reply queues has lower performance than its alternatives Temporary and Exclusive.		ReplyToType
preserveMessageQos (producer)	Set to true, if you want to send message using the QoS settings specified on the message, instead of the QoS settings on the JMS endpoint. The following three headers are considered JMSPriority, JMSDeliveryMode, and JMSExpiration. You can provide all or only some of them. If not provided, Camel will fall back to use the values from the endpoint instead. So, when using this option, the headers override the values from the endpoint. The explicitQosEnabled option, by contrast, will only use options set on the endpoint, and not values from the message header.	false	boolean

Name	Description	Default	Type
asyncConsumer (consumer)	Whether the JmsConsumer processes the Exchange asynchronously. If enabled then the JmsConsumer may pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). This means that messages may be processed not 100% strictly in order. If disabled (as default) then the Exchange is fully processed before the JmsConsumer will pickup the next message from the JMS queue. Note if transacted has been enabled, then asyncConsumer=true does not run asynchronously, as transaction must be executed synchronously (Camel 3.0 may support async transactions).	false	boolean
allowNullBody (producer)	Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.	true	boolean
includeSentJMSMessageID (producer)	Only applicable when sending to JMS destination using InOnly (eg fire and forget). Enabling this option will enrich the Camel Exchange with the actual JMSMessageID that was used by the JMS client when the message was sent to the JMS destination.	false	boolean
includeAllJMSXProperties (advanced)	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.	false	boolean
defaultTaskExecutorType (consumer)	Specifies what default TaskExecutor type to use in the DefaultMessageListenerContainer, for both consumer endpoints and the ReplyTo consumer of producer endpoints. Possible values: SimpleAsync (uses Spring's SimpleAsyncTaskExecutor) or ThreadPool (uses Spring's ThreadPoolTaskExecutor with optimal values - cached threadpool-like). If not set, it defaults to the previous behaviour, which uses a cached thread pool for consumer endpoints and SimpleAsync for reply consumers. The use of ThreadPool is recommended to reduce thread trash in elastic configurations with dynamically increasing and decreasing concurrent consumers.		DefaultTaskExecutor Type

Name	Description	Default	Type
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		JmsKeyFormatStrategy
allowAdditionalHeaders (producer)	This option is used to allow additional headers which may have values that are invalid according to JMS specification. For example some message systems such as WMQ do this with header names using prefix JMS_IBM_MQMD_ containing values with byte array or other invalid types. You can specify multiple header names separated by comma, and use as suffix for wildcard matching.		String
queueBrowseStrategy (advanced)	To use a custom QueueBrowseStrategy when browsing queues		QueueBrowseStrategy
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
waitForProvisionCorrelationToBeUpdated Counter (advanced)	Number of times to wait for provisional correlation id to be updated to the actual correlation id when doing request/reply over JMS and when the option useMessageIDAsCorrelationID is enabled.	50	int
waitForProvisionCorrelationToBeUpdated ThreadSleepingTime (advanced)	Interval in millis to sleep each time while waiting for provisional correlation id to be updated.	100	long
correlationProperty (producer)	Use this JMS property to correlate messages in InOut exchange pattern (request-reply) instead of JMSCorrelationID property. This allows you to exchange messages with systems that do not correlate messages using JMSCorrelationID JMS property. If used JMSCorrelationID will not be used or set by Camel. The value of here named property will be generated if not supplied in the header of the message under the same name.		String

Name	Description	Default	Type
subscriptionDurable (consumer)	Set whether to make the subscription durable. The durable subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a durable subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well.	false	boolean
subscriptionShared (consumer)	Set whether to make the subscription shared. The shared subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a shared subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Note that shared subscriptions may also be durable, so this flag can (and often will) be combined with subscriptionDurable as well. Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well. Requires a JMS 2.0 compatible message broker.	false	boolean
subscriptionName (consumer)	Set the name of a subscription to create. To be applied in case of a topic (pub-sub domain) with a shared or durable subscription. The subscription name needs to be unique within this client's JMS client id. Default is the class name of the specified message listener. Note: Only 1 concurrent consumer (which is the default of this message listener container) is allowed for each subscription, except for a shared subscription (which requires JMS 2.0).		String
streamMessageType Enabled (producer)	Sets whether StreamMessage type is enabled or not. Message payloads of streaming kind such as files, InputStream, etc will either by sent as BytesMessage or StreamMessage. This option controls which kind will be used. By default BytesMessage is used which enforces the entire message payload to be read into memory. By enabling this option the message payload is read into memory in chunks and each chunk is then written to the StreamMessage until no more data.	false	boolean
formatDateHeadersTo Iso8601 (producer)	Sets whether date headers should be formatted according to the ISO 8601 standard.	false	boolean

Name	Description	Default	Type
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

170.4.2. Endpoint options

The JMS endpoint is configured using URI syntax:

```
jms:destinationType:destinationName
```

with the following path and query parameters:

170.4.3. Path Parameters (2 parameters):

Name	Description	Default	Type
destinationType	The kind of destination to use	queue	String
destinationName	**Required** Name of the queue or topic to use as destination		String

170.4.4. Query Parameters (91 parameters):

Name	Description	Default	Type
clientId (common)	Sets the JMS client ID to use. Note that this value, if specified, must be unique and can only be used by a single JMS connection instance. It is typically only required for durable topic subscriptions. If using Apache ActiveMQ you may prefer to use Virtual Topics instead.		String
connectionFactory (common)	Sets the default connection factory to be used if a connection factory is not specified for either link setTemplateConnectionFactory(ConnectionFactory) or link setListenerConnectionFactory(ConnectionFactory)		ConnectionFactory

Name	Description	Default	Type
disableReplyTo (common)	Specifies whether Camel ignores the JMSReplyTo header in messages. If true, Camel does not send a reply back to the destination specified in the JMSReplyTo header. You can use this option if you want Camel to consume from a route and you do not want Camel to automatically send back a reply message because another component in your code handles the reply message. You can also use this option if you want to use Camel as a proxy between different message brokers and you want to route message from one system to another.	false	boolean
durableSubscriptionName (common)	The durable subscriber name for specifying durable topic subscriptions. The clientId option must be configured as well.		String
jmsMessageType (common)	Allows you to force the use of a specific javax.jms.Message implementation for sending JMS messages. Possible values are: Bytes, Map, Object, Stream, Text. By default, Camel would determine which JMS message type to use from the In body type. This option allows you to specify it.		JmsMessageType
testConnectionOnStartup (common)	Specifies whether to test the connection on startup. This ensures that when Camel starts that all the JMS consumers have a valid connection to the JMS broker. If a connection cannot be granted then Camel throws an exception on startup. This ensures that Camel is not started with failed connections. The JMS producers is tested as well.	false	boolean
acknowledgementModeName (consumer)	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE	AUTO_ACKNOWLEDGE	String

Name	Description	Default	Type
asyncConsumer (consumer)	Whether the JmsConsumer processes the Exchange asynchronously. If enabled then the JmsConsumer may pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). This means that messages may be processed not 100% strictly in order. If disabled (as default) then the Exchange is fully processed before the JmsConsumer will pickup the next message from the JMS queue. Note if transacted has been enabled, then asyncConsumer=true does not run asynchronously, as transaction must be executed synchronously (Camel 3.0 may support async transactions).	false	boolean
autoStartup (consumer)	Specifies whether the consumer container should auto-startup.	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
cacheLevel (consumer)	Sets the cache level by ID for the underlying JMS resources. See cacheLevelName option for more details.		int
cacheLevelName (consumer)	Sets the cache level by name for the underlying JMS resources. Possible values are: CACHE_AUTO, CACHE_CONNECTION, CACHE_CONSUMER, CACHE_NONE, and CACHE_SESSION. The default setting is CACHE_AUTO. See the Spring documentation and Transactions Cache Levels for more information.	CACHE_AUTO	String

Name	Description	Default	Type
concurrentConsumers (consumer)	Specifies the default number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.	1	int
maxConcurrentConsumers (consumer)	Specifies the maximum number of concurrent consumers when consuming from JMS (not for request/reply over JMS). See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. When doing request/reply over JMS then the option replyToMaxConcurrentConsumers is used to control number of concurrent consumers on the reply message listener.		int
replyTo (consumer)	Provides an explicit ReplyTo destination, which overrides any incoming value of Message.getJMSReplyTo().		String
replyToDeliveryPersistent (consumer)	Specifies whether to use persistent delivery by default for replies.	true	boolean
selector (consumer)	Sets the JMS selector to use		String
subscriptionDurable (consumer)	Set whether to make the subscription durable. The durable subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a durable subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well.	false	boolean

Name	Description	Default	Type
subscriptionName (consumer)	Set the name of a subscription to create. To be applied in case of a topic (pub-sub domain) with a shared or durable subscription. The subscription name needs to be unique within this client's JMS client id. Default is the class name of the specified message listener. Note: Only 1 concurrent consumer (which is the default of this message listener container) is allowed for each subscription, except for a shared subscription (which requires JMS 2.0).		String
subscriptionShared (consumer)	Set whether to make the subscription shared. The shared subscription name to be used can be specified through the subscriptionName property. Default is false. Set this to true to register a shared subscription, typically in combination with a subscriptionName value (unless your message listener class name is good enough as subscription name). Note that shared subscriptions may also be durable, so this flag can (and often will) be combined with subscriptionDurable as well. Only makes sense when listening to a topic (pub-sub domain), therefore this method switches the pubSubDomain flag as well. Requires a JMS 2.0 compatible message broker.	false	boolean
acceptMessagesWhileStopping (consumer)	Specifies whether the consumer accept messages while it is stopping. You may consider enabling this option, if you start and stop JMS routes at runtime, while there are still messages enqueued on the queue. If this option is false, and you stop the JMS route, then messages may be rejected, and the JMS broker would have to attempt redeliveries, which yet again may be rejected, and eventually the message may be moved at a dead letter queue on the JMS broker. To avoid this its recommended to enable this option.	false	boolean
allowReplyManagerQuickStop (consumer)	Whether the DefaultMessageListenerContainer used in the reply managers for request-reply messaging allow the link DefaultMessageListenerContainerrunningAllowed() flag to quick stop in case link JmsConfigurationisAcceptMessagesWhileStopping() is enabled, and org.apache.camel.CamelContext is currently being stopped. This quick stop ability is enabled by default in the regular JMS consumers but to enable for reply managers you must enable this flag.	false	boolean

Name	Description	Default	Type
consumerType (consumer)	The consumer type to use, which can be one of: Simple, Default, or Custom. The consumer type determines which Spring JMS listener to use. Default will use org.springframework.jms.listener.DefaultMessageListenerContainer, Simple will use org.springframework.jms.listener.SimpleMessageListenerContainer. When Custom is specified, the MessageListenerContainerFactory defined by the messageListenerContainerFactory option will determine what org.springframework.jms.listener.AbstractMessageListenerContainer to use.	Default	ConsumerType
defaultTaskExecutorType (consumer)	Specifies what default TaskExecutor type to use in the DefaultMessageListenerContainer, for both consumer endpoints and the ReplyTo consumer of producer endpoints. Possible values: SimpleAsync (uses Spring's SimpleAsyncTaskExecutor) or ThreadPool (uses Spring's ThreadPoolTaskExecutor with optimal values - cached threadpool-like). If not set, it defaults to the previous behaviour, which uses a cached thread pool for consumer endpoints and SimpleAsync for reply consumers. The use of ThreadPool is recommended to reduce thread trash in elastic configurations with dynamically increasing and decreasing concurrent consumers.		DefaultTaskExecutor Type
eagerLoadingOfProperties (consumer)	Enables eager loading of JMS properties and payload as soon as a message is loaded which generally is inefficient as the JMS properties may not be required but sometimes can catch early any issues with the underlying JMS provider and the use of JMS properties	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
exposeListenerSession (consumer)	Specifies whether the listener session should be exposed when consuming messages.	false	boolean

Name	Description	Default	Type
replyToSameDestination Allowed (consumer)	Whether a JMS consumer is allowed to send a reply message to the same destination that the consumer is using to consume from. This prevents an endless loop by consuming and sending back the same message to itself.	false	boolean
taskExecutor (consumer)	Allows you to specify a custom task executor for consuming messages.		TaskExecutor
deliveryMode (producer)	Specifies the delivery mode to be used. Possibles values are those defined by javax.jms.DeliveryMode. NON_PERSISTENT = 1 and PERSISTENT = 2.		Integer
deliveryPersistent (producer)	Specifies whether persistent delivery is used by default.	true	boolean
explicitQosEnabled (producer)	Set if the deliveryMode, priority or timeToLive qualities of service should be used when sending messages. This option is based on Spring's JmsTemplate. The deliveryMode, priority and timeToLive options are applied to the current endpoint. This contrasts with the preserveMessageQos option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.	false	Boolean
formatDateHeadersToIso8601 (producer)	Sets whether date headers should be formatted according to the ISO 8601 standard.	false	boolean
preserveMessageQos (producer)	Set to true, if you want to send message using the QoS settings specified on the message, instead of the QoS settings on the JMS endpoint. The following three headers are considered JMSPriority, JMSDeliveryMode, and JMSExpiration. You can provide all or only some of them. If not provided, Camel will fall back to use the values from the endpoint instead. So, when using this option, the headers override the values from the endpoint. The explicitQosEnabled option, by contrast, will only use options set on the endpoint, and not values from the message header.	false	boolean

Name	Description	Default	Type
priority (producer)	Values greater than 1 specify the message priority when sending (where 0 is the lowest priority and 9 is the highest). The explicitQosEnabled option must also be enabled in order for this option to have any effect.	4	int
replyToConcurrentConsumers (producer)	Specifies the default number of concurrent consumers when doing request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.	1	int
replyToMaxConcurrent Consumers (producer)	Specifies the maximum number of concurrent consumers when using request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads.		int
replyToOnTimeoutMax ConcurrentConsumers (producer)	Specifies the maximum number of concurrent consumers for continue routing when timeout occurred when using request/reply over JMS.	1	int
replyToOverride (producer)	Provides an explicit ReplyTo destination in the JMS message, which overrides the setting of replyTo. It is useful if you want to forward the message to a remote Queue and receive the reply message from the ReplyTo destination.		String
replyToType (producer)	Allows for explicitly specifying which kind of strategy to use for replyTo queues when doing request/reply over JMS. Possible values are: Temporary, Shared, or Exclusive. By default Camel will use temporary queues. However if replyTo has been configured, then Shared is used by default. This option allows you to use exclusive queues instead of shared ones. See Camel JMS documentation for more details, and especially the notes about the implications if running in a clustered environment, and the fact that Shared reply queues has lower performance than its alternatives Temporary and Exclusive.		ReplyToType
requestTimeout (producer)	The timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds). The default is 20 seconds. You can include the header CamelJmsRequestTimeout to override this endpoint configured timeout value, and thus have per message individual timeout values. See also the requestTimeoutCheckerInterval option.	20000	long

Name	Description	Default	Type
timeToLive (producer)	When sending messages, specifies the time-to-live of the message (in milliseconds).	-1	long
allowAdditionalHeaders (producer)	This option is used to allow additional headers which may have values that are invalid according to JMS specification. For example some message systems such as WMQ do this with header names using prefix JMS_IBM_MQMD_ containing values with byte array or other invalid types. You can specify multiple header names separated by comma, and use as suffix for wildcard matching.		String
allowNullBody (producer)	Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.	true	boolean
alwaysCopyMessage (producer)	If true, Camel will always make a JMS message copy of the message when it is passed to the producer for sending. Copying the message is needed in some situations, such as when a replyToDestinationSelectorName is set (incidentally, Camel will set the alwaysCopyMessage option to true, if a replyToDestinationSelectorName is set)	false	boolean
correlationProperty (producer)	Use this JMS property to correlate messages in InOut exchange pattern (request-reply) instead of JMSCorrelationID property. This allows you to exchange messages with systems that do not correlate messages using JMSCorrelationID JMS property. If used JMSCorrelationID will not be used or set by Camel. The value of here named property will be generated if not supplied in the header of the message under the same name.		String
disableTimeToLive (producer)	Use this option to force disabling time to live. For example when you do request/reply over JMS, then Camel will by default use the requestTimeout value as time to live on the message being sent. The problem is that the sender and receiver systems have to have their clocks synchronized, so they are in sync. This is not always so easy to archive. So you can use disableTimeToLive=true to not set a time to live value on the sent message. Then the message will not expire on the receiver system. See below in section About time to live for more details.	false	boolean

Name	Description	Default	Type
forceSendOriginalMessage (producer)	When using mapJmsMessage=false Camel will create a new JMS message to send to a new JMS destination if you touch the headers (get or set) during the route. Set this option to true to force Camel to send the original JMS message that was received.	false	boolean
includeSentJMSMessageID (producer)	Only applicable when sending to JMS destination using InOnly (eg fire and forget). Enabling this option will enrich the Camel Exchange with the actual JMSMessageID that was used by the JMS client when the message was sent to the JMS destination.	false	boolean
replyToCacheLevelName (producer)	Sets the cache level by name for the reply consumer when doing request/reply over JMS. This option only applies when using fixed reply queues (not temporary). Camel will by default use: CACHE_CONSUMER for exclusive or shared w/ replyToSelectorName. And CACHE_SESSION for shared without replyToSelectorName. Some JMS brokers such as IBM WebSphere may require to set the replyToCacheLevelName=CACHE_NONE to work. Note: If using temporary queues then CACHE_NONE is not allowed, and you must use a higher value such as CACHE_CONSUMER or CACHE_SESSION.		String
replyToDestinationSelector Name (producer)	Sets the JMS Selector using the fixed name to be used so you can filter out your own replies from the others when using a shared queue (that is, if you are not using a temporary reply queue).		String
streamMessageTypeEnabled (producer)	Sets whether StreamMessage type is enabled or not. Message payloads of streaming kind such as files, InputStream, etc will either by sent as BytesMessage or StreamMessage. This option controls which kind will be used. By default BytesMessage is used which enforces the entire message payload to be read into memory. By enabling this option the message payload is read into memory in chunks and each chunk is then written to the StreamMessage until no more data.	false	boolean
allowSerializedHeaders (advanced)	Controls whether or not to include serialized headers. Applies only when link isTransferExchange() is true. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean

Name	Description	Default	Type
asyncStartListener (advanced)	Whether to startup the JmsConsumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean
asyncStopListener (advanced)	Whether to stop the JmsConsumer message listener asynchronously, when stopping a route.	false	boolean
destinationResolver (advanced)	A pluggable org.springframework.jms.support.destination.DestinationResolver that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).		DestinationResolver
errorHandler (advanced)	Specifies a org.springframework.util.ErrorHandler to be invoked in case of any uncaught exceptions thrown while processing a Message. By default these exceptions will be logged at the WARN level, if no errorHandler has been configured. You can configure logging level and whether stack traces should be logged using errorHandlerLoggingLevel and errorHandlerLogStackTrace options. This makes it much easier to configure, than having to code a custom errorHandler.		ErrorHandler
exceptionListener (advanced)	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.		ExceptionListener
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
idleConsumerLimit (advanced)	Specify the limit for the number of consumers that are allowed to be idle at any given time.	1	int

Name	Description	Default	Type
idleTaskExecutionLimit (advanced)	Specifies the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the maxConcurrentConsumers setting). There is additional doc available from Spring.	1	int
includeAllJMSXProperties (advanced)	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.	false	boolean
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		String
mapJmsMessage (advanced)	Specifies whether Camel should auto map the received JMS message to a suited payload type, such as javax.jms.TextMessage to a String etc.	true	boolean
maxMessagesPerTask (advanced)	The number of messages per task. -1 is unlimited. If you use a range for concurrent consumers (eg min max), then this option can be used to set a value to eg 100 to control how fast the consumers will shrink when less work is required.	-1	int
messageConverter (advanced)	To use a custom Spring org.springframework.jms.support.converter.MessageConverter so you can be in control how to map to/from a javax.jms.Message.		MessageConverter
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy

Name	Description	Default	Type
messageIdEnabled (advanced)	When sending, specifies whether message IDs should be added. This is just an hint to the JMS Broker. If the JMS provider accepts this hint, these messages must have the message ID set to null; if the provider ignores the hint, the message ID must be set to its normal unique value	true	boolean
messageListenerContainerFactory (advanced)	Registry ID of the MessageListenerContainerFactory used to determine what org.springframework.jms.listener.AbstractMessageListenerContainer to use to consume messages. Setting this will automatically set consumerType to Custom.		MessageListenerContainerFactory
messageTimestampEnabled (advanced)	Specifies whether timestamps should be enabled by default on sending messages. This is just an hint to the JMS Broker. If the JMS provider accepts this hint, these messages must have the timestamp set to zero; if the provider ignores the hint, the timestamp must be set to its normal value	true	boolean
pubSubNoLocal (advanced)	Specifies whether to inhibit the delivery of messages published by its own connection.	false	boolean
receiveTimeout (advanced)	The timeout for receiving messages (in milliseconds).	1000	long
recoveryInterval (advanced)	Specifies the interval between recovery attempts, i.e. when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.	5000	long
requestTimeoutCheckerInterval (advanced)	Configures how often Camel should check for timed out Exchanges when doing request/reply over JMS. By default Camel checks once per second. But if you must react faster when a timeout occurs, then you can lower this interval, to check more frequently. The timeout is determined by the option requestTimeout.	1000	long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
transferException (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed on the consumer side, then the caused Exception will be send back in response as a javax.jms.ObjectMessage. If the client is Camel, the returned Exception is rethrown. This allows you to use Camel JMS as a bridge in your routing - for example, using persistent queues to enable robust routing. Notice that if you also have transferExchange enabled, this option takes precedence. The caught exception is required to be serializable. The original Exception on the consumer side can be wrapped in an outer exception such as org.apache.camel.RuntimeCamelException when returned to the producer.	false	boolean
transferExchange (advanced)	You can transfer the exchange over the wire instead of just the body and headers. The following fields are transferred: In body, Out body, Fault body, In headers, Out headers, Fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level. You must enable this option on both the producer and consumer side, so Camel knows the payloads is an Exchange and not a regular payload.	false	boolean
transferFault (advanced)	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed with a SOAP fault (not exception) on the consumer side, then the fault flag on link org.apache.camel.MessageisFault() will be send back in the response as a JMS header with the key link JmsConstantsJMS_TRANSFER_FAULT. If the client is Camel, the returned fault flag will be set on the link org.apache.camel.MessagesetFault(boolean). You may want to enable this when using Camel components that support faults such as SOAP based such as cxf or spring-ws.	false	boolean
useMessageIDAsCorrelationID (advanced)	Specifies whether JMSMessageID should always be used as JMSCorrelationID for InOut messages.	false	boolean
waitForProvisionCorrelationToBeUpdatedCounter (advanced)	Number of times to wait for provisional correlation id to be updated to the actual correlation id when doing request/reply over JMS and when the option useMessageIDAsCorrelationID is enabled.	50	int

Name	Description	Default	Type
waitForProvisionCorrelationToBeUpdatedThreadSleeping Time (advanced)	Interval in millis to sleep each time while waiting for provisional correlation id to be updated.	100	long
errorHandlerLoggingLevel (logging)	Allows to configure the default errorHandler logging level for logging uncaught exceptions.	WARN	LoggingLevel
errorHandlerLogStackTrace (logging)	Allows to control whether stacktraces should be logged or not, by the default errorHandler.	true	boolean
password (security)	Password to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String
username (security)	Username to use with the ConnectionFactory. You can also configure username/password directly on the ConnectionFactory.		String
transacted (transaction)	Specifies whether to use transacted mode	false	boolean
lazyCreateTransaction Manager (transaction)	If true, Camel will create a JmsTransactionManager, if there is no transactionManager injected when option transacted=true.	true	boolean
transactionManager (transaction)	The Spring transaction manager to use.		PlatformTransaction Manager
transactionName (transaction)	The name of the transaction to use.		String
transactionTimeout (transaction)	The timeout value of the transaction (in seconds), if using transacted mode.	-1	int

170.5. MESSAGE MAPPING BETWEEN JMS AND CAMEL

Camel automatically maps messages between **javax.jms.Message** and **org.apache.camel.Message**.

When sending a JMS message, Camel converts the message body to the following JMS message types:

Body Type	JMS Message	Comment
String	**javax.jms.TextMessage**	
org.w3c.dom.Node	**javax.jms.TextMessage**	The DOM will be converted to **String**.
Map	**javax.jms.MapMessage**	
java.io.Serializable	**javax.jms.ObjectMessage**	
byte[]	**javax.jms.BytesMessage**	
java.io.File	**javax.jms.BytesMessage**	
java.io.Reader	**javax.jms.BytesMessage**	
java.io.InputStream	**javax.jms.BytesMessage**	
java.nio.ByteBuffer	**javax.jms.BytesMessage**	

When receiving a JMS message, Camel converts the JMS message to the following body type:

JMS Message	Body Type
javax.jms.TextMessage	**String**
javax.jms.BytesMessage	**byte[]**
javax.jms.MapMessage	**Map<String, Object>**
javax.jms.ObjectMessage	**Object**

170.5.1. Disabling auto-mapping of JMS messages

You can use the **mapJmsMessage** option to disable the auto-mapping above. If disabled, Camel will not try to map the received JMS message, but instead uses it directly as the payload. This allows you to avoid the overhead of mapping and let Camel just pass through the JMS message. For instance, it even allows you to route **javax.jms.ObjectMessage** JMS messages with classes you do **not** have on the classpath.

170.5.2. Using a custom MessageConverter

You can use the **messageConverter** option to do the mapping yourself in a Spring **org.springframework.jms.support.converter.MessageConverter** class.

For example, in the route below we use a custom message converter when sending a message to the JMS order queue:

```
from("file://inbox/order").to("jms:queue:order?messageConverter=#myMessageConverter");
```

You can also use a custom message converter when consuming from a JMS destination.

170.5.3. Controlling the mapping strategy selected

You can use the **jmsMessageType** option on the endpoint URL to force a specific message type for all messages.

In the route below, we poll files from a folder and send them as **javax.jms.TextMessage** as we have forced the JMS producer endpoint to use text messages:

```
from("file://inbox/order").to("jms:queue:order?jmsMessageType=Text");
```

You can also specify the message type to use for each message by setting the header with the key **CamelJmsMessageType**. For example:

```
from("file://inbox/order").setHeader("CamelJmsMessageType",
JmsMessageType.Text).to("jms:queue:order");
```

The possible values are defined in the **enum** class, **org.apache.camel.jms.JmsMessageType**.

170.6. MESSAGE FORMAT WHEN SENDING

The exchange that is sent over the JMS wire must conform to the JMS Message spec.

For the **exchange.in.header** the following rules apply for the header **keys**:

- Keys starting with **JMS** or **JMSX** are reserved.
- **exchange.in.headers** keys must be literals and all be valid Java identifiers (do not use dots in the key name).
- Camel replaces dots & hyphens and the reverse when when consuming JMS messages:
  . is replaced by ***DOT*** and the reverse replacement when Camel consumes the message.
  - is replaced by ***HYPHEN*** and the reverse replacement when Camel consumes the message.
- See also the option **jmsKeyFormatStrategy**, which allows use of your own custom strategy for formatting keys.

For the **exchange.in.header**, the following rules apply for the header **values**:

- The values must be primitives or their counter objects (such as **Integer**, **Long**, **Character**). The types, **String**, **CharSequence**, **Date**, **BigDecimal** and **BigInteger** are all converted to their **toString()** representation. All other types are dropped.

Camel will log with category **org.apache.camel.component.jms.JmsBinding** at **DEBUG** level if it drops a given header value. For example:

```
2008-07-09 06:43:04,046 [main           ] DEBUG JmsBinding
  - Ignoring non primitive header: order of class:
org.apache.camel.component.jms.issues.DummyOrder with value: DummyOrder{orderId=333,
itemId=4444, quantity=2}
```

170.7. MESSAGE FORMAT WHEN RECEIVING

Camel adds the following properties to the **Exchange** when it receives a message:

Property	Type	Description
org.apache.camel.jms.replyDestination	**javax.jms.Destination**	The reply destination.

Camel adds the following JMS properties to the In message headers when it receives a JMS message:

Header	Type	Description
JMSCorrelationID	**String**	The JMS correlation ID.

Header	Type	Description
JMSDeliveryMode	**int**	The JMS delivery mode.
JMSDestination	**javax.jms.Destination**	The JMS destination.
JMSExpiration	**long**	The JMS expiration.
JMSMessageID	**String**	The JMS unique message ID.
JMSPriority	**int**	The JMS priority (with 0 as the lowest priority and 9 as the highest).
JMSRedelivered	**boolean**	Is the JMS message redelivered.
JMSReplyTo	**javax.jms.Destination**	The JMS reply-to destination.
JMSTimestamp	**long**	The JMS timestamp.
JMSType	**String**	The JMS type.
JMSXGroupID	**String**	The JMS group ID.

As all the above information is standard JMS you can check the JMS documentation for further details.

170.8. ABOUT USING CAMEL TO SEND AND RECEIVE MESSAGES AND JMSREPLYTO

The JMS component is complex and you have to pay close attention to how it works in some cases. So this is a short summary of some of the areas/pitfalls to look for.

When Camel sends a message using its **JMSProducer**, it checks the following conditions:

- The message exchange pattern,
- Whether a **JMSReplyTo** was set in the endpoint or in the message headers,
- Whether any of the following options have been set on the JMS endpoint: **disableReplyTo**, **preserveMessageQos**, **explicitQosEnabled**.

All this can be a tad complex to understand and configure to support your use case.

170.8.1. JmsProducer

The **JmsProducer** behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
InOut	-	Camel will expect a reply, set a temporary **JMSReplyTo**, and after sending the message, it will start to listen for the reply message on the temporary queue.
InOut	**JMSReplyTo** is set	Camel will expect a reply and, after sending the message, it will start to listen for the reply message on the specified **JMSReplyTo** queue.
InOnly	-	Camel will send the message and **not** expect a reply.
InOnly	**JMSReplyTo** is set	By default, Camel discards the **JMSReplyTo** destination and clears the **JMSReplyTo** header before sending the message. Camel then sends the message and does **not** expect a reply. Camel logs this in the log at **WARN** level (changed to **DEBUG** level from **Camel 2.6** onwards. You can use **preserveMessageQuo=true** to instruct Camel to keep the **JMSReplyTo**. In all situations the **JmsProducer** does **not** expect any reply and thus continue after sending the message.

170.8.2. JmsConsumer

The **JmsConsumer** behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
InOut	-	Camel will send the reply back to the **JMSReplyTo** queue.
InOnly	-	Camel will not send a reply back, as the pattern is *InOnly*.

Exchange Pattern	Other options	Description
-	**disableReplyTo=true**	This option suppresses replies.

So pay attention to the message exchange pattern set on your exchanges.

If you send a message to a JMS destination in the middle of your route you can specify the exchange pattern to use, see more at Request Reply.
This is useful if you want to send an **InOnly** message to a JMS topic:

```
from("activemq:queue:in")
  .to("bean:validateOrder")
  .to(ExchangePattern.InOnly, "activemq:topic:order")
  .to("bean:handleOrder");
```

170.9. REUSE ENDPOINT AND SEND TO DIFFERENT DESTINATIONS COMPUTED AT RUNTIME

If you need to send messages to a lot of different JMS destinations, it makes sense to reuse a JMS endpoint and specify the real destination in a message header. This allows Camel to reuse the same endpoint, but send to different destinations. This greatly reduces the number of endpoints created and economizes on memory and thread resources.

You can specify the destination in the following headers:

Header	Type	Description
CamelJmsDestination	**javax.jms.Destination**	A destination object.
CamelJmsDestinationName	**String**	The destination name.

For example, the following route shows how you can compute a destination at run time and use it to override the destination appearing in the JMS URL:

```
from("file://inbox")
  .to("bean:computeDestination")
  .to("activemq:queue:dummy");
```

The queue name, **dummy**, is just a placeholder. It must be provided as part of the JMS endpoint URL, but it will be ignored in this example.

In the **computeDestination** bean, specify the real destination by setting the **CamelJmsDestinationName** header as follows:

```
public void setJmsHeader(Exchange exchange) {
   String id = ....
   exchange.getIn().setHeader("CamelJmsDestinationName", "order:" + id");
}
```

Then Camel will read this header and use it as the destination instead of the one configured on the endpoint. So, in this example Camel sends the message to **activemq:queue:order:2**, assuming the **id** value was 2.

If both the **CamelJmsDestination** and the **CamelJmsDestinationName** headers are set, **CamelJmsDestination** takes priority. Keep in mind that the JMS producer removes both **CamelJmsDestination** and **CamelJmsDestinationName** headers from the exchange and do not propagate them to the created JMS message in order to avoid the accidental loops in the routes (in scenarios when the message will be forwarded to the another JMS endpoint).

170.10. CONFIGURING DIFFERENT JMS PROVIDERS

You can configure your JMS provider in Spring XML as follows:

Basically, you can configure as many JMS component instances as you wish and give them **a unique name using the id attribute**. The preceding example configures an **activemq** component. You could do the same to configure MQSeries, TibCo, BEA, Sonic and so on.

Once you have a named JMS component, you can then refer to endpoints within that component using URIs. For example for the component name, **activemq**, you can then refer to destinations using the URI format, **activemq:[queue:|topic:]destinationName**. You can use the same approach for all other JMS providers.

This works by the SpringCamelContext lazily fetching components from the spring context for the scheme name you use for Endpoint URIs and having the Component resolve the endpoint URIs.

170.10.1. Using JNDI to find the ConnectionFactory

If you are using a J2EE container, you might need to look up JNDI to find the JMS **ConnectionFactory** rather than use the usual **<bean>** mechanism in Spring. You can do this using Spring's factory bean or the new Spring XML namespace. For example:

```
<bean id="weblogic" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory" ref="myConnectionFactory"/>
</bean>

<jee:jndi-lookup id="myConnectionFactory" jndi-name="jms/connectionFactory"/>
```

See The jee schema in the Spring reference documentation for more details about JNDI lookup.

170.11. CONCURRENT CONSUMING

A common requirement with JMS is to consume messages concurrently in multiple threads in order to make an application more responsive. You can set the **concurrentConsumers** option to specify the number of threads servicing the JMS endpoint, as follows:

```
from("jms:SomeQueue?concurrentConsumers=20").
  bean(MyClass.class);
```

You can configure this option in one of the following ways:

- On the **JmsComponent**,
- On the endpoint URI or,
- By invoking **setConcurrentConsumers()** directly on the **JmsEndpoint**.

170.11.1. Concurrent Consuming with async consumer

Notice that each concurrent consumer will only pickup the next available message from the JMS broker, when the current message has been fully processed. You can set the option **asyncConsumer=true** to let the consumer pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). See more details in the table on top of the page about the **asyncConsumer** option.

```
from("jms:SomeQueue?concurrentConsumers=20&asyncConsumer=true").
  bean(MyClass.class);
```

170.12. REQUEST-REPLY OVER JMS

Camel supports Request Reply over JMS. In essence the MEP of the Exchange should be **InOut** when you send a message to a JMS queue.

Camel offers a number of options to configure request/reply over JMS that influence performance and clustered environments. The table below summaries the options.

Option	Performance	Cluster	Description
Temporary	Fast	Yes	A temporary queue is used as reply queue, and automatic created by Camel. To use this do **not** specify a replyTo queue name. And you can optionally configure **replyToType=Temporary** to make it stand out that temporary queues are in use.
Shared	Slow	Yes	A shared persistent queue is used as reply queue. The queue must be created beforehand, although some brokers can create them on the fly such as Apache ActiveMQ. To use this you must specify the replyTo queue name. And you can optionally configure **replyToType=Shared** to make it stand out that shared queues are in use. A shared queue can be used in a clustered environment with multiple nodes running this Camel application at the same time. All using the same shared reply queue. This is possible because JMS Message selectors are used to correlate expected reply messages; this impacts performance though. JMS Message selectors is slower, and therefore not as fast as **Temporary** or **Exclusive** queues. See further below how to tweak this for better performance.

Option	Performance	Cluster	Description
Exclusive	Fast	No (*Yes)	An exclusive persistent queue is used as reply queue. The queue must be created beforehand, although some brokers can create them on the fly such as Apache ActiveMQ. To use this you must specify the replyTo queue name. And you **must** configure **replyToType=Exclusive** to instruct Camel to use exclusive queues, as **Shared** is used by default, if a**replyTo** queue name was configured. When using exclusive reply queues, then JMS Message selectors are **not** in use, and therefore other applications must not use this queue as well. An exclusive queue **cannot** be used in a clustered environment with multiple nodes running this Camel application at the same time; as we do not have control if the reply queue comes back to the same node that sent the request message; that is why shared queues use JMS Message selectors to make sure of this. **Though** if you configure each Exclusive reply queue with an unique name per node, then you can run this in a clustered environment. As then the reply message will be sent back to that queue for the given node, that awaits the reply message.
concurrentConsumers	Fast	Yes	**Camel 2.10.3:** Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the **concurrentConsumers** and **maxConcurrentConsumers** options. **Notice:** That using **Shared** reply queues may not work as well with concurrent listeners, so use this option with care.
maxConcurrentConsumers	Fast	Yes	**Camel 2.10.3:** Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the **concurrentConsumers** and **maxConcurrentConsumers** options. **Notice:** That using **Shared** reply queues may not work as well with concurrent listeners, so use this option with care.

The **JmsProducer** detects the **InOut** and provides a **JMSReplyTo** header with the reply destination to be used. By default Camel uses a temporary queue, but you can use the **replyTo** option on the endpoint to specify a fixed reply queue (see more below about fixed reply queue).

Camel will automatic setup a consumer which listen on the reply queue, so you should **not** do anything. This consumer is a Spring **DefaultMessageListenerContainer** which listen for replies. However it's fixed to 1 concurrent consumer.
That means replies will be processed in sequence as there are only 1 thread to process the replies. If you want to process replies faster, then we need to use concurrency. But **not** using the **concurrentConsumer** option. We should use the **threads** from the Camel DSL instead, as shown in the route below:

Instead of using threads, then use concurrentConsumers option if using Camel 2.10.3 or better. See further below.

```
from(xxx)
.inOut().to("activemq:queue:foo")
.threads(5)
.to(yyy)
.to(zzz);
```

In this route we instruct Camel to route replies asynchronously using a thread pool with 5 threads.

From **Camel 2.10.3** onwards you can now configure the listener to use concurrent threads using the **concurrentConsumers** and **maxConcurrentConsumers** options. This allows you to easier configure this in Camel as shown below:

```
from(xxx)
.inOut().to("activemq:queue:foo?concurrentConsumers=5")
.to(yyy)
.to(zzz);
```

170.12.1. Request-reply over JMS and using a shared fixed reply queue

If you use a fixed reply queue when doing Request Reply over JMS as shown in the example below, then pay attention.

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar")
.to(yyy)
```

In this example the fixed reply queue named "bar" is used. By default Camel assumes the queue is shared when using fixed reply queues, and therefore it uses a **JMSSelector** to only pickup the expected reply messages (eg based on the **JMSCorrelationID**). See next section for exclusive fixed reply queues. That means its not as fast as temporary queues. You can speedup how often Camel will pull for reply messages using the **receiveTimeout** option. By default its 1000 millis. So to make it faster you can set it to 250 millis to pull 4 times per second as shown:

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar&receiveTimeout=250")
.to(yyy)
```

Notice this will cause the Camel to send pull requests to the message broker more frequent, and thus require more network traffic.
It is generally recommended to use temporary queues if possible.

170.12.2. Request-reply over JMS and using an exclusive fixed reply queue

Available as of Camel 2.9

In the previous example, Camel would anticipate the fixed reply queue named "bar" was shared, and thus it uses a **JMSSelector** to only consume reply messages which it expects. However there is a drawback doing this as JMS selectos is slower. Also the consumer on the reply queue is slower to update with new JMS selector ids. In fact it only updates when the **receiveTimeout** option times out, which by default is 1 second. So in theory the reply messages could take up till about 1 sec to be detected. On the other hand if the fixed reply queue is exclusive to the Camel reply consumer, then we can avoid using the JMS selectors, and thus be more performant. In fact as fast as using temporary queues. So in **Camel 2.9** onwards we introduced the **ReplyToType** option which you can configure to **Exclusive** to tell Camel that the reply queue is exclusive as shown in the example below:

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar&replyToType=Exclusive")
.to(yyy)
```

Mind that the queue must be exclusive to each and every endpoint. So if you have two routes, then they each need an unique reply queue as shown in the next example:

-

```
from(xxx)
.inOut().to("activemq:queue:foo?replyTo=bar&replyToType=Exclusive")
.to(yyy)

from(aaa)
.inOut().to("activemq:queue:order?replyTo=order.reply&replyToType=Exclusive")
.to(bbb)
```

The same applies if you run in a clustered environment. Then each node in the cluster must use an unique reply queue name. As otherwise each node in the cluster may pickup messages which was intended as a reply on another node. For clustered environments its recommended to use shared reply queues instead.

170.13. SYNCHRONIZING CLOCKS BETWEEN SENDERS AND RECEIVERS

When doing messaging between systems, its desirable that the systems have synchronized clocks. For example when sending a JMS message, then you can set a time to live value on the message. Then the receiver can inspect this value, and determine if the message is already expired, and thus drop the message instead of consume and process it. However this requires that both sender and receiver have synchronized clocks. If you are using ActiveMQ then you can use the timestamp plugin to synchronize clocks.

170.14. ABOUT TIME TO LIVE

Read first above about synchronized clocks.

When you do request/reply (InOut) over JMS with Camel then Camel uses a timeout on the sender side, which is default 20 seconds from the **requestTimeout** option. You can control this by setting a higher/lower value. However the time to live value is still set on the JMS message being send. So that requires the clocks to be synchronized between the systems. If they are not, then you may want to disable the time to live value being set. This is now possible using the **disableTimeToLive** option from **Camel 2.8** onwards. So if you set this option to **disableTimeToLive=true**, then Camel does **not** set any time to live value when sending JMS messages. **But** the request timeout is still active. So for example if you do request/reply over JMS and have disabled time to live, then Camel will still use a timeout by 20 seconds (the **requestTimeout** option). That option can of course also be configured. So the two options **requestTimeout** and **disableTimeToLive** gives you fine grained control when doing request/reply.

From **Camel 2.13/2.12.3** onwards you can provide a header in the message to override and use as the request timeout value instead of the endpoint configured value. For example:

```
from("direct:someWhere")
  .to("jms:queue:foo?replyTo=bar&requestTimeout=30s")
  .to("bean:processReply");
```

In the route above we have a endpoint configured **requestTimeout** of 30 seconds. So Camel will wait up till 30 seconds for that reply message to come back on the bar queue. If no reply message is received then a **org.apache.camel.ExchangeTimedOutException** is set on the Exchange and Camel continues routing the message, which would then fail due the exception, and Camel's error handler reacts.

If you want to use a per message timeout value, you can set the header with key **org.apache.camel.component.jms.JmsConstants#JMS_REQUEST_TIMEOUT** which has constant value **"CamelJmsRequestTimeout"** with a timeout value as long type.

For example we can use a bean to compute the timeout value per individual message, such as calling the **"whatIsTheTimeout"** method on the service bean as shown below:

```
from("direct:someWhere")
  .setHeader("CamelJmsRequestTimeout", method(ServiceBean.class, "whatIsTheTimeout"))
  .to("jms:queue:foo?replyTo=bar&requestTimeout=30s")
  .to("bean:processReply");
```

When you do fire and forget (InOut) over JMS with Camel then Camel by default does **not** set any time to live value on the message. You can configure a value by using the **timeToLive** option. For example to indicate a 5 sec., you set **timeToLive=5000**. The option **disableTimeToLive** can be used to force disabling the time to live, also for InOnly messaging. The **requestTimeout** option is not being used for InOnly messaging.

170.15. ENABLING TRANSACTED CONSUMPTION

A common requirement is to consume from a queue in a transaction and then process the message using the Camel route. To do this, just ensure that you set the following properties on the component/endpoint:

- **transacted** = true
- **transactionManager** = a *Transsaction Manager* - typically the **JmsTransactionManager**

See the Transactional Client EIP pattern for further details.

Transactions and [Request Reply] over JMS

When using Request Reply over JMS you cannot use a single transaction; JMS will not send any messages until a commit is performed, so the server side won't receive anything at all until the transaction commits. Therefore to use Request Reply you must commit a transaction after sending the request and then use a separate transaction for receiving the response.

To address this issue the JMS component uses different properties to specify transaction use for oneway messaging and request reply messaging:

The **transacted** property applies **only** to the InOnly message Exchange Pattern (MEP).

The **transactedInOut** property applies to the InOut(Request Reply) message Exchange Pattern (MEP).

If you want to use transactions for Request Reply (InOut MEP), you **must** set **transactedInOut=true**.

Available as of Camel 2.10

You can leverage the DMLC transacted session API using the following properties on component/endpoint:

- **transacted** = true
- **lazyCreateTransactionManager** = false

The benefit of doing so is that the cacheLevel setting will be honored when using local transactions without a configured TransactionManager. When a TransactionManager is configured, no caching happens at DMLC level and its necessary to rely on a pooled connection factory. For more details about this kind of setup see here and here.

170.16. USING JMSREPLYTO FOR LATE REPLIES

When using Camel as a JMS listener, it sets an Exchange property with the value of the ReplyTo **javax.jms.Destination** object, having the key **ReplyTo**. You can obtain this **Destination** as follows:

```
Destination replyDestination =
exchange.getIn().getHeader(JmsConstants.JMS_REPLY_DESTINATION, Destination.class);
```

And then later use it to send a reply using regular JMS or Camel.

```
// we need to pass in the JMS component, and in this sample we use ActiveMQ
JmsEndpoint endpoint = JmsEndpoint.newInstance(replyDestination, activeMQComponent);
// now we have the endpoint we can use regular Camel API to send a message to it
template.sendBody(endpoint, "Here is the late reply.");
```

A different solution to sending a reply is to provide the **replyDestination** object in the same Exchange property when sending. Camel will then pick up this property and use it for the real destination. The endpoint URI must include a dummy destination, however. For example:

```
// we pretend to send it to some non existing dummy queue
template.send("activemq:queue:dummy, new Processor() {
    public void process(Exchange exchange) throws Exception {
        // and here we override the destination with the ReplyTo destination object so the message is
sent to there instead of dummy
        exchange.getIn().setHeader(JmsConstants.JMS_DESTINATION, replyDestination);
        exchange.getIn().setBody("Here is the late reply.");
    }
}
```

170.17. USING A REQUEST TIMEOUT

In the sample below we send a Request Reply style message Exchange (we use the **requestBody** method = **InOut**) to the slow queue for further processing in Camel and we wait for a return reply:

170.18. SAMPLES

JMS is used in many examples for other components as well. But we provide a few samples below to get started.

170.18.1. Receiving from JMS

In the following sample we configure a route that receives JMS messages and routes the message to a POJO:

```
from("jms:queue:foo").
  to("bean:myBusinessLogic");
```

You can of course use any of the EIP patterns so the route can be context based. For example, here's how to filter an order topic for the big spenders:

```
from("jms:topic:OrdersTopic").
  filter().method("myBean", "isGoldCustomer").
    to("jms:queue:BigSpendersQueue");
```

170.18.2. Sending to JMS

In the sample below we poll a file folder and send the file content to a JMS topic. As we want the content of the file as a **TextMessage** instead of a **BytesMessage**, we need to convert the body to a **String**:

```
from("file://orders").
  convertBodyTo(String.class).
  to("jms:topic:OrdersTopic");
```

170.18.3. Using Annotations

Camel also has annotations so you can use POJO Consuming and POJO Producing.

170.18.4. Spring DSL sample

The preceding examples use the Java DSL. Camel also supports Spring XML DSL. Here is the big spender sample using Spring DSL:

```
<route>
  <from uri="jms:topic:OrdersTopic"/>
  <filter>
    <method bean="myBean" method="isGoldCustomer"/>
    <to uri="jms:queue:BigSpendersQueue"/>
  </filter>
</route>
```

170.18.5. Other samples

JMS appears in many of the examples for other components and EIP patterns, as well in this Camel documentation. So feel free to browse the documentation. If you have time, check out the this tutorial that uses JMS but focuses on how well Spring Remoting and Camel works together Tutorial-JmsRemoting.

170.18.6. Using JMS as a Dead Letter Queue storing Exchange

Normally, when using JMS as the transport, it only transfers the body and headers as the payload. If you want to use JMS with a Dead Letter Channel, using a JMS queue as the Dead Letter Queue, then normally the caused Exception is not stored in the JMS message. You can, however, use the **transferExchange** option on the JMS dead letter queue to instruct Camel to store the entire Exchange in the queue as a **javax.jms.ObjectMessage** that holds a **org.apache.camel.impl.DefaultExchangeHolder**. This allows you to consume from the Dead Letter Queue and retrieve the caused exception from the Exchange property with the key **Exchange.EXCEPTION_CAUGHT**. The demo below illustrates this:

```
// setup error handler to use JMS as queue and store the entire Exchange
errorHandler(deadLetterChannel("jms:queue:dead?transferExchange=true"));
```

Then you can consume from the JMS queue and analyze the problem:

```
from("jms:queue:dead").to("bean:myErrorAnalyzer");

// and in our bean
String body = exchange.getIn().getBody();
Exception cause = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);
// the cause message is
String problem = cause.getMessage();
```

170.18.7. Using JMS as a Dead Letter Channel storing error only

You can use JMS to store the cause error message or to store a custom body, which you can initialize yourself. The following example uses the Message Translator EIP to do a transformation on the failed exchange before it is moved to the JMS dead letter queue:

```
// we sent it to a seda dead queue first
errorHandler(deadLetterChannel("seda:dead"));

// and on the seda dead queue we can do the custom transformation before its sent to the JMS queue
from("seda:dead").transform(exceptionMessage()).to("jms:queue:dead");
```

Here we only store the original cause error message in the transform. You can, however, use any Expression to send whatever you like. For example, you can invoke a method on a Bean or use a custom processor.

170.19. SENDING AN INONLY MESSAGE AND KEEPING THE JMSREPLYTO HEADER

When sending to a JMS destination using **camel-jms** the producer will use the MEP to detect if its *InOnly* or *InOut* messaging. However there can be times where you want to send an *InOnly* message but keeping the **JMSReplyTo** header. To do so you have to instruct Camel to keep it, otherwise the **JMSReplyTo** header will be dropped.

For example to send an *InOnly* message to the foo queue, but with a **JMSReplyTo** with bar queue you can do as follows:

```
template.send("activemq:queue:foo?preserveMessageQos=true", new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getIn().setBody("World");
        exchange.getIn().setHeader("JMSReplyTo", "bar");
    }
});
```

Notice we use **preserveMessageQos=true** to instruct Camel to keep the **JMSReplyTo** header.

170.20. SETTING JMS PROVIDER OPTIONS ON THE DESTINATION

Some JMS providers, like IBM's WebSphere MQ need options to be set on the JMS destination. For example, you may need to specify the **targetClient** option. Since **targetClient** is a WebSphere MQ option and not a Camel URI option, you need to set that on the JMS destination name like so:

```
// ...
.setHeader("CamelJmsDestinationName", constant("queue:///MY_QUEUE?targetClient=1"))
.to("wmq:queue:MY_QUEUE?useMessageIDAsCorrelationID=true");
```

Some versions of WMQ won't accept this option on the destination name and you will get an exception like:

```
com.ibm.msg.client.jms.DetailedJMSException: JMSCC0005: The specified
value 'MY_QUEUE?targetClient=1' is not allowed for
'XMSC_DESTINATION_NAME'
```

A workaround is to use a custom DestinationResolver:

```
JmsComponent wmq = new JmsComponent(connectionFactory);

wmq.setDestinationResolver(new DestinationResolver() {
    public Destination resolveDestinationName(Session session, String destinationName, boolean
pubSubDomain) throws JMSException {
        MQQueueSession wmqSession = (MQQueueSession) session;
        return wmqSession.createQueue("queue:///" + destinationName + "?targetClient=1");
    }
});
```

170.21. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Transactional Client
- Bean Integration
- Tutorial-JmsRemoting
- JMSTemplate gotchas

CHAPTER 171. JMX COMPONENT

171.1. CAMEL JMX

Apache Camel has extensive support for JMX to allow you to monitor and control the Camel managed objects with a JMX client.

Camel also provides a JMX component that allows you to subscribe to MBean notifications. This page is about how to manage and monitor Camel using JMX.

171.2. OPTIONS

The JMX component has no options.

The JMX endpoint is configured using URI syntax:

```
jmx:serverURL
```

with the following path and query parameters:

171.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
serverURL	server url comes from the remaining endpoint		String

171.2.2. Query Parameters (29 parameters):

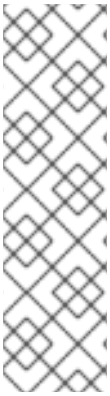
Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
format (consumer)	URI Property: Format for the message body. Either xml or raw. If xml, the notification is serialized to xml. If raw, then the raw java object is set as the body.	xml	String
granularityPeriod (consumer)	URI Property: monitor types only The frequency to poll the bean to check the monitor.	10000	long

Name	Description	Default	Type
monitorType (consumer)	URI Property: monitor types only The type of monitor to create. One of string, gauge, counter.		String
objectDomain (consumer)	**Required** URI Property: The domain for the mbean you're connecting to		String
objectName (consumer)	URI Property: The name key for the mbean you're connecting to. This value is mutually exclusive with the object properties that get passed.		String
observedAttribute (consumer)	URI Property: monitor types only The attribute to observe for the monitor bean.		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
handback (advanced)	URI Property: Value to handback to the listener when a notification is received. This value will be put in the message header with the key jmx.handback		Object
notificationFilter (advanced)	URI Property: Reference to a bean that implements the NotificationFilter.		NotificationFilter
objectProperties (advanced)	URI Property: properties for the object name. These values will be used if the objectName param is not set		Map
reconnectDelay (advanced)	URI Property: The number of seconds to wait before attempting to retry establishment of the initial connection or attempt to reconnect a lost connection	10	int
reconnectOnConnection Failure (advanced)	URI Property: If true the consumer will attempt to reconnect to the JMX server when any connection failure occurs. The consumer will attempt to re-establish the JMX connection every 'x' seconds until the connection is made-- where 'x' is the configured reconnectionDelay	false	boolean

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
testConnectionOnStartup (advanced)	URI Property: If true the consumer will throw an exception if unable to establish the JMX connection upon startup. If false, the consumer will attempt to establish the JMX connection every 'x' seconds until the connection is made – where 'x' is the configured reconnectionDelay	true	boolean
initThreshold (counter)	URI Property: counter monitor only Initial threshold for the monitor. The value must exceed this before notifications are fired.		int
modulus (counter)	URI Property: counter monitor only The value at which the counter is reset to zero		int
offset (counter)	URI Property: counter monitor only The amount to increment the threshold after it's been exceeded.		int
differenceMode (gauge)	URI Property: counter gauge monitor only If true, then the value reported in the notification is the difference from the threshold as opposed to the value itself.	false	boolean
notifyHigh (gauge)	URI Property: gauge monitor only If true, the gauge will fire a notification when the high threshold is exceeded	false	boolean
notifyLow (gauge)	URI Property: gauge monitor only If true, the gauge will fire a notification when the low threshold is exceeded	false	boolean
thresholdHigh (gauge)	URI Property: gauge monitor only Value for the gauge's high threshold		Double
thresholdLow (gauge)	URI Property: gauge monitor only Value for the gauge's low threshold		Double
password (security)	URI Property: credentials for making a remote connection		String
user (security)	URI Property: credentials for making a remote connection		String

Name	Description	Default	Type
notifyDiffer (string)	URI Property: string monitor only If true, the string monitor will fire a notification when the string attribute differs from the string to compare.	false	boolean
notifyMatch (string)	URI Property: string monitor only If true, the string monitor will fire a notification when the string attribute matches the string to compare.	false	boolean
stringToCompare (string)	URI Property: string monitor only Value for the string monitor's string to compare.		String

171.3. ACTIVATING JMX IN CAMEL



NOTE

Spring JAR dependency, required for Camel 2.8 or older

spring-context.jar, **spring-aop.jar**, **spring-beans.jar**, and **spring-core.jar** are needed on the classpath by Camel to be able to use JMX instrumentation. If these .jars are not on the classpath, Camel will fallback to non JMX mode. This situation is logged at **WARN** level using logger name **org.apache.camel.impl.DefaultCamelContext**.

From **Camel 2.9** onwards, the Spring JARs are **no** longer required to run Camel in JMX mode.

171.3.1. Using JMX to manage Apache Camel

By default, JMX instrumentation agent is enabled in Camel, which means that Camel runtime creates and registers MBean management objects with a **MBeanServer** instance in the VM. This allows Camel users to instantly obtain insights into how Camel routes perform down to the individual processor level.

The supported types of management objects are endpoint, route, service, and processor. Some of these management objects also expose lifecycle operations in addition to performance counter attributes.

The **DefaultManagementNamingStrategy** is the default naming strategy which builds object names used for MBean registration. By default, **org.apache.camel** is the domain name for all object names created by **CamelNamingStrategy**. The domain name of the MBean object can be configured by Java VM system property:

```
-Dorg.apache.camel.jmx.mbeanObjectDomainName=your.domain.name
```

Or, by adding a **jmxAgent** element inside the camelContext element in Spring configuration:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" mbeanObjectDomainName="your.domain.name"/>
    ...
</camelContext>
```

Spring configuration always takes precedence over system properties when they both present. It is true for all JMX related configurations.

171.3.2. Disabling JMX instrumentation agent in Camel

You can disable JMX instrumentation agent by setting the Java VM system property as follow:

```
-Dorg.apache.camel.jmx.disabled=true
```

The property value is treated as **boolean**.

Or, by adding a **jmxAgent** element inside the **camelContext** element in Spring configuration:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" disabled="true"/>
    ...
</camelContext>
```

Or in **Camel 2.1** its a bit easier (not having to use JVM system property) if using pure Java as you can disable it as follows:

```
CamelContext camel = new DefaultCamelContext();
camel.disableJMX();
```

171.3.3. Locating a MBeanServer in the Java VM

Each CamelContext can have an instance of **InstrumentationAgent** wrapped inside the **InstrumentationLifecycleStrategy**. The InstrumentationAgent is the object that interfaces with a **MBeanServer** to register / unregister Camel MBeans. Multiple CamelContexts / InstrumentationAgents can / should share a **MBeanServer**. By default, Camel runtime picks the first **MBeanServer** returned by MBeanServerFactory.findMBeanServer method that matches the default domain name of **org.apache.camel**.

You may want to change the default domain name to match the **MBeanServer** instance that you are already using in your application. Especially, if your **MBeanServer** is attached to a JMX connector server, you will not need to create a connector server in Camel.

You can configure the matching default domain name via system property.

```
-Dorg.apache.camel.jmx.mbeanServerDefaultDomain=<your.domain.name>
```

Or, by adding a **jmxAgent** element inside the camelContext element in Spring configuration:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" mbeanServerDefaultDomain="your.domain.name"/>
    ...
</camelContext>
```

If no matching **MBeanServer** can be found, a new one is created and the new `MBeanServer's default domain name is set according to the default and configuration as mentioned above.

It is also possible to use the **PlatformMBeanServer** when it is desirable to manage JVM MBeans by setting the system property. The **MBeanServer** default domain name configuration is ignored as it is not applicable.

CAUTION

Starting in next release (1.5), the default value of **usePlatformMBeanServer** will be changed to **true**. You can set the property to **false** to disable using platform **MBeanServer**.

```
-Dorg.apache.camel.jmx.usePlatformMBeanServer=True
```

Or, by adding a **jmxAgent** element inside the **camelContext** element in Spring configuration:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" usePlatformMBeanServer="true"/>
    ...
</camelContext>
```

171.3.4. Creating JMX RMI Connector Server

JMX connector server enables MBeans to be remotely managed by a JMX client such as JConsole; Camel JMX RMI connector server can be optionally turned on by setting system property and the **MBeanServer** used by Camel is attached to that connector server.

```
-Dorg.apache.camel.jmx.createRmiConnector=True
```

Or, by adding a **jmxAgent** element inside the **camelContext** element in Spring configuration:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" createConnector="true"/>
    ...
</camelContext>
```

171.3.5. JMX Service URL

The default JMX Service URL has the format:

```
service:jmx:rmi:///jndi/rmi://localhost:<registryPort>/<serviceUrlPath>
```

registryPort is the RMI registry port and the default value is **1099**.

You can set the RMI registry port by system property.

```
-Dorg.apache.camel.jmx.rmiConnector.registryPort=<port number>
```

Or, by adding a **jmxAgent** element inside the **camelContext** element in Spring configuration:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" createConnector="true" registryPort="port number"/>
    ...
</camelContext>
```

serviceUrlPath is the path name in the URL and the default value is **/jmxrmi/camel**.

You can set the service URL path by system property.

```
-Dorg.apache.camel.jmx.serviceUrlPath=<path>
```

TIP

Setting ManagementAgent settings in Java

In **Camel 2.4** onwards you can also set the various options on the **ManagementAgent**:

```
context.getManagementStrategy().getManagementAgent().setServiceUrlPath("/foo/bar");
context.getManagementStrategy().getManagementAgent().setRegistryPort(2113);
context.getManagementStrategy().getManagementAgent().setCreateConnector(true);
```

Or, by adding a **jmxAgent** element inside the camelContext element in Spring configuration:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" createConnector="true" serviceUrlPath="path"/>
    ...
</camelContext>
```

By default, RMI server object listens on a dynamically generated port, which can be a problem for connections established through a firewall. In such situations, RMI connection port can be explicitly set by the system property.

```
-Dorg.apache.camel.jmx.rmiConnector.connectorPort=<port number>
```

Or, by adding a **jmxAgent** element inside the **camelContext** element in Spring configuration:

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
  <jmxAgent id="agent" createConnector="true" connectorPort="port number"/>
    ...
</camelContext>
```

When the connector port option is set, the JMX service URL will become:

```
service:jmx:rmi://localhost:<connectorPort>/jndi/rmi://localhost:<registryPort>/<serviceUrlPath>
```

171.3.6. The System Properties for Camel JMX support

Property Name	value	Description
org.apache.camel.jmx	**true** or **false**	if is **true**, it will enable jmx feature in Camel

See more system properties in this section below: *jmxAgent Properties Reference*.

171.3.7. How to use authentication with JMX

JMX in the JDK have features for authentication and also for using secure connections over SSL. You have to refer to the SUN documentation how to use this:

- http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html
- http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html

171.3.8. JMX inside an Application Server

171.3.8.1. Tomcat 6

See this page for details about enabling JMX in Tomcat.

In short, modify your catalina.sh (or catalina.bat in Windows) file to set the following options...

```
set CATALINA_OPTS=-Dcom.sun.management.jmxremote \
  -Dcom.sun.management.jmxremote.port=1099 \
  -Dcom.sun.management.jmxremote.ssl=false \
  -Dcom.sun.management.jmxremote.authenticate=false
```

171.3.8.2. JBoss AS 4

By default JBoss creates its own **MBeanServer**. To allow Camel to expose to the same server follow these steps:

1. Tell Camel to use the Platform **MBeanServer** (This defaults to true in Camel 1.5)

```
<camel:camelContext id="camelContext">
  <camel:jmxAgent id="jmxAgent" mbeanObjectDomainName="org.yourname"
usePlatformMBeanServer="true"  />
</camel:camelContext>
```

1. Alter your JBoss instance to use the Platform **MBeanServer**.
   Add the following property to your **JAVA_OPTS** by editing **run.sh** or **run.conf -Djboss.platform.mbeanserver**. See http://wiki.jboss.org/wiki/JBossMBeansInJConsole

171.3.8.3. WebSphere

Alter the **mbeanServerDefaultDomain** to be **WebSphere**:

```
<camel:jmxAgent id="agent" createConnector="true" mbeanObjectDomainName="org.yourname"
usePlatformMBeanServer="false" mbeanServerDefaultDomain="WebSphere"/>
```

171.3.8.4. Oracle OC4j

The Oracle OC4J J2EE application server will not allow Camel to access the platform **MBeanServer**. You can identify this in the log as Camel will log a **WARNING**.

```
xxx xx, xxxx xx:xx:xx xx org.apache.camel.management.InstrumentationLifecycleStrategy
onContextStart
WARNING: Could not register CamelContext MBean
```

```
java.lang.SecurityException: Unauthorized access from application: xx to MBean:
java.lang:type=ClassLoading
        at
oracle.oc4j.admin.jmx.shared.UserMBeanServer.checkRegisterAccess(UserMBeanServer.java:873)
```

To resolve this you should disable the JMX agent in Camel, see section *Disabling JMX instrumentation agent in Camel*.

171.3.9. Advanced JMX Configuration

The Spring configuration file allows you to configure how Camel is exposed to JMX for management. In some cases, you could specify more information here, like the connector's port or the path name.

171.3.10. Example:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" createConnector="true" registryPort="2000"
mbeanServerDefaultDomain="org.apache.camel.test"/>
    <route>
      <from uri="seda:start"/>
      <to uri="mock:result"/>
    </route>
</camelContext>
```

If you wish to change the Java 5 JMX settings you can use various JMX system properties

For example you can enable remote JMX connections to the Sun JMX connector, via setting the following environment variable (using **set** or **export** depending on your platform). These settings only configure the Sun JMX connector within Java 1.5+, not the JMX connector that Camel creates by default.

```
SUNJMX=-Dcom.sun.management.jmxremote=true -Dcom.sun.management.jmxremote.port=1616 \
-Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
```

(The SUNJMX environment variable is simple used by the startup script for Camel, as additional startup parameters for the JVM. If you start Camel directly, you'll have to pass these parameters yourself.)

171.3.11. jmxAgent Properties Reference

Spring property	System property	Default Value	Description
id			The JMX agent name, and it is not optional
usePlatformMBeanServer	**org.apache.camel.jmx.usePlatformMBeanServer**	**false**, **true** - Release 1.5 or later	If **true**, it will use the **MBeanServer** from the JVM
mbeanServerDefaultDomain	**org.apache.camel.jmx.mbeanServerDefaultDomain**	**org.apache.camel**	The default JMX domain of the **MBeanServer**

Spring property	System property	Default Value	Description
mbeanObjectDomainName	**org.apache.camel.jmx.mbeanObjectDomainName**	**org.apache.camel**	The JMX domain that all object names will use
createConnector	**org.apache.camel.jmx.createRmiConnect**	**false**	If we should create a JMX connector (to allow remote management) for the **MBeanServer**
registryPort	**org.apache.camel.jmx.rmiConnector.registryPort**	**1099**	The port that the JMX RMI registry will use
connectorPort	**org.apache.camel.jmx.rmiConnector.connectorPort**	-1 (dynamic)	The port that the JMX RMI server will use
serviceUrlPath	**org.apache.camel.jmx.serviceUrlPath**	**/jmxrmi/camel**	The path that JMX connector will be registered under
onlyRegisterProcessorWithCustomId	**org.apache.camel.jmx.onlyRegisterProcessorWithCustomId**	**false**	**Camel 2.0:** If this option is enabled then only processors with a custom id set will be registered. This allows you to filer out unwanted processors in the JMX console.
statisticsLevel		**All** / **Default**	**Camel 2.1:** Configures the level for whether performance statistics is enabled for the MBean. See section *Configuring level of granularity for performance statistics* for more details. From **Camel 2.16** onwards the All option is renamed to Default, and a new Extended option has been introduced which allows gathered additional runtime JMX metrics.

Spring property	System property	Default Value	Description
includeHostName	**org.apache.camel.jmx.includeHostName**		**Camel 2.13:** Whether to include the hostname in the MBean naming. From Camel 2.13 onwards this is default **false**, where as in older releases its default **true**. You can use this option to restore old behavior if really needed.
useHostIPAddress	**org.apache.camel.jmx.useHostIPAddress**	**false**	**Camel 2.16:** Whether to use hostname or IP Address in the service url when creating the remote connector. By default the hostname will be used.
loadStatisticsEnabled	**org.apache.camel.jmx.loadStatisticsEnabled**	**false**	**Camel 2.16:**Whether load statistics is enabled (gathers load statistics using a background thread per CamelContext).
endpointRuntimeStatisticsEnabled	**org.apache.camel.jmx.endpointRuntimeStatisticsEnabled**	**true**	**Camel 2.16:** Whether endpoint runtime statistics is enabled (gathers runtime usage of each incoming and outgoing endpoints).

171.3.12. Configuring whether to register MBeans always, for new routes or just by default

Available as of Camel 2.7

Camel now offers 2 settings to control whether or not to register mbeans

Option	Default	Description
registerAlways	**false**	If enabled then MBeans is always registered.

Option	Default	Description
registerNewRoutes	**true**	If enabled then adding new routes after CamelContext has been started will also register MBeans from that given route.

By default Camel registers MBeans for all the routes configured when its starting. The **registerNewRoutes** option control if MBeans should also be registered if you add new routes thereafter. You can disable this, if you for example add and remove temporary routes where management is not needed.

Be a bit caution to use the **registerAlways** option when using dynamic EIP patterns such as the Recipient List having unique endpoints. If so then each unique endpoint and its associated services/producers would also be registered. This could potential lead to system degration due the rising number of mbeans in the registry. A MBean is not a light-weight object and thus consumes memory.

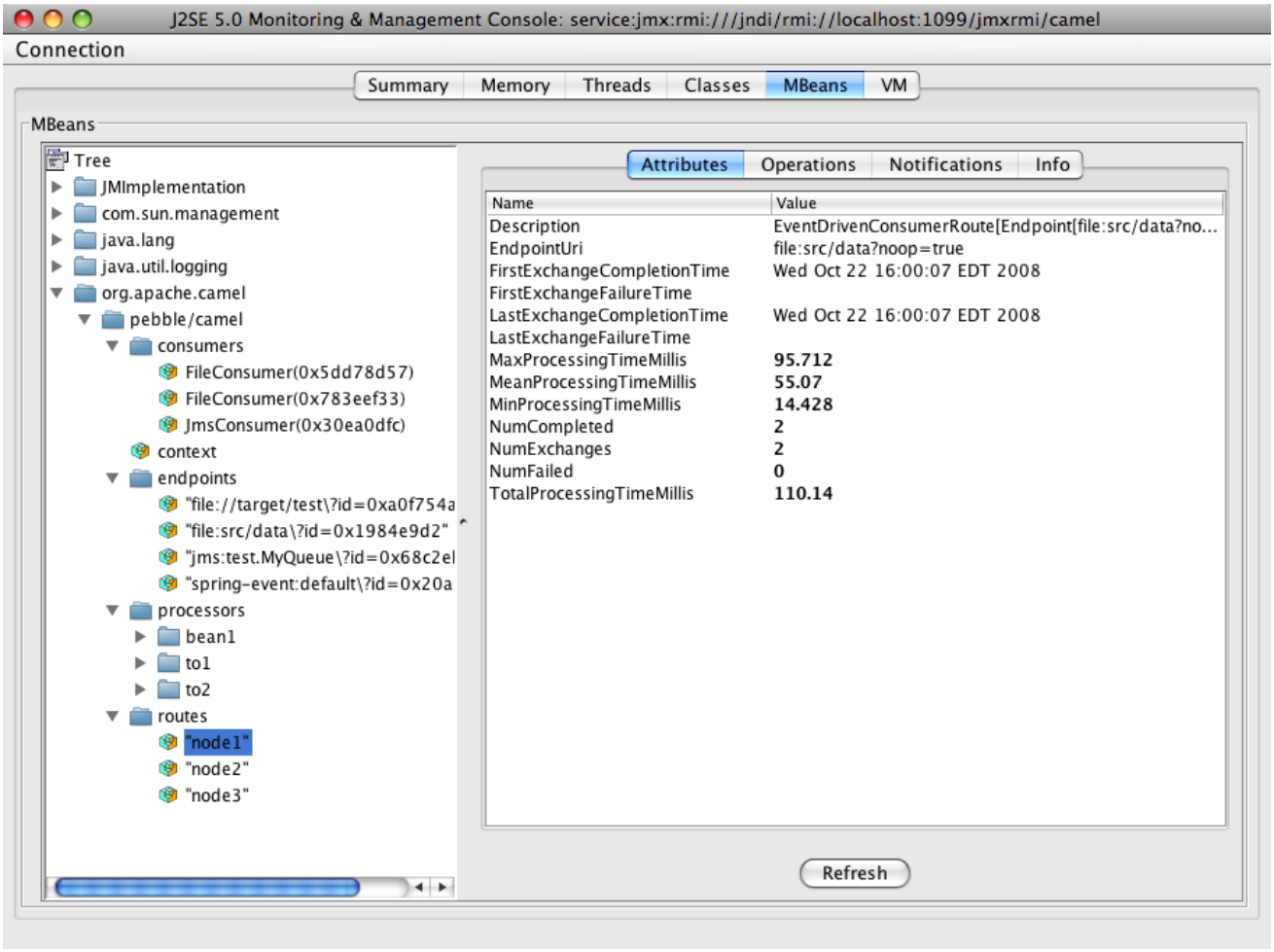
171.4. MONITORING CAMEL USING JMX

171.4.1. Using JConsole to monitor Camel

The **CamelContext** should appear in the list of local connections, if you are running JConsole on the same host as Camel.

To connect to a remote Camel instance, or if the local process does not show up, use Remote Process option, and enter an URL. Here is an example localhost **URL:service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/camel**.

Using the Apache Camel with JConsole:



171.4.2. Which endpoints are registered

In **Camel 2.1** onwards **only singleton** endpoints are registered as the overhead for non singleton will be substantial in cases where thousands or millions of endpoints are used. This can happens when using a Recipient List EIP or from a **ProducerTemplate** that sends a lot of messages.

171.4.3. Which processors are registered

See this FAQ.

171.4.4. How to use the JMX NotificationListener to listen the camel events?

The Camel notification events give a coarse grained overview what is happening. You can see lifecycle event from context and endpoints and you can see exchanges being received by and sent to endpoints.

From **Camel 2.4** you can use a custom JMX NotificationListener to listen the camel events.

First you need to set up a **JmxNotificationEventNotifier** before you start the CamelContext:

```
// Set up the JmxNotificationEventNotifier
notifier = new JmxNotificationEventNotifier();
notifier.setSource("MyCamel");
notifier.setIgnoreCamelContextEvents(true);
notifier.setIgnoreRouteEvents(true);
notifier.setIgnoreServiceEvents(true);
```

```
CamelContext context = new DefaultCamelContext(createRegistry());
context.getManagementStrategy().addEventNotifier(notifier);
```

Second you can register your listener for listening the event:

```
// register the NotificationListener
ObjectName on = ObjectName.getInstance("org.apache.camel:context=camel-
1,type=eventnotifiers,name=JmxEventNotifier");
MyNotificationListener listener = new MyNotificationListener();
context.getManagementStrategy().getManagementAgent().getMBeanServer().addNotificationListener(o
n,
    listener,
    new NotificationFilter() {
        private static final long serialVersionUID = 1L;

        public boolean isNotificationEnabled(Notification notification) {
            return notification.getSource().equals("MyCamel");
        }
    }, null);
```

171.4.5. Using the Tracer MBean to get fine grained tracing

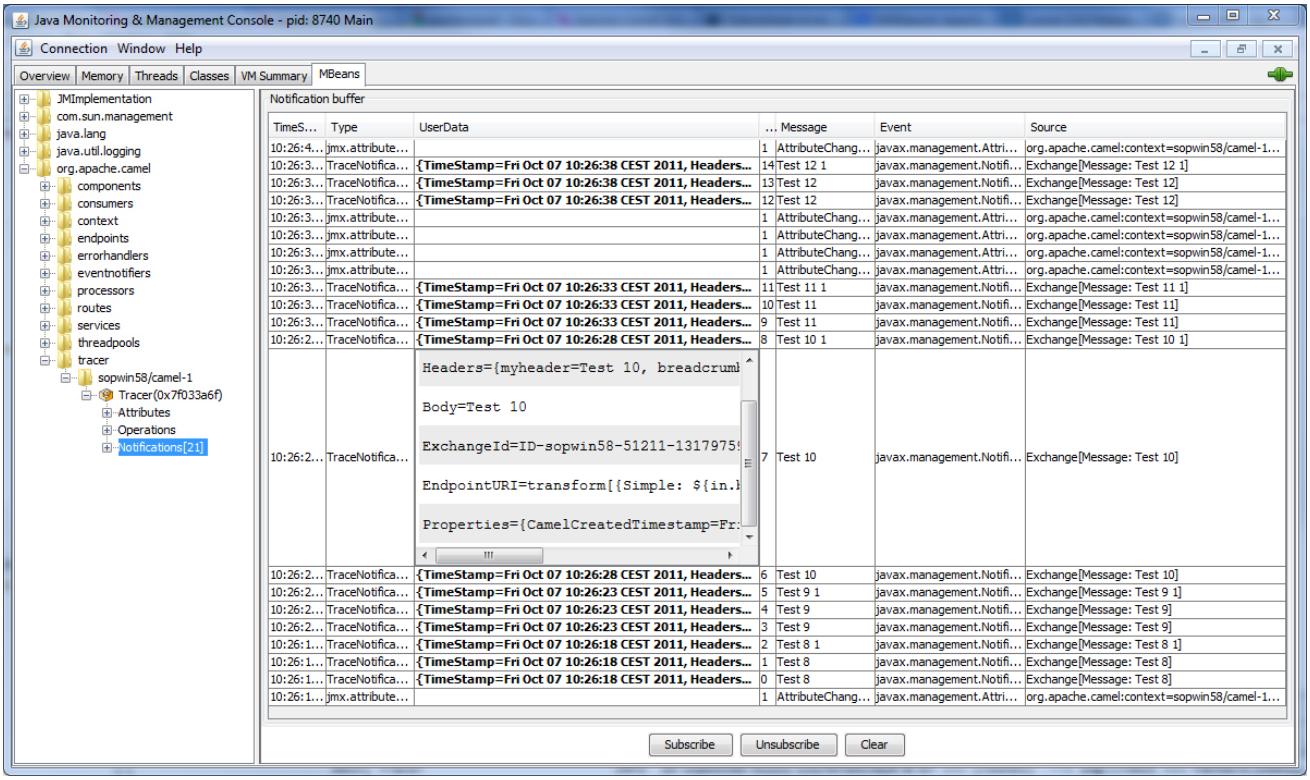
Additionally to the coarse grained notifications above **Camel 2.9.0** support JMX Notification for fine grained trace events.

These can be found in the Tracer MBean. To activate fine grained tracing you first need to activate tracing on the context or on a route.

This can either be done when configuring the context or on the context / route MBeans.

As a second step you have to set the **jmxTraceNotifications** attribute to **true** on the tracer. This can again be done when configuring the context or at runtime on the tracer MBean.

Now you can register for TraceEvent Notifications on the Tracer MBean using JConsole. There will be one Notification for every step on the route with all exchange and message details:

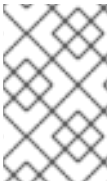


171.5. USING JMX FOR YOUR OWN CAMEL CODE

171.5.1. Registering your own Managed Endpoints

Available as of Camel 2.0

You can decorate your own endpoints with Spring managed annotations **@ManagedResource** to allow to register them in the Camel **MBeanServer** and thus access your custom MBeans using JMX.



NOTE

In **Camel 2.1** we have changed this to apply other than just endpoints but then you need to implement the interface **org.apache.camel.spi.ManagementAware** as well. More about this later.

For example we have the following custom endpoint where we define some options to be managed:

```
@ManagedResource(description = "Our custom managed endpoint")
public class CustomEndpoint extends MockEndpoint implements
ManagementAware<CustomEndpoint> {

    public CustomEndpoint(final String endpointUri, final Component component) {
        super(endpointUri, component);
    }

    public Object getManagedObject(CustomEndpoint object) {
        return this;
    }

    public boolean isSingleton() {
        return true;
    }
```

```
    protected String createEndpointUri() {
        return "custom";
    }

    @ManagedAttribute
    public String getFoo() {
        return "bar";
    }

    @ManagedAttribute
    public String getEndpointUri() {
        return super.getEndpointUri();
    }
}
```

Notice from **Camel 2.9** onwards its encouraged to use the **@ManagedResource**, **@ManagedAttribute**, and **@ManagedOperation** from the **org.apache.camel.api.management** package. This allows your custom code to not depend on Spring JARs.

171.5.2. Programming your own Managed Services

Available as of Camel 2.1

Camel now offers to use your own MBeans when registering services for management. What that means is for example you can develop a custom Camel component and have it expose MBeans for endpoints, consumers and producers etc. All you need to do is to implement the interface **org.apache.camel.spi.ManagementAware** and return the managed object Camel should use.

Now before you think oh boys the JMX API is really painful and terrible, then yeah you are right. Lucky for us Spring though too and they created a range of annotations you can use to export management on an existing bean. That means that you often use that and just return **this** in the **getManagedObject** from the **ManagementAware** interface. For an example see the code example above with the **CustomEndpoint**.

Now in **Camel 2.1** you can do this for all the objects that Camel registers for management which are quite a bunch, but not all.

For services which do not implement this **ManagementAware** interface then Camel will fallback to using default wrappers as defined in the table below:

Type	MBean wrapper
CamelContext	**ManagedCamelContext**
Component	**ManagedComponent**
Endpoint	**ManagedEndpoint**
Consumer	**ManagedConsumer**
Producer	**ManagedProducer**

Type	MBean wrapper
Route	**ManagedRoute**
Processor	**ManagedProcessor**
Tracer	**ManagedTracer**
Service	**ManagedService**

In addition to that there are some extended wrappers for specialized types such as:

Type	MBean wrapper
ScheduledPollConsumer	**ManagedScheduledPollConsumer**
BrowsableEndpoint	**ManagedBrowseableEndpoint**
Throttler	**ManagedThrottler**
Delayer	**ManagedDelayer**
SendProcessor	**ManagedSendProcessor**

And in the future we will add additional wrappers for more EIP patterns.

171.5.3. ManagementNamingStrategy

Available as of Camel 2.1

Camel provides a pluggable API for naming strategy by **org.apache.camel.spi.ManagementNamingStrategy**. A default implementation is used to compute the MBean names that all MBeans are registered with.

171.5.4. Management naming pattern

Available as of Camel 2.10

From **Camel 2.10** onwards we made it easier to configure a naming pattern for the MBeans. The pattern is used as part of the **ObjectName** as they key after the domain name.

By default Camel will use MBean names for the **ManagedCamelContextMBean** as follows:

```
org.apache.camel:context=localhost/camel-1,type=context,name=camel-1
```

And from **Camel 2.13** onwards the hostname is not included in the MBean names, so the above example would be as follows:

```
org.apache.camel:context=camel-1,type=context,name=camel-1
```

If you configure a name on the **CamelContext** then that name is part of the **ObjectName** as well. For example if we have

```
<camelContext id="myCamel" ...>
```

Then the MBean names will be as follows:

```
org.apache.camel:context=localhost/myCamel,type=context,name=myCamel
```

Now if there is a naming clash in the JVM, such as there already exists a MBean with that given name above, then Camel will by default try to auto correct this by finding a new free name in the **JMXMBeanServer** by using a counter. As shown below the counter is now appended, so we have **myCamel-1** as part of the **ObjectName**:

```
org.apache.camel:context=localhost/myCamel-1,type=context,name=myCamel
```

This is possible because Camel uses a naming pattern by default that supports the following tokens:

- **camelId** = the CamelContext id (eg the name)
- **name** - same as **camelId**
- **counter** - an incrementing counter * **bundleId** - the OSGi bundle id (only for OSGi environments)
- **symbolicName** - the OSGi symbolic name (only for OSGi environments)
- **version** - the OSGi bundle version (only for OSGi environments)

The default naming pattern is differentiated between OSGi and non-OSGi as follows:

- non OSGI: **name**
- OSGi: **bundleId-name**
- OSGi **Camel 2.13: symbolicName**

However if there is a naming clash in the **JMXMBeanServer** then Camel will automatic fallback and use the **counter** in the pattern to remedy this. And thus the following patterns will then be used:

- non OSGI: **name-counter**
- OSGi: **bundleId-name-counter**
- OSGi **Camel 2.13: symbolicName-counter**

If you set an explicit naming pattern, then that pattern is always used, and the default patterns above is **not** used.

This allows us to have full control, very easily, of the naming for both the **CamelContext** id in the Registry as well the JMX MBeans in the **JMXMBeanRegistry**.

From **Camel 2.15** onwards you can configure the default management name pattern using a JVM system property, to configure this globally for the JVM. Notice that you can override this pattern by configure it explicit, as shown in the examples further below.

Set a JVM system property to use a default management name pattern that prefixes the name with cool.

```
System.setProperty(JmxSystemPropertyKeys.MANAGEMENT_NAME_PATTERN, "cool-#name#");
```

So if we want to explicit name both the **CamelContext** and to use fixed MBean names, that do not change (eg has no counters), then we can use the new **managementNamePattern** attribute:

```
<camelContext id="myCamel" managementNamePattern="#name#">
```

Then the MBean names will always be as follows:

```
org.apache.camel:context=localhost/myCamel,type=context,name=myCamel
```

In Java, you can configure the **managementNamePattern** as follows:

```
context.getManagementNameStrategy().setNamePattern("#name#");
```

You can also use a different name in the **managementNamePattern** than the id, so for example we can do:

```
<camelContext id="myCamel" managementNamePattern="coolCamel">
```

You may want to do this in OSGi environments in case you do not want the OSGi bundle id as part of the MBean names. As the OSGi bundle id can change if you restart the server, or uninstall and install the same application. You can then do as follows to not use the OSGi bundle id as part of the name:

```
<camelContext id="myCamel" managementNamePattern="#name#">
```

Note this requires that **myCamel** is unique in the entire JVM. If you install a 2nd Camel application that has the same **CamelContext** id and **managementNamePattern** then Camel will fail upon starting, and report a MBean already exists exception.

171.5.5. ManagementStrategy

Available as of Camel 2.1

Camel now provides a totally pluggable management strategy that allows you to be 100% in control of management. It is a rich interface with many methods for management. Not only for adding and removing managed objects from the **MBeanServer**, but also event notification is provided as well using the **org.apache.camel.spi.EventNotifier** API. What it does, for example, is make it easier to provide an adapter for other management products. In addition, it also allows you to provide more details and features that are provided out of the box at Apache.

171.5.6. Configuring level of granularity for performance statistics

Available as of Camel 2.1

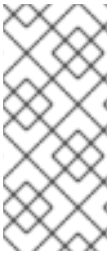
You can now set a pre set level whether performance statistics is enabled or not when Camel start ups. The levels are

- **Extended** - As default but with additional statistics gathered during runtime such as fine grained level of usage of endpoints and more. This options requires Camel 2.16

- **All** / **Default** - Camel will enable statistics for both routes and processors (fine grained). From **Camel 2.16** onwards the All option was renamed to Default.
- **RoutesOnly** - Camel will only enable statistics for routes (coarse grained)
- **Off** - Camel will not enable statistics for any.

From **Camel 2.9** onwards the performance statistics also include average load statistics per CamelContext and Route MBeans. The statistics is average load based on the number of in-flight exchanges, on a per 1, 5, and 15 minute rate. This is similar to load statistics on Unix systems. **Camel 2.11** onwards allows you to explicit disable load performance statistics by setting **loadStatisticsEnabled=false** on the **<jmxAgent>**. Note that it will be off if the statics level is configured to off as well. From **Camel 2.13** onwards the load performance statistics is by default disabled. You can enable this by setting **loadStatisticsEnabled=true** on the **<jmxAgent>**.

At runtime you can always use the management console (such as JConsole) to change on a given route or processor whether its statistics are enabled or not.



NOTE

What does statistics enabled mean?

Statistics enabled means that Camel will do fine grained performance statistics for that particular MBean. The statistics you can see are many, such as: number of exchanges completed/failed, last/total/mina/max/mean processing time, first/last failed time, etc.

Using Java DSL you set this level by:

```
// only enable routes when Camel starts
context.getManagementStrategy().setStatisticsLevel(ManagementStatisticsLevel.RoutesOnly);
```

And from Spring DSL you do:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
   <jmxAgent id="agent" statisticsLevel="RoutesOnly"/>
 ...
</camelContext>
```

171.6. HIDING SENSITIVE INFORMATION

Available as of Camel 2.12

By default, Camel enlists MBeans in JMX such as endpoints configured using URIs. In this configuration, there may be sensitive information such as passwords.

This information can be hidden by enabling the **mask** option as shown below:

Using Java DSL you turn this on by:

```
// only enable routes when Camel starts
context.getManagementStrategy().getManagementAgent().setMask(true);
```

And from Spring DSL you do:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <jmxAgent id="agent" mask="true"/>
 ...
</camelContext>
```

This will mask URIs having options such as password and passphrase, and use **xxxxxx** as the replacement value.

171.6.1. Declaring which JMX attributes and operations to mask

On the **org.apache.camel.api.management.ManagedAttribute** and **org.apache.camel.api.management.ManagedOperation**, the attribute **mask** can be set to **true** to indicate that the result of this JMX attribute/operation should be masked (if enabled on JMX agent, see above).

For example, on the default managed endpoints from camel-core **org.apache.camel.api.management.mbean.ManagedEndpointMBean**, we have declared that the **EndpointUri** JMX attribute is masked:

```
@ManagedAttribute(description = "Endpoint URI", mask = true)
String getEndpointUri();
```

171.7. SEE ALSO

- Management Example
- Why is my processor not showing up in JConsole

CHAPTER 172. JOLT COMPONENT

Available as of Camel version 2.16

The **jolt:** component allows you to process a JSON messages using an JOLT specification. This can be ideal when doing JSON to JSON transformation.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jolt</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

172.1. URI FORMAT

```
jolt:specName[?options]
```

Where **specName** is the classpath-local URI of the specification to invoke; or the complete URL of the remote specification (eg: file://folder/myfile.json).

You can append query options to the URI in the following format, **?option=value&option=value&...**

172.2. OPTIONS

The JOLT component supports 2 options which are listed below.

Name	Description	Default	Type
transform (advanced)	Explicitly sets the Transform to use. If not set a Transform specified by the transformDsl will be created		Transform
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The JOLT endpoint is configured using URI syntax:

```
jolt:resourceUri
```

with the following path and query parameters:

172.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

172.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
inputType (producer)	Specifies if the input is hydrated JSON or a JSON String.	Hydrated	JoltInputOutputType
outputType (producer)	Specifies if the output should be hydrated JSON or a JSON String.	Hydrated	JoltInputOutputType
transformDsl (producer)	Specifies the Transform DSL of the endpoint resource. If none is specified Chainr will be used.	Chainr	JoltTransformType
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

172.3. SAMPLES

For example you could use something like

```
from("activemq:My.Queue").
  to("jolt:com/acme/MyResponse.json");
```

And a file based resource:

```
from("activemq:My.Queue").
  to("jolt:file://myfolder/MyResponse.json?contentCache=true").
  to("activemq:Another.Queue");
```

You can also specify what specification the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelJoltResourceUri").constant("path/to/my/spec.json").
  to("jolt:dummy");
```

172.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 173. JPA COMPONENT

Available as of Camel version 1.0

The **jpa** component enables you to store and retrieve Java objects from persistent storage using EJB 3's Java Persistence Architecture (JPA), which is a standard interface layer that wraps Object/Relational Mapping (ORM) products such as OpenJPA, Hibernate, TopLink, and so on.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jpa</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

173.1. SENDING TO THE ENDPOINT

You can store a Java entity bean in a database by sending it to a JPA producer endpoint. The body of the *In* message is assumed to be an entity bean (that is, a POJO with an `@Entity` annotation on it) or a collection or array of entity beans.

If the body is a List of entities, make sure to use **entityType=java.util.ArrayList** as a configuration passed to the producer endpoint.

If the body does not contain one of the previous listed types, put a Message Translator in front of the endpoint to perform the necessary conversion first.

From **Camel 2.19** onwards you can use **query**, **namedQuery** or **nativeQuery** for the producer as well. Also in the value of the **parameters**, you can use Simple expression which allows you to retrieve parameter values from Message body, header and etc. Those query can be used for retrieving a set of data with using **SELECT** JPQL/SQL statement as well as executing bulk update/delete with using **UPDATE**/**DELETE** JPQL/SQL statement. Please note that you need to specify **useExecuteUpdate** to **true** if you execute **UPDATE**/**DELETE** with **namedQuery** as camel don't look into the named query unlike **query** and **nativeQuery**.

173.2. CONSUMING FROM THE ENDPOINT

Consuming messages from a JPA consumer endpoint removes (or updates) entity beans in the database. This allows you to use a database table as a logical queue: consumers take messages from the queue and then delete/update them to logically remove them from the queue.

If you do not wish to delete the entity bean when it has been processed (and when routing is done), you can specify **consumeDelete=false** on the URI. This will result in the entity being processed each poll.

If you would rather perform some update on the entity to mark it as processed (such as to exclude it from a future query) then you can annotate a method with `@Consumed` which will be invoked on your entity bean when the entity bean when it has been processed (and when routing is done).

From **Camel 2.13** onwards you can use `@PreConsumed` which will be invoked on your entity bean before it has been processed (before routing).

If you are consuming a lot (100K+) of rows and experience OutOfMemory problems you should set the maximumResults to sensible value.

173.3. URI FORMAT

```
jpa:entityClassName[?options]
```

For sending to the endpoint, the *entityClassName* is optional. If specified, it helps the Type Converter to ensure the body is of the correct type.

For consuming, the *entityClassName* is mandatory.

You can append query options to the URI in the following format, **?option=value&option=value&…**

173.4. OPTIONS

The JPA component supports 5 options which are listed below.

Name	Description	Default	Type
entityManagerFactory (common)	To use the EntityManagerFactory. This is strongly recommended to configure.		EntityManagerFactory
transactionManager (common)	To use the PlatformTransactionManager for managing transactions.		PlatformTransactionManager
joinTransaction (common)	The camel-jpa component will join transaction by default. You can use this option to turn this off, for example if you use LOCAL_RESOURCE and join transaction doesn't work with your JPA provider. This option can also be set globally on the JpaComponent, instead of having to set it on all endpoints.	true	boolean
sharedEntityManager (common)	Whether to use Spring's SharedEntityManager for the consumer/producer. Note in most cases joinTransaction should be set to false as this is not an EXTENDED EntityManager.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The JPA endpoint is configured using URI syntax:

```
jpa:entityType
```

with the following path and query parameters:

173.4.1. Path Parameters (1 parameters):

Name	Description	Default	Type
entityType	**Required** The JPA annotated class to use as entity.		Class<?>

173.4.2. Query Parameters (42 parameters):

Name	Description	Default	Type
joinTransaction (common)	The camel-jpa component will join transaction by default. You can use this option to turn this off, for example if you use LOCAL_RESOURCE and join transaction doesn't work with your JPA provider. This option can also be set globally on the JpaComponent, instead of having to set it on all endpoints.	true	boolean
maximumResults (common)	Set the maximum number of results to retrieve on the Query.	-1	int
namedQuery (common)	To use a named query.		String
nativeQuery (common)	To use a custom native query. You may want to use the option resultClass also when using native queries.		String
parameters (common)	This key/value mapping is used for building the query parameters. It is expected to be of the generic type java.util.Map where the keys are the named parameters of a given JPA query and the values are their corresponding effective values you want to select for. When it's used for producer, Simple expression can be used as a parameter value. It allows you to retrieve parameter values from the message body, header and etc.		Map
persistenceUnit (common)	**Required** The JPA persistence unit used by default.	camel	String
query (common)	To use a custom query.		String
resultClass (common)	Defines the type of the returned payload (we will call entityManager.createNativeQuery(nativeQuery, resultClass) instead of entityManager.createNativeQuery(nativeQuery)). Without this option, we will return an object array. Only has an affect when using in conjunction with native query when consuming data.		Class<?>

Name	Description	Default	Type
sharedEntityManager (common)	Whether to use Spring's SharedEntityManager for the consumer/producer. Note in most cases joinTransaction should be set to false as this is not an EXTENDED EntityManager.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
consumeDelete (consumer)	If true, the entity is deleted after it is consumed; if false, the entity is not deleted.	true	boolean
consumeLockEntity (consumer)	Specifies whether or not to set an exclusive lock on each entity bean while processing the results from polling.	true	boolean
deleteHandler (consumer)	To use a custom DeleteHandler to delete the row after the consumer is done processing the exchange		Object>
lockModeType (consumer)	To configure the lock mode on the consumer.	PESSIMISTIC_WRITE	LockModeType
maxMessagesPerPoll (consumer)	An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set. Can be used to avoid polling many thousands of messages when starting up the server. Set a value of 0 or negative to disable.		int
preDeleteHandler (consumer)	To use a custom Pre-DeleteHandler to delete the row after the consumer has read the entity.		Object>
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
skipLockedEntity (consumer)	To configure whether to use NOWAIT on lock and silently skip the entity.	false	boolean

Name	Description	Default	Type
transacted (consumer)	Whether to run the consumer in transacted mode, by which all messages will either commit or rollback, when the entire batch has been processed. The default behavior (false) is to commit all the previously successfully processed messages, and only rollback the last failed message.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
flushOnSend (producer)	Flushes the EntityManager after the entity bean has been persisted.	true	boolean
remove (producer)	Indicates to use entityManager.remove(entity).	false	boolean
useExecuteUpdate (producer)	To configure whether to use executeUpdate() when producer executes a query. When you use INSERT, UPDATE or DELETE statement as a named query, you need to specify this option to 'true'.		Boolean
usePassedInEntityManager (producer)	If set to true, then Camel will use the EntityManager from the header JpaConstants.ENTITYMANAGER instead of the configured entity manager on the component/endpoint. This allows end users to control which entity manager will be in use.	false	boolean
usePersist (producer)	Indicates to use entityManager.persist(entity) instead of entityManager.merge(entity). Note: entityManager.persist(entity) doesn't work for detached entities (where the EntityManager has to execute an UPDATE instead of an INSERT query)!	false	boolean

Name	Description	Default	Type
entityManagerProperties (advanced)	Additional properties for the entity manager to use.		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler

Name	Description	Default	Type
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

173.5. MESSAGE HEADERS

Camel adds the following message headers to the exchange:

Header	Type	Description
CamelJpaTemplate	**JpaTemplate**	**Not supported anymore since Camel 2.12:** The **JpaTemplate** object that is used to access the entity bean. You need this object in some situations, for instance in a type converter or when you are doing some custom processing. See CAMEL-5932 for the reason why the support for this header has been dropped.
CamelEntityManager	**EntityManager**	**Camel 2.12: JPA consumer / Camel 2.12.2: JPA producer:** The JPA **EntityManager** object being used by **JpaConsumer** or **JpaProducer**.

173.6. CONFIGURING ENTITYMANAGERFACTORY

Its strongly advised to configure the JPA component to use a specific **EntityManagerFactory** instance. If failed to do so each **JpaEndpoint** will auto create their own instance of **EntityManagerFactory** which most often is not what you want.

For example, you can instantiate a JPA component that references the **myEMFactory** entity manager factory, as follows:

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
  <property name="entityManagerFactory" ref="myEMFactory"/>
</bean>
```

In **Camel 2.3** the **JpaComponent** will auto lookup the **EntityManagerFactory** from the Registry which means you do not need to configure this on the **JpaComponent** as shown above. You only need to do so if there is ambiguity, in which case Camel will log a WARN.

173.7. CONFIGURING TRANSACTIONMANAGER

Since **Camel 2.3** the **JpaComponent** will auto lookup the **TransactionManager** from the Registry. If Camel won't find any **TransactionManager** instance registered, it will also look up for the **TransactionTemplate** and try to extract **TransactionManager** from it.

If none **TransactionTemplate** is available in the registry, **JpaEndpoint** will auto create their own instance of **TransactionManager** which most often is not what you want.

If more than single instance of the **TransactionManager** is found, Camel will log a WARN. In such cases you might want to instantiate and explicitly configure a JPA component that references the **myTransactionManager** transaction manager, as follows:

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
  <property name="entityManagerFactory" ref="myEMFactory"/>
  <property name="transactionManager" ref="myTransactionManager"/>
</bean>
```

173.8. USING A CONSUMER WITH A NAMED QUERY

For consuming only selected entities, you can use the **consumer.namedQuery** URI query option. First, you have to define the named query in the JPA Entity class:

```
@Entity
@NamedQuery(name = "step1", query = "select x from MultiSteps x where x.step = 1")
public class MultiSteps {
  ...
}
```

After that you can define a consumer uri like this one:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.namedQuery=step1")
.to("bean:myBusinessLogic");
```

173.9. USING A CONSUMER WITH A QUERY

For consuming only selected entities, you can use the **consumer.query** URI query option. You only have to define the query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.query=select o from
org.apache.camel.examples.MultiSteps o where o.step = 1")
.to("bean:myBusinessLogic");
```

173.10. USING A CONSUMER WITH A NATIVE QUERY

For consuming only selected entities, you can use the **consumer.nativeQuery** URI query option. You only have to define the native query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.nativeQuery=select * from MultiSteps
where step = 1")
.to("bean:myBusinessLogic");
```

If you use the native query option, you will receive an object array in the message body.

173.11. USING A PRODUCER WITH A NAMED QUERY

For retrieving selected entities or execute bulk update/delete, you can use the **namedQuery** URI query option. First, you have to define the named query in the JPA Entity class:

```
@Entity
@NamedQuery(name = "step1", query = "select x from MultiSteps x where x.step = 1")
public class MultiSteps {
   ...
}
```

After that you can define a producer uri like this one:

```
from("direct:namedQuery")
.to("jpa://org.apache.camel.examples.MultiSteps?namedQuery=step1");
```

Note that you need to specify **useExecuteUpdate** option to **true** to execute **UPDATE/DELETE** statement as a named query.

173.12. USING A PRODUCER WITH A QUERY

For retrieving selected entities or execute bulk update/delete, you can use the **query** URI query option. You only have to define the query option:

```
from("direct:query")
.to("jpa://org.apache.camel.examples.MultiSteps?query=select o from
org.apache.camel.examples.MultiSteps o where o.step = 1");
```

173.13. USING A PRODUCER WITH A NATIVE QUERY

For retrieving selected entities or execute bulk update/delete, you can use the **nativeQuery** URI query option. You only have to define the native query option:

```
from("direct:nativeQuery")
.to("jpa://org.apache.camel.examples.MultiSteps?
resultClass=org.apache.camel.examples.MultiSteps&nativeQuery=select * from MultiSteps where
step = 1");
```

If you use the native query option without specifying **resultClass**, you will receive an object array in the message body.

173.14. EXAMPLE

See Tracer Example for an example using JPA to store traced messages into a database.

173.15. USING THE JPA-BASED IDEMPOTENT REPOSITORY

The Idempotent Consumer from the EIP patterns is used to filter out duplicate messages. A JPA-based idempotent repository is provided.

To use the JPA based idempotent repository.

Procedure

1. Set up a **persistence-unit** in the persistence.xml file:
2. Set up a **org.springframework.orm.jpa.JpaTemplate** which is used by the **org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository**:
3. Configure the error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20
4. Configure the idempotent repository: **org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository**:
5. Create the JPA idempotent repository in the Spring XML file:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route id="JpaMessageIdRepositoryTest">
    <from uri="direct:start" />
    <idempotentConsumer messageIdRepositoryRef="jpaStore">
      <header>messageId</header>
      <to uri="mock:result" />
    </idempotentConsumer>
  </route>
</camelContext>
```

When running this Camel component tests inside your IDE

If you run the tests of this component directly inside your IDE, and not through Maven, then you could see exceptions like these:

```
org.springframework.transaction.CannotCreateTransactionException: Could not open JPA
EntityManager for transaction; nested exception is
<openjpa-2.2.1-r422266:1396819 nonfatal user error>
org.apache.openjpa.persistence.ArgumentException: This configuration disallows runtime
optimization,
but the following listed types were not enhanced at build time or at class load time with a javaagent:
"org.apache.camel.examples.SendEmail".
    at
org.springframework.orm.jpa.JpaTransactionManager.doBegin(JpaTransactionManager.java:427)
    at
org.springframework.transaction.support.AbstractPlatformTransactionManager.getTransaction(Abstra
ctPlatformTransactionManager.java:371)
    at
org.springframework.transaction.support.TransactionTemplate.execute(TransactionTemplate.java:12
7)
    at org.apache.camel.processor.jpa.JpaRouteTest.cleanupRepository(JpaRouteTest.java:96)
    at org.apache.camel.processor.jpa.JpaRouteTest.createCamelContext(JpaRouteTest.java:67)
    at org.apache.camel.test.junit4.CamelTestSupport.doSetUp(CamelTestSupport.java:238)
    at org.apache.camel.test.junit4.CamelTestSupport.setUp(CamelTestSupport.java:208)
```

The problem here is that the source has been compiled or recompiled through your IDE and not through Maven, which would enhance the byte-code at build time . To overcome this you need to enable dynamic byte-code enhancement of OpenJPA . For example, assuming the current OpenJPA version

being used in Camel is 2.2.1, to run the tests inside your IDE you would need to pass the following argument to the JVM:

```
-javaagent:<path_to_your_local_m2_cache>/org/apache/openjpa/openjpa/2.2.1/openjpa-2.2.1.jar
```

173.16. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Tracer Example

CHAPTER 174. JSON FASTJSON DATAFORMAT

Available as of Camel version 2.20

Fastjson is a Data Format which uses the Fastjson Library

```
from("activemq:My.Queue").
  marshal().json(JsonLibrary.Fastjson).
  to("mqseries:Another.Queue");
```

174.1. FASTJSON OPTIONS

The JSon Fastjson dataformat supports 19 options which are listed below.

Name	Default	Java Type	Description
objectMapper		**String**	Lookup and use the existing ObjectMapper with the given id when using Jackson.
useDefaultObjectMapper	**true**	**Boolean**	Whether to lookup and use default Jackson ObjectMapper from the registry.
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
library	**XStream**	**JsonLibrary**	Which json library to use.
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
jsonView		**Class<?>**	When marshalling a POJO to JSON you might want to exclude certain fields from the JSON output. With Jackson you can use JSON views to accomplish this. This option is to refer to the class which has JsonView annotations
include		**String**	If you want to marshal a pojo to JSON, and the pojo has some fields with null values. And you want to skip these null values, you can set this option to NOT_NULL
allowJmsType	**false**	**Boolean**	Used for JMS users to allow the JMSType header from the JMS spec to specify a FQN classname to use to unmarshal to.
collectionTypeName		**String**	Refers to a custom collection type to lookup in the registry to use. This option should rarely be used, but allows to use different collection types than java.util.Collection based as default.
useList	**false**	**Boolean**	To unarmshal to a List of Map or a List of Pojo.

Name	Default	Java Type	Description
enableJaxbAnnotationModule	false	Boolean	Whether to enable the JAXB annotations module when using jackson. When enabled then JAXB annotations can be used by Jackson.
moduleClassNames		String	To use custom Jackson modules com.fasterxml.jackson.databind.Module specified as a String with FQN class names. Multiple classes can be separated by comma.
moduleRefs		String	To use custom Jackson modules referred from the Camel registry. Multiple modules can be separated by comma.
enableFeatures		String	Set of features to enable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
disableFeatures		String	Set of features to disable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
permissions		String	Adds permissions that controls which Java packages and classes XStream is allowed to use during unmarshal from xml/json to Java beans. A permission must be configured either here or globally using a JVM system property. The permission can be specified in a syntax where a plus sign is allow, and minus sign is deny. Wildcards is supported by using . as prefix. For example to allow com.foo and all subpackages then specfy com.foo.. Multiple permissions can be configured separated by comma, such as com.foo.,-com.foo.bar.MySecretBean. The following default permission is always included: -,java.lang.,java.util. unless its overridden by specifying a JVM system property with they key org.apache.camel.xstream.permissions.
allowUnmarshallType	false	Boolean	If enabled then Jackson is allowed to attempt to use the CamelJacksonUnmarshalType header during the unmarshalling. This should only be enabled when desired to be used.
timezone		String	If set then Jackson will use the Timezone when marshalling/unmarshalling. This option will have no effect on the others Json DataFormat, like gson, fastjson and xstream.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

174.2. DEPENDENCIES

To use Fastjson in your camel routes you need to add the dependency on **camel-fastjson** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-fastjson</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 175. JSON GSON DATAFORMAT

Available as of Camel version 2.10

Gson is a Data Format which uses the Gson Library

```
from("activemq:My.Queue").
  marshal().json(JsonLibrary.Gson).
  to("mqseries:Another.Queue");
```

175.1. GSON OPTIONS

The JSon GSon dataformat supports 19 options which are listed below.

Name	Default	Java Type	Description
objectMapper		**String**	Lookup and use the existing ObjectMapper with the given id when using Jackson.
useDefaultObjectMapper	**true**	**Boolean**	Whether to lookup and use default Jackson ObjectMapper from the registry.
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
library	**XStream**	**JsonLibrary**	Which json library to use.
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
jsonView		**Class<?>**	When marshalling a POJO to JSON you might want to exclude certain fields from the JSON output. With Jackson you can use JSON views to accomplish this. This option is to refer to the class which has JsonView annotations
include		**String**	If you want to marshal a pojo to JSON, and the pojo has some fields with null values. And you want to skip these null values, you can set this option to NOT_NULL
allowJmsType	**false**	**Boolean**	Used for JMS users to allow the JMSType header from the JMS spec to specify a FQN classname to use to unmarshal to.
collectionTypeName		**String**	Refers to a custom collection type to lookup in the registry to use. This option should rarely be used, but allows to use different collection types than java.util.Collection based as default.
useList	**false**	**Boolean**	To unarmshal to a List of Map or a List of Pojo.

Name	Default	Java Type	Description
enableJaxbAnnotationModule	false	Boolean	Whether to enable the JAXB annotations module when using jackson. When enabled then JAXB annotations can be used by Jackson.
moduleClassNames		String	To use custom Jackson modules com.fasterxml.jackson.databind.Module specified as a String with FQN class names. Multiple classes can be separated by comma.
moduleRefs		String	To use custom Jackson modules referred from the Camel registry. Multiple modules can be separated by comma.
enableFeatures		String	Set of features to enable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
disableFeatures		String	Set of features to disable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
permissions		String	Adds permissions that controls which Java packages and classes XStream is allowed to use during unmarshal from xml/json to Java beans. A permission must be configured either here or globally using a JVM system property. The permission can be specified in a syntax where a plus sign is allow, and minus sign is deny. Wildcards is supported by using . as prefix. For example to allow com.foo and all subpackages then specfy com.foo.. Multiple permissions can be configured separated by comma, such as com.foo.,-com.foo.bar.MySecretBean. The following default permission is always included: -,java.lang.,java.util. unless its overridden by specifying a JVM system property with they key org.apache.camel.xstream.permissions.
allowUnmarshallType	false	Boolean	If enabled then Jackson is allowed to attempt to use the CamelJacksonUnmarshalType header during the unmarshalling. This should only be enabled when desired to be used.
timezone		String	If set then Jackson will use the Timezone when marshalling/unmarshalling. This option will have no effect on the others Json DataFormat, like gson, fastjson and xstream.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

175.2. DEPENDENCIES

To use Gson in your camel routes you need to add the dependency on **camel-gson** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-gson</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 176. JSON JACKSON DATAFORMAT

Available as of Camel version 2.0

Jackson is a Data Format which uses the Jackson Library

```
from("activemq:My.Queue").
  marshal().json(JsonLibrary.Jackson).
  to("mqseries:Another.Queue");
```

176.1. JACKSON OPTIONS

The JSon Jackson dataformat supports 19 options which are listed below.

Name	Default	Java Type	Description
objectMapper		**String**	Lookup and use the existing ObjectMapper with the given id when using Jackson.
useDefaultObjectMapper	**true**	**Boolean**	Whether to lookup and use default Jackson ObjectMapper from the registry.
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
library	**XStream**	**JsonLibrary**	Which json library to use.
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
jsonView		**Class<?>**	When marshalling a POJO to JSON you might want to exclude certain fields from the JSON output. With Jackson you can use JSON views to accomplish this. This option is to refer to the class which has JsonView annotations
include		**String**	If you want to marshal a pojo to JSON, and the pojo has some fields with null values. And you want to skip these null values, you can set this option to NOT_NULL
allowJmsType	**false**	**Boolean**	Used for JMS users to allow the JMSType header from the JMS spec to specify a FQN classname to use to unmarshal to.
collectionTypeName		**String**	Refers to a custom collection type to lookup in the registry to use. This option should rarely be used, but allows to use different collection types than java.util.Collection based as default.
useList	**false**	**Boolean**	To unarmshal to a List of Map or a List of Pojo.

Name	Default	Java Type	Description
enableJaxbAnnotationModule	false	Boolean	Whether to enable the JAXB annotations module when using jackson. When enabled then JAXB annotations can be used by Jackson.
moduleClassNames		String	To use custom Jackson modules com.fasterxml.jackson.databind.Module specified as a String with FQN class names. Multiple classes can be separated by comma.
moduleRefs		String	To use custom Jackson modules referred from the Camel registry. Multiple modules can be separated by comma.
enableFeatures		String	Set of features to enable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
disableFeatures		String	Set of features to disable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
permissions		String	Adds permissions that controls which Java packages and classes XStream is allowed to use during unmarshal from xml/json to Java beans. A permission must be configured either here or globally using a JVM system property. The permission can be specified in a syntax where a plus sign is allow, and minus sign is deny. Wildcards is supported by using . as prefix. For example to allow com.foo and all subpackages then specfy com.foo.. Multiple permissions can be configured separated by comma, such as com.foo.,-com.foo.bar.MySecretBean. The following default permission is always included: -,java.lang.,java.util. unless its overridden by specifying a JVM system property with they key org.apache.camel.xstream.permissions.
allowUnmarshallType	false	Boolean	If enabled then Jackson is allowed to attempt to use the CamelJacksonUnmarshalType header during the unmarshalling. This should only be enabled when desired to be used.
timezone		String	If set then Jackson will use the Timezone when marshalling/unmarshalling. This option will have no effect on the others Json DataFormat, like gson, fastjson and xstream.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

176.2. USING CUSTOM OBJECTMAPPER

You can configure **JacksonDataFormat** to use a custom **ObjectMapper** in case you need more control of the mapping configuration.

If you setup a single **ObjectMapper** in the registry, then Camel will automatic lookup and use this **ObjectMapper**. For example if you use Spring Boot, then Spring Boot can provide a default **ObjectMapper** for you if you have Spring MVC enabled. And this would allow Camel to detect that there is one bean of **ObjectMapper** class type in the Spring Boot bean registry and then use it. When this happens you should set a **INFO** logging from Camel.

176.3. DEPENDENCIES

To use Jackson in your camel routes you need to add the dependency on **camel-jackson** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jackson</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 177. JSON JOHNZON DATAFORMAT

Available as of Camel version 2.18

Johnzon is a Data Format which uses the Johnzon Library

```
from("activemq:My.Queue").
  marshal().json(JsonLibrary.Johnzon).
  to("mqseries:Another.Queue");
```

177.1. JOHNZON OPTIONS

The JSon Johnzon dataformat supports 19 options which are listed below.

Name	Default	Java Type	Description
objectMapper		**String**	Lookup and use the existing ObjectMapper with the given id when using Jackson.
useDefaultObjectMapper	**true**	**Boolean**	Whether to lookup and use default Jackson ObjectMapper from the registry.
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
library	**XStream**	**JsonLibrary**	Which json library to use.
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
jsonView		**Class<?>**	When marshalling a POJO to JSON you might want to exclude certain fields from the JSON output. With Jackson you can use JSON views to accomplish this. This option is to refer to the class which has JsonView annotations
include		**String**	If you want to marshal a pojo to JSON, and the pojo has some fields with null values. And you want to skip these null values, you can set this option to NOT_NULL
allowJmsType	**false**	**Boolean**	Used for JMS users to allow the JMSType header from the JMS spec to specify a FQN classname to use to unmarshal to.
collectionTypeName		**String**	Refers to a custom collection type to lookup in the registry to use. This option should rarely be used, but allows to use different collection types than java.util.Collection based as default.
useList	**false**	**Boolean**	To unarmshal to a List of Map or a List of Pojo.

Name	Default	Java Type	Description
enableJaxbAnnotationModule	**false**	**Boolean**	Whether to enable the JAXB annotations module when using jackson. When enabled then JAXB annotations can be used by Jackson.
moduleClassNames		**String**	To use custom Jackson modules com.fasterxml.jackson.databind.Module specified as a String with FQN class names. Multiple classes can be separated by comma.
moduleRefs		**String**	To use custom Jackson modules referred from the Camel registry. Multiple modules can be separated by comma.
enableFeatures		**String**	Set of features to enable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
disableFeatures		**String**	Set of features to disable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
permissions		**String**	Adds permissions that controls which Java packages and classes XStream is allowed to use during unmarshal from xml/json to Java beans. A permission must be configured either here or globally using a JVM system property. The permission can be specified in a syntax where a plus sign is allow, and minus sign is deny. Wildcards is supported by using . as prefix. For example to allow com.foo and all subpackages then specfy com.foo.. Multiple permissions can be configured separated by comma, such as com.foo.,-com.foo.bar.MySecretBean. The following default permission is always included: -,java.lang.,java.util. unless its overridden by specifying a JVM system property with they key org.apache.camel.xstream.permissions.
allowUnmarshallType	**false**	**Boolean**	If enabled then Jackson is allowed to attempt to use the CamelJacksonUnmarshalType header during the unmarshalling. This should only be enabled when desired to be used.
timezone		**String**	If set then Jackson will use the Timezone when marshalling/unmarshalling. This option will have no effect on the others Json DataFormat, like gson, fastjson and xstream.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

177.2. DEPENDENCIES

To use Johnzon in your camel routes you need to add the dependency on **camel-johnzon** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-johnzon</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 178. JSON SCHEMA VALIDATOR COMPONENT

Available as of Camel version 2.20

The JSON Schema Validator component performs bean validation of the message body against JSON Schemas v4 draft using the NetworkNT JSON Schema library (https://github.com/networknt/json-schema-validator).

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-json-validator</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

178.1. URI FORMAT

```
json-validator:resourceUri[?options]
```

Where **resourceUri** is some URL to a local resource on the classpath or a full URL to a remote resource or resource on the file system which contains the JSON Schema to validate against.

178.2. URI OPTIONS

The JSON Schema Validator component has no options.

The JSON Schema Validator endpoint is configured using URI syntax:

```
json-validator:resourceUri
```

with the following path and query parameters:

178.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

178.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
failOnNullBody (producer)	Whether to fail if no body exists.	true	boolean
failOnNullHeader (producer)	Whether to fail if no header exists when validating against a header.	true	boolean
headerName (producer)	To validate against a header instead of the message body.		String
errorHandler (advanced)	To use a custom ValidatorErrorHandler. The default error handler captures the errors and throws an exception.		JsonValidatorError Handler
schemaLoader (advanced)	To use a custom schema loader allowing for adding custom format validation. The default implementation will create a schema loader with draft v4 support.		JsonSchemaLoader
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

178.3. EXAMPLE

Assumed we have the following JSON Schema

myschema.json

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "definitions": {},
  "id": "my-schema",
  "properties": {
    "id": {
      "default": 1,
      "description": "An explanation about the purpose of this instance.",
      "id": "/properties/id",
      "title": "The id schema",
      "type": "integer"
    },
    "name": {
      "default": "A green door",
      "description": "An explanation about the purpose of this instance.",
      "id": "/properties/name",
      "title": "The name schema",
```

```
      "type": "string"
    },
    "price": {
      "default": 12.5,
      "description": "An explanation about the purpose of this instance.",
      "id": "/properties/price",
      "title": "The price schema",
      "type": "number"
    }
  },
  "required": [
    "name",
    "id",
    "price"
  ],
  "type": "object"
}
```

we can validate incoming JSON with the following Camel route, where **myschema.json** is loaded from the classpath.

```
from("direct:start")
  .to("json-validator:myschema.json")
  .to("mock:end")
```

CHAPTER 179. JSON XSTREAM DATAFORMAT

Available as of Camel version 2.0

XStream is a Data Format which uses the XStream library to marshal and unmarshal Java objects to and from XML.

To use XStream in your camel routes you need to add the a dependency on **camel-xstream** which implements this data format.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xstream</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

179.1. OPTIONS

The JSon XStream dataformat supports 19 options which are listed below.

Name	Default	Java Type	Description
objectMapper		**String**	Lookup and use the existing ObjectMapper with the given id when using Jackson.
useDefaultObjectMapper	**true**	**Boolean**	Whether to lookup and use default Jackson ObjectMapper from the registry.
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
library	**XStream**	**JsonLibrary**	Which json library to use.
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
jsonView		**Class<?>**	When marshalling a POJO to JSON you might want to exclude certain fields from the JSON output. With Jackson you can use JSON views to accomplish this. This option is to refer to the class which has JsonView annotations
include		**String**	If you want to marshal a pojo to JSON, and the pojo has some fields with null values. And you want to skip these null values, you can set this option to NOT_NULL

Name	Default	Java Type	Description
allowJmsType	false	Boolean	Used for JMS users to allow the JMSType header from the JMS spec to specify a FQN classname to use to unmarshal to.
collectionTypeName		String	Refers to a custom collection type to lookup in the registry to use. This option should rarely be used, but allows to use different collection types than java.util.Collection based as default.
useList	false	Boolean	To unarmshal to a List of Map or a List of Pojo.
enableJaxbAnnotationModule	false	Boolean	Whether to enable the JAXB annotations module when using jackson. When enabled then JAXB annotations can be used by Jackson.
moduleClassNames		String	To use custom Jackson modules com.fasterxml.jackson.databind.Module specified as a String with FQN class names. Multiple classes can be separated by comma.
moduleRefs		String	To use custom Jackson modules referred from the Camel registry. Multiple modules can be separated by comma.
enableFeatures		String	Set of features to enable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma
disableFeatures		String	Set of features to disable on the Jackson com.fasterxml.jackson.databind.ObjectMapper. The features should be a name that matches a enum from com.fasterxml.jackson.databind.SerializationFeature, com.fasterxml.jackson.databind.DeserializationFeature, or com.fasterxml.jackson.databind.MapperFeature Multiple features can be separated by comma

Name	Default	Java Type	Description
permissions		**String**	Adds permissions that controls which Java packages and classes XStream is allowed to use during unmarshal from xml/json to Java beans. A permission must be configured either here or globally using a JVM system property. The permission can be specified in a syntax where a plus sign is allow, and minus sign is deny. Wildcards is supported by using . as prefix. For example to allow com.foo and all subpackages then specfy com.foo.. Multiple permissions can be configured separated by comma, such as com.foo.,-com.foo.bar.MySecretBean. The following default permission is always included: -,java.lang.,java.util. unless its overridden by specifying a JVM system property with they key org.apache.camel.xstream.permissions.
allowUnmarshallType	**false**	**Boolean**	If enabled then Jackson is allowed to attempt to use the CamelJacksonUnmarshalType header during the unmarshalling. This should only be enabled when desired to be used.
timezone		**String**	If set then Jackson will use the Timezone when marshalling/unmarshalling. This option will have no effect on the others Json DataFormat, like gson, fastjson and xstream.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

179.2. USING THE JAVA DSL

```
// lets turn Object messages into XML then send to MQSeries
from("activemq:My.Queue").
  marshal().xstream().
  to("mqseries:Another.Queue");
```

If you would like to configure the **XStream** instance used by the Camel for the message transformation, you can simply pass a reference to that instance on the DSL level.

```
XStream xStream = new XStream();
xStream.aliasField("money", PurchaseOrder.class, "cash");
// new Added setModel option since Camel 2.14
xStream.setModel("NO_REFERENCES");
...

from("direct:marshal").
  marshal(new XStreamDataFormat(xStream)).
  to("mock:marshaled");
```

179.3. XMLINPUTFACTORY AND XMLOUTPUTFACTORY

The XStream library uses the **javax.xml.stream.XMLInputFactory** and **javax.xml.stream.XMLOutputFactory**, you can control which implementation of this factory should be used.

The Factory is discovered using this algorithm: 1. Use the **javax.xml.stream.XMLInputFactory** , **javax.xml.stream.XMLOutputFactory** system property. 2. Use the **lib/xml.stream.properties** file in the **JRE_HOME** directory. 3. Use the Services API, if available, to determine the classname by looking in the **META-INF/services/javax.xml.stream.XMLInputFactory**, **META-INF/services/javax.xml.stream.XMLOutputFactory** files in jars available to the JRE. 4. Use the platform default XMLInputFactory,XMLOutputFactory instance.

179.4. HOW TO SET THE XML ENCODING IN XSTREAM DATAFORMAT?

From Camel 2.2.0, you can set the encoding of XML in Xstream DataFormat by setting the Exchange's property with the key **Exchange.CHARSET_NAME**, or setting the encoding property on Xstream from DSL or Spring config.

```
from("activemq:My.Queue").
  marshal().xstream("UTF-8").
  to("mqseries:Another.Queue");
```

179.5. SETTING THE TYPE PERMISSIONS OF XSTREAM DATAFORMAT

In Camel, one can always use its own processing step in the route to filter and block certain XML documents to be routed to the XStream's unmarhall step. From Camel 2.16.1, 2.15.5, you can set XStream's type permissions to automatically allow or deny the instantiation of certain types.

The default type permissions setting used by Camel denies all types except for those from java.lang and java.util packages. This setting can be changed by setting System property org.apache.camel.xstream.permissions. Its value is a string of comma-separated permission terms, each representing a type being allowed or denied, depending on whether the term is prefixed with '' (note '' may be omitted) or with '-', respectively.

Each term may contain a wildcard character '**. For example, value "-**java.lang.**,java.util.**" indicates denying all types except for java.lang.* and java.util.* classes. Setting this value to an empty string "" reverts to the default XStream's type permissions handling which denies certain blacklisted classes and allow others.

The type permissions setting can be extended at an individual XStream DataFormat instance by setting its type permissions property.

```
<dataFormats>
   <xstream id="xstream-default"
            permissions="org.apache.camel.samples.xstream.*"/>
   ...
```

CHAPTER 180. JSONPATH LANGUAGE

Available as of Camel version 2.13

Camel supports JSonPath to allow using Expression or Predicate on json messages.

```
from("queue:books.new")
  .choice()
    .when().jsonpath("$.store.book[?(@.price < 10)]")
      .to("jms:queue:book.cheap")
    .when().jsonpath("$.store.book[?(@.price < 30)]")
      .to("jms:queue:book.average")
    .otherwise()
      .to("jms:queue:book.expensive")
```

180.1. JSONPATH OPTIONS

The JsonPath language supports 7 options which are listed below.

Name	Default	Java Type	Description
resultType		**String**	Sets the class name of the result type (type from output)
suppressExceptions	**false**	**Boolean**	Whether to suppress exceptions such as PathNotFoundException.
allowSimple	**true**	**Boolean**	Whether to allow in inlined simple exceptions in the JsonPath expression
allowEasyPredicate	**true**	**Boolean**	Whether to allow using the easy predicate parser to pre-parse predicates.
writeAsString	**false**	**Boolean**	Whether to write the output of each row/element as a JSON String value instead of a Map/POJO value.
headerName		**String**	Name of header to use as input, instead of the message body
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

180.2. USING XML CONFIGURATION

If you prefer to configure your routes in your Spring XML file then you can use JSonPath expressions as follows

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <choice>
```

```
      <when>
        <jsonpath>$.store.book[?(@.price &lt; 10)]</jsonpath>
        <to uri="mock:cheap"/>
      </when>
      <when>
        <jsonpath>$.store.book[?(@.price &lt; 30)]</jsonpath>
        <to uri="mock:average"/>
      </when>
      <otherwise>
        <to uri="mock:expensive"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

180.3. SYNTAX

See the JSonPath project page for further examples.

180.4. EASY SYNTAX

Available as of Camel 2.19

When you just want to define a basic predicate using jsonpath syntax it can be a bit hard to remember the syntax. So for example to find out all the cheap books you have to do

```
$.store.book[?(@.price < 20)]
```

However what if you could just write it as

```
store.book.price < 20
```

And you can omit the path if you just want to look at nodes with a price key

```
price < 20
```

To support this there is a **EasyPredicateParser** which kicks-in if you have define the predicate using a basic style. That means the predicate must not start with the **$** sign, and only include one operator.

The easy syntax is:

```
left OP right
```

You can use Camel simple language in the right operator, eg

```
store.book.price < ${header.limit}
```

180.5. SUPPORTED MESSAGE BODY TYPES

Camel JSonPath supports message body using the following types:

Type	Comment
File	Reading from files
String	Plain strings
Map	Message bodies as **java.util.Map** types
List	Message bodies as **java.util.List** types
POJO	**Optional** If Jackson is on the classpath, then camel-jsonpath is able to use Jackson to read the message body as POJO and convert to **java.util.Map** which is supported by JSonPath. For example you can add **camel-jackson** as dependency to include Jackson.
InputStream	If none of the above types matches, then Camel will attempt to read the message body as an **java.io.InputStream**.

If a message body is of unsupported type then an exception is thrown by default, however you can configure JSonPath to suppress exceptions (see below)

180.6. SUPPRESS EXCEPTIONS

Available as of Camel 2.16

By default jsonpath will throw an exception if the json payload does not have a valid path accordingly to the configured jsonpath expression. In some use-cases you may want to ignore this in case the json payload contains optional data. Therefore you can set the option suppressExceptions to true to ignore this as shown:

```
from("direct:start")
    .choice()
        // use true to suppress exceptions
        .when().jsonpath("person.middlename", true)
            .to("mock:middle")
        .otherwise()
            .to("mock:other");
```

And in XML DSL:

```
<route>
  <from uri="direct:start"/>
  <choice>
    <when>
      <jsonpath suppressExceptions="true">person.middlename</jsonpath>
      <to uri="mock:middle"/>
    </when>
    <otherwise>
      <to uri="mock:other"/>
```

```
    </otherwise>
  </choice>
</route>
```

This option is also available on the **@JsonPath** annotation.

180.7. INLINE SIMPLE EXCEPTIONS

Available as of Camel 2.18

Its now possible to inlined Simple language expressions in the JSonPath expression using the simple syntax ${xxx}. An example is shown below:

```
from("direct:start")
  .choice()
    .when().jsonpath("$.store.book[?(@.price < ${header.cheap})]")
      .to("mock:cheap")
    .when().jsonpath("$.store.book[?(@.price < ${header.average})]")
      .to("mock:average")
    .otherwise()
      .to("mock:expensive");
```

And in XML DSL:

```
<route>
  <from uri="direct:start"/>
  <choice>
    <when>
      <jsonpath>$.store.book[?(@.price < ${header.cheap})]</jsonpath>
      <to uri="mock:cheap"/>
    </when>
    <when>
      <jsonpath>$.store.book[?(@.price < ${header.average})]</jsonpath>
      <to uri="mock:average"/>
    </when>
    <otherwise>
      <to uri="mock:expensive"/>
    </otherwise>
  </choice>
</route>
```

You can turn off support for inlined simple expression by setting the option allowSimple to false as shown:

```
.when().jsonpath("$.store.book[?(@.price < 10)]", false, false)
```

And in XML DSL:

```
<jsonpath allowSimple="false">$.store.book[?(@.price < 10)]</jsonpath>
```

180.8. JSONPATH INJECTION

You can use Bean Integration to invoke a method on a bean and use various languages such as JSonPath to extract a value from the message and bind it to a method parameter.

For example

```
public class Foo {

    @Consume(uri = "activemq:queue:books.new")
    public void doSomething(@JsonPath("$.store.book[*].author") String author, @Body String json) {
      // process the inbound message here
    }
}
```

180.9. ENCODING DETECTION

Since Camel version 2.16, the encoding of the JSON document is detected automatically, if the document is encoded in unicode (UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE) as specified in RFC-4627. If the encoding is a non-unicode encoding, you can either make sure that you enter the document in String format to the JSONPath component or you can specify the encoding in the header "**CamelJsonPathJsonEncoding**" (JsonpathConstants.HEADER_JSON_ENCODING).

180.10. SPLIT JSON DATA INTO SUB ROWS AS JSON

You can use jsonpath to split a JSon document, such as:

```
from("direct:start")
    .split().jsonpath("$.store.book[*]")
    .to("log:book");
```

Then each book is logged, however the message body is a **Map** instance. Sometimes you may want to output this as plain String JSon value instead, which can be done from **Camel 2.20** onwards with the **writeAsString** option as shown:

```
from("direct:start")
    .split().jsonpathWriteAsString("$.store.book[*]")
    .to("log:book");
```

Then each book is logged as a String JSon value. For earlier versions of Camel you would need to use camel-jackson dataformat and marshal the message body to make it convert the message body from **Map** to a **String** type.

180.11. USING HEADER AS INPUT

Available as of Camel 2.20

By default jsonpath uses the message body as the input source. However you can also use a header as input by specifying the **headerName** option.

For example to count the number of books from a json document that was stored in a header named **books** you can do:

```
from("direct:start")
    .setHeader("numberOfBooks")
```

```
        .jsonpath("$..store.book.length()", false, int.class, "books")
    .to("mock:result");
```

In the **jsonpath** expression above we specify the name of the header as **books** and we also told that we wanted the result to be converted as an integer by **int.class**.

The same example in XML DSL would be:

```
<route>
  <from uri="direct:start"/>
  <setHeader headerName="numberOfBooks">
    <jsonpath headerName="books" resultType="int">$..store.book.length()</jsonpath>
  </transform>
  <to uri="mock:result"/>
</route>
```

180.12. DEPENDENCIES

To use JSonPath in your camel routes you need to add the a dependency on **camel-jsonpath** which implements the JSonPath language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jsonpath</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 181. JT400 COMPONENT

Available as of Camel version 1.5

The **jt400** component allows you to exchanges messages with an AS/400 system using data queues.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jt400</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

181.1. URI FORMAT

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/QUEUE.DTAQ[?options]
```

To call remote program (**Camel 2.7**)

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/program.PGM[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

181.2. JT400 OPTIONS

The JT400 component supports 2 options which are listed below.

Name	Description	Default	Type
connectionPool (advanced)	Returns the default connection pool used by this component.		AS400ConnectionPool
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The JT400 endpoint is configured using URI syntax:

```
jt400:userID:password/systemName/objectPath.type
```

with the following path and query parameters:

181.2.1. Path Parameters (5 parameters):

Name	Description	Default	Type
userID	**Required** Returns the ID of the AS/400 user.		String
password	**Required** Returns the password of the AS/400 user.		String
systemName	**Required** Returns the name of the AS/400 system.		String
objectPath	**Required** Returns the fully qualified integrated file system path name of the target object of this endpoint.		String
type	**Required** Whether to work with data queues or remote program call		Jt400Type

181.2.2. Query Parameters (30 parameters):

Name	Description	Default	Type
ccsid (common)	Sets the CCSID to use for the connection with the AS/400 system.		int
format (common)	Sets the data format for sending messages.	text	Format
guiAvailable (common)	Sets whether AS/400 prompting is enabled in the environment running Camel.	false	boolean
keyed (common)	Whether to use keyed or non-keyed data queues.	false	boolean
outputFieldsIdxArray (common)	Specifies which fields (program parameters) are output parameters.		Integer[]
outputFieldsLengthArray (common)	Specifies the fields (program parameters) length as in the AS/400 program definition.		Integer[]
searchKey (common)	Search key for keyed data queues.		String
searchType (common)	Search type such as EQ for equal etc.	EQ	SearchType

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
readTimeout (consumer)	Timeout in millis the consumer will wait while trying to read a new message of the data queue.	30000	int
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
procedureName (procedureName)	Procedure name from a service program to call		String

Name	Description	Default	Type
secured (security)	Whether connections to AS/400 are secured with SSL.	false	boolean

181.3. USAGE

When configured as a consumer endpoint, the endpoint will poll a data queue on a remote system. For every entry on the data queue, a new **Exchange** is sent with the entry's data in the *In* message's body, formatted either as a **String** or a **byte[]**, depending on the format. For a provider endpoint, the *In* message body contents will be put on the data queue as either raw bytes or text.

181.4. CONNECTION POOL

Available as of Camel 2.10

Connection pooling is in use from Camel 2.10 onwards. You can explicit configure a connection pool on the Jt400Component, or as an uri option on the endpoint.

181.4.1. Remote program call (Camel 2.7)

This endpoint expects the input to be either a String array or byte[] array (depending on format) and handles all the CCSID handling through the native jt400 library mechanisms. A parameter can be *omitted* by passing null as the value in its position (the remote program has to support it). After the program execution the endpoint returns either a String array or byte[] array with the values as they were returned by the program (the input only parameters will contain the same data as the beginning of the invocation). This endpoint does not implement a provider endpoint!

181.5. EXAMPLE

In the snippet below, the data for an exchange sent to the **direct:george** endpoint will be put in the data queue **PENNYLANE** in library **BEATLES** on a system named **LIVERPOOL**.
Another user connects to the same data queue to receive the information from the data queue and forward it to the **mock:ringo** endpoint.

```
public class Jt400RouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {

from("direct:george").to("jt400://GEORGE:EGROEG@LIVERPOOL/QSYS.LIB/BEATLES.LIB/PENNYLANE.DTAQ");

from("jt400://RINGO:OGNIR@LIVERPOOL/QSYS.LIB/BEATLES.LIB/PENNYLANE.DTAQ").to("mock:ringo");
    }
}
```

181.5.1. Remote program call example (Camel 2.7)

In the snippet below, the data Exchange sent to the direct:work endpoint will contain three string that

will be used as the arguments for the program “compute” in the library “assets”. This program will write the output values in the 2nd and 3rd parameters. All the parameters will be sent to the direct:play endpoint.

```
public class Jt400RouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
       from("direct:work").to("jt400://GRUPO:ATWORK@server/QSYS.LIB/assets.LIB/compute.PGM?
fieldsLength=10,10,512&ouputFieldsIdx=2,3").to(“direct:play”);
    }
}
```

181.5.2. Writing to keyed data queues

```
from("jms:queue:input")
.to("jt400://username:password@system/lib.lib/MSGINDQ.DTAQ?keyed=true");
```

181.5.3. Reading from keyed data queues

```
from("jt400://username:password@system/lib.lib/MSGOUTDQ.DTAQ?
keyed=true&searchKey=MYKEY&searchType=GE")
.to("jms:queue:output");
```

181.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 182. KAFKA COMPONENT

Available as of Camel version 2.13

The **kafka:** component is used for communicating with Apache Kafka message broker.

Maven users will need to add the following dependency to their **pom.xml** for this component.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-kafka</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

182.1. URI FORMAT

```
kafka:topic[?options]
```

182.2. OPTIONS

The Kafka component supports 8 options which are listed below.

Name	Description	Default	Type
configuration (common)	Allows to pre-configure the Kafka component with common options that the endpoints will reuse.		KafkaConfiguration
brokers (common)	URL of the Kafka brokers to use. The format is host1:port1,host2:port2, and the list can be a subset of brokers or a VIP pointing to a subset of brokers. This option is known as bootstrap.servers in the Kafka documentation.		String
workerPool (advanced)	To use a shared custom worker pool for continue routing Exchange after kafka server has acknowledge the message that was sent to it from KafkaProducer using asynchronous non-blocking processing. If using this option then you must handle the lifecycle of the thread pool to shut the pool down when no longer needed.		ExecutorService
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean

Name	Description	Default	Type
breakOnFirstError (consumer)	This options controls what happens when a consumer is processing an exchange and it fails. If the option is false then the consumer continues to the next message and processes it. If the option is true then the consumer breaks out, and will seek back to offset of the message that caused a failure, and then re-attempt to process this message. However this can lead to endless processing of the same message if its bound to fail every time, eg a poison message. Therefore its recommended to deal with that for example by using Camel's error handler.	false	boolean
allowManualCommit (consumer)	Whether to allow doing manual commits via KafkaManualCommit. If this option is enabled then an instance of KafkaManualCommit is stored on the Exchange message header, which allows end users to access this API and perform manual offset commits via the Kafka consumer.	false	boolean
kafkaManualCommit Factory (consumer)	Factory to use for creating KafkaManualCommit instances. This allows to plugin a custom factory to create custom KafkaManualCommit instances in case special logic is needed when doing manual commits that deviates from the default implementation that comes out of the box.		KafkaManualCommit Factory
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Kafka endpoint is configured using URI syntax:

```
kafka:topic
```

with the following path and query parameters:

182.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
topic	**Required** Name of the topic to use. On the consumer you can use comma to separate multiple topics. A producer can only send a message to a single topic.		String

182.2.2. Query Parameters (93 parameters):

Name	Description	Default	Type
brokers (common)	URL of the Kafka brokers to use. The format is host1:port1,host2:port2, and the list can be a subset of brokers or a VIP pointing to a subset of brokers. This option is known as bootstrap.servers in the Kafka documentation.		String
clientId (common)	The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request.		String
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
reconnectBackoffMaxMs (common)	The maximum amount of time in milliseconds to wait when reconnecting to a broker that has repeatedly failed to connect. If provided, the backoff per host will increase exponentially for each consecutive connection failure, up to this maximum. After calculating the backoff increase, 20% random jitter is added to avoid connection storms.	1000	Integer
allowManualCommit (consumer)	Whether to allow doing manual commits via KafkaManualCommit. If this option is enabled then an instance of KafkaManualCommit is stored on the Exchange message header, which allows end users to access this API and perform manual offset commits via the Kafka consumer.	false	boolean
autoCommitEnable (consumer)	If true, periodically commit to ZooKeeper the offset of messages already fetched by the consumer. This committed offset will be used when the process fails as the position from which the new consumer will begin.	true	Boolean
autoCommitIntervalMs (consumer)	The frequency in ms that the consumer offsets are committed to zookeeper.	5000	Integer
autoCommitOnStop (consumer)	Whether to perform an explicit auto commit when the consumer stops to ensure the broker has a commit from the last consumed message. This requires the option autoCommitEnable is turned on. The possible values are: sync, async, or none. And sync is the default value.	sync	String
autoOffsetReset (consumer)	What to do when there is no initial offset in ZooKeeper or if an offset is out of range: smallest : automatically reset the offset to the smallest offset largest : automatically reset the offset to the largest offset fail: throw exception to the consumer	latest	String

Name	Description	Default	Type
breakOnFirstError (consumer)	This options controls what happens when a consumer is processing an exchange and it fails. If the option is false then the consumer continues to the next message and processes it. If the option is true then the consumer breaks out, and will seek back to offset of the message that caused a failure, and then re-attempt to process this message. However this can lead to endless processing of the same message if its bound to fail every time, eg a poison message. Therefore its recommended to deal with that for example by using Camel's error handler.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
checkCrcs (consumer)	Automatically check the CRC32 of the records consumed. This ensures no on-the-wire or on-disk corruption to the messages occurred. This check adds some overhead, so it may be disabled in cases seeking extreme performance.	true	Boolean
consumerRequestTimeoutMs (consumer)	The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.	40000	Integer
consumersCount (consumer)	The number of consumers that connect to kafka server	1	int
consumerStreams (consumer)	Number of concurrent consumers on the consumer	10	int

Name	Description	Default	Type
fetchMaxBytes (consumer)	The maximum amount of data the server should return for a fetch request This is not an absolute maximum, if the first message in the first non-empty partition of the fetch is larger than this value, the message will still be returned to ensure that the consumer can make progress. The maximum message size accepted by the broker is defined via message.max.bytes (broker config) or max.message.bytes (topic config). Note that the consumer performs multiple fetches in parallel.	52428800	Integer
fetchMinBytes (consumer)	The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request.	1	Integer
fetchWaitMaxMs (consumer)	The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy fetch.min.bytes	500	Integer
groupId (consumer)	A string that uniquely identifies the group of consumer processes to which this consumer belongs. By setting the same group id multiple processes indicate that they are all part of the same consumer group. This option is required for consumers.		String
heartbeatIntervalMs (consumer)	The expected time between heartbeats to the consumer coordinator when using Kafka's group management facilities. Heartbeats are used to ensure that the consumer's session stays active and to facilitate rebalancing when new consumers join or leave the group. The value must be set lower than session.timeout.ms, but typically should be set no higher than 1/3 of that value. It can be adjusted even lower to control the expected time for normal rebalances.	3000	Integer
kafkaHeaderDeserializer (consumer)	Sets custom KafkaHeaderDeserializer for deserialization kafka headers values to camel headers values.		KafkaHeaderDeserializer

Name	Description	Default	Type
keyDeserializer (consumer)	Deserializer class for key that implements the Deserializer interface.	org.apache.kafka.common.serialization.StringDeserializer	String
maxPartitionFetchBytes (consumer)	The maximum amount of data per-partition the server will return. The maximum total memory used for a request will be partitions max.partition.fetch.bytes. This size must be at least as large as the maximum message size the server allows or else it is possible for the producer to send messages larger than the consumer can fetch. If that happens, the consumer can get stuck trying to fetch a large message on a certain partition.	1048576	Integer
maxPollIntervalMs (consumer)	The maximum delay between invocations of poll() when using consumer group management. This places an upper bound on the amount of time that the consumer can be idle before fetching more records. If poll() is not called before expiration of this timeout, then the consumer is considered failed and the group will rebalance in order to reassign the partitions to another member.		Long
maxPollRecords (consumer)	The maximum number of records returned in a single call to poll()	500	Integer
offsetRepository (consumer)	The offset repository to use in order to locally store the offset of each partition of the topic. Defining one will disable the autocommit.		String>
partitionAssignor (consumer)	The class name of the partition assignment strategy that the client will use to distribute partition ownership amongst consumer instances when group management is used	org.apache.kafka.clients.consumer.RangeAssignor	String
pollTimeoutMs (consumer)	The timeout used when polling the KafkaConsumer.	5000	Long

Name	Description	Default	Type
seekTo (consumer)	Set if KafkaConsumer will read from beginning or end on startup: beginning : read from beginning end : read from end This is replacing the earlier property seekToBeginning		String
sessionTimeoutMs (consumer)	The timeout used to detect failures when using Kafka's group management facilities.	10000	Integer
topicIsPattern (consumer)	Whether the topic is a pattern (regular expression). This can be used to subscribe to dynamic number of topics matching the pattern.	false	boolean
valueDeserializer (consumer)	Deserializer class for value that implements the Deserializer interface.	org.apache.kafka.common.serialization.StringDeserializer	String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
bridgeEndpoint (producer)	If the option is true, then KafkaProducer will ignore the KafkaConstants.TOPIC header setting of the inbound message.	false	boolean
bufferMemorySize (producer)	The total bytes of memory the producer can use to buffer records waiting to be sent to the server. If records are sent faster than they can be delivered to the server the producer will either block or throw an exception based on the preference specified by block.on.buffer.full.This setting should correspond roughly to the total memory the producer will use, but is not a hard bound since not all memory the producer uses is used for buffering. Some additional memory will be used for compression (if compression is enabled) as well as for maintaining in-flight requests.	33554432	Integer

Name	Description	Default	Type
circularTopicDetection (producer)	If the option is true, then KafkaProducer will detect if the message is attempted to be sent back to the same topic it may come from, if the message was original from a kafka consumer. If the KafkaConstants.TOPIC header is the same as the original kafka consumer topic, then the header setting is ignored, and the topic of the producer endpoint is used. In other words this avoids sending the same message back to where it came from. This option is not in use if the option bridgeEndpoint is set to true.	true	boolean
compressionCodec (producer)	This parameter allows you to specify the compression codec for all data generated by this producer. Valid values are none, gzip and snappy.	none	String
connectionMaxIdleMs (producer)	Close idle connections after the number of milliseconds specified by this config.	540000	Integer
enableIdempotence (producer)	If set to 'true' the producer will ensure that exactly one copy of each message is written in the stream. If 'false', producer retries may write duplicates of the retried message in the stream. If set to true this option will require max.in.flight.requests.per.connection to be set to 1 and retries cannot be zero and additionally acks must be set to 'all'.	false	boolean
kafkaHeaderSerializer (producer)	Sets custom KafkaHeaderDeserializer for serialization camel headers values to kafka headers values.		KafkaHeaderSerializer
key (producer)	The record key (or null if no key is specified). If this option has been configured then it take precedence over header link KafkaConstantsKEY		String
keySerializerClass (producer)	The serializer class for keys (defaults to the same as for messages if nothing is given).	org.apache.kafka.common.serialization.StringSerializer	String

Name	Description	Default	Type
lingerMs (producer)	The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under moderate load. This setting accomplishes this by adding a small amount of artificial delaythat is, rather than immediately sending out a record the producer will wait for up to the given delay to allow other records to be sent so that the sends can be batched together. This can be thought of as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on the delay for batching: once we get batch.size worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up. This setting defaults to 0 (i.e. no delay). Setting linger.ms=5, for example, would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absense of load.	0	Integer
maxBlockMs (producer)	The configuration controls how long sending to kafka will block. These methods can be blocked for multiple reasons. For e.g: buffer full, metadata unavailable.This configuration imposes maximum limit on the total time spent in fetching metadata, serialization of key and value, partitioning and allocation of buffer memory when doing a send(). In case of partitionsFor(), this configuration imposes a maximum time threshold on waiting for metadata	60000	Integer
maxInFlightRequest (producer)	The maximum number of unacknowledged requests the client will send on a single connection before blocking. Note that if this setting is set to be greater than 1 and there are failed sends, there is a risk of message re-ordering due to retries (i.e., if retries are enabled).	5	Integer
maxRequestSize (producer)	The maximum size of a request. This is also effectively a cap on the maximum record size. Note that the server has its own cap on record size which may be different from this. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests.	1048576	Integer

Name	Description	Default	Type
metadataMaxAgeMs (producer)	The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.	300000	Integer
metricReporters (producer)	A list of classes to use as metrics reporters. Implementing the MetricReporter interface allows plugging in classes that will be notified of new metric creation. The JmxReporter is always included to register JMX statistics.		String
metricsSampleWindowMs (producer)	The number of samples maintained to compute metrics.	30000	Integer
noOfMetricsSample (producer)	The number of samples maintained to compute metrics.	2	Integer
partitioner (producer)	The partitioner class for partitioning messages amongst sub-topics. The default partitioner is based on the hash of the key.	org.apache.kafka.clients.producer.internals.DefaultPartitioner	String
partitionKey (producer)	The partition to which the record will be sent (or null if no partition was specified). If this option has been configured then it take precedence over header link KafkaConstantsPARTITION_KEY		Integer
producerBatchSize (producer)	The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. This configuration controls the default batch size in bytes. No attempt will be made to batch records larger than this size.Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent.A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.	16384	Integer

Name	Description	Default	Type
queueBufferingMaxMessages (producer)	The maximum number of unsent messages that can be queued up the producer when using async mode before either the producer must be blocked or data must be dropped.	10000	Integer
receiveBufferBytes (producer)	The size of the TCP receive buffer (SO_RCVBUF) to use when reading data.	65536	Integer
reconnectBackoffMs (producer)	The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.	50	Integer
recordMetadata (producer)	Whether the producer should store the RecordMetadata results from sending to Kafka. The results are stored in a List containing the RecordMetadata metadata's. The list is stored on a header with the key link KafkaConstantsKAFKA_RECORDMETA	true	boolean
requestRequiredAcks (producer)	The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The following settings are common: acks=0 If set to zero then the producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the retries configuration will not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to -1. acks=1 This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost. acks=all This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.	1	String
requestTimeoutMs (producer)	The amount of time the broker will wait trying to meet the request.required.acks requirement before sending back an error to the client.	305000	Integer

Name	Description	Default	Type
retries (producer)	Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. Note that this retry is no different than if the client resent the record upon receiving the error. Allowing retries will potentially change the ordering of records because if two records are sent to a single partition, and the first fails and is retried but the second succeeds, then the second record may appear first.	0	Integer
retryBackoffMs (producer)	Before each retry, the producer refreshes the metadata of relevant topics to see if a new leader has been elected. Since leader election takes a bit of time, this property specifies the amount of time that the producer waits before refreshing the metadata.	100	Integer
sendBufferBytes (producer)	Socket write buffer size	131072	Integer
serializerClass (producer)	The serializer class for messages.	org.apache.kafka.common.serialization.StringSerializer	String
workerPool (producer)	To use a custom worker pool for continue routing Exchange after kafka server has acknowledge the message that was sent to it from KafkaProducer using asynchronous non-blocking processing.		ExecutorService
workerPoolCoreSize (producer)	Number of core threads for the worker pool for continue routing Exchange after kafka server has acknowledge the message that was sent to it from KafkaProducer using asynchronous non-blocking processing.	10	Integer
workerPoolMaxSize (producer)	Maximum number of threads for the worker pool for continue routing Exchange after kafka server has acknowledge the message that was sent to it from KafkaProducer using asynchronous non-blocking processing.	20	Integer

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
interceptorClasses (monitoring)	Sets interceptors for producer or consumers. Producer interceptors have to be classes implementing org.apache.kafka.clients.producer.ProducerInterceptor Consumer interceptors have to be classes implementing org.apache.kafka.clients.consumer.ConsumerInterceptor Note that if you use Producer interceptor on a consumer it will throw a class cast exception in runtime		String
kerberosBeforeReloginMin Time (security)	Login thread sleep time between refresh attempts.	60000	Integer
kerberosInitCmd (security)	Kerberos kinit command path. Default is /usr/bin/kinit	/usr/bin/kinit	String
kerberosPrincipalToLocal Rules (security)	A list of rules for mapping from principal names to short names (typically operating system usernames). The rules are evaluated in order and the first rule that matches a principal name is used to map it to a short name. Any later rules in the list are ignored. By default, principal names of the form username/hostnameREALM are mapped to username. For more details on the format please see security authorization and acls. Multiple values can be separated by comma	DEFAULT	String
kerberosRenewJitter (security)	Percentage of random jitter added to the renewal time.	0.05	Double
kerberosRenewWindowFactor (security)	Login thread will sleep until the specified window factor of time from last refresh to ticket's expiry has been reached, at which time it will try to renew the ticket.	0.8	Double
saslJaasConfig (security)	Expose the kafka sasl.jaas.config parameter Example: org.apache.kafka.common.security.plain.PlainLoginModule required username=USERNAME password=PASSWORD;		String

Name	Description	Default	Type
saslKerberosServiceName (security)	The Kerberos principal name that Kafka runs as. This can be defined either in Kafka's JAAS config or in Kafka's config.		String
saslMechanism (security)	The Simple Authentication and Security Layer (SASL) Mechanism used. For the valid values see http://www.iana.org/assignments/sasl-mechanisms/sasl-mechanisms.xhtml	GSSAPI	String
securityProtocol (security)	Protocol used to communicate with brokers. Currently only PLAINTEXT and SSL are supported.	PLAINTEXT	String
sslCipherSuites (security)	A list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol.By default all the available cipher suites are supported.		String
sslContextParameters (security)	SSL configuration using a Camel SSLContextParameters object. If configured it's applied before the other SSL endpoint parameters.		SSLContextParameters
sslEnabledProtocols (security)	The list of protocols enabled for SSL connections. TLSv1.2, TLSv1.1 and TLSv1 are enabled by default.	TLSv1.2,TLSv1.1,TLSv1	String
sslEndpointAlgorithm (security)	The endpoint identification algorithm to validate server hostname using server certificate.		String
sslKeymanagerAlgorithm (security)	The algorithm used by key manager factory for SSL connections. Default value is the key manager factory algorithm configured for the Java Virtual Machine.	SunX509	String
sslKeyPassword (security)	The password of the private key in the key store file. This is optional for client.		String
sslKeystoreLocation (security)	The location of the key store file. This is optional for client and can be used for two-way authentication for client.		String
sslKeystorePassword (security)	The store password for the key store file.This is optional for client and only needed if ssl.keystore.location is configured.		String

Name	Description	Default	Type
sslKeystoreType (security)	The file format of the key store file. This is optional for client. Default value is JKS	JKS	String
sslProtocol (security)	The SSL protocol used to generate the SSLContext. Default setting is TLS, which is fine for most cases. Allowed values in recent JVMs are TLS, TLSv1.1 and TLSv1.2. SSL, SSLv2 and SSLv3 may be supported in older JVMs, but their usage is discouraged due to known security vulnerabilities.	TLS	String
sslProvider (security)	The name of the security provider used for SSL connections. Default value is the default security provider of the JVM.		String
sslTrustmanagerAlgorithm (security)	The algorithm used by trust manager factory for SSL connections. Default value is the trust manager factory algorithm configured for the Java Virtual Machine.	PKIX	String
sslTruststoreLocation (security)	The location of the trust store file.		String
sslTruststorePassword (security)	The password for the trust store file.		String
sslTruststoreType (security)	The file format of the trust store file. Default value is JKS.	JKS	String

For more information about Producer/Consumer configuration:

http://kafka.apache.org/documentation.html#newconsumerconfigshttp://kafka.apache.org/documentatic

182.3. MESSAGE HEADERS

182.3.1. Consumer headers

The following headers are available when consuming messages from Kafka.

Header constant	Header value	Type	Description
KafkaConstants.TOPIC	**"kafka.TOPIC"**	**String**	The topic from where the message originated
KafkaConstants.PARTITION	**"kafka.PARTITION"**	**Integer**	The partition where the message was stored

Header constant	Header value	Type	Description
KafkaConstants.OFFSET	**"kafka.OFFSET"**	**Long**	The offset of the message
KafkaConstants.KEY	**"kafka.KEY"**	**Object**	The key of the message if configured
KafkaConstants.HEADERS	**"kafka.HEADERS"**	**org.apache.kafka.common.header.Headers**	The record headers
KafkaConstants.LAST_RECORD_BEFORE_COMMIT	**"kafka.LAST_RECORD_BEFORE_COMMIT"**	**Boolean**	Whether or not it's the last record before commit (only available if **autoCommitEnable** endpoint parameter is **false**)
KafkaConstants.MANUAL_COMMIT	**"CamelKafkaManualCommit"**	**KafkaManualCommit**	Can be used for forcing manual offset commit when using Kafka consumer.

182.3.2. Producer headers

Before sending a message to Kafka you can configure the following headers.

Header constant	Header value	Type	Description
KafkaConstants.KEY	**"kafka.KEY"**	**Object**	**Required** The key of the message in order to ensure that all related message goes in the same partition
KafkaConstants.TOPIC	**"kafka.TOPIC"**	**String**	The topic to which send the message (only read if the **bridgeEndpoint** endpoint parameter is **true**)
KafkaConstants.PARTITION_KEY	**"kafka.PARTITION_KEY"**	**Integer**	Explicitly specify the partition (only used if the **KafkaConstants.KEY** header is defined)

After the message is sent to Kafka, the following headers are available

Header constant	Header value	Type	Description
KafkaConstants.KAFKA_RECORDMETA	**"org.apache.kafka.clients.producer.RecordMetadata"**	**List<RecordMetadata>**	The metadata (only configured if **recordMetadata** endpoint parameter is **true**

182.4. SAMPLES

182.4.1. Consuming messages from Kafka

Here is the minimal route you need in order to read messages from Kafka.

```
from("kafka:test?brokers=localhost:9092")
    .log("Message received from Kafka : ${body}")
    .log("    on the topic ${headers[kafka.TOPIC]}")
    .log("    on the partition ${headers[kafka.PARTITION]}")
    .log("    with the offset ${headers[kafka.OFFSET]}")
    .log("    with the key ${headers[kafka.KEY]}")
```

When consuming messages from Kafka you can use your own offset management and not delegate this management to Kafka. In order to keep the offsets the component needs a **StateRepository** implementation such as **FileStateRepository**. This bean should be available in the registry. Here how to use it :

```
// Create the repository in which the Kafka offsets will be persisted
FileStateRepository repository = FileStateRepository.fileStateRepository(new
File("/path/to/repo.dat"));

// Bind this repository into the Camel registry
JndiRegistry registry = new JndiRegistry();
registry.bind("offsetRepo", repository);

// Configure the camel context
DefaultCamelContext camelContext = new DefaultCamelContext(registry);
camelContext.addRoutes(new RouteBuilder() {
   @Override
   public void configure() throws Exception {
      from("kafka:" + TOPIC + "?brokers=localhost:{{kafkaPort}}" +
                "&groupId=A" +                          //
                "&autoOffsetReset=earliest" +           // Ask to start from the beginning if we have
unknown offset
                "&offsetRepository=#offsetRepo")        // Keep the offsets in the previously configured
repository
            .to("mock:result");
   }
});
```

182.4.2. Producing messages to Kafka

Here is the minimal route you need in order to write messages to Kafka.

```
from("direct:start")
    .setBody(constant("Message from Camel"))          // Message to send
    .setHeader(KafkaConstants.KEY, constant("Camel")) // Key of the message
    .to("kafka:test?brokers=localhost:9092");
```

182.5. SSL CONFIGURATION

You have 2 different ways to configure the SSL communication on the Kafka` component.

The first way is through the many SSL endpoint parameters

```
from("kafka:" + TOPIC + "?brokers=localhost:{{kafkaPort}}" +
            "&groupId=A" +
            "&sslKeystoreLocation=/path/to/keystore.jks" +
            "&sslKeystorePassword=changeit" +
            "&sslKeyPassword=changeit")
        .to("mock:result");
```

The second way is to use the **sslContextParameters** endpoint parameter.

```
// Configure the SSLContextParameters object
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/path/to/keystore.jks");
ksp.setPassword("changeit");
KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("changeit");
SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

// Bind this SSLContextParameters into the Camel registry
JndiRegistry registry = new JndiRegistry();
registry.bind("ssl", scp);

// Configure the camel context
DefaultCamelContext camelContext = new DefaultCamelContext(registry);
camelContext.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("kafka:" + TOPIC + "?brokers=localhost:{{kafkaPort}}" +
                "&groupId=A" +                          //
                "&sslContextParameters=#ssl")           // Reference the SSL configuration
            .to("mock:result");
    }
});
```

182.6. USING THE KAFKA IDEMPOTENT REPOSITORY

Available from Camel 2.19

The **camel-kafka** library provides a Kafka topic-based idempotent repository. This repository stores broadcasts all changes to idempotent state (add/remove) in a Kafka topic, and populates a local in-memory cache for each repository's process instance through event sourcing.

The topic used must be unique per idempotent repository instance. The mechanism does not have any requirements about the number of topic partitions; as the repository consumes from all partitions at the same time. It also does not have any requirements about the replication factor of the topic.

Each repository instance that uses the topic (e.g. typically on different machines running in parallel) controls its own consumer group, so in a cluster of 10 Camel processes using the same topic each will control its own offset.

On startup, the instance subscribes to the topic and rewinds the offset to the beginning, rebuilding the cache to the latest state. The cache will not be considered warmed up until one poll of **pollDurationMs** in length returns 0 records. Startup will not be completed until either the cache has warmed up, or 30 seconds go by; if the latter happens the idempotent repository may be in an inconsistent state until its consumer catches up to the end of the topic.

A **KafkaIdempotentRepository** has the following properties:

Property	Description
topic	The name of the Kafka topic to use to broadcast changes. (required)
bootstrapServers	The **bootstrap.servers** property on the internal Kafka producer and consumer. Use this as shorthand if not setting **consumerConfig** and **producerConfig**. If used, this component will apply sensible default configurations for the producer and consumer.
producerConfig	Sets the properties that will be used by the Kafka producer that broadcasts changes. Overrides **bootstrapServers**, so must define the Kafka **bootstrap.servers** property itself
consumerConfig	Sets the properties that will be used by the Kafka consumer that populates the cache from the topic. Overrides **bootstrapServers**, so must define the Kafka **bootstrap.servers** property itself
maxCacheSize	How many of the most recently used keys should be stored in memory (default 1000).
pollDurationMs	The poll duration of the Kafka consumer. The local caches are updated immediately. This value will affect how far behind other peers that update their caches from the topic are relative to the idempotent consumer instance that sent the cache action message. The default value of this is 100 ms. If setting this value explicitly, be aware that there is a tradeoff between the remote cache liveness and the volume of network traffic between this repository's consumer and the Kafka brokers. The cache warmup process also depends on there being one poll that fetches nothing - this indicates that the stream has been consumed up to the current point. If the poll duration is excessively long for the rate at which messages are sent on the topic, there exists a possibility that the cache cannot be warmed up and will operate in an inconsistent state relative to its peers until it catches up.

The repository can be instantiated by defining the **topic** and **bootstrapServers**, or the **producerConfig** and **consumerConfig** property sets can be explicitly defined to enable features such as SSL/SASL.

To use, this repository must be placed in the Camel registry, either manually or by registration as a bean in Spring/Blueprint, as it is **CamelContext** aware.

Sample usage is as follows:

```
KafkaIdempotentRepository kafkaIdempotentRepository = new
KafkaIdempotentRepository("idempotent-db-inserts", "localhost:9091");

SimpleRegistry registry = new SimpleRegistry();
registry.put("insertDbIdemRepo", kafkaIdempotentRepository); // must be registered in the registry, to
enable access to the CamelContext
CamelContext context = new CamelContext(registry);

// later in RouteBuilder...
from("direct:performInsert")
    .idempotentConsumer(header("id")).messageIdRepositoryRef("insertDbIdemRepo")
       // once-only insert into database
    .end()
```

In XML:

```
<!-- simple -->
<bean id="insertDbIdemRepo"
class="org.apache.camel.processor.idempotent.kafka.KafkaIdempotentRepository">
  <property name="topic" value="idempotent-db-inserts"/>
  <property name="bootstrapServers" value="localhost:9091"/>
</bean>

<!-- complex -->
<bean id="insertDbIdemRepo"
class="org.apache.camel.processor.idempotent.kafka.KafkaIdempotentRepository">
  <property name="topic" value="idempotent-db-inserts"/>
  <property name="maxCacheSize" value="10000"/>
  <property name="consumerConfig">
    <props>
      <prop key="bootstrap.servers">localhost:9091</prop>
    </props>
  </property>
  <property name="producerConfig">
    <props>
      <prop key="bootstrap.servers">localhost:9091</prop>
    </props>
  </property>
</bean>
```

182.7. USING MANUAL COMMIT WITH KAFKA CONSUMER

Available as of Camel 2.21

By default the Kafka consumer will use auto commit, where the offset will be committed automaticcally in the background using a given interval.

In case you want to force manual commits, you can use **KafkaManualCommit** API from the Camel Exchange, stored on the message header. This requires to turn on manual commits by either setting the option **allowManualCommit** to **true** on the **KafkaComponent** or on the endpoint, for example:

```
KafkaComponent kafka = new KafkaComponent();
kafka.setAllowManualCommit(true);
...
camelContext.addComponent("kafka", kafka);
```

You can then use the **KafkaManualCommit** from Java code such as a Camel **Processor**:

```
public void process(Exchange exchange) {
    KafkaManualCommit manual = exchange.getIn().getHeader(KafkaConstants.MANUAL_COMMIT,
KafkaManualCommit.class);
    manual.commitSync();
}
```

This will force a synchronous commit which will block until the commit is acknowledge on Kafka, or if it fails an exception is thrown.

If you want to use a custom implementation of **KafkaManualCommit** then you can configure a custom **KafkaManualCommitFactory** on the **KafkaComponent** that creates instances of your custom implementation.

182.8. KAFKA HEADERS PROPAGATION

Available as of Camel 2.22

When consuming messages from Kafka, headers will be propagated to camel exchange headers automatically. Producing flow backed by same behaviour - camel headers of particular exchange will be propagated to kafka message headers.

Since kafka headers allows only **byte[]** values, in order camel exchnage header to be propagated its value should be serialized to **bytes[]**, otherwise header will be skipped. Following header value types are supported: **String**, **Integer**, **Long**, **Double**, **Boolean**, **byte[]**. Note: all headers propagated **from** kafka **to** camel exchange will contain **byte[]** value by default. In order to override default functionality uri parameters can be set: **kafkaHeaderDeserializer** for **from** route and **kafkaHeaderSerializer** for **to** route. Example:

```
from("kafka:my_topic?kafkaHeaderDeserializer=#myDeserializer")
...
.to("kafka:my_topic?kafkaHeaderSerializer=#mySerializer")
```

By default all headers are being filtered by **KafkaHeaderFilterStrategy**. Strategy filters out headers which start with **Camel** or **org.apache.camel** prefixes. Default strategy can be overridden by using **headerFilterStrategy** uri parameter in both **to** and **from** routes:

```
from("kafka:my_topic?headerFilterStrategy=#myStrategy")
...
.to("kafka:my_topic?headerFilterStrategy=#myStrategy")
```

myStrategy object should be subclass of **HeaderFilterStrategy** and must be placed in the Camel registry, either manually or by registration as a bean in Spring/Blueprint, as it is **CamelContext** aware.

CHAPTER 183. KESTREL COMPONENT (DEPRECATED)

Available as of Camel version 2.6

The Kestrel component allows messages to be sent to a Kestrel queue, or messages to be consumed from a Kestrel queue. This component uses the spymemcached client for memcached protocol communication with Kestrel servers.

WARNING

The kestrel project is inactive and the Camel team regard this components as deprecated.

183.1. URI FORMAT

```
kestrel://[addresslist/]queuename[?options]
```

Where **queuename** is the name of the queue on Kestrel. The **addresslist** part of the URI may include one or more **host:port** pairs. For example, to connect to the queue **foo** on **kserver01:22133**, use:

```
kestrel://kserver01:22133/foo
```

If the addresslist is omitted, **localhost:22133** is assumed, i.e.:

```
kestrel://foo
```

Likewise, if a port is omitted from a **host:port** pair in addresslist, the default port 22133 is assumed, i.e.:

```
kestrel://kserver01/foo
```

Here is an example of a Kestrel endpoint URI used for producing to a clustered queue:

```
kestrel://kserver01:22133,kserver02:22133,kserver03:22133/massive
```

Here is an example of a Kestrel endpoint URI used for consuming concurrently from a queue:

```
kestrel://kserver03:22133/massive?concurrentConsumers=25&waitTimeMs=500
```

183.2. OPTIONS

The Kestrel component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use a shared configured configuration as base for creating new endpoints.		KestrelConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Kestrel endpoint is configured using URI syntax:

```
kestrel:addresses/queue
```

with the following path and query parameters:

183.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
addresses	The address(es) on which kestrel is running	localhost:22133	String[]
queue	**Required** The queue we are polling		String

183.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
concurrentConsumers (common)	How many concurrent listeners to schedule for the thread pool	1	int
waitTimeMs (common)	How long a given wait should block (server side), in milliseconds	100	int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

183.3. CONFIGURING THE KESTREL COMPONENT USING SPRING XML

The simplest form of explicit configuration is as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <bean id="kestrel" class="org.apache.camel.component.kestrel.KestrelComponent"/>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  </camelContext>

</beans>
```

That will enable the Kestrel component with all default settings, i.e. it will use **localhost:22133**, 100ms wait time, and a single non-concurrent consumer by default.

To use specific options in the base configuration (which supplies configuration to endpoints whose **?** **properties** are not specified), you can set up a KestrelConfiguration POJO as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <bean id="kestrelConfiguration" class="org.apache.camel.component.kestrel.KestrelConfiguration">
    <property name="addresses" value="kestrel01:22133"/>
```

```
    <property name="waitTimeMs" value="100"/>
    <property name="concurrentConsumers" value="1"/>
  </bean>

  <bean id="kestrel" class="org.apache.camel.component.kestrel.KestrelComponent">
    <property name="configuration" ref="kestrelConfiguration"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  </camelContext>

</beans>
```

183.4. USAGE EXAMPLES

183.4.1. Example 1: Consuming

```
from("kestrel://kserver02:22133/massive?concurrentConsumers=10&waitTimeMs=500")
  .bean("myConsumer", "onMessage");
```

```
public class MyConsumer {
    public void onMessage(String message) {
        ...
    }
}
```

183.4.2. Example 2: Producing

```
public class MyProducer {
    @EndpointInject(uri = "kestrel://kserver01:22133,kserver02:22133/myqueue")
    ProducerTemplate producerTemplate;

    public void produceSomething() {
        producerTemplate.sendBody("Hello, world.");
    }
}
```

183.4.3. Example 3: Spring XML Configuration

```
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="kestrel://ks01:22133/sequential?concurrentConsumers=1&waitTimeMs=500"/>
      <bean ref="myBean" method="onMessage"/>
    </route>
    <route>
      <from uri="direct:start"/>
      <to uri="kestrel://ks02:22133/stuff"/>
    </route>
  </camelContext>
```

```
public class MyBean {
```

```
    public void onMessage(String message) {
        ...
    }
}
```

183.5. DEPENDENCIES

The Kestrel component has the following dependencies:

- **spymemcached** 2.5 (or greater)

183.5.1. spymemcached

You **must** have the **spymemcached** jar on your classpath. Here is a snippet you can use in your pom.xml:

```
<dependency>
  <groupId>spy</groupId>
  <artifactId>memcached</artifactId>
  <version>2.5</version>
</dependency>
```

Alternatively, you can download the jar directly.

Warning: Limitations

NOTE

The spymemcached client library does **not** work properly with kestrel when JVM assertions are enabled. There is a known issue with spymemcached when assertions are enabled and a requested key contains the **/t=...** extension (i.e. if you're using the **waitTimeMs** option on an endpoint URI, which is highly encouraged). Fortunately, JVM assertions are **disabled by default**, unless you explicitly enable them, so this should not present a problem under normal circumstances. Something to note is that Maven's Surefire test plugin **enables** assertions. If you're using this component in a Maven test environment, you may need to set **enableAssertions** to **false**. Please refer to the surefire:test reference for details.

183.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 184. KIE-CAMEL

184.1. OVERVIEW

Kie-Camel is an Apache Camel component (endpoint) that integrates with KIE (Drools). It allows you to specify a **KIE** module (using maven GAV) which you can pull into the route and execute. Also, It enables you to specify portions of the message body as facts.

For more details about the kie-camel component, see Apache Camel Integration.

CHAPTER 185. KRATI COMPONENT (DEPRECATED)

Available as of Camel version 2.9

This component allows the use krati datastores and datasets inside Camel. Krati is a simple persistent data store with very low latency and high throughput. It is designed for easy integration with read-write-intensive applications with little effort in tuning configuration, performance and JVM garbage collection.

Camel provides a producer and consumer for krati datastore_(key/value engine)_. It also provides an idempotent repository for filtering out duplicate messages.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-krati</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

185.1. URI FORMAT

```
krati:[the path of the datastore][?options]
```

The **path of the datastore** is the relative path of the folder that krati will use for its datastore.

You can append query options to the URI in the following format, **?option=value&option=value&…**

185.2. KRATI OPTIONS

The Krati component has no options.

The Krati endpoint is configured using URI syntax:

```
krati:path
```

with the following path and query parameters:

185.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
path	**Required** Path of the datastore is the relative path of the folder that krati will use for its datastore.		String

185.2.2. Query Parameters (29 parameters):

Name	Description	Default	Type
hashFunction (common)	The hash function to use.		HashFunction<byte[]>
initialCapacity (common)	The inital capcity of the store.	100	int
keySerializer (common)	The serializer that will be used to serialize the key.		Object>
segmentFactory (common)	Sets the segment factory of the target store.		SegmentFactory
segmentFileSize (common)	Data store segments size in MB.	64	int
valueSerializer (common)	The serializer that will be used to serialize the value.		Object>
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
maxMessagesPerPoll (consumer)	The maximum number of messages which can be received in one poll. This can be used to avoid reading in too much data and taking up too much memory.		int
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
key (producer)	The key.		String
operation (producer)	Specifies the type of operation that will be performed to the datastore.		String
value (producer)	The Value.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long

Name	Description	Default	Type
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

```
krati:/tmp/krati?operation=CamelKratiGet&initialCapacity=10000&keySerializer=#myCustomSerializer
```

For producer endpoint you can override all of the above URI options by passing the appropriate headers to the message.

185.2.3. Message Headers for datastore

Header	Description
CamelKratiOperation	The operation to be performed on the datastore. The valid options are CamelKratiAdd, CamelKratiGet, CamelKratiDelete, CamelKratiDeleteAll
CamelKratiKey	The key.

Header	Description
CamelKratiValue	The value.

185.3. USAGE SAMPLES

185.3.1. Example 1: Putting to the datastore.

This example will show you how you can store any message inside a datastore.

```
from("direct:put").to("krati:target/test/producertest");
```

In the above example you can override any of the URI parameters with headers on the message.
Here is how the above example would look like using xml to define our route.

```
<route>
    <from uri="direct:put"/>
    <to uri="krati:target/test/producerspringtest"/>
</route>
```

185.3.2. Example 2: Getting/Reading from a datastore

This example will show you how you can read the contnet of a datastore.

```
from("direct:get")
    .setHeader(KratiConstants.KRATI_OPERATION,
constant(KratiConstants.KRATI_OPERATION_GET))
    .to("krati:target/test/producertest");
```

In the above example you can override any of the URI parameters with headers on the message.
Here is how the above example would look like using xml to define our route.

```
<route>
    <from uri="direct:get"/>
    <to uri="krati:target/test/producerspringtest?operation=CamelKratiGet"/>
</route>
```

185.3.3. Example 3: Consuming from a datastore

This example will consume all items that are under the specified datastore.

```
from("krati:target/test/consumertest")
    .to("direct:next");
```

You can achieve the same goal by using xml, as you can see below.

```
<route>
    <from uri="krati:target/test/consumerspringtest"/>
    <to uri="mock:results"/>
</route>
```

185.4. IDEMPOTENT REPOSITORY

As already mentioned this component also offers and idemptonet repository which can be used for filtering out duplicate messages.

```
from("direct://in").idempotentConsumer(header("messageId"), new
KratiIdempotentRepositroy("/tmp/idempotent").to("log://out");
```

185.4.1. See also

Krati Website

CHAPTER 186. KUBERNETES COMPONENTS

Available as of Camel version 2.17

The **Kubernetes** components integrate your application with Kubernetes standalone or on top of Openshift.

The camel-kubernetes consists of 13 components:

- Kubernetes ConfigMap
- Kubernetes Namespace
- Kubernetes Node
- Kubernetes Persistent Volume
- Kubernetes Persistent Volume Claim
- Kubernetes Pod
- Kubernetes Replication Controller
- Kubernetes Resource Quota
- Kubernetes Secrets
- Kubernetes Service Account
- Kubernetes Service

In OpenShift, also:

- Kubernetes Build Config
- Kubernetes Build

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-kubernetes</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

186.1. HEADERS

Name	Type	Description
CamelKubernetesOperation	String	The Producer operation

Name	Type	Description
CamelKubernetesNamespaceName	String	The Namespace name
CamelKubernetesNamespaceLabels	Map	The Namespace Labels
CamelKubernetesServiceLabels	Map	The Service labels
CamelKubernetesServiceName	String	The Service name
CamelKubernetesServiceSpec	io.fabric8.kubernetes.api.model.ServiceSpec	The Spec for a Service
CamelKubernetesReplicationControllersLabels	Map	Replication controller labels
CamelKubernetesReplicationControllerName	String	Replication controller name

Name	Type	Description
CamelKubernetesReplicationControllerSpec	io.fabric8.kubernetes.api.model.ReplicationControllerSpec	The Spec for a Replication Controller
CamelKubernetesReplicationControllerReplicas	Integer	The number of replicas for a Replication Controller during the Scale operation
CamelKubernetesPodsLabels	Map	Pod labels
CamelKubernetesPodName	String	Pod name
CamelKubernetesPodSpec	io.fabric8.kubernetes.api.model.PodSpec	The Spec for a Pod
CamelKubernetesPersistentVolumesLabels	Map	Persistent Volume labels
CamelKubernetesPersistentVolumesName	String	Persistent Volume name

Name	Type	Description
CamelKubernetesPersistentVolumesClaimsLabels	Map	Persistent Volume Claim labels
CamelKubernetesPersistentVolumesClaimsName	String	Persistent Volume Claim name
CamelKubernetesPersistentVolumesClaimsSpec	io.fabric8.kubernetes.api.model.PersistentVolumeClaimSpec	The Spec for a Persistent Volume claim
CamelKubernetesSecretsLabels	Map	Secret labels
CamelKubernetesSecretsName	String	Secret name
CamelKubernetesSecret	io.fabric8.kubernetes.api.model.Secret	A Secret Object

Name	Type	Description
CamelKubernetesResourcesQuotaLabels	Map	Resource Quota labels
CamelKubernetesResourcesQuotaName	String	Resource Quota name
CamelKubernetesResourceQuotaSpec	io.fabric8.kubernetes.api.model.ResourceQuotaSpec	The Spec for a Resource Quota
CamelKubernetesServiceAccountsLabels	Map	Service Account labels
CamelKubernetesServiceAccountName	String	Service Account name
CamelKubernetesServiceAccount	io.fabric8.kubernetes.api.model.ServiceAccount	A Service Account object
CamelKubernetesNodesLabels	Map	Node labels

Name	Type	Description
CamelKubernetesNodeName	String	Node name
CamelKubernetesBuildsLabels	Map	Openshift Build labels
CamelKubernetesBuildName	String	Openshift Build name
CamelKubernetesBuildConfigsLabels	Map	Openshift Build Config labels
CamelKubernetesBuildConfigName	String	Openshift Build Config name
CamelKubernetesEventAction	io.fabric8.kubernetes.client.Watcher.Action	Action watched by the consumer
CamelKubernetesEventTimestamp	String	Timestamp of the action watched by the consumer
CamelKubernetesConfigMapName	String	ConfigMap name

Name	Type	Description
CamelKubernetesConfigMapsLabels	Map	ConfigMap labels
CamelKubernetesConfigData	Map	ConfigMap Data

186.2. USAGE

186.2.1. Producer examples

Here we show some examples of producer using camel-kubernetes.

186.2.2. Create a pod

```
from("direct:createPod")
    .toF("kubernetes-pods://%s?oauthToken=%s&operation=createPod", host, authToken);
```

By using the KubernetesConstants.KUBERNETES_POD_SPEC header you can specify your PodSpec and pass it to this operation.

186.2.3. Delete a pod

```
from("direct:createPod")
    .toF("kubernetes-pods://%s?oauthToken=%s&operation=deletePod", host, authToken);
```

By using the KubernetesConstants.KUBERNETES_POD_NAME header you can specify your Pod name and pass it to this operation.

CHAPTER 187. KUBERNETES COMPONENT (DEPRECATED)

Available as of Camel version 2.17

IMPORTANT

The composite kubernetes component has been deprecated. Use individual component splitted as following.

- Kubernetes Components
  - Kubernetes Build Config
  - Kubernetes Build
  - Kubernetes ConfigMap
  - Kubernetes Namespace
  - Kubernetes Node
  - Kubernetes Persistent Volume
  - Kubernetes Persistent Volume Claim
  - Kubernetes Pod
  - Kubernetes Replication Controller
  - Kubernetes Resource Quota
  - Kubernetes Secrets
  - Kubernetes Service Account
  - Kubernetes Service

The **Kubernetes** component is a component for integrating your application with Kubernetes standalone or on top of Openshift.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-kubernetes</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

187.1. URI FORMAT

```
kubernetes:masterUrl[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&...**

187.2. OPTIONS

The Kubernetes component has no options.

The Kubernetes endpoint is configured using URI syntax:

```
kubernetes:masterUrl
```

with the following path and query parameters:

187.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

187.2.2. Query Parameters (28 parameters):

Name	Description	Default	Type
apiVersion (common)	The Kubernetes API Version to use		String
category (common)	**Required** Kubernetes Producer and Consumer category		String
dnsDomain (common)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (common)	Default KubernetesClient to use if provided		KubernetesClient
portName (common)	The port name, used for ServiceCall EIP		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
labelKey (consumer)	The Consumer Label key when watching at some resources		String

Name	Description	Default	Type
labelValue (consumer)	The Consumer Label value when watching at some resources		String
namespace (consumer)	The namespace		String
poolSize (consumer)	The Consumer pool size	1	int
resourceName (consumer)	The Consumer Resource Name we would like to watch		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Producer operation to do on Kubernetes		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String

Name	Description	Default	Type
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

187.3. HEADERS

Name	Type	Description
CamelKubernetesOperation	String	The Producer operation
CamelKubernetesNamespaceName	String	The Namespace name
CamelKubernetesNamespaceLabels	Map	The Namespace Labels

Name	Type	Description
CamelKubernetesServiceLabels	Map	The Service labels
CamelKubernetesServiceName	String	The Service name
CamelKubernetesServiceSpec	io.fabric8.kubernetes.api.model.ServiceSpec	The Spec for a Service
CamelKubernetesReplicationControllersLabels	Map	Replication controller labels
CamelKubernetesReplicationControllerName	String	Replication controller name
CamelKubernetesReplicationControllerSpec	io.fabric8.kubernetes.api.model.ReplicationControllerSpec	The Spec for a Replication Controller

Name	Type	Description
CamelKubernetesReplicationControllerReplicas	Integer	The number of replicas for a Replication Controller during the Scale operation
CamelKubernetesPodsLabels	Map	Pod labels
CamelKubernetesPodName	String	Pod name
CamelKubernetesPodSpec	io.fabric8.kubernetes.api.model.PodSpec	The Spec for a Pod
CamelKubernetesPersistentVolumesLabels	Map	Persistent Volume labels
CamelKubernetesPersistentVolumesName	String	Persistent Volume name
CamelKubernetesPersistentVolumesClaimsLabels	Map	Persistent Volume Claim labels

Name	Type	Description
CamelKubernetesPersistentVolumesClaimsName	String	Persistent Volume Claim name
CamelKubernetesPersistentVolumesClaimsSpec	io.fabric8.kubernetes.api.model.PersistentVolumeClaimSpec	The Spec for a Persistent Volume claim
CamelKubernetesSecretsLabels	Map	Secret labels
CamelKubernetesSecretsName	String	Secret name
CamelKubernetesSecret	io.fabric8.kubernetes.api.model.Secret	A Secret Object
CamelKubernetesResourcesQuotaLabels	Map	Resource Quota labels

Name	Type	Description
CamelKubernetesResourcesQuotaName	String	Resource Quota name
CamelKubernetesResourceQuotaSpec	io.fabric8.kubernetes.api.model.ResourceQuotaSpec	The Spec for a Resource Quota
CamelKubernetesServiceAccountsLabels	Map	Service Account labels
CamelKubernetesServiceAccountName	String	Service Account name
CamelKubernetesServiceAccount	io.fabric8.kubernetes.api.model.ServiceAccount	A Service Account object
CamelKubernetesNodesLabels	Map	Node labels
CamelKubernetesNodeName	String	Node name

Name	Type	Description
CamelKubernetesBuildsLabels	Map	Openshift Build labels
CamelKubernetesBuildName	String	Openshift Build name
CamelKubernetesBuildConfigsLabels	Map	Openshift Build Config labels
CamelKubernetesBuildConfigName	String	Openshift Build Config name
CamelKubernetesEventAction	io.fabric8.kubernetes.client.Watcher.Action	Action watched by the consumer
CamelKubernetesEventTimestamp	String	Timestamp of the action watched by the consumer
CamelKubernetesConfigMapName	String	ConfigMap name

Name	Type	Description
CamelKubernetesConfigMapsLabels	Map	ConfigMap labels
CamelKubernetesConfigData	Map	ConfigMap Data

187.4. CATEGORIES

Actually the camel-kubernetes component supports the following Kubernetes resources

- Namespaces
- Pods
- Replication Controllers
- Services
- Persistent Volumes
- Persistent Volume Claims
- Secrets
- Resource Quota
- Service Accounts
- Nodes
- Configmaps

In Openshift also

- Builds
- BuildConfigs

187.5. USAGE

187.5.1. Producer examples

Here we show some examples of producer using camel-kubernetes.

187.5.2. Create a pod

```
from("direct:createPod")
    .toF("kubernetes://%s?oauthToken=%s&category=pods&operation=createPod", host, authToken);
```

By using the KubernetesConstants.KUBERNETES_POD_SPEC header you can specify your PodSpec and pass it to this operation.

187.5.3. Delete a pod

```
from("direct:createPod")
    .toF("kubernetes://%s?oauthToken=%s&category=pods&operation=deletePod", host, authToken);
```

By using the KubernetesConstants.KUBERNETES_POD_NAME header you can specify your Pod name and pass it to this operation.

CHAPTER 188. KUBERNETES CONFIGMAP COMPONENT

Available as of Camel version 2.17

The **Kubernetes ConfigMap** component is one of Kubernetes Components which provides a producer to execute kubernetes ConfigMap operations.

188.1. COMPONENT OPTIONS

The Kubernetes ConfigMap component has no options.

188.2. ENDPOINT OPTIONS

The Kubernetes ConfigMap endpoint is configured using URI syntax:

```
kubernetes-config-maps:masterUrl
```

with the following path and query parameters:

188.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

188.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 189. KUBERNETES DEPLOYMENTS COMPONENT

Available as of Camel version 2.20

The **Kubernetes Deployments** component is one of Kubernetes Components which provides a producer to execute kubernetes secret operations.

189.1. COMPONENT OPTIONS

The Kubernetes Deployments component has no options.

189.2. ENDPOINT OPTIONS

The Kubernetes Deployments endpoint is configured using URI syntax:

```
kubernetes-deployments:masterUrl
```

with the following path and query parameters:

189.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

189.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
apiVersion (common)	The Kubernetes API Version to use		String
dnsDomain (common)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (common)	Default KubernetesClient to use if provided		KubernetesClient
portName (common)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
labelKey (consumer)	The Consumer Label key when watching at some resources		String
labelValue (consumer)	The Consumer Label value when watching at some resources		String
namespace (consumer)	The namespace		String
poolSize (consumer)	The Consumer pool size	1	int
resourceName (consumer)	The Consumer Resource Name we would like to watch		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Producer operation to do on Kubernetes		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String

Name	Description	Default	Type
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 190. KUBERNETES NAMESPACES COMPONENT

Available as of Camel version 2.17

The **Kubernetes Namespaces** component is one of Kubernetes Components which provides a producer to execute kubernetes namespace operations and a consumer to consume kubernetes namespace events.

190.1. COMPONENT OPTIONS

The Kubernetes Namespaces component has no options.

190.2. ENDPOINT OPTIONS

The Kubernetes Namespaces endpoint is configured using URI syntax:

```
kubernetes-namespaces:masterUrl
```

with the following path and query parameters:

190.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

190.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
apiVersion (common)	The Kubernetes API Version to use		String
dnsDomain (common)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (common)	Default KubernetesClient to use if provided		KubernetesClient
portName (common)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
labelKey (consumer)	The Consumer Label key when watching at some resources		String
labelValue (consumer)	The Consumer Label value when watching at some resources		String
namespace (consumer)	The namespace		String
poolSize (consumer)	The Consumer pool size	1	int
resourceName (consumer)	The Consumer Resource Name we would like to watch		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Producer operation to do on Kubernetes		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String

Name	Description	Default	Type
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 191. KUBERNETES NODES COMPONENT

Available as of Camel version 2.17

The **Kubernetes Nodes** component is one of Kubernetes Components which provides a producer to execute kubernetes node operations and a consumer to consume kubernetes node events.

191.1. COMPONENT OPTIONS

The Kubernetes Nodes component has no options.

191.2. ENDPOINT OPTIONS

The Kubernetes Nodes endpoint is configured using URI syntax:

```
kubernetes-nodes:masterUrl
```

with the following path and query parameters:

191.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

191.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
apiVersion (common)	The Kubernetes API Version to use		String
dnsDomain (common)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (common)	Default KubernetesClient to use if provided		KubernetesClient
portName (common)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
labelKey (consumer)	The Consumer Label key when watching at some resources		String
labelValue (consumer)	The Consumer Label value when watching at some resources		String
namespace (consumer)	The namespace		String
poolSize (consumer)	The Consumer pool size	1	int
resourceName (consumer)	The Consumer Resource Name we would like to watch		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Producer operation to do on Kubernetes		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String

Name	Description	Default	Type
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 192. KUBERNETES PERSISTENT VOLUME CLAIM COMPONENT

Available as of Camel version 2.17

The **Kubernetes Persistent Volume Claim** component is one of Kubernetes Components which provides a producer to execute kubernetes persistent volume claim operations.

192.1. COMPONENT OPTIONS

The Kubernetes Persistent Volume Claim component has no options.

192.2. ENDPOINT OPTIONS

The Kubernetes Persistent Volume Claim endpoint is configured using URI syntax:

```
kubernetes-persistent-volumes-claims:masterUrl
```

with the following path and query parameters:

192.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

192.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 193. KUBERNETES PERSISTENT VOLUME COMPONENT

Available as of Camel version 2.17

The **Kubernetes Persistent Volume** component is one of Kubernetes Components which provides a producer to execute kubernetes persistent volume operations.

193.1. COMPONENT OPTIONS

The Kubernetes Persistent Volume component has no options.

193.2. ENDPOINT OPTIONS

The Kubernetes Persistent Volume endpoint is configured using URI syntax:

```
kubernetes-persistent-volumes:masterUrl
```

with the following path and query parameters:

193.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

193.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 194. KUBERNETES PODS COMPONENT

Available as of Camel version 2.17

The **Kubernetes Pods** component is one of Kubernetes Components which provides a producer to execute kubernetes pod operations.

194.1. COMPONENT OPTIONS

The Kubernetes Pods component has no options.

194.2. ENDPOINT OPTIONS

The Kubernetes Pods endpoint is configured using URI syntax:

```
kubernetes-pods:masterUrl
```

with the following path and query parameters:

194.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

194.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
apiVersion (common)	The Kubernetes API Version to use		String
dnsDomain (common)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (common)	Default KubernetesClient to use if provided		KubernetesClient
portName (common)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
labelKey (consumer)	The Consumer Label key when watching at some resources		String
labelValue (consumer)	The Consumer Label value when watching at some resources		String
namespace (consumer)	The namespace		String
poolSize (consumer)	The Consumer pool size	1	int
resourceName (consumer)	The Consumer Resource Name we would like to watch		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Producer operation to do on Kubernetes		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String

Name	Description	Default	Type
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 195. KUBERNETES REPLICATION CONTROLLER COMPONENT

Available as of Camel version 2.17

The **Kubernetes Replication Controller** component is one of Kubernetes Components which provides a producer to execute kubernetes replication controller operations and a consumer to consume kubernetes replication controller events.

195.1. COMPONENT OPTIONS

The Kubernetes Replication Controller component has no options.

195.2. ENDPOINT OPTIONS

The Kubernetes Replication Controller endpoint is configured using URI syntax:

```
kubernetes-replication-controllers:masterUrl
```

with the following path and query parameters:

195.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

195.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
apiVersion (common)	The Kubernetes API Version to use		String
dnsDomain (common)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (common)	Default KubernetesClient to use if provided		KubernetesClient
portName (common)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
labelKey (consumer)	The Consumer Label key when watching at some resources		String
labelValue (consumer)	The Consumer Label value when watching at some resources		String
namespace (consumer)	The namespace		String
poolSize (consumer)	The Consumer pool size	1	int
resourceName (consumer)	The Consumer Resource Name we would like to watch		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Producer operation to do on Kubernetes		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String

Name	Description	Default	Type
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 196. KUBERNETES RESOURCES QUOTA COMPONENT

Available as of Camel version 2.17

The **Kubernetes Resources Quota** component is one of Kubernetes Components which provides a producer to execute kubernetes resource quota operations.

196.1. COMPONENT OPTIONS

The Kubernetes Resources Quota component has no options.

196.2. ENDPOINT OPTIONS

The Kubernetes Resources Quota endpoint is configured using URI syntax:

```
kubernetes-resources-quota:masterUrl
```

with the following path and query parameters:

196.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

196.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 197. KUBERNETES SECRETS COMPONENT

Available as of Camel version 2.17

The **Kubernetes Secrets** component is one of Kubernetes Components which provides a producer to execute kubernetes secret operations.

197.1. COMPONENT OPTIONS

The Kubernetes Secrets component has no options.

197.2. ENDPOINT OPTIONS

The Kubernetes Secrets endpoint is configured using URI syntax:

```
kubernetes-secrets:masterUrl
```

with the following path and query parameters:

197.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

197.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 198. KUBERNETES SERVICE ACCOUNT COMPONENT

Available as of Camel version 2.17

The **Kubernetes Service Account** component is one of Kubernetes Components which provides a producer to execute kubernetes Service Account operations.

198.1. COMPONENT OPTIONS

The Kubernetes Service Account component has no options.

198.2. ENDPOINT OPTIONS

The Kubernetes Service Account endpoint is configured using URI syntax:

```
kubernetes-service-accounts:masterUrl
```

with the following path and query parameters:

198.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

198.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 199. KUBERNETES SERVICES COMPONENT

Available as of Camel version 2.17

The **Kubernetes Services** component is one of Kubernetes Components which provides a producer to execute kubernetes service operations and a consumer to consume kubernetes service events.

199.1. COMPONENT OPTIONS

The Kubernetes Services component has no options.

199.2. ENDPOINT OPTIONS

The Kubernetes Services endpoint is configured using URI syntax:

```
kubernetes-services:masterUrl
```

with the following path and query parameters:

199.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

199.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
apiVersion (common)	The Kubernetes API Version to use		String
dnsDomain (common)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (common)	Default KubernetesClient to use if provided		KubernetesClient
portName (common)	The port name, used for ServiceCall EIP		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
labelKey (consumer)	The Consumer Label key when watching at some resources		String
labelValue (consumer)	The Consumer Label value when watching at some resources		String
namespace (consumer)	The namespace		String
poolSize (consumer)	The Consumer pool size	1	int
resourceName (consumer)	The Consumer Resource Name we would like to watch		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Producer operation to do on Kubernetes		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String

Name	Description	Default	Type
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

199.3. ECLIPSE KURA COMPONENT

Available as of Camel 2.15

This documentation page covers the integration options of Camel with the Eclipse Kura M2M gateway. The common reason to deploy Camel routes into the Eclipse Kura is to provide enterprise integration patterns and Camel components to the messaging M2M gateway. For example you might want to install Kura on Raspberry PI, then read temperature from the sensor attached to that Raspberry PI using Kura services and finally forward the current temperature value to your data center service using Camel EIP and components.

199.3.1. KuraRouter activator

Bundles deployed to the Eclipse Kura are usually developed as bundle activators. So the easiest way to deploy Apache Camel routes into the Kura is to create an OSGi bundle containing the class extending **org.apache.camel.kura.KuraRouter** class:

```
public class MyKuraRouter extends KuraRouter {

  @Override
  public void configure() throws Exception {
    from("timer:trigger").
      to("netty-http:http://app.mydatacenter.com/api");
  }

}
```

Keep in mind that **KuraRouter** implements the **org.osgi.framework.BundleActivator** interface, so you need to register its **start** and **stop** lifecycle methods while creating Kura bundle component class.

Kura router starts its own OSGi-aware **CamelContext**. It means that for every class extending **KuraRouter**, there will be a dedicated **CamelContext** instance. Ideally we recommend to deploy one **KuraRouter** per OSGi bundle.

199.3.2. Deploying KuraRouter

Bundle containing your Kura router class should import the following packages in the OSGi manifest:

```
Import-Package: org.osgi.framework;version="1.3.0",
  org.slf4j;version="1.6.4",

org.apache.camel,org.apache.camel.impl,org.apache.camel.core.osgi,org.apache.camel.builder,org.apa
che.camel.model,
  org.apache.camel.component.kura
```

Keep in mind that you don't have to import every Camel component bundle you plan to use in your routes, as Camel components are resolved as the services on the runtime level.

Before you deploy your router bundle, be sure that you have deployed (and started) the following Camel core bundles (using Kura GoGo shell)...

```
install file:///home/user/.m2/repository/org/apache/camel/camel-core/2.15.0/camel-core-2.15.0.jar
start <camel-core-bundle-id>
install file:///home/user/.m2/repository/org/apache/camel/camel-core-osgi/2.15.0/camel-core-osgi-
2.15.0.jar
start <camel-core-osgi-bundle-id>
install file:///home/user/.m2/repository/org/apache/camel/camel-kura/2.15.0/camel-kura-2.15.0.jar
start <camel-kura-bundle-id>
```

...and all the components you plan to use in your routes:

```
install file:///home/user/.m2/repository/org/apache/camel/camel-stream/2.15.0/camel-stream-
2.15.0.jar
start <camel-stream-bundle-id>
```

Then finally deploy your router bundle:

```
install file:///home/user/.m2/repository/com/example/myrouter/1.0/myrouter-1.0.jar
start <your-bundle-id>
```

199.3.3. KuraRouter utilities

Kura router base class provides many useful utilities. This section explores each of them.

199.3.3.1. SLF4J logger

Kura uses SLF4J facade for logging purposes. Protected member **log** returns SLF4J logger instance associated with the given Kura router.

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        log.info("Configuring Camel routes!");
        ...
    }

}
```

199.3.3.2. BundleContext

Protected member **bundleContext** returns bundle context associated with the given Kura router.

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        ServiceReference<MyService> serviceRef =
bundleContext.getServiceReference(LogService.class.getName());
        MyService myService = bundleContext.getService(serviceRef);
        ...
    }

}
```

199.3.3.3. CamelContext

Protected member **camelContext** is the **CamelContext** associated with the given Kura router.

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        camelContext.getStatus();
        ...
    }

}
```

199.3.3.4. ProducerTemplate

Protected member **producerTemplate** is the **ProducerTemplate** instance associated with the given Camel context.

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        producerTemplate.sendBody("jms:temperature", 22.0);
        ...
    }

}
```

199.3.3.5. ConsumerTemplate

Protected member **consumerTemplate** is the **ConsumerTemplate** instance associated with the given Camel context.

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        double currentTemperature = producerTemplate.receiveBody("jms:temperature", Double.class);
        ...
    }

}
```

199.3.3.6. OSGi service resolver

OSGi service resolver (**service(Class<T> serviceType)**) can be used to easily retrieve service by type from the OSGi bundle context.

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        MyService myService = service(MyService.class);
        ...
    }

}
```

If service is not found, a **null** value is returned. If you want your application to fail if the service is not available, use **requiredService(Class)** method instead. The **requiredService** throws **IllegalStateException** if a service cannot be found.

```
public class MyKuraRouter extends KuraRouter {

    @Override
    public void configure() throws Exception {
        MyService myService = requiredService(MyService.class);
```

```
        ...
    }

}
```

199.3.4. KuraRouter activator callbacks

Kura router comes with the lifecycle callbacks that can be used to customize the way the Camel router works. For example to configure the **CamelContext** instance associated with the router just before the former is started, override **beforeStart** method of the **KuraRouter** class:

```
public class MyKuraRouter extends KuraRouter {

  ...

  protected void beforeStart(CamelContext camelContext) {
    OsgiDefaultCamelContext osgiContext = (OsgiCamelContext) camelContext;
    osgiContext.setName("NameOfTheRouter");
  }

}
```

199.3.5. Loading XML routes from ConfigurationAdmin

Sometimes it is desired to read the XML definition of the routes from the server configuration. This a common scenario for IoT gateways where over-the-air redeployment cost may be significant. To address this requirement each **KuraRouter** looks for the **kura.camel.BUNDLE-SYMBOLIC-NAME.route** property from the **kura.camel** PID using the OSGi ConfigurationAdmin. This approach allows you to define Camel XML routes file per deployed **KuraRouter**. In order to update a route, just edit an appropriate configuration property and restart a bundle associated with it. The content of the **kura.camel.BUNDLE-SYMBOLIC-NAME.route** property is expected to be Camel XML route file, for example:

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route id="loaded">
        <from uri="direct:bar"/>
        <to uri="mock:bar"/>
    </route>
</routes>
```

199.3.6. Deploying Kura router as a declarative OSGi service

If you would like to deploy your Kura router as a declarative OSGi service, you can use **activate** and **deactivate** methods provided by **KuraRouter**.

```
<scr:component name="org.eclipse.kura.example.camel.MyKuraRouter" activate="activate"
deactivate="deactivate" enabled="true" immediate="true">
  <implementation class="org.eclipse.kura.example.camel.MyKuraRouter"/>
</scr:component>
```

199.3.7. See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 200. LANGUAGE COMPONENT

Available as of Camel version 2.5

The language component allows you to send Exchange to an endpoint which executes a script by any of the supported Languages in Camel.
By having a component to execute language scripts, it allows more dynamic routing capabilities. For example by using the Routing Slip or Dynamic Router EIPs you can send messages to **language** endpoints where the script is dynamic defined as well.

This component is provided out of the box in **camel-core** and hence no additional JARs is needed. You only have to include additional Camel components if the language of choice mandates it, such as using Groovy or JavaScript languages.

200.1. URI FORMAT

```
language://languageName[:script][?options]
```

And from Camel 2.11 onwards you can refer to an external resource for the script using same notation as supported by the other Languages in Camel

```
language://languageName:resource:scheme:location][?options]
```

200.2. URI OPTIONS

The Language component has no options.

The Language endpoint is configured using URI syntax:

```
language:languageName:resourceUri
```

with the following path and query parameters:

200.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
languageName	**Required** Sets the name of the language to use		String
resourceUri	Path to the resource, or a reference to lookup a bean in the Registry to use as the resource		String

200.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
binary (producer)	Whether the script is binary content or text content. By default the script is read as text content (eg java.lang.String)	false	boolean
cacheScript (producer)	Whether to cache the compiled script and reuse Notice reusing the script can cause side effects from processing one Camel org.apache.camel.Exchange to the next org.apache.camel.Exchange.	false	boolean
contentCache (producer)	Sets whether to use resource content cache or not.	false	boolean
script (producer)	Sets the script to execute		String
transform (producer)	Whether or not the result of the script should be used as message body. This options is default true.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

200.3. MESSAGE HEADERS

The following message headers can be used to affect the behavior of the component

Header	Description
CamelLanguageScript	The script to execute provided in the header. Takes precedence over script configured on the endpoint.

200.4. EXAMPLES

For example you can use the Simple language to Message Translator a message:

In case you want to convert the message body type you can do this as well:

You can also use the Groovy language, such as this example where the input message will by multiplied with 2:

You can also provide the script as a header as shown below. Here we use XPath language to extract the text from the **<foo>** tag.

```
Object out = producer.requestBodyAndHeader("language:xpath", "<foo>Hello World</foo>",
Exchange.LANGUAGE_SCRIPT, "/foo/text()");
assertEquals("Hello World", out);
```

200.5. LOADING SCRIPTS FROM RESOURCES

Available as of Camel 2.9

You can specify a resource uri for a script to load in either the endpoint uri, or in the **Exchange.LANGUAGE_SCRIPT** header.
The uri must start with one of the following schemes: file:, classpath:, or http:

For example to load a script from the classpath:

By default the script is loaded once and cached. However you can disable the **contentCache** option and have the script loaded on each evaluation.
For example if the file myscript.txt is changed on disk, then the updated script is used:

From **Camel 2.11** onwards you can refer to the resource similar to the other Languages in Camel by prefixing with **"resource:"** as shown below:

CHAPTER 201. LDAP COMPONENT

Available as of Camel version 1.5

The **ldap** component allows you to perform searches in LDAP servers using filters as the message payload.
This component uses standard JNDI (**javax.naming** package) to access the server.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ldap</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

201.1. URI FORMAT

```
ldap:ldapServerBean[?options]
```

The *ldapServerBean* portion of the URI refers to a DirContext bean in the registry. The LDAP component only supports producer endpoints, which means that an **ldap** URI cannot appear in the **from** at the start of a route.

You can append query options to the URI in the following format, **?option=value&option=value&…**

201.2. OPTIONS

The LDAP component has no options.

The LDAP endpoint is configured using URI syntax:

```
ldap:dirContextName
```

with the following path and query parameters:

201.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
dirContextName	**Required** Name of either a javax.naming.directory.DirContext, or java.util.Hashtable, or Map bean to lookup in the registry. If the bean is either a Hashtable or Map then a new javax.naming.directory.DirContext instance is created for each use. If the bean is a javax.naming.directory.DirContext then the bean is used as given. The latter may not be possible in all situations where the javax.naming.directory.DirContext must not be shared, and in those situations it can be better to use java.util.Hashtable or Map instead.		String

201.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
base (producer)	The base DN for searches.	ou=system	String
pageSize (producer)	When specified the ldap module uses paging to retrieve all results (most LDAP Servers throw an exception when trying to retrieve more than 1000 entries in one query). To be able to use this a LdapContext (subclass of DirContext) has to be passed in as ldapServerBean (otherwise an exception is thrown)		Integer
returnedAttributes (producer)	Comma-separated list of attributes that should be set in each entry of the result		String
scope (producer)	Specifies how deeply to search the tree of entries, starting at the base DN.	subtree	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

201.3. RESULT

The result is returned in the Out body as a **ArrayList<javax.naming.directory.SearchResult>** object.

201.4. DIRCONTEXT

The URI, **ldap:ldapserver**, references a Spring bean with the ID, **ldapserver**. The **ldapserver** bean may be defined as follows:

```
<bean id="ldapserver" class="javax.naming.directory.InitialDirContext" scope="prototype">
  <constructor-arg>
    <props>
      <prop key="java.naming.factory.initial">com.sun.jndi.ldap.LdapCtxFactory</prop>
      <prop key="java.naming.provider.url">ldap://localhost:10389</prop>
      <prop key="java.naming.security.authentication">none</prop>
    </props>
  </constructor-arg>
</bean>
```

The preceding example declares a regular Sun based LDAP **DirContext** that connects anonymously to a locally hosted LDAP server.

NOTE

DirContext objects are **not** required to support concurrency by contract. It is therefore important that the directory context is declared with the setting, **scope="prototype"**, in the **bean** definition or that the context supports concurrency. In the Spring framework, **prototype** scoped objects are instantiated each time they are looked up.

201.5. SAMPLES

Following on from the Spring configuration above, the code sample below sends an LDAP request to filter search a group for a member. The Common Name is then extracted from the response.

```
ProducerTemplate<Exchange> template = exchange
  .getContext().createProducerTemplate();

Collection<?> results = (Collection<?>) (template
  .sendBody(
    "ldap:ldapserver?base=ou=mygroup,ou=groups,ou=system",
    "(member=uid=huntc,ou=users,ou=system)"));

if (results.size() > 0) {
  // Extract what we need from the device's profile

  Iterator<?> resultIter = results.iterator();
  SearchResult searchResult = (SearchResult) resultIter
      .next();
  Attributes attributes = searchResult
      .getAttributes();
  Attribute deviceCNAttr = attributes.get("cn");
  String deviceCN = (String) deviceCNAttr.get();

  ...
```

If no specific filter is required - for example, you just need to look up a single entry - specify a wildcard filter expression. For example, if the LDAP entry has a Common Name, use a filter expression like:

```
(cn=*)
```

201.5.1. Binding using credentials

A Camel end user donated this sample code he used to bind to the ldap server using credentials.

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
props.setProperty(Context.PROVIDER_URL, "ldap://localhost:389");
props.setProperty(Context.URL_PKG_PREFIXES, "com.sun.jndi.url");
props.setProperty(Context.REFERRAL, "ignore");
props.setProperty(Context.SECURITY_AUTHENTICATION, "simple");
props.setProperty(Context.SECURITY_PRINCIPAL, "cn=Manager");
props.setProperty(Context.SECURITY_CREDENTIALS, "secret");

SimpleRegistry reg = new SimpleRegistry();
reg.put("myldap", new InitialLdapContext(props, null));

CamelContext context = new DefaultCamelContext(reg);
context.addRoutes(
   new RouteBuilder() {
      public void configure() throws Exception {
         from("direct:start").to("ldap:myldap?base=ou=test");
      }
   }
);
context.start();

ProducerTemplate template = context.createProducerTemplate();

Endpoint endpoint = context.getEndpoint("direct:start");
Exchange exchange = endpoint.createExchange();
exchange.getIn().setBody("(uid=test)");
Exchange out = template.send(endpoint, exchange);

Collection<SearchResult> data = out.getOut().getBody(Collection.class);
assert data != null;
assert !data.isEmpty();

System.out.println(out.getOut().getBody());

context.stop();
```

201.6. CONFIGURING SSL

All required is to create a custom socket factory and reference it in the InitialDirContext bean - see below sample.

SSL Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
                http://camel.apache.org/schema/blueprint http://camel.apache.org/schema/blueprint/camel-
blueprint.xsd">
```

```
    <sslContextParameters xmlns="http://camel.apache.org/schema/blueprint"
                          id="sslContextParameters">
      <keyManagers
            keyPassword="{{keystore.pwd}}">
        <keyStore
              resource="{{keystore.url}}"
              password="{{keystore.pwd}}"/>
      </keyManagers>
    </sslContextParameters>

    <bean id="customSocketFactory" class="zotix.co.util.CustomSocketFactory">
      <argument ref="sslContextParameters" />
    </bean>
    <bean id="ldapserver" class="javax.naming.directory.InitialDirContext" scope="prototype">
      <argument>
        <props>
          <prop key="java.naming.factory.initial" value="com.sun.jndi.ldap.LdapCtxFactory"/>
          <prop key="java.naming.provider.url" value="ldaps://lab.zotix.co:636"/>
          <prop key="java.naming.security.protocol" value="ssl"/>
          <prop key="java.naming.security.authentication" value="simple" />
          <prop key="java.naming.security.principal" value="cn=Manager,dc=example,dc=com"/>
          <prop key="java.naming.security.credentials" value="passw0rd"/>
          <prop key="java.naming.ldap.factory.socket"
              value="zotix.co.util.CustomSocketFactory"/>
        </props>
      </argument>
    </bean>
</blueprint>
```

Custom Socket Factory

```
import org.apache.camel.util.jsse.SSLContextParameters;

import javax.net.SocketFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;
import java.security.KeyStore;

/**
 * The CustomSocketFactory. Loads the KeyStore and creates an instance of SSLSocketFactory
 */
public class CustomSocketFactory extends SSLSocketFactory {

    private static SSLSocketFactory socketFactory;

    /**
     * Called by the getDefault() method.
     */
    public CustomSocketFactory() {
```

```
    }

    /**
     * Called by Blueprint DI to initialise an instance of SocketFactory
     *
     * @param sslContextParameters
     */
    public CustomSocketFactory(SSLContextParameters sslContextParameters) {
        try {
            KeyStore keyStore =
sslContextParameters.getKeyManagers().getKeyStore().createKeyStore();
            TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
            tmf.init(keyStore);
            SSLContext ctx = SSLContext.getInstance("TLS");
            ctx.init(null, tmf.getTrustManagers(), null);
            socketFactory = ctx.getSocketFactory();
        } catch (Exception ex) {
            ex.printStackTrace(System.err);  /* handle exception */
        }
    }

    /**
     * Getter for the SocketFactory
     *
     * @return
     */
    public static SocketFactory getDefault() {
        return new CustomSocketFactory();
    }

    @Override
    public String[] getDefaultCipherSuites() {
        return socketFactory.getDefaultCipherSuites();
    }

    @Override
    public String[] getSupportedCipherSuites() {
        return socketFactory.getSupportedCipherSuites();
    }

    @Override
    public Socket createSocket(Socket socket, String string, int i, boolean bln) throws IOException {
        return socketFactory.createSocket(socket, string, i, bln);
    }

    @Override
    public Socket createSocket(String string, int i) throws IOException {
        return socketFactory.createSocket(string, i);
    }

    @Override
    public Socket createSocket(String string, int i, InetAddress ia, int i1) throws IOException {
        return socketFactory.createSocket(string, i, ia, i1);
    }

    @Override
```

```
    public Socket createSocket(InetAddress ia, int i) throws IOException {
        return socketFactory.createSocket(ia, i);
    }

    @Override
    public Socket createSocket(InetAddress ia, int i, InetAddress ia1, int i1) throws IOException {
        return socketFactory.createSocket(ia, i, ia1, i1);
    }
}
```

201.7. SEE ALSO

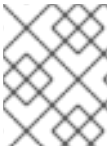
- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 202. LDIF COMPONENT

Available as of Camel version 2.20

The **ldif** component allows you to do updates on an LDAP server from a LDIF body content.

This component uses a basic URL syntax to access the server. It uses the Apache DS LDAP library to process the LDIF. After processing the LDIF, the response body will be a list of statuses for success/failure of each entry.



NOTE

The Apache LDAP API is very sensitive to LDIF syntax errors. If in doubt, refer to the unit tests to see an example of each change type.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ldif</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

202.1. URI FORMAT

```
ldap:ldapServerBean[?options]
```

The *ldapServerBean* portion of the URI refers to a LdapConnection. This should be constructed from a factory at the point of use to avoid connection timeouts. The LDIF component only supports producer endpoints, which means that an **ldif** URI cannot appear in the **from** at the start of a route.

For SSL configuration, refer to the **camel-ldap** component where there is an example of setting up a custom SocketFactory instance.

You can append query options to the URI in the following format, **?option=value&option=value&…**

202.2. OPTIONS

The LDIF component has no options.

The LDIF endpoint is configured using URI syntax:

```
ldif:ldapConnectionName
```

with the following path and query parameters:

202.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
ldapConnectionName	**Required** The name of the LdapConnection bean to pull from the registry. Note that this must be of scope prototype to avoid it being shared among threads or using a connection that has timed out.		String

202.2.2. Query Parameters (1 parameters):

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

202.3. BODY TYPES:

The body can be a URL to an LDIF file or an inline LDIF file. To signify the difference in body types, an inline LDIF must start with:

```
version: 1
```

If not, the component will try to parse the body as a URL.

202.4. RESULT

The result is returned in the Out body as a **ArrayList<java.lang.String>** object. This contains either "success" or an Exception message for each LDIF entry.

202.5. LDAPCONNECTION

The URI, **ldif:ldapConnectionName**, references a bean with the ID, **ldapConnectionName**. The ldapConnection can be configured using a **LdapConnectionConfig** bean. Note that the scope must have a scope of **prototype** to avoid the connection being shared or picking up a stale connection.

The **LdapConnection** bean may be defined as follows in Spring XML:

```
<bean id="ldapConnectionOptions"
class="org.apache.directory.ldap.client.api.LdapConnectionConfig">
  <property name="ldapHost" value="${ldap.host}"/>
  <property name="ldapPort" value="${ldap.port}"/>
  <property name="name" value="${ldap.username}"/>
  <property name="credentials" value="${ldap.password}"/>
  <property name="useSsl" value="false"/>
  <property name="useTls" value="false"/>
</bean>
```

```
<bean id="ldapConnectionFactory"
class="org.apache.directory.ldap.client.api.DefaultLdapConnectionFactory">
  <constructor-arg index="0" ref="ldapConnectionOptions"/>
</bean>

<bean id="ldapConnection" factory-bean="ldapConnectionFactory" factory-
method="newLdapConnection" scope="prototype"/>
```

or in a OSGi blueprint.xml:

```
<bean id="ldapConnectionOptions"
class="org.apache.directory.ldap.client.api.LdapConnectionConfig">
  <property name="ldapHost" value="${ldap.host}"/>
  <property name="ldapPort" value="${ldap.port}"/>
  <property name="name" value="${ldap.username}"/>
  <property name="credentials" value="${ldap.password}"/>
  <property name="useSsl" value="false"/>
  <property name="useTls" value="false"/>
</bean>

<bean id="ldapConnectionFactory"
class="org.apache.directory.ldap.client.api.DefaultLdapConnectionFactory">
  <argument ref="ldapConnectionOptions"/>
</bean>

<bean id="ldapConnection" factory-ref="ldapConnectionFactory" factory-
method="newLdapConnection" scope="prototype"/>
```

202.6. SAMPLES

Following on from the Spring configuration above, the code sample below sends an LDAP request to filter search a group for a member. The Common Name is then extracted from the response.

```
ProducerTemplate<Exchange> template = exchange.getContext().createProducerTemplate();

List<?> results = (Collection<?>) template.sendBody("ldap:ldapConnection, "LDiff goes here");

if (results.size() > 0) {
  // Check for no errors

  for (String result : results) {
    if ("success".equalTo(result)) {
      // LDIF entry success
    } else {
      // LDIF entry failure
    }
  }
}
```

202.7. LEVELDB

Available as of Camel 2.10

Leveldb is a very lightweight and embedable key value database. It allows together with Camel to provide persistent support for various Camel features such as Aggregator.

Current features it provides:

- LevelDBAggregationRepository

202.7.1. Using LevelDBAggregationRepository

LevelDBAggregationRepository is an **AggregationRepository** which on the fly persists the aggregated messages. This ensures that you will not loose messages, as the default aggregator will use an in memory only **AggregationRepository**.

It has the following options:

Option	Type	Description
repositoryName	String	A mandatory repository name. Allows you to use a shared **LevelDBFile** for multiple repositories.
persistentFileName	String	Filename for the persistent storage. If no file exists on startup a new file is created.
levelDBFile	LevelDBFile	Use an existing configured **org.apache.camel.component.leveldb.LevelDBFile** instance.
sync	boolean	**Camel 2.12:** Whether or not the LevelDBFile should sync on write or not. Default is false. By sync on write ensures that its always waiting for all writes to be spooled to disk and thus will not loose updates. See LevelDB docs for more details about async vs sync writes.
returnOldExchange	boolean	Whether the get operation should return the old existing Exchange if any existed. By default this option is **false** to optimize as we do not need the old exchange when aggregating.
useRecovery	boolean	Whether or not recovery is enabled. This option is by default **true**. When enabled the Camel Aggregator automatic recover failed aggregated exchange and have them resubmitted.
recoveryInterval	long	If recovery is enabled then a background task is run every x'th time to scan for failed exchanges to recover and resubmit. By default this interval is 5000 millis.
maximumRedeliveries	int	Allows you to limit the maximum number of redelivery attempts for a recovered exchange. If enabled then the Exchange will be moved to the dead letter channel if all redelivery attempts failed. By default this option is disabled. If this option is used then the **deadLetterUri** option must also be provided.

Option	Type	Description
deadLetterUri	String	An endpoint uri for a Dead Letter Channel where exhausted recovered Exchanges will be moved. If this option is used then the **maximumRedeliveries** option must also be provided.

The **repositoryName** option must be provided. Then either the **persistentFileName** or the **levelDBFile** must be provided.

202.7.2. What is preserved when persisting

LevelDBAggregationRepository will only preserve any **Serializable** compatible message body data types. Message headers must be primitive / string / numbers / etc. If a data type is not such a type its dropped and a **WARN** is logged. And it only persists the **Message** body and the **Message** headers. The **Exchange** properties are **not** persisted.

202.7.3. Recovery

The **LevelDBAggregationRepository** will by default recover any failed Exchange. It does this by having a background tasks that scans for failed Exchanges in the persistent store. You can use the **checkInterval** option to set how often this task runs. The recovery works as transactional which ensures that Camel will try to recover and redeliver the failed Exchange. Any Exchange which was found to be recovered will be restored from the persistent store and resubmitted and send out again.

The following headers is set when an Exchange is being recovered/redelivered:

Header	Type	Description
Exchange.REDELIVERED	Boolean	Is set to true to indicate the Exchange is being redelivered.
Exchange.REDELIVERY_COUNTER	Integer	The redelivery attempt, starting from 1.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the **confirm** method is invoked on the **AggregationRepository**. This means if the same Exchange fails again it will be kept retried until it success.

You can use option **maximumRedeliveries** to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the **deadLetterUri** option so Camel knows where to send the Exchange when the **maximumRedeliveries** was hit.

You can see some examples in the unit tests of camel-leveldb, for example this test.

202.7.3.1. Using LevelDBAggregationRepository in Java DSL

In this example we want to persist aggregated messages in the **target/data/leveldb.dat** file.

202.7.3.2. Using LevelDBAggregationRepository in Spring XML

The same example but using Spring XML instead:

202.7.4. Dependencies

To use LevelDB in your camel routes you need to add the a dependency on **camel-leveldb**.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-leveldb</artifactId>
  <version>2.10.0</version>
</dependency>
```

202.7.5. See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Aggregator
- HawtDB
- Components

CHAPTER 203. LINKEDIN COMPONENT

Available as of Camel version 2.14

The LinkedIn component provides access to all of LinkedIn REST APIs documented at https://developer.linkedin.com/rest.

LinkedIn uses OAuth2.0 for all client application authentication. In order to use camel-linkedin with your account, you'll need to create a new application for LinkedIn at https://www.linkedin.com/secure/developer. The LinkedIn application's client id and secret will allow access to LinkedIn REST APIs which require a current user. A user access token is generated and managed by component for an end user. Alternatively the Camel application can register an implementation of org.apache.camel.component.linkedin.api.OAuthSecureStorage to provide an org.apache.camel.component.linkedin.apiOAuthToken OAuth token.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-linkedin</artifactId>
    <version>${camel-version}</version>
</dependency>
```

203.1. URI FORMAT

```
linkedin://endpoint-prefix/endpoint?[options]
```

Endpoint prefix can be one of:

- comments
- companies
- groups
- jobs
- people
- posts
- search

203.2. LINKEDINCOMPONENT

The Linkedin component supports 2 options, which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		LinkedInConfiguration

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Linkedin endpoint is configured using URI syntax:

```
linkedin:apiName/methodName
```

with the following path and query parameters:

203.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		LinkedInApiName
methodName	**Required** What sub operation to use for the selected operation		String

203.2.2. Query Parameters (16 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
accessToken (common)	LinkedIn access token to avoid username and password login procedure. LinkedIn responds to login forms by using a CAPTCHA. This makes it impossible for a standalone, headless process to log in to LinkedIn by specifying a username and password. To work around this, obtain a LinkedIn access token and provide the token as the setting of the accessToken parameter. Obtaining a LinkedIn access token is a multi-step procedure. You must configure your LinkedIn application, obtain a LinkedIn authorization code, and exchange that code for the LinkedIn access token. For details, see: https://developer.linkedin.com/docs/oauth2 The default behavior is that the access token expires after 60 days. To change this, specify a value for the expiryTime paramter. If the access token expires, the LinkedIn component tries to log in to LinkedIn by providing a username and password, which results in a CAPTCHA so the login fails. The LinkedIn component cannot refresh the access token. You must manually obtain a new access token each time an access token expires. When you update the access token you must restart the application so that it uses the new token.		String
clientId (common)	LinkedIn application client ID		String
clientSecret (common)	LinkedIn application client secret		String
expiryTime (common)	A number of milliseconds since the UNIX Epoch. The default is 60 days. A LinkedIn access token expires when this amount of time elapses after the token is in use.		Long
httpParams (common)	Custom HTTP parameters, for example, proxy host and port. Use constants from AllClientPNames.		Map
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
lazyAuth (common)	Flag to enable/disable lazy OAuth, default is true. When enabled, OAuth token retrieval or generation is not done until the first REST call.	true	boolean

Name	Description	Default	Type
redirectUri (common)	Application redirect URI, although the component never redirects to this page to avoid having to have a functioning redirect server. For testing, one could use https://localhost.		String
scopes (common)	List of LinkedIn scopes as specified at https://developer.linkedin.com/documents/authentication#granting		OAuthScope[]
secureStorage (common)	Callback interface for providing an OAuth token or to store the token generated by the component. The callback should return null on the first call and then save the created token in the saveToken() callback. If the callback returns null the first time, a userPassword MUST be provided.		OAuthSecureStorage
userName (common)	LinkedIn user account name, MUST be provided		String
userPassword (common)	LinkedIn account password		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this option is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

203.3. SPRING BOOT AUTO-CONFIGURATION

The component supports 15 options, which are listed below.

Name	Description	Default	Type
camel.component.linkedin.configuration.access-token	LinkedIn access token to avoid username and password login procedure. LinkedIn responds to login forms by using a CAPTCHA. This makes it impossible for a standalone, headless process to log in to LinkedIn by specifying a username and password. To work around this, obtain a LinkedIn access token and provide the token as the setting of the accessToken parameter. Obtaining a LinkedIn access token is a multi-step procedure. You must configure your LinkedIn application, obtain a LinkedIn authorization code, and exchange that code for the LinkedIn access token. For details, see: https://developer.linkedin.com/docs/oauth2 The default behavior is that the access token expires after 60 days. To change this, specify a value for the expiryTime paramter. If the access token expires, the LinkedIn component tries to log in to LinkedIn by providing a username and password, which results in a CAPTCHA so the login fails. The LinkedIn component cannot refresh the access token. You must manually obtain a new access token each time an access token expires. When you update the access token you must restart the application so that it uses the new token.		String
camel.component.linkedin.configuration.api-name	What kind of operation to perform		LinkedInApiName
camel.component.linkedin.configuration.client-id	LinkedIn application client ID		String
camel.component.linkedin.configuration.client-secret	LinkedIn application client secret		String
camel.component.linkedin.configuration.expiry-time	A number of milliseconds since the UNIX Epoch. The default is 60 days. A LinkedIn access token expires when this amount of time elapses after the token is in use.		Long
camel.component.linkedin.configuration.http-params	Custom HTTP parameters, for example, proxy host and port. Use constants from AllClientPNames.		Map

Name	Description	Default	Type
camel.component.linkedin.configuration.lazy-auth	Flag to enable/disable lazy OAuth, default is true. When enabled, OAuth token retrieval or generation is not done until the first REST call.	true	Boolean
camel.component.linkedin.configuration.method-name	What sub operation to use for the selected operation		String
camel.component.linkedin.configuration.redirect-uri	Application redirect URI, although the component never redirects to this page to avoid having to have a functioning redirect server. For testing, one could use https://localhost.		String
camel.component.linkedin.configuration.scopes	List of LinkedIn scopes as specified at https://developer.linkedin.com/documents/authentication#granting		OAuthScope[]
camel.component.linkedin.configuration.secure-storage	Callback interface for providing an OAuth token or to store the token generated by the component. The callback should return null on the first call and then save the created token in the saveToken() callback. If the callback returns null the first time, a userPassword MUST be provided.		OAuthSecureStorage
camel.component.linkedin.configuration.user-name	LinkedIn user account name, MUST be provided		String
camel.component.linkedin.configuration.user-password	LinkedIn account password		String
camel.component.linkedin.enabled	Enable linkedin component	true	Boolean
camel.component.linkedin.resolve-property-placeholders	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	Boolean

203.4. PRODUCER ENDPOINTS:

Producer endpoints can use endpoint prefixes followed by endpoint names and associated options described next. A shorthand alias can be used for some endpoints. The endpoint URI MUST contain a prefix.

Endpoint options that are not mandatory are denoted by []. When there are no mandatory options for an endpoint, one of the set of [] options MUST be provided. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelLinkedIn.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelLinkedIn.option** header.

For more information on the endpoints and options see LinkedIn REST API documentation at https://developer.linkedin.com/rest.

203.4.1. Endpoint prefix *comments*

The following endpoints can be invoked with the prefix **comments** as follows:

```
linkedin://comments/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
getComment	comment	comment_id, fields	org.apache.camel.component.linkedin.api.model.Comment
removeComment	comment	comment_id	

URI Options for *comments*

Name	Type
comment_id	String
fields	String

203.4.2. Endpoint prefix *companies*

The following endpoints can be invoked with the prefix **companies** as follows:

```
linkedin://companies/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type

Endpoint	Shorthand Alias	Options	Result Body Type
addCompanyUpdateComment	companyUpdateComment	company_id, update_key, updatecomment	
addCompanyUpdateCommentAsCompany	companyUpdateCommentAsCompany	company_id, update_key, updatecomment	
addShare	share	company_id, share	
getCompanies	companies	email_domain, fields, is_company_admin	org.apache.camel.component.linkedin.api.model.Companies
getCompanyById	companyById	company_id, fields	org.apache.camel.component.linkedin.api.model.Company
getCompanyByName	companyByName	fields, universal_name	org.apache.camel.component.linkedin.api.model.Company
getCompanyUpdateComments	companyUpdateComments	company_id, fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Comments
getCompanyUpdateLikes	companyUpdateLikes	company_id, fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Likes
getCompanyUpdates	companyUpdates	company_id, count, event_type, fields, start	org.apache.camel.component.linkedin.api.model.Updates
getHistoricalFollowStatistics	historicalFollowStatistics	company_id, end_timestamp, start_timestamp, time_granularity	org.apache.camel.component.linkedin.api.model.HistoricalFollowStatistics
getHistoricalStatusUpdateStatistics	historicalStatusUpdateStatistics	company_id, end_timestamp, start_timestamp, time_granularity, update_key	org.apache.camel.component.linkedin.api.model.HistoricalStatusUpdateStatistics
getNumberOfFollowers	numberOfFollowers	companySizes, company_id, geos, industries, jobFunc, seniorities	org.apache.camel.component.linkedin.api.model.NumFollowers

Endpoint	Shorthand Alias	Options	Result Body Type
getStatistics	statistics	company_id	org.apache.camel.component.linkedin.api.model.CompanyStatistics
isShareEnabled		company_id	org.apache.camel.component.linkedin.api.model.IsCompanyShareEnabled
isViewerShareEnabled		company_id	org.apache.camel.component.linkedin.api.model.IsCompanyShareEnabled
likeCompanyUpdate		company_id, isliked, update_key	

URI Options for *companies*

If a value is not provided for one of the option(s) **[companySizes, count, email_domain, end_timestamp, event_type, geos, industries, is_company_admin, jobFunc, secure_urls, seniorities, start, start_timestamp, time_granularity]** either in the endpoint URI or in a message header, it will be assumed to be **null**. Note that the **null** value(s) will only be used if other options do not satisfy matching endpoints.

Name	Type
companySizes	java.util.List
company_id	Long
count	Long
email_domain	String
end_timestamp	Long
event_type	org.apache.camel.component.linkedin.api.Eventtype
fields	String
geos	java.util.List
industries	java.util.List

Name	Type
is_company_admin	Boolean
isliked	org.apache.camel.component.linkedin.api.model.IsLiked
jobFunc	java.util.List
secure_urls	Boolean
seniorities	java.util.List
share	org.apache.camel.component.linkedin.api.model.Share
start	Long
start_timestamp	Long
time_granularity	org.apache.camel.component.linkedin.api.TimeGranularity
universal_name	String
update_key	String
updatecomment	org.apache.camel.component.linkedin.api.model.UpdateComment

203.4.3. Endpoint prefix *groups*

The following endpoints can be invoked with the prefix **groups** as follows:

```
linkedin://groups/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
addPost	post	group_id, post	
getGroup	group	group_id	org.apache.camel.component.linkedin.api.model.Group

URI Options for *groups*

Name	Type
group_id	Long
post	org.apache.camel.component.linkedin.api.model.Post

203.4.4. Endpoint prefix *jobs*

The following endpoints can be invoked with the prefix **jobs** as follows:

```
linkedin://jobs/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
addJob	job	job	
editJob		job, partner_job_id	
getJob	job	fields, job_id	org.apache.camel.component.linkedin.api.model.Job

URI Options for *jobs*

Name	Type
fields	String
job	org.apache.camel.component.linkedin.api.model.Job
job_id	Long
partner_job_id	Long

203.4.5. Endpoint prefix *people*

The following endpoints can be invoked with the prefix **people** as follows:

```
linkedin://people/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
addActivity	activity	activity	
addGroupMembership	groupMembership	groupmembership	

Endpoint	Shorthand Alias	Options	Result Body Type
addInvite	invite	mailboxitem	
addJobBookmark	jobBookmark	jobbookmark	
addUpdateComment	updateComment	update_key, updatecomment	
followCompany		company	
getConnections	connections	fields, secure_urls	org.apache.camel.component.linkedin.api.model.Connections
getConnectionsById	connectionsById	fields, person_id, secure_urls	org.apache.camel.component.linkedin.api.model.Connections
getConnectionsByUrl	connectionsByUrl	fields, public_profile_url, secure_urls	org.apache.camel.component.linkedin.api.model.Connections
getFollowedCompanies	followedCompanies	fields	org.apache.camel.component.linkedin.api.model.Companies
getGroupMembershipSettings	groupMembershipSettings	count, fields, group_id, start	org.apache.camel.component.linkedin.api.model.GroupMemberships
getGroupMemberships	groupMemberships	count, fields, membership_state, start	org.apache.camel.component.linkedin.api.model.GroupMemberships
getJobBookmarks	jobBookmarks		org.apache.camel.component.linkedin.api.model.JobBookmarks
getNetworkStats	networkStats		org.apache.camel.component.linkedin.api.model.NetworkStats
getNetworkUpdates	networkUpdates	after, before, count, fields, scope, secure_urls, show_hidden_members, start, type	org.apache.camel.component.linkedin.api.model.Updates

Endpoint	Shorthand Alias	Options	Result Body Type
getNetworkUpdatesById	networkUpdatesById	after, before, count, fields, person_id, scope, secure_urls, show_hidden_members, start, type	org.apache.camel.component.linkedin.api.model.Updates
getPerson	person	fields, secure_urls	org.apache.camel.component.linkedin.api.model.Person
getPersonById	personById	fields, person_id, secure_urls	org.apache.camel.component.linkedin.api.model.Person
getPersonByUrl	personByUrl	fields, public_profile_url, secure_urls	org.apache.camel.component.linkedin.api.model.Person
getPosts	posts	category, count, fields, group_id, modified_since, order, role, start	org.apache.camel.component.linkedin.api.model.Posts
getSuggestedCompanies	suggestedCompanies	fields	org.apache.camel.component.linkedin.api.model.Companies
getSuggestedGroupPosts	suggestedGroupPosts	category, count, fields, group_id, modified_since, order, role, start	org.apache.camel.component.linkedin.api.model.Posts
getSuggestedGroups	suggestedGroups	fields	org.apache.camel.component.linkedin.api.model.Groups
getSuggestedJobs	suggestedJobs	fields	org.apache.camel.component.linkedin.api.model.JobSuggestions
getUpdateComments	updateComments	fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Comments
getUpdateLikes	updateLikes	fields, secure_urls, update_key	org.apache.camel.component.linkedin.api.model.Likes
likeUpdate		isliked, update_key	

Endpoint	Shorthand Alias	Options	Result Body Type
removeGroupMembership	groupMembership	group_id	
removeGroupSuggestion	groupSuggestion	group_id	
removeJobBookmark	jobBookmark	job_id	
share		share	org.apache.camel.component.linkedin.api.model.Update
stopFollowingCompany		company_id	
updateGroupMembership		group_id, groupmembership	

URI Options for *people*

If a value is not provided for one of the option(s) **[after, before, category, count, membership_state, modified_since, order, public_profile_url, role, scope, secure_urls, show_hidden_members, start, type]** either in the endpoint URI or in a message header, it will be assumed to be **null**. Note that the **null** value(s) will only be used if other options do not satisfy matching endpoints.

Name	Type
activity	org.apache.camel.component.linkedin.api.model.Activity
after	Long
before	Long
category	org.apache.camel.component.linkedin.api.Category
company	org.apache.camel.component.linkedin.api.model.Company
company_id	Long
count	Long
fields	String
group_id	Long

Name	Type
groupmembership	org.apache.camel.component.linkedin.api.model.GroupMembership
isliked	org.apache.camel.component.linkedin.api.model.IsLiked
job_id	Long
jobbookmark	org.apache.camel.component.linkedin.api.model.JobBookmark
mailboxitem	org.apache.camel.component.linkedin.api.model.MailboxItem
membership_state	org.apache.camel.component.linkedin.api.model.MembershipState
modified_since	Long
order	org.apache.camel.component.linkedin.api.Order
person_id	String
public_profile_url	String
role	org.apache.camel.component.linkedin.api.Role
scope	String
secure_urls	Boolean
share	org.apache.camel.component.linkedin.api.model.Share
show_hidden_members	Boolean
start	Long
type	org.apache.camel.component.linkedin.api.Type
update_key	String
updatecomment	org.apache.camel.component.linkedin.api.model.UpdateComment

203.4.6. Endpoint prefix *posts*

The following endpoints can be invoked with the prefix **posts** as follows:

```
linkedin://posts/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
addComment	comment	comment, post_id	
flagCategory		post_id, postcategorycode	
followPost		isfollowing, post_id	
getPost	post	count, fields, post_id, start	org.apache.camel.component.linkedin.api.model.Post
getPostComments	postComments	count, fields, post_id, start	org.apache.camel.component.linkedin.api.model.Comments
likePost		isliked, post_id	
removePost	post	post_id	

URI Options for *posts*

If a value is not provided for one of the option(s) **[count, start]** either in the endpoint URI or in a message header, it will be assumed to be **null**. Note that the **null** value(s) will only be used if other options do not satisfy matching endpoints.

Name	Type
comment	org.apache.camel.component.linkedin.api.model.Comment
count	Long
fields	String
isfollowing	org.apache.camel.component.linkedin.api.model.IsFollowing
isliked	org.apache.camel.component.linkedin.api.model.IsLiked

Name	Type
post_id	String
postcategorycode	org.apache.camel.component.linkedin.api.model.PostCategoryCode
start	Long

203.4.7. Endpoint prefix *search*

The following endpoints can be invoked with the prefix **search** as follows:

```
linkedin://search/endpoint?[options]
```

Endpoint	Shorthand Alias	Options	Result Body Type
searchCompanies	companies	count, facet, facets, fields, hq_only, keywords, sort, start	org.apache.camel.component.linkedin.api.model.CompanySearch
searchJobs	jobs	company_name, count, country_code, distance, facet, facets, fields, job_title, keywords, postal_code, sort, start	org.apache.camel.component.linkedin.api.model.JobSearch
searchPeople	people	company_name, count, country_code, current_company, current_school, current_title, distance, facet, facets, fields, first_name, keywords, last_name, postal_code, school_name, sort, start, title	org.apache.camel.component.linkedin.api.model.PeopleSearch

URI Options for *search*

If a value is not provided for one of the option(s) **[company_name, count, country_code, current_company, current_school, current_title, distance, facet, facets, first_name, hq_only, job_title, keywords, last_name, postal_code, school_name, sort, start, title]** either in the endpoint URI or in a message header, it will be assumed to be **null**. Note that the **null** value(s) will only be used if other options do not satisfy matching endpoints.

Name	Type
company_name	String

Name	Type
count	Long
country_code	String
current_company	String
current_school	String
current_title	String
distance	org.apache.camel.component.linkedin.api.model.Distance
facet	String
facets	String
fields	String
first_name	String
hq_only	String
job_title	String
keywords	String
last_name	String
postal_code	String
school_name	String
sort	String
start	Long
title	String

203.5. CONSUMER ENDPOINTS

Any of the producer endpoints can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. By default Consumer endpoints that return an array or collection will generate one exchange per element,

and their routes will be executed once for each exchange. To change this behavior use the property **consumer.splitResults=true** to return a single exchange for the entire list or array.

203.6. MESSAGE HEADERS

Any URI option can be provided in a message header for producer endpoints with a **CamelLinkedIn.** prefix.

203.7. MESSAGE BODY

All result message bodies utilize objects provided by the Camel LinkedIn API SDK, which is built using Apache CXF JAX-RS. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint parameter.

203.8. USE CASES

The following route gets user's profile:

```
from("direct:foo")
    .to("linkedin://people/person");
```

The following route polls user's connections every 30 seconds:

```
from("linkedin://people/connections?consumer.timeUnit=SECONDS&consumer.delay=30")
    .to("bean:foo");
```

The following route uses a producer with dynamic header options. The **personId** header has the LinkedIn person ID, so its assigned to the **CamelLinkedIn.person_id** header as follows:

```
from("direct:foo")
    .setHeader("CamelLinkedIn.person_id", header("personId"))
    .to("linkedin://people/connectionsById")
    .to("bean://bar");
```

CHAPTER 204. LOG COMPONENT

Available as of Camel version 1.1

The **log:** component logs message exchanges to the underlying logging mechanism.

Camel uses sfl4j which allows you to configure logging via, among others:

- Log4j
- Logback
- Java Util Logging

204.1. URI FORMAT

```
log:loggingCategory[?options]
```

Where **loggingCategory** is the name of the logging category to use. You can append query options to the URI in the following format, **?option=value&option=value&...**

INFO:*Using Logger instance from the the Registry* As of **Camel 2.12.4/2.13.1**, if there's single instance of **org.slf4j.Logger** found in the Registry, the **loggingCategory** is no longer used to create logger instance. The registered instance is used instead. Also it is possible to reference particular **Logger** instance using **?logger=#myLogger** URI parameter. Eventually, if there's no registered and URI **logger** parameter, the logger instance is created using **loggingCategory**.

For example, a log endpoint typically specifies the logging level using the **level** option, as follows:

```
log:org.apache.camel.example?level=DEBUG
```

The default logger logs every exchange (*regular logging*). But Camel also ships with the **Throughput** logger, which is used whenever the **groupSize** option is specified.

TIP:*Also a log in the DSL* There is also a **log** directly in the DSL, but it has a different purpose. Its meant for lightweight and human logs. See more details at LogEIP.

204.2. OPTIONS

The Log component supports 2 options which are listed below.

Name	Description	Default	Type
exchangeFormatter (advanced)	Sets a custom ExchangeFormatter to convert the Exchange to a String suitable for logging. If not specified, we default to DefaultExchangeFormatter.		ExchangeFormatter
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Log endpoint is configured using URI syntax:

```
log:loggerName
```

with the following path and query parameters:

204.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
loggerName	**Required** The logger name to use		String

204.2.2. Query Parameters (26 parameters):

Name	Description	Default	Type
groupActiveOnly (producer)	If true, will hide stats when no new messages have been received for a time interval, if false, show stats regardless of message traffic.	true	Boolean
groupDelay (producer)	Set the initial delay for stats (in millis)		Long
groupInterval (producer)	If specified will group message stats by this time interval (in millis)		Long
groupSize (producer)	An integer that specifies a group size for throughput logging.		Integer
level (producer)	Logging level to use. The default value is INFO.	INFO	String
logMask (producer)	If true, mask sensitive information like password or passphrase in the log.		Boolean
marker (producer)	An optional Marker name to use.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
maxChars (formatting)	Limits the number of characters logged per line.	10000	int
multiline (formatting)	If enabled then each information is outputted on a newline.	false	boolean

Name	Description	Default	Type
showAll (formatting)	Quick option for turning all options on. (multiline, maxChars has to be manually set if to be used)	false	boolean
showBody (formatting)	Show the message body.	true	boolean
showBodyType (formatting)	Show the body Java type.	true	boolean
showCaughtException (formatting)	f the exchange has a caught exception, show the exception message (no stack trace). A caught exception is stored as a property on the exchange (using the key link org.apache.camel.ExchangeEXCEPTION_CAUGHT and for instance a doCatch can catch exceptions.	false	boolean
showException (formatting)	If the exchange has an exception, show the exception message (no stacktrace)	false	boolean
showExchangeId (formatting)	Show the unique exchange ID.	false	boolean
showExchangePattern (formatting)	Shows the Message Exchange Pattern (or MEP for short).	true	boolean
showFiles (formatting)	If enabled Camel will output files	false	boolean
showFuture (formatting)	If enabled Camel will on Future objects wait for it to complete to obtain the payload to be logged.	false	boolean
showHeaders (formatting)	Show the message headers.	false	boolean
showOut (formatting)	If the exchange has an out message, show the out message.	false	boolean
showProperties (formatting)	Show the exchange properties.	false	boolean
showStackTrace (formatting)	Show the stack trace, if an exchange has an exception. Only effective if one of showAll, showException or showCaughtException are enabled.	false	boolean

Name	Description	Default	Type
showStreams (formatting)	Whether Camel should show stream bodies or not (eg such as java.io.InputStream). Beware if you enable this option then you may not be able later to access the message body as the stream have already been read by this logger. To remedy this you will have to use Stream Caching.	false	boolean
skipBodyLineSeparator (formatting)	Whether to skip line separators when logging the message body. This allows to log the message body in one line, setting this option to false will preserve any line separators from the body, which then will log the body as is.	true	boolean
style (formatting)	Sets the outputs style to use.	Default	OutputStyle

204.3. REGULAR LOGGER SAMPLE

In the route below we log the incoming orders at **DEBUG** level before the order is processed:

```
from("activemq:orders").to("log:com.mycompany.order?level=DEBUG").to("bean:processOrder");
```

Or using Spring XML to define the route:

```
<route>
  <from uri="activemq:orders"/>
  <to uri="log:com.mycompany.order?level=DEBUG"/>
  <to uri="bean:processOrder"/>
</route>
```

204.4. REGULAR LOGGER WITH FORMATTER SAMPLE

In the route below we log the incoming orders at **INFO** level before the order is processed.

```
from("activemq:orders").
    to("log:com.mycompany.order?showAll=true&multiline=true").to("bean:processOrder");
```

204.5. THROUGHPUT LOGGER WITH GROUPSIZE SAMPLE

In the route below we log the throughput of the incoming orders at **DEBUG** level grouped by 10 messages.

```
from("activemq:orders").
    to("log:com.mycompany.order?level=DEBUG&groupSize=10").to("bean:processOrder");
```

204.6. THROUGHPUT LOGGER WITH GROUPINTERVAL SAMPLE

This route will result in message stats logged every 10s, with an initial 60s delay and stats should be displayed even if there isn't any message traffic.

```
from("activemq:orders").
    to("log:com.mycompany.order?
level=DEBUG&groupInterval=10000&groupDelay=60000&groupActiveOnly=false").to("bean:process
Order");
```

The following will be logged:

```
"Received: 1000 new messages, with total 2000 so far. Last group took: 10000 millis which is: 100
messages per second. average: 100"
```

204.7. MASKING SENSITIVE INFORMATION LIKE PASSWORD

Available as of Camel 2.19

You can enable security masking for logging by setting **logMask** flag to **true**. Note that this option also affects Log EIP.

To enable mask in Java DSL at CamelContext level:

```
camelContext.setLogMask(true);
```

And in XML:

```
<camelContext logMask="true">
```

You can also turn it on|off at endpoint level. To enable mask in Java DSL at endpoint level, add logMask=true option in the URI for the log endpoint:

```
from("direct:start").to("log:foo?logMask=true");
```

And in XML:

```
<route>
  <from uri="direct:foo"/>
  <to uri="log:foo?logMask=true"/>
</route>
```

org.apache.camel.processor.DefaultMaskingFormatter is used for the masking by default. If you want to use a custom masking formatter, put it into registry with the name **CamelCustomLogMask**. Note that the masking formatter must implement **org.apache.camel.spi.MaskingFormatter**.

204.8. FULL CUSTOMIZATION OF THE LOGGING OUTPUT

Available as of Camel 2.11

With the options outlined in the #Formatting section, you can control much of the output of the logger. However, log lines will always follow this structure:

```
Exchange[Id:ID-machine-local-50656-1234567901234-1-2, ExchangePattern:InOut,
```

```
Properties:{CamelToEndpoint=log://org.apache.camel.component.log.TEST?showAll=true,
CamelCreatedTimestamp=Thu Mar 28 00:00:00 WET 2013},
Headers:{breadcrumbId=ID-machine-local-50656-1234567901234-1-1}, BodyType:String, Body:Hello
World, Out: null]
```

This format is unsuitable in some cases, perhaps because you need to...

- ... filter the headers and properties that are printed, to strike a balance between insight and verbosity.
- ... adjust the log message to whatever you deem most readable.
- ... tailor log messages for digestion by log mining systems, e.g. Splunk.
- ... print specific body types differently.
- ... etc.

Whenever you require absolute customization, you can create a class that implements the **ExchangeFormatter** interface. Within the **format(Exchange)** method you have access to the full Exchange, so you can select and extract the precise information you need, format it in a custom manner and return it. The return value will become the final log message.

You can have the Log component pick up your custom **ExchangeFormatter** in either of two ways:

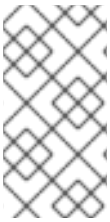
Explicitly instantiating the LogComponent in your Registry:

```
<bean name="log" class="org.apache.camel.component.log.LogComponent">
  <property name="exchangeFormatter" ref="myCustomFormatter" />
</bean>
```

204.8.1. Convention over configuration:*

Simply by registering a bean with the name **logFormatter**; the Log Component is intelligent enough to pick it up automatically.

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter" />
```



NOTE

the **ExchangeFormatter** gets applied to **all Log endpoints within that Camel Context** If you need different ExchangeFormatters for different endpoints, just instantiate the LogComponent as many times as needed, and use the relevant bean name as the endpoint prefix.

From **Camel 2.11.2/2.12** onwards when using a custom log formatter, you can specify parameters in the log uri, which gets configured on the custom log formatter. Though when you do that you should define the "logFormatter" as prototype scoped so its not shared if you have different parameters, eg:

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter" scope="prototype"/>
```

And then we can have Camel routes using the log uri with different options:

```
<to uri="log:foo?param1=foo&amp;param2=100"/>

<to uri="log:bar?param1=bar&amp;param2=200"/>
```

204.9. USING LOG COMPONENT IN OSGI

Improvement as of Camel 2.12.4/2.13.1

When using Log component inside OSGi (e.g., in Karaf), the underlying logging mechanisms are provided by PAX logging. It searches for a bundle which invokes **org.slf4j.LoggerFactory.getLogger()** method and associates the bundle with the logger instance. Without specifying custom **org.sfl4j.Logger** instance, the logger created by Log component is associated with **camel-core** bundle.

In some scenarios it is required that the bundle associated with logger should be the bundle which contains route definition. To do this, either register single instance of **org.slf4j.Logger** in the Registry or reference it using **logger** URI parameter.

204.10. SEE ALSO

- LogEIP for using **log** directly in the DSL for human logs.

CHAPTER 205. LUCENE COMPONENT

Available as of Camel version 2.2

The **lucene** component is based on the Apache Lucene project. Apache Lucene is a powerful high-performance, full-featured text search engine library written entirely in Java. For more details about Lucene, please see the following links

- http://lucene.apache.org/java/docs/
- http://lucene.apache.org/java/docs/features.html

The lucene component in camel facilitates integration and utilization of Lucene endpoints in enterprise integration patterns and scenarios. The lucene component does the following

- builds a searchable index of documents when payloads are sent to the Lucene Endpoint
- facilitates performing of indexed searches in Camel

This component only supports producer endpoints.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-lucene</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

205.1. URI FORMAT

```
lucene:searcherName:insert[?options]
lucene:searcherName:query[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

205.2. INSERT OPTIONS

The Lucene component supports 2 options which are listed below.

Name	Description	Default	Type
config (advanced)	To use a shared lucene configuration		LuceneConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Lucene endpoint is configured using URI syntax:

```
lucene:host:operation
```

with the following path and query parameters:

205.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** The URL to the lucene server		String
operation	**Required** Operation to do such as insert or query.		LuceneOperation

205.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
analyzer (producer)	An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The value for analyzer can be any class that extends the abstract class org.apache.lucene.analysis.Analyzer. Lucene also offers a rich set of analyzers out of the box		Analyzer
indexDir (producer)	A file system directory in which index files are created upon analysis of the document by the specified analyzer		File
maxHits (producer)	An integer value that limits the result set of the search operation		int
srcDir (producer)	An optional directory containing files to be used to be analyzed and added to the index at producer startup.		File
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

205.3. SENDING/RECEIVING MESSAGES TO/FROM THE CACHE

205.3.1. Message Headers

Header	Description
QUERY	The Lucene Query to performed on the index. The query may include wildcards and phrases
RETURN_LUCENE_DOCS	**Camel 2.15:** Set this header to true to include the actual Lucene documentation when returning hit information.

205.3.2. Lucene Producers

This component supports 2 producer endpoints.

insert - The insert producer builds a searchable index by analyzing the body in incoming exchanges and associating it with a token ("content"). **query** - The query producer performs searches on a pre-created index. The query uses the searchable index to perform score & relevance based searches. Queries are sent via the incoming exchange contains a header property name called 'QUERY'. The value of the header property 'QUERY' is a Lucene Query. For more details on how to create Lucene Queries check out http://lucene.apache.org/java/3_0_0/queryparsersyntax.html

205.3.3. Lucene Processor

There is a processor called LuceneQueryProcessor available to perform queries against lucene without the need to create a producer.

205.4. LUCENE USAGE SAMPLES

205.4.1. Example 1: Creating a Lucene index

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start").
            to("lucene:whitespaceQuotesIndex:insert?
                analyzer=#whitespaceAnalyzer&indexDir=#whitespace&srcDir=#load_dir").
            to("mock:result");
    }
};
```

205.4.2. Example 2: Loading properties into the JNDI registry in the Camel Context

```
@Override
protected JndiRegistry createRegistry() throws Exception {
  JndiRegistry registry =
        new JndiRegistry(createJndiContext());
  registry.bind("whitespace", new File("./whitespaceIndexDir"));
  registry.bind("load_dir",
       new File("src/test/resources/sources"));
  registry.bind("whitespaceAnalyzer",
```

```
        new WhitespaceAnalyzer());
  return registry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

205.4.3. Example 2: Performing searches using a Query Producer

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      from("direct:start").
        setHeader("QUERY", constant("Seinfeld")).
        to("lucene:searchIndex:query?
          analyzer=#whitespaceAnalyzer&indexDir=#whitespace&maxHits=20").
        to("direct:next");

      from("direct:next").process(new Processor() {
        public void process(Exchange exchange) throws Exception {
          Hits hits = exchange.getIn().getBody(Hits.class);
          printResults(hits);
        }

        private void printResults(Hits hits) {
           LOG.debug("Number of hits: " + hits.getNumberOfHits());
           for (int i = 0; i < hits.getNumberOfHits(); i++) {
             LOG.debug("Hit " + i + " Index Location:" + hits.getHit().get(i).getHitLocation());
             LOG.debug("Hit " + i + " Score:" + hits.getHit().get(i).getScore());
             LOG.debug("Hit " + i + " Data:" + hits.getHit().get(i).getData());
           }
         }
      }).to("mock:searchResult");
   }
};
```

205.4.4. Example 3: Performing searches using a Query Processor

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      try {
         from("direct:start").
           setHeader("QUERY", constant("Rodney Dangerfield")).
           process(new LuceneQueryProcessor("target/stdindexDir", analyzer, null, 20)).
           to("direct:next");
      } catch (Exception e) {
         e.printStackTrace();
      }

      from("direct:next").process(new Processor() {
         public void process(Exchange exchange) throws Exception {
           Hits hits = exchange.getIn().getBody(Hits.class);
           printResults(hits);
         }

         private void printResults(Hits hits) {
```

```
                LOG.debug("Number of hits: " + hits.getNumberOfHits());
                for (int i = 0; i < hits.getNumberOfHits(); i++) {
                    LOG.debug("Hit " + i + " Index Location:" + hits.getHit().get(i).getHitLocation());
                    LOG.debug("Hit " + i + " Score:" + hits.getHit().get(i).getScore());
                    LOG.debug("Hit " + i + " Data:" + hits.getHit().get(i).getData());
                }
            }
        }).to("mock:searchResult");
    }
};
```

CHAPTER 206. LUMBERJACK COMPONENT

Available as of Camel version 2.18

The **Lumberjack** component retrieves logs sent over the network using the Lumberjack protocol, from Filebeat for instance. The network communication can be secured with SSL.

This component only supports consumer endpoints.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-lumberjack</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

206.1. URI FORMAT

```
lumberjack:host
lumberjack:host:port
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

206.2. OPTIONS

The Lumberjack component supports 3 options which are listed below.

Name	Description	Default	Type
sslContextParameters (security)	Sets the default SSL configuration to use for all the endpoints. You can also configure it directly at the endpoint level.		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Lumberjack endpoint is configured using URI syntax:

```
lumberjack:host:port
```

with the following path and query parameters:

206.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** Network interface on which to listen for Lumberjack		String
port	Network port on which to listen for Lumberjack	5044	int

206.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sslContextParameters (consumer)	SSL configuration		SSLContextParameters
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

206.3. RESULT

The result body is a **Map<String, Object>** object.

206.4. LUMBERJACK USAGE SAMPLES

206.4.1. Example 1: Streaming the log messages

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("lumberjack:0.0.0.0").                  // Listen on all network interfaces using the default port
            setBody(simple("${body[message]}")).     // Select only the log message
            to("stream:out");                        // Write it into the output stream
    }
};
```

CHAPTER 207. LZF DEFLATE COMPRESSION DATAFORMAT

Available as of Camel version 2.17

The LZF Data Format is a message compression and de-compression format. It uses the LZF deflate algorithm. Messages marshalled using LZF compression can be unmarshalled using LZF decompression just prior to being consumed at the endpoint. The compression capability is quite useful when you deal with large XML and Text based payloads or when you read messages previously comressed using LZF algotithm.

207.1. OPTIONS

The LZF Deflate Compression dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
usingParallelCompression	**false**	**Boolean**	Enable encoding (compress) using multiple processing cores.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

207.2. MARSHAL

In this example we marshal a regular text/XML payload to a compressed payload employing LZF compression format and send it an ActiveMQ queue called MY_QUEUE.

```
from("direct:start").marshal().lzf().to("activemq:queue:MY_QUEUE");
```

207.3. UNMARSHAL

In this example we unmarshal a LZF payload from an ActiveMQ queue called MY_QUEUE to its original format, and forward it for processing to the **UnGZippedMessageProcessor**.

```
from("activemq:queue:MY_QUEUE").unmarshal().lzf().process(new
UnCompressedMessageProcessor());
```

207.4. DEPENDENCIES

To useLZF compression in your camel routes you need to add a dependency on **camel-lzf** which implements this data format.

If you use Maven you can just add the following to your **pom.xml**, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
```

```
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-lzf</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 208. MAIL COMPONENT

Available as of Camel version 1.0

The mail component provides access to Email via Spring's Mail support and the underlying JavaMail system.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mail</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

WARNING

Geronimo mail .jar

We have discovered that the geronimo mail **.jar** (v1.6) has a bug when polling mails with attachments. It cannot correctly identify the **Content-Type**. So, if you attach a **.jpeg** file to a mail and you poll it, the **Content-Type** is resolved as **text/plain** and not as **image/jpeg**. For that reason, we have added an **org.apache.camel.component.ContentTypeResolver** SPI interface which enables you to provide your own implementation and fix this bug by returning the correct Mime type based on the file name. So if the file name ends with **jpeg/jpg**, you can return **image/jpeg**.

You can set your custom resolver on the **MailComponent** instance or on the **MailEndpoint** instance.

TIP

POP3 or IMAP POP3 has some limitations and end users are encouraged to use IMAP if possible.

INFO: **Using mock-mail for testing** You can use a mock framework for unit testing, which allows you to test without the need for a real mail server. However you should remember to not include the mock-mail when you go into production or other environments where you need to send mails to a real mail server. Just the presence of the mock-javamail.jar on the classpath means that it will kick in and avoid sending the mails.

208.1. URI FORMAT

Mail endpoints can have one of the following URI formats (for the protocols, SMTP, POP3, or IMAP, respectively):

```
smtp://[username@]host[:port][?options]
pop3://[username@]host[:port][?options]
imap://[username@]host[:port][?options]
```

The mail component also supports secure variants of these protocols (layered over SSL). You can enable the secure protocols by adding **s** to the scheme:

```
smtps://[username@]host[:port][?options]
pop3s://[username@]host[:port][?options]
imaps://[username@]host[:port][?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

208.2.

The Mail component supports 4 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	Sets the Mail configuration		MailConfiguration
contentTypeResolver (advanced)	Resolver to determine Content-Type for file attachments.		ContentTypeResolver
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

208.3.

The Mail endpoint is configured using URI syntax:

```
imap:host:port
```

with the following path and query parameters:

208.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** The mail server host name		String
port	The port number of the mail server		int

208.3.2. Query Parameters (62 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
closeFolder (consumer)	Whether the consumer should close the folder after polling. Setting this option to false and having disconnect=false as well, then the consumer keep the folder open between polls.	true	boolean
copyTo (consumer)	After processing a mail message, it can be copied to a mail folder with the given name. You can override this configuration value, with a header with the key copyTo, allowing you to copy messages to folder names configured at runtime.		String
delete (consumer)	Deletes the messages after they have been processed. This is done by setting the DELETED flag on the mail message. If false, the SEEN flag is set instead. As of Camel 2.10 you can override this configuration option by setting a header with the key delete to determine if the mail should be deleted or not.	false	boolean
disconnect (consumer)	Whether the consumer should disconnect after polling. If enabled this forces Camel to connect on each poll.	false	boolean
handleFailedMessage (consumer)	If the mail consumer cannot retrieve a given mail message, then this option allows to handle the caused exception by the consumer's error handler. By enable the bridge error handler on the consumer, then the Camel routing error handler can handle the exception instead. The default behavior would be the consumer throws an exception and no mails from the batch would be able to be routed by Camel.	false	boolean
maxMessagesPerPoll (consumer)	Specifies the maximum number of messages to gather per poll. By default, no maximum is set. Can be used to set a limit of e.g. 1000 to avoid downloading thousands of files when the server starts up. Set a value of 0 or negative to disable this option.		int

Name	Description	Default	Type
mimeDecodeHeaders (consumer)	This option enables transparent MIME decoding and unfolding for mail headers.	false	boolean
peek (consumer)	Will mark the javax.mail.Message as peeked before processing the mail message. This applies to IMAPMessage messages types only. By using peek the mail will not be eager marked as SEEN on the mail server, which allows us to rollback the mail message if there is an error processing in Camel.	true	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
skipFailedMessage (consumer)	If the mail consumer cannot retrieve a given mail message, then this option allows to skip the message and move on to retrieve the next mail message. The default behavior would be the consumer throws an exception and no mails from the batch would be able to be routed by Camel.	false	boolean
unseen (consumer)	Whether to limit by unseen mails only.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
fetchSize (consumer)	Sets the maximum number of messages to consume during a poll. This can be used to avoid overloading a mail server, if a mailbox folder contains a lot of messages. Default value of -1 means no fetch size and all messages will be consumed. Setting the value to 0 is a special corner case, where Camel will not consume any messages at all.	-1	int
folderName (consumer)	The folder to poll.	INBOX	String
mailUidGenerator (consumer)	A pluggable MailUidGenerator that allows to use custom logic to generate UUID of the mail message.		MailUidGenerator

Name	Description	Default	Type
mapMailMessage (consumer)	Specifies whether Camel should map the received mail message to Camel body/headers. If set to true, the body of the mail message is mapped to the body of the Camel IN message and the mail headers are mapped to IN headers. If this option is set to false then the IN message contains a raw javax.mail.Message. You can retrieve this raw message by calling exchange.getIn().getBody(javax.mail.Message.class).	true	boolean
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
postProcessAction (consumer)	Refers to an MailBoxPostProcessAction for doing post processing tasks on the mailbox once the normal processing ended.		MailBoxPostProcess Action
bcc (producer)	Sets the BCC email address. Separate multiple email addresses with comma.		String
cc (producer)	Sets the CC email address. Separate multiple email addresses with comma.		String
from (producer)	The from email address	camel@localhost	String
replyTo (producer)	The Reply-To recipients (the receivers of the response mail). Separate multiple email addresses with a comma.		String
subject (producer)	The Subject of the message being sent. Note: Setting the subject in the header takes precedence over this option.		String
to (producer)	Sets the To email address. Separate multiple email addresses with comma.		String
javaMailSender (producer)	To use a custom org.apache.camel.component.mail.JavaMailSender for sending emails.		JavaMailSender

Name	Description	Default	Type
additionalJavaMail Properties (advanced)	Sets additional java mail properties, that will append/override any default properties that is set based on all the other options. This is useful if you need to add some special options but want to keep the others as is.		Properties
alternativeBodyHeader (advanced)	Specifies the key to an IN message header that contains an alternative email body. For example, if you send emails in text/html format and want to provide an alternative mail body for non-HTML email clients, set the alternative mail body with this key as a header.	CamelMailAlternativeBody	String
attachmentsContentTransferEncodingResolver (advanced)	To use a custom AttachmentsContentTransferEncodingResolver to resolve what content-type-encoding to use for attachments.		AttachmentsContentTransferEncodingResolver
binding (advanced)	Sets the binding used to convert from a Camel message to and from a Mail message		MailBinding
connectionTimeout (advanced)	The connection timeout in milliseconds.	30000	int
contentType (advanced)	The mail message content type. Use text/html for HTML mails.	text/plain	String
contentTypeResolver (advanced)	Resolver to determine Content-Type for file attachments.		ContentTypeResolver
debugMode (advanced)	Enable debug mode on the underlying mail framework. The SUN Mail framework logs the debug messages to System.out by default.	false	boolean
headerFilterStrategy (advanced)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter headers.		HeaderFilterStrategy
ignoreUnsupportedCharset (advanced)	Option to let Camel ignore unsupported charset in the local JVM when sending mails. If the charset is unsupported then charset=XXX (where XXX represents the unsupported charset) is removed from the content-type and it relies on the platform default instead.	false	boolean

Name	Description	Default	Type
ignoreUriScheme (advanced)	Option to let Camel ignore unsupported charset in the local JVM when sending mails. If the charset is unsupported then charset=XXX (where XXX represents the unsupported charset) is removed from the content-type and it relies on the platform default instead.	false	boolean
session (advanced)	Specifies the mail session that camel should use for all mail interactions. Useful in scenarios where mail sessions are created and managed by some other resource, such as a JavaEE container. If this is not specified, Camel automatically creates the mail session for you.		Session
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
useInlineAttachments (advanced)	Whether to use disposition inline or attachment.	false	boolean
idempotentRepository (filter)	A pluggable repository org.apache.camel.spi.IdempotentRepository which allows to cluster consuming from the same mailbox, and let the repository coordinate whether a mail message is valid for the consumer to process. By default no repository is in use.		String>
idempotentRepositoryRemoveOnCommit (filter)	When using idempotent repository, then when the mail message has been successfully processed and is committed, should the message id be removed from the idempotent repository (default) or be kept in the repository. By default its assumed the message id is unique and has no value to be kept in the repository, because the mail message will be marked as seen/moved or deleted to prevent it from being consumed again. And therefore having the message id stored in the idempotent repository has little value. However this option allows to store the message id, for whatever reason you may have.	true	boolean
searchTerm (filter)	Refers to a javax.mail.search.SearchTerm which allows to filter mails based on search criteria such as subject, body, from, sent after a certain date etc.		SearchTerm
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll.	60000	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

Name	Description	Default	Type
sortTerm (sort)	Sorting order for messages. Only natively supported for IMAP. Emulated to some degree when using POP3 or when IMAP server does not have the SORT capability.		String
dummyTrustManager (security)	To use a dummy security setting for trusting all certificates. Should only be used for development mode, and not production.	false	boolean
password (security)	The password for login		String
sslContextParameters (security)	To configure security using SSLContextParameters.		SSLContextParameters
username (security)	The username for login		String

208.3.3. Sample endpoints

Typically, you specify a URI with login credentials as follows (taking SMTP as an example):

```
smtp://[username@]host[:port][?password=somepwd]
```

Alternatively, it is possible to specify both the user name and the password as query options:

```
smtp://host[:port]?password=somepwd&username=someuser
```

For example:

```
smtp://mycompany.mailserver:30?password=tiger&username=scott
```

208.4. COMPONENTS

- IMAP
- IMAPs
- POP3s
- POP3s
- SMTP
- SMTPs

208.4.1. Default ports

Default port numbers are supported. If the port number is omitted, Camel determines the port number to use based on the protocol.

Protocol	Default Port Number
SMTP	**25**
SMTPS	**465**
POP3	**110**
POP3S	**995**
IMAP	**143**
IMAPS	**993**

208.5. SSL SUPPORT

The underlying mail framework is responsible for providing SSL support. You may either configure SSL/TLS support by completely specifying the necessary Java Mail API configuration options, or you may provide a configured SSLContextParameters through the component or endpoint configuration.

208.5.1. Using the JSSE Configuration Utility

As of **Camel 2.10**, the mail component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the mail component.

Programmatic configuration of the endpoint

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/truststore.jks");
ksp.setPassword("keystorePassword");
TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);
SSLContextParameters scp = new SSLContextParameters();
scp.setTrustManagers(tmp);
Registry registry = ...
registry.bind("sslContextParameters", scp);
...
from(...)
&amp;nbsp; &amp;nbsp; .to("smtps://smtp.google.com?
username=user@gmail.com&password=password&sslContextParameters=#sslContextParameters");
```

Spring DSL based configuration of endpoint

```
...
<camel:sslContextParameters id="sslContextParameters">
  <camel:trustManagers>
    <camel:keyStore resource="/users/home/server/truststore.jks" password="keystorePassword"/>
  </camel:trustManagers>
</camel:sslContextParameters>...
...
<to uri="smtps://smtp.google.com?
username=user@gmail.com&password=password&sslContextParameters=#sslContextParameters"/
>...
```

208.5.2. Configuring JavaMail Directly

Camel uses SUN JavaMail, which only trusts certificates issued by well known Certificate Authorities (the default JVM trust configuration). If you issue your own certificates, you have to import the CA certificates into the JVM's Java trust/key store files, override the default JVM trust/key store files (see **SSLNOTES.txt** in JavaMail for details).

208.6. MAIL MESSAGE CONTENT

Camel uses the message exchange's IN body as the MimeMessage text content. The body is converted to **String.class**.

Camel copies all of the exchange's IN headers to the MimeMessage headers.

The subject of the MimeMessage can be configured using a header property on the IN message. The code below demonstrates this:

The same applies for other MimeMessage headers such as recipients, so you can use a header property as **To**:

Since Camel 2.11 When using the MailProducer the send the mail to server, you should be able to get the message id of the MimeMessage with the key **CamelMailMessageId** from the Camel message header.

208.7. HEADERS TAKE PRECEDENCE OVER PRE-CONFIGURED RECIPIENTS

The recipients specified in the message headers always take precedence over recipients pre-configured in the endpoint URI. The idea is that if you provide any recipients in the message headers, that is what you get. The recipients pre-configured in the endpoint URI are treated as a fallback.

In the sample code below, the email message is sent to **davsclaus@apache.org**, because it takes precedence over the pre-configured recipient, **info@mycompany.com**. Any **CC** and **BCC** settings in the endpoint URI are also ignored and those recipients will not receive any mail. The choice between headers and pre-configured settings is all or nothing: the mail component *either* takes the recipients exclusively from the headers or exclusively from the pre-configured settings. It is not possible to mix and match headers and pre-configured settings.

```
    Map<String, Object> headers = new HashMap<String, Object>();
    headers.put("to", "davsclaus@apache.org");

    template.sendBodyAndHeaders("smtp://admin@localhost?to=info@mycompany.com", "Hello
World", headers);
```

208.8. MULTIPLE RECIPIENTS FOR EASIER CONFIGURATION

It is possible to set multiple recipients using a comma-separated or a semicolon-separated list. This applies both to header settings and to settings in an endpoint URI. For example:

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("to", "davsclaus@apache.org ; jstrachan@apache.org ; ningjiang@apache.org");
```

The preceding example uses a semicolon, **;**, as the separator character.

208.9. SETTING SENDER NAME AND EMAIL

You can specify recipients in the format, **name <email>**, to include both the name and the email address of the recipient.

For example, you define the following headers on the a Message:

```
Map headers = new HashMap();
map.put("To", "Claus Ibsen <davsclaus@apache.org>");
map.put("From", "James Strachan <jstrachan@apache.org>");
map.put("Subject", "Camel is cool");
```

208.10. JAVAMAIL API (EX SUN JAVAMAIL)

JavaMail API is used under the hood for consuming and producing mails.
We encourage end-users to consult these references when using either POP3 or IMAP protocol. Note particularly that POP3 has a much more limited set of features than IMAP.

- JavaMail POP3 API
- JavaMail IMAP API
- And generally about the MAIL Flags

208.11. SAMPLES

We start with a simple route that sends the messages received from a JMS queue as emails. The email account is the **admin** account on **mymailserver.com**.

```
from("jms://queue:subscription").to("smtp://admin@mymailserver.com?password=secret");
```

In the next sample, we poll a mailbox for new emails once every minute. Notice that we use the special **consumer** option for setting the poll interval, **consumer.delay**, as 60000 milliseconds = 60 seconds.

```
from("imap://admin@mymailserver.com
    password=secret&unseen=true&consumer.delay=60000")
   .to("seda://mails");
```

In this sample we want to send a mail to multiple recipients:

208.12. SENDING MAIL WITH ATTACHMENT SAMPLE

WARNING

Attachments are not support by all Camel components The *Attachments API* is based on the Java Activation Framework and is generally only used by the Mail API. Since many of the other Camel components do not support attachments, the attachments could potentially be lost as they propagate along the route. The rule of thumb, therefore, is to add attachments just before sending a message to the mail endpoint.

The mail component supports attachments. In the sample below, we send a mail message containing a plain text message with a logo file attachment.

208.13. SSL SAMPLE

In this sample, we want to poll our Google mail inbox for mails. To download mail onto a local mail client, Google mail requires you to enable and configure SSL. This is done by logging into your Google mail account and changing your settings to allow IMAP access. Google have extensive documentation on how to do this.

```
from("imaps://imap.gmail.com?
username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD"
    + "&delete=false&unseen=true&consumer.delay=60000").to("log:newmail");
```

The preceding route polls the Google mail inbox for new mails once every minute and logs the received messages to the **newmail** logger category.
Running the sample with **DEBUG** logging enabled, we can monitor the progress in the logs:

```
2008-05-08 06:32:09,640 DEBUG MailConsumer - Connecting to MailStore
imaps//imap.gmail.com:993 (SSL enabled), folder=INBOX
2008-05-08 06:32:11,203 DEBUG MailConsumer - Polling mailfolder: imaps//imap.gmail.com:993
(SSL enabled), folder=INBOX
2008-05-08 06:32:11,640 DEBUG MailConsumer - Fetching 1 messages. Total 1 messages.
2008-05-08 06:32:12,171 DEBUG MailConsumer - Processing message: messageNumber=[332],
from=[James Bond <007@mi5.co.uk>], to=YOUR_USERNAME@gmail.com], subject=[...
2008-05-08 06:32:12,187 INFO  newmail - Exchange[MailMessage: messageNumber=[332], from=
[James Bond <007@mi5.co.uk>], to=YOUR_USERNAME@gmail.com], subject=[...
```

208.14. CONSUMING MAILS WITH ATTACHMENT SAMPLE

In this sample we poll a mailbox and store all attachments from the mails as files. First, we define a route to poll the mailbox. As this sample is based on google mail, it uses the same route as shown in the SSL sample:

```
from("imaps://imap.gmail.com?
username=YOUR_USERNAME@gmail.com&password=YOUR_PASSWORD"
    + "&delete=false&unseen=true&consumer.delay=60000").process(new MyMailProcessor());
```

Instead of logging the mail we use a processor where we can process the mail from java code:

-

```
public void process(Exchange exchange) throws Exception {
    // the API is a bit clunky so we need to loop
    Map<String, DataHandler> attachments = exchange.getIn().getAttachments();
    if (attachments.size() > 0) {
        for (String name : attachments.keySet()) {
            DataHandler dh = attachments.get(name);
            // get the file name
            String filename = dh.getName();

            // get the content and convert it to byte[]
            byte[] data = exchange.getContext().getTypeConverter()
                            .convertTo(byte[].class, dh.getInputStream());

            // write the data to a file
            FileOutputStream out = new FileOutputStream(filename);
            out.write(data);
            out.flush();
            out.close();
        }
    }
}
```

As you can see the API to handle attachments is a bit clunky but it's there so you can get the **javax.activation.DataHandler** so you can handle the attachments using standard API.

208.15. HOW TO SPLIT A MAIL MESSAGE WITH ATTACHMENTS

In this example we consume mail messages which may have a number of attachments. What we want to do is to use the Splitter EIP per individual attachment, to process the attachments separately. For example if the mail message has 5 attachments, we want the Splitter to process five messages, each having a single attachment. To do this we need to provide a custom Expression to the Splitter where we provide a List<Message> that contains the five messages with the single attachment.

The code is provided out of the box in Camel 2.10 onwards in the **camel-mail** component. The code is in the class: **org.apache.camel.component.mail.SplitAttachmentsExpression**, which you can find the source code here

In the Camel route you then need to use this Expression in the route as shown below:

If you use XML DSL then you need to declare a method call expression in the Splitter as shown below

```
<split>
  <method beanType="org.apache.camel.component.mail.SplitAttachmentsExpression"/>
  <to uri="mock:split"/>
</split>
```

From Camel 2.16 onwards you can also split the attachments as byte[] to be stored as the message body. This is done by creating the expression with boolean true

```
SplitAttachmentsExpression split = SplitAttachmentsExpression(true);
```

And then use the expression with the splitter eip.

208.16. USING CUSTOM SEARCHTERM

Available as of Camel 2.11

You can configure a **searchTerm** on the **MailEndpoint** which allows you to filter out unwanted mails.

For example to filter mails to contain Camel in either Subject or Text you can do as follows:

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm.subjectOrBody=Camel"/>
  <to uri="bean:myBean"/>
</route>
```

Notice we use the **"searchTerm.subjectOrBody"** as parameter key to indicate that we want to search on mail subject or body, to contain the word "Camel".
The class **org.apache.camel.component.mail.SimpleSearchTerm** has a number of options you can configure:

Or to get the new unseen emails going 24 hours back in time you can do. Notice the "now-24h" syntax. See the table below for more details.

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm.fromSentDate=now-24h"/>
  <to uri="bean:myBean"/>
</route>
```

You can have multiple searchTerm in the endpoint uri configuration. They would then be combined together using AND operator, eg so both conditions must match. For example to get the last unseen emails going back 24 hours which has Camel in the mail subject you can do:

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm.subject=Camel&searchTerm.fromSentDate=now-
24h"/>
  <to uri="bean:myBean"/>
</route>
```

The **SimpleSearchTerm** is designed to be easily configurable from a POJO, so you can also configure it using a <bean> style in XML

```
<bean id="mySearchTerm" class="org.apache.camel.component.mail.SimpleSearchTerm">
  <property name="subject" value="Order"/>
  <property name="to" value="acme-order@acme.com"/>
  <property name="fromSentDate" value="now"/>
 </bean>
```

You can then refer to this bean, using #beanId in your Camel route as shown:

```
<route>
  <from uri="imaps://mymailseerver?
username=foo&password=secret&searchTerm=#mySearchTerm"/>
```

```
  <to uri="bean:myBean"/>
</route>
```

In Java there is a builder class to build compound **SearchTerms** using the **org.apache.camel.component.mail.SearchTermBuilder** class. This allows you to build complex terms such as:

```
// we just want the unseen mails which is not spam
SearchTermBuilder builder = new SearchTermBuilder();

builder.unseen().body(Op.not, "Spam").subject(Op.not, "Spam")
  // which was sent from either foo or bar
  .from("foo@somewhere.com").from(Op.or, "bar@somewhere.com");
  // .. and we could continue building the terms

SearchTerm term = builder.build();
```

208.17. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 209. MASTER COMPONENT

Available as of Camel version 2.20

The **camel-master:** endpoint provides a way to ensure only a single consumer in a cluster consumes from a given endpoint; with automatic failover if that JVM dies.

This can be very useful if you need to consume from some legacy back end which either doesn't support concurrent consumption or due to commercial or stability reasons you can only have a single connection at any point in time.

209.1. USING THE MASTER ENDPOINT

Just prefix any camel endpoint with **master:someName:** where *someName* is a logical name and is used to acquire the master lock. e.g.

```
from("master:cheese:jms:foo").to("activemq:wine");
```

The above simulates the [Exclusive Consumers](http://activemq.apache.org/exclusive-consumer.html) type feature in ActiveMQ; but on any third party JMS provider which maybe doesn't support exclusive consumers.

209.2. URI FORMAT

```
master:namespace:endpoint[?options]
```

Where endpoint is any Camel endpoint you want to run in master/slave mode.

209.3. OPTIONS

The Master component supports 3 options which are listed below.

Name	Description	Default	Type
service (advanced)	Inject the service to use.		CamelClusterService
serviceSelector (advanced)	Inject the service selector used to lookup the CamelClusterService to use.		Selector
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Master endpoint is configured using URI syntax:

```
master:namespace:delegateUri
```

with the following path and query parameters:

209.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
namespace	**Required** The name of the cluster namespace to use		String
delegateUri	**Required** The endpoint uri to use in master/slave mode		String

209.3.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

209.4. EXAMPLE

You can protect a clustered Camel application to only consume files from one active node.

```
// the file endpoint we want to consume from
String url = "file:target/inbox?delete=true";

// use the camel master component in the clustered group named myGroup
// to run a master/slave mode in the following Camel url
from("master:myGroup:" + url)
```

```
    .log(name + " - Received file: ${file:name}")
    .delay(delay)
    .log(name + " - Done file:     ${file:name}")
    .to("file:target/outbox");
```

The master component leverages CamelClusterService you can configure using

- **Java**

```
ZooKeeperClusterService service = new ZooKeeperClusterService();
service.setId("camel-node-1");
service.setNodes("myzk:2181");
service.setBasePath("/camel/cluster");

context.addService(service)
```

- **Xml (Spring/Blueprint)**

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/spring
    http://camel.apache.org/schema/spring/camel-spring.xsd">


 <bean id="cluster"
class="org.apache.camel.component.zookeeper.cluster.ZooKeeperClusterService">
   <property name="id" value="camel-node-1"/>
   <property name="basePath" value="/camel/cluster"/>
   <property name="nodes" value="myzk:2181"/>
 </bean>

 <camelContext xmlns="http://camel.apache.org/schema/spring" autoStartup="false">
   ...
 </camelContext>

</beans>
```

- **Spring boot**

```
camel.component.zookeeper.cluster.service.enabled   = true
camel.component.zookeeper.cluster.service.id        = camel-node-1
camel.component.zookeeper.cluster.service.base-path = /camel/cluster
camel.component.zookeeper.cluster.service.nodes     = myzk:2181
```

209.5. IMPLEMENTATIONS

Camel provide the following ClusterService implementations:

- camel-atomix
- camel-consul

- camel-file
- camel-kubernetes
- camel-zookeeper

209.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 210. METRICS COMPONENT

210.1. METRICS COMPONENT

The **metrics:** component allows to collect various metrics directly from Camel routes. Supported metric types are counter, histogram, meter, timer and gauge. Metrics provides simple way to measure behaviour of application. Configurable reporting backend is enabling different integration options for collecting and visualizing statistics. The component also provides a **MetricsRoutePolicyFactory** which allows to expose route statistics using Dropwizard Metrics, see bottom of page for details.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-metrics</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

210.2. URI FORMAT

```
metrics:[ meter | counter | histogram | timer | gauge ]:metricname[?options]
```

210.3. OPTIONS

The Metrics component supports 2 options which are listed below.

Name	Description	Default	Type
metricRegistry (advanced)	To use a custom configured MetricRegistry.		MetricRegistry
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Metrics endpoint is configured using URI syntax:

```
metrics:metricsType:metricsName
```

with the following path and query parameters:

210.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
metricsType	**Required** Type of metrics		MetricsType
metricsName	**Required** Name of metrics		String

210.3.2. Query Parameters (7 parameters):

Name	Description	Default	Type
action (producer)	Action when using timer type		MetricsTimerAction
decrement (producer)	Decrement value when using counter type		Long
increment (producer)	Increment value when using counter type		Long
mark (producer)	Mark when using meter type		Long
subject (producer)	Subject value when using gauge type		Object
value (producer)	Value value when using histogram type		Long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

210.4. METRIC REGISTRY

Camel Metrics component uses by default a **MetricRegistry** instance with a **Slf4jReporter** that has a 60 second reporting interval. This default registry can be replaced with a custom one by providing a **MetricRegistry** bean. If multiple **MetricRegistry** beans exist in the application, the one with name **metricRegistry** is used.

For example using Spring Java Configuration:

```
@Configuration
public static class MyConfig extends SingleRouteCamelConfiguration {

    @Bean
    @Override
    public RouteBuilder route() {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
```

```
            // define Camel routes here
        }
    };
}

@Bean(name = MetricsComponent.METRIC_REGISTRY_NAME)
public MetricRegistry getMetricRegistry() {
    MetricRegistry registry = ...;
    return registry;
}
}
```

Or using CDI:

```
class MyBean extends RouteBuilder {

    @Override
    public void configure() {
      from("...")
          // Register the 'my-meter' meter in the MetricRegistry below
          .to("metrics:meter:my-meter");
    }

    @Produces
    // If multiple MetricRegistry beans
    // @Named(MetricsComponent.METRIC_REGISTRY_NAME)
    MetricRegistry registry() {
        MetricRegistry registry = new MetricRegistry();
        // ...
        return registry;
    }
}
```

CAUTION

MetricRegistry uses internal thread(s) for reporting. There is no public API in version DropWizard **3.0.1** for users to clean up on exit. Thus using Camel Metrics Component leads to Java classloader leak and my cause **OutOfMemoryErrors** in some cases.

210.5. USAGE

Each metric has type and name. Supported types are counter, histogram, meter, timer and gauge. Metric name is simple string. If metric type is not provided then type meter is used by default.

210.5.1. Headers

Metric name defined in URI can be overridden by using header with name **CamelMetricsName**.

For example

```
from("direct:in")
    .setHeader(MetricsConstants.HEADER_METRIC_NAME, constant("new.name"))
    .to("metrics:counter:name.not.used")
    .to("direct:out");
```

will update counter with name **new.name** instead of **name.not.used**.

All Metrics specific headers are removed from the message once Metrics endpoint finishes processing of exchange. While processing exchange Metrics endpoint will catch all exceptions and write log entry using level **warn**.

210.6. METRICS TYPE COUNTER

```
metrics:counter:metricname[?options]
```

210.6.1. Options

Name	Default	Description
increment	-	Long value to add to the counter
decrement	-	Long value to subtract from the counter

If neither **increment** or **decrement** is defined then counter value will be incremented by one. If **increment** and **decrement** are both defined only increment operation is called.

```
// update counter simple.counter by 7
from("direct:in")
    .to("metric:counter:simple.counter?increment=7")
    .to("direct:out");
```

```
// increment counter simple.counter by 1
from("direct:in")
    .to("metric:counter:simple.counter")
    .to("direct:out");
```

```
// decrement counter simple.counter by 3
from("direct:in")
    .to("metric:counter:simple.counter?decrement=3")
    .to("direct:out");
```

210.6.2. Headers

Message headers can be used to override **increment** and **decrement** values specified in Metrics component URI.

Name	Description	Expected type
Camel Metrics CounterIncrement	Override increment value in URI	Long
Camel Metrics CounterDecrement	Override decrement value in URI	Long

```
// update counter simple.counter by 417
from("direct:in")
    .setHeader(MetricsConstants.HEADER_COUNTER_INCREMENT, constant(417L))
    .to("metric:counter:simple.counter?increment=7")
    .to("direct:out");
```

```
// updates counter using simple language to evaluate body.length
from("direct:in")
    .setHeader(MetricsConstants.HEADER_COUNTER_INCREMENT, simple("${body.length}"))
    .to("metrics:counter:body.length")
    .to("mock:out");
```

210.7. METRIC TYPE HISTOGRAM

```
metrics:histogram:metricname[?options]
```

210.7.1. Options

Name	Default	Description
value	-	Value to use in histogram

If no **value** is not set nothing is added to histogram and warning is logged.

```
// adds value 9923 to simple.histogram
from("direct:in")
    .to("metric:histogram:simple.histogram?value=9923")
    .to("direct:out");
```

```
// nothing is added to simple.histogram; warning is logged
from("direct:in")
```

```
    .to("metric:histogram:simple.histogram")
    .to("direct:out");
```

210.7.2. Headers

Message header can be used to override value specified in Metrics component URI.

Name	Description	Expected type
Camel Metrics HistogramValue	Override histogram value in URI	Long

```
// adds value 992 to simple.histogram
from("direct:in")
    .setHeader(MetricsConstants.HEADER_HISTOGRAM_VALUE, constant(992L))
    .to("metric:histogram:simple.histogram?value=700")
    .to("direct:out")
```

210.8. METRIC TYPE METER

```
metrics:meter:metricname[?options]
```

210.8.1. Options

Name	Default	Description
mark	-	Long value to use as mark

If **mark** is not set then **meter.mark()** is called without argument.

```
// marks simple.meter without value
from("direct:in")
    .to("metric:simple.meter")
    .to("direct:out");
```

```
// marks simple.meter with value 81
from("direct:in")
    .to("metric:meter:simple.meter?mark=81")
    .to("direct:out");
```

210.8.2. Headers

Message header can be used to override **mark** value specified in Metrics component URI.

Name	Description	Expected type
Camel Metrics Meter Mark	Override mark value in URI	Long

```
// updates meter simple.meter with value 345
from("direct:in")
    .setHeader(MetricsConstants.HEADER_METER_MARK, constant(345L))
    .to("metric:meter:simple.meter?mark=123")
    .to("direct:out");
```

210.9. METRICS TYPE TIMER

```
metrics:timer:metricname[?options]
```

210.9.1. Options

Name	Default	Description
action	-	start or stop

If no **action** or invalid value is provided then warning is logged without any timer update. If action **start** is called on already running timer or **stop** is called on not running timer then nothing is updated and warning is logged.

```
// measure time taken by route "calculate"
from("direct:in")
    .to("metrics:timer:simple.timer?action=start")
    .to("direct:calculate")
    .to("metrics:timer:simple.timer?action=stop");
```

TimerContext objects are stored as Exchange properties between different Metrics component calls.

210.9.2. Headers

Message header can be used to override action value specified in Metrics component URI.

Name	Description	Expected type
Camel Metrics TimerAction	Override timer action in URI	**org.apache.camel.component.metrics.timer.TimerEndpoint.TimerAction**

```
// sets timer action using header
from("direct:in")
    .setHeader(MetricsConstants.HEADER_TIMER_ACTION, TimerAction.start)
    .to("metric:timer:simple.timer")
    .to("direct:out");
```

210.10. METRIC TYPE GAUGE

```
metrics:gauge:metricname[?options]
```

210.10.1. Options

Name	Default	Description
subject	-	Any object to be observed by the gauge

If **subject** is not defined it's simply ignored, i.e. the gauge is not registered.

```
// update gauge "simple.gauge" by a bean "mySubjectBean"
from("direct:in")
    .to("metric:gauge:simple.gauge?subject=#mySubjectBean")
    .to("direct:out");
```

210.10.2. Headers

Message headers can be used to override **subject** values specified in Metrics component URI. Note: if **CamelMetricsName** header is specified, then new gauge is registered in addition to default one specified in a URI.

Name	Description	Expected type
Camel Metrics Gauge Subject	Override subject value in URI	Object

```
// update gauge simple.gauge by a String literal "myUpdatedSubject"
from("direct:in")
    .setHeader(MetricsConstants.HEADER_GAUGE_SUBJECT, constant("myUpdatedSubject"))
    .to("metric:counter:simple.gauge?subject=#mySubjectBean")
    .to("direct:out");
```

210.11. METRICSROUTEPOLICYFACTORY

This factory allows to add a RoutePolicy for each route which exposes route utilization statistics using Dropwizard metrics. This factory can be used in Java and XML as the examples below demonstrates.

NOTE

Instead of using the MetricsRoutePolicyFactory you can define a MetricsRoutePolicy per route you want to instrument, in case you only want to instrument a few selected routes.

From Java you just add the factory to the **CamelContext** as shown below:

```
context.addRoutePolicyFactory(new MetricsRoutePolicyFactory());
```

And from XML DSL you define a <bean> as follows:

```
<!-- use camel-metrics route policy to gather metrics for all routes -->
<bean id="metricsRoutePolicyFactory"
class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

The **MetricsRoutePolicyFactory** and **MetricsRoutePolicy** supports the following options:

Name	Default	Description
useJmx	false	Whether to report fine grained statistics to JMX by using the **com.codahale.metrics.JmxReporter**. Notice that if JMX is enabled on CamelContext then a **MetricsRegistryService** mbean is enlisted under the services type in the JMX tree. That mbean has a single operation to output the statistics using json. Setting **useJmx** to true is only needed if you want fine grained mbeans per statistics type.

Name	Default	Description
jmxDomain	org.apache.camel.metrics	The JMX domain name
prettyPrint	false	Whether to use pretty print when outputting statistics in json format
metricsRegistry		Allow to use a shared **com.codahale.metrics.MetricRegistry**. If none is provided then Camel will create a shared instance used by the this CamelContext.
rateUnit	TimeUnit.SECONDS	The unit to use for rate in the metrics reporter or when dumping the statistics as json.
durationUnit	TimeUnit.MILLISECONDS	The unit to use for duration in the metrics reporter or when dumping the statistics as json.
namePattern	**name.routeId.type**	**Camel 2.17:** The name pattern to use. Uses dot as separators, but you can change that. The values **name**, **routeId**, and **type** will be replaced with actual value. Where**name** is the name of the CamelContext. **routeId** is the name of the route. And**type** is the value of responses.

From Java code tou can get hold of the **com.codahale.metrics.MetricRegistry** from the **org.apache.camel.component.metrics.routepolicy.MetricsRegistryService** as shown below:

```
MetricRegistryService registryService = context.hasService(MetricsRegistryService.class);
if (registryService != null) {
  MetricsRegistry registry = registryService.getMetricsRegistry();
  ...
}
```

210.12. METRICSMESSAGEHISTORYFACTORY

Available as of Camel 2.17

This factory allows to use metrics to capture Message History performance statistics while routing messages. It works by using a metrics Timer for each node in all the routes. This factory can be used in Java and XML as the examples below demonstrates.

From Java you just set the factory to the **CamelContext** as shown below:

```
context.setMessageHistoryFactory(new MetricsMessageHistoryFactory());
```

And from XML DSL you define a <bean> as follows:

```
<!-- use camel-metrics message history to gather metrics for all messages being routed -->
<bean id="metricsMessageHistoryFactory"
class="org.apache.camel.component.metrics.messagehistory.MetricsMessageHistoryFactory"/>
```

The following options is supported on the factory:

Name	Default	Description
useJmx	false	Whether to report fine grained statistics to JMX by using the **com.codahale.metrics.JmxReporter**. Notice that if JMX is enabled on CamelContext then a **MetricsRegistryService** mbean is enlisted under the services type in the JMX tree. That mbean has a single operation to output the statistics using json. Setting **useJmx** to true is only needed if you want fine grained mbeans per statistics type.
jmxDomain	org.apache.camel.metrics	The JMX domain name
prettyPrint	false	Whether to use pretty print when outputting statistics in json format
metricsRegistry		Allow to use a shared **com.codahale.metrics.MetricRegistry**. If none is provided then Camel will create a shared instance used by the this CamelContext.
rateUnit	TimeUnit.SECONDS	The unit to use for rate in the metrics reporter or when dumping the statistics as json.
durationUnit	TimeUnit.MILLISECONDS	The unit to use for duration in the metrics reporter or when dumping the statistics as json.
namePattern	**name.routeId.id.type**	The name pattern to use. Uses dot as separators, but you can change that. The values **name**, **routeId**, **type**, and **id** will be replaced with actual value. Where**name** is the name of the CamelContext. **routeId** is the name of the route. The**id** pattern represents the node id. And **type** is the value of history.

At runtime the metrics can be accessed from Java API or JMX which allows to gather the data as json output.

From Java code you can do get the service from the CamelContext as shown:

```
MetricsMessageHistoryService service = context.hasService(MetricsMessageHistoryService.class);
String json = service.dumpStatisticsAsJson();
```

And the JMX API the MBean is registered in the **type=services** tree with **name=MetricsMessageHistoryService**.

210.13. INSTRUMENTEDTHREADPOOLFACTORY

Available as of Camel 2.18

This factory allows you to gather performance information about Camel Thread Pools by injecting a InstrumentedThreadPoolFactory which collects information from inside of Camel. See more details at Advanced configuration of CamelContext using Spring

210.14. SEE ALSO

- The **camel-example-cdi-metrics** example that illustrates the integration between Camel, Metrics and CDI.

CHAPTER 211. OPC UA CLIENT COMPONENT

Available as of Camel version 2.19

The **Milo Client** component provides access to OPC UA servers using the Eclipse Milo™ implementation.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-milo</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

The OPC UA Client component supports 6 options which are listed below.

Name	Description	Default	Type
defaultConfiguration (common)	All default options for client		MiloClientConfiguration
applicationName (common)	Default application name		String
applicationUri (common)	Default application URI		String
productUri (common)	Default product URI		String
reconnectTimeout (common)	Default reconnect timeout		Long
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

211.1. URI FORMAT

The URI syntax of the endpoint is:

```
milo-client:tcp://[user:password@]host:port/path/to/service?node=RAW(nsu=urn:foo:bar;s=item-1)
```

If the server does not use a path, then it is possible to simply omit it:

```
milo-client:tcp://[user:password@]host:port?node=RAW(nsu=urn:foo:bar;s=item-1)
```

If no user credentials are provided the client will switch to anonymous mode.

211.2. URI OPTIONS

All configuration options in the group client are applicable to the shared client instance. Endpoints will share client instances for each endpoint URI. So the first time a request for that endpoint URI is made, the options of the client group are applied. All further instances will be ignored.

If you need alternate options for the same endpoint URI it is possible though to set the clientId option which will by added internally to the endpoint URI in order to select a different shared connection instance. In other words, shared connections located by the combination of endpoint URI and client id.

The OPC UA Client endpoint is configured using URI syntax:

```
milo-client:endpointUri
```

with the following path and query parameters:

211.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
endpointUri	**Required** The OPC UA server endpoint		String

211.2.2. Query Parameters (24 parameters):

Name	Description	Default	Type
clientId (common)	A virtual client id to force the creation of a new connection instance		String
defaultAwaitWrites (common)	Default await setting for writes	false	boolean
node (common)	The node definition (see Node ID)		ExpandedNodeId
samplingInterval (common)	The sampling interval in milliseconds		Double
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
allowedSecurityPolicies (client)	A set of allowed security policy URIs. Default is to accept all and use the highest.		String
applicationName (client)	The application name	Apache Camel adapter for Eclipse Milo	String
applicationUri (client)	The application URI	http://camel.apache.org/EclipseMilo/Client	String
channelLifetime (client)	Channel lifetime in milliseconds		Long
keyAlias (client)	The name of the key in the keystore file		String
keyPassword (client)	The key password		String
keyStorePassword (client)	The keystore password		String
keyStoreType (client)	The key store type		String

Name	Description	Default	Type
keyStoreUrl (client)	The URL where the key should be loaded from		URL
maxPendingPublishRequests (client)	The maximum number of pending publish requests		Long
maxResponseMessageSize (client)	The maximum number of bytes a response message may have		Long
overrideHost (client)	Override the server reported endpoint host with the host from the endpoint URI.	false	boolean
productUri (client)	The product URI	http://camel.apache.org/EclipseMilo	String
requestTimeout (client)	Request timeout in milliseconds		Long
sessionName (client)	Session name		String
sessionTimeout (client)	Session timeout in milliseconds		Long

211.2.3. Node ID

In order to define a target node a namespace and node id is required. In previous versions this was possible by specifying **nodeId** and either **namespaceUri** or **namespaceIndex**. However this only allowed for using string based node IDs. And while this configuration is still possible, the newer one is preferred.

The new approach is to specify a full namespace+node ID in the format **ns=1;i=1** which also allows to use the other node ID formats (like numeric, GUID/UUID or opaque). If the **node** parameter is used the older ones must not be used. The syntax of this node format is a set of **key=value** pairs delimited by a semi-colon (**;**).

Exactly one namespace and one node id key must be used. See the following table for possible keys:

Key	Type	Description
ns	namespace	Numeric namespace index

Key	Type	Description
nsu	namespace	Namespace URI
s	node	String node ID
i	node	Numeric node ID
g	node	GUID/UUID node ID
b	node	Base64 encoded string for opaque node ID

As the values generated by the syntax cannot be transparently encoded into a URI parameter value, it is necessary to escape them. However Camel allows to wrap the actual value inside **RAW(...)**, which makes escaping unnecessary. For example:

```
milo-client://user:password@localhost:12345?node=RAW(nsu=http://foo.bar;s=foo/bar)
```

211.2.4. Security policies

When setting the allowing security policies is it possible to use the well known OPC UA URIs (e.g. **http://opcfoundation.org/UA/SecurityPolicy#Basic128Rsa15**) or to use the Milo enum literals (e.g. **None**). Specifying an unknown security policy URI or enum is an error.

The known security policy URIs and enum literals are can be seen here: SecurityPolicy.java

Note: In any case security policies are considered case sensitive.

211.3. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 212. OPC UA SERVER COMPONENT

Available as of Camel version 2.19

The **Milo Server** component provides an OPC UA server using the Eclipse Milo™ implementation.

Java 8: This component requires Java 8 at runtime.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-milo</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Messages sent to the endpoint from Camel will be available from the OPC UA server to OPC UA Clients. Value write requests from OPC UA Client will trigger messages which are sent into Apache Camel.

The OPC UA Server component supports 19 options which are listed below.

Name	Description	Default	Type
namespaceUri (common)	The URI of the namespace, defaults to urn:org:apache:camel		String
applicationName (common)	The application name		String
applicationUri (common)	The application URI		String
productUri (common)	The product URI		String
bindPort (common)	The TCP port the server binds to		int
strictEndpointUrls Enabled (common)	Set whether strict endpoint URLs are enforced	false	boolean
serverName (common)	Server name		String
hostname (common)	Server hostname		String

Name	Description	Default	Type
securityPolicies (common)	Security policies		Set
securityPoliciesById (common)	Security policies by URI or name		String>
userAuthentication Credentials (common)	Set user password combinations in the form of user1:pwd1,user2:pwd2 Usernames and passwords will be URL decoded		String
enableAnonymous Authentication (common)	Enable anonymous authentication, disabled by default	false	boolean
bindAddresses (common)	Set the addresses of the local addresses the server should bind to		String
buildInfo (common)	Server build info		BuildInfo
serverCertificate (common)	Server certificate		Result
certificateManager (common)	Server certificate manager		CertificateManager
certificateValidator (common)	Validator for client certificates		CertificateValidator>
defaultCertificate Validator (common)	Validator for client certificates using default file based approach		File
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

212.1. URI FORMAT

```
milo-server:itemId[?options]
```

212.2. URI OPTIONS

The OPC UA Server endpoint is configured using URI syntax:

```
milo-server:itemId
```

with the following path and query parameters:

212.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
itemId	**Required** ID of the item		String

212.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

212.3. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 213. MIME MULTIPART DATAFORMAT

Available as of Camel version 2.17

This data format that can convert a Camel message with attachments into a Camel message having a MIME-Multipart message as message body (and no attachments).

The use case for this is to enable the user to send attachments over endpoints that do not directly support attachments, either as special protocol implementation (e.g. send a MIME-multipart over an HTTP endpoint) or as a kind of tunneling solution (e.g. because camel-jms does not support attachments but by marshalling the message with attachments into a MIME-Multipart, sending that to a JMS queue, receiving the message from the JMS queue and unmarshalling it again (into a message body with attachments).

The marshal option of the mime-multipart data format will convert a message with attachments into a MIME-Multipart message. If the parameter "multipartWithoutAttachment" is set to true it will also marshal messages without attachments into a multipart message with a single part, if the parameter is set to false it will leave the message alone.

MIME headers of the mulitpart as "MIME-Version" and "Content-Type" are set as camel headers to the message. If the parameter "headersInline" is set to true it will also create a MIME multipart message in any case.
Furthermore the MIME headers of the multipart are written as part of the message body, not as camel headers.

The unmarshal option of the mime-multipart data format will convert a MIME-Multipart message into a camel message with attachments and leaves other messages alone. MIME-Headers of the MIME-Multipart message have to be set as Camel headers. The unmarshalling will only take place if the "Content-Type" header is set to a "multipart" type. If the option "headersInline" is set to true, the body is always parsed as a MIME message.As a consequence if the message body is a stream and stream caching is not enabled, a message body that is actually not a MIME message with MIME headers in the message body will be replaced by an empty message. Up to Camel version 2.17.1 this will happen all message bodies that do not contain a MIME multipart message regardless of body type and stream cache setting.

213.1. OPTIONS

The MIME Multipart dataformat supports 6 options which are listed below.

Name	Default	Java Type	Description
multipartSubType	**mixed**	**String**	Specify the subtype of the MIME Multipart. Default is mixed.
multipartWithoutAttachment	**false**	**Boolean**	Defines whether a message without attachment is also marshaled into a MIME Multipart (with only one body part). Default is false.
headersInline	**false**	**Boolean**	Defines whether the MIME-Multipart headers are part of the message body (true) or are set as Camel headers (false). Default is false.

Name	Default	Java Type	Description
includeHeaders		**String**	A regex that defines which Camel headers are also included as MIME headers into the MIME multipart. This will only work if headersInline is set to true. Default is to include no headers
binaryContent	**false**	**Boolean**	Defines whether the content of binary parts in the MIME multipart is binary (true) or Base-64 encoded (false) Default is false.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

213.2. MESSAGE HEADERS (MARSHAL)

Name	Type	Description
Message-Id	String	The marshal operation will set this parameter to the generated MIME message id if the "headersInline" parameter is set to false.
MIME-Version	String	The marshal operation will set this parameter to the applied MIME version (1.0) if the "headersInline" parameter is set to false.
Content-Type	String	The content of this header will be used as a content type for the message body part. If no content type is set, "application/octet-stream" is assumed. After the marshal operation the content type is set to "multipart/related" or empty if the "headersInline" parameter is set to true.
Content-Encoding	String	If the incoming content type is "text/*" the content encoding will be set to the encoding parameter of the Content-Type MIME header of the body part. Furthermore the given charset is applied for text to binary conversions.

213.3. MESSAGE HEADERS (UNMARSHAL)

Name	Type	Description

Name	Type	Description
Content-Type	String	If this header is not set to "multipart/*" the unmarshal operation will not do anything. In other cases the multipart will be parsed into a camel message with attachments and the header is set to the Content-Type header of the body part, except if this is application/octet-stream. In the latter case the header is removed.
Content-Encoding	String	If the content-type of the body part contains an encoding parameter this header will be set to the value of this encoding parameter (converted from MIME endoding descriptor to Java encoding descriptor)
MIME-Version	String	The unmarshal operation will read this header and use it for parsing the MIME multipart. The header is removed afterwards

213.4. EXAMPLES

```
from(...).marshal().mimeMultipart()
```

With a message where no Content-Type header is set, will create a Message with the following message Camel headers:

Camel Message Headers

```
Content-Type=multipart/mixed; \n boundary="----=_Part_0_14180567.1447658227051"
Message-Id=<...>
MIME-Version=1.0
```

The message body will be:

Camel Message Body

```
------=_Part_0_14180567.1447658227051
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64
Qm9keSB0ZXh0
------=_Part_0_14180567.1447658227051
Content-Type: application/binary
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="Attachment File Name"
AAECAwQFBgc=
------=_Part_0_14180567.1447658227051--
```

A message with the header Content-Type set to "text/plain" sent to the route

```
from("...").marshal().mimeMultipart("related", true, true, "(included|x-.*)", true);
```

will create a message without any specific MIME headers set as Camel headers (the Content-Type header is removed from the Camel message) and the following message body that includes also all headers of the original message starting with "x-" and the header with name "included":

Camel Message Body

```
Message-ID: <...>
MIME-Version: 1.0
Content-Type: multipart/related;
    boundary="----=_Part_0_1134128170.1447659361365"
x-bar: also there
included: must be included
x-foo: any value

------=_Part_0_1134128170.1447659361365
Content-Type: text/plain
Content-Transfer-Encoding: 8bit

Body text
------=_Part_0_1134128170.1447659361365
Content-Type: application/binary
Content-Transfer-Encoding: binary
Content-Disposition: attachment; filename="Attachment File Name"

[binary content]
------=_Part_0_1134128170.1447659361365
```

213.5. DEPENDENCIES

To use MIME-Multipart in your Camel routes you need to add a dependency on **camel-mail** which implements this data format.

If you use Maven you can just add the following to your pom.xml:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mail</artifactId>
  <version>x.x.x</version> <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 214. MINA2 COMPONENT

Available as of Camel version 2.10

The **mina2:** component is a transport for working with Apache MINA 2.x

TIP

Favor using Netty as Netty is a much more active maintained and popular project than Apache Mina currently is

INFO: Be careful with sync=false on consumer endpoints. Since camel-mina2 all consumer exchanges are InOut. This is different to camel-mina.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mina2</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

214.1. URI FORMAT

```
mina2:tcp://hostname[:port][?options]
mina2:udp://hostname[:port][?options]
mina2:vm://hostname[:port][?options]
```

You can specify a codec in the Registry using the **codec** option. If you are using TCP and no codec is specified then the **textline** flag is used to determine if text line based codec or object serialization should be used instead. By default the object serialization is used.

For UDP if no codec is specified the default uses a basic **ByteBuffer** based codec.

The VM protocol is used as a direct forwarding mechanism in the same JVM.

A Mina producer has a default timeout value of 30 seconds, while it waits for a response from the remote server.

In normal use, **camel-mina** only supports marshalling the body content—message headers and exchange properties are not sent.
However, the option, **transferExchange**, does allow you to transfer the exchange itself over the wire. See options below.

You can append query options to the URI in the following format, **?option=value&option=value&…**

214.2. OPTIONS

The Mina2 component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared mina configuration.		Mina2Configuration
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Mina2 endpoint is configured using URI syntax:

```
mina2:protocol:host:port
```

with the following path and query parameters:

214.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
protocol	**Required** Protocol to use		String
host	**Required** Hostname to use. Use localhost or 0.0.0.0 for local server as consumer. For producer use the hostname or ip address of the remote server.		String
port	**Required** Port number		int

214.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
disconnect (common)	Whether or not to disconnect(close) from Mina session right after use. Can be used for both consumer and producer.	false	boolean
minaLogger (common)	You can enable the Apache MINA logging filter. Apache MINA uses slf4j logging at INFO level to log all input and output.	false	boolean

Name	Description	Default	Type
sync (common)	Setting to set endpoint as one-way or request-response.	true	boolean
timeout (common)	You can configure the timeout that specifies how long to wait for a response from a remote server. The timeout unit is in milliseconds, so 60000 is 60 seconds.	30000	long
writeTimeout (common)	Maximum amount of time it should take to send data to the MINA session. Default is 10000 milliseconds.	10000	long
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
clientMode (consumer)	If the clientMode is true, mina consumer will connect the address as a TCP client.	false	boolean
disconnectOnNoReply (consumer)	If sync is enabled then this option dictates MinaConsumer if it should disconnect where there is no reply to send back.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
noReplyLogLevel (consumer)	If sync is enabled this option dictates MinaConsumer which logging level to use when logging a there is no reply to send back.	WARN	LoggingLevel
cachedAddress (producer)	Whether to create the InetAddress once and reuse. Setting this to false allows to pickup DNS changes in the network.	true	boolean

Name	Description	Default	Type
lazySessionCreation (producer)	Sessions can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.	true	boolean
maximumPoolSize (advanced)	Number of worker threads in the worker pool for TCP and UDP	16	int
orderedThreadPoolExecutor (advanced)	Whether to use ordered thread pool, to ensure events are processed orderly on the same channel.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
transferExchange (advanced)	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
allowDefaultCodec (codec)	The mina component installs a default codec if both, codec is null and textline is false. Setting allowDefaultCodec to false prevents the mina component from installing a default codec as the first element in the filter chain. This is useful in scenarios where another filter must be the first in the filter chain, like the SSL filter.	true	boolean
codec (codec)	To use a custom minda codec implementation.		ProtocolCodecFactory
decoderMaxLineLength (codec)	To set the textline protocol decoder max line length. By default the default value of Mina itself is used which are 1024.	1024	int
encoderMaxLineLength (codec)	To set the textline protocol encoder max line length. By default the default value of Mina itself is used which are Integer.MAX_VALUE.	-1	int
encoding (codec)	You can configure the encoding (a charset name) to use for the TCP textline codec and the UDP protocol. If not provided, Camel will use the JVM default Charset		String

Name	Description	Default	Type
filters (codec)	You can set a list of Mina IoFilters to use.		List
textline (codec)	Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is false, then Object Serialization is assumed over TCP.	false	boolean
textlineDelimiter (codec)	Only used for TCP and if textline=true. Sets the text line delimiter to use. If none provided, Camel will use DEFAULT. This delimiter is used to mark the end of text.		Mina2TextLineDelimiter
autoStartTls (security)	Whether to auto start SSL handshake.	true	boolean
sslContextParameters (security)	To configure SSL security.		SSLContextParameters

214.3. USING A CUSTOM CODEC

See the Mina how to write your own codec. To use your custom codec with **camel-mina**, you should register your codec in the Registry; for example, by creating a bean in the Spring XML file. Then use the **codec** option to specify the bean ID of your codec. See HL7 that has a custom codec.

214.4. SAMPLE WITH SYNC=FALSE

In this sample, Camel exposes a service that listens for TCP connections on port 6200. We use the **textline** codec. In our route, we create a Mina consumer endpoint that listens on port 6200:

```
from("mina2:tcp://localhost:" + port1 + "?textline=true&sync=false").to("mock:result");
```

As the sample is part of a unit test, we test it by sending some data to it on port 6200.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedBodiesReceived("Hello World");

template.sendBody("mina2:tcp://localhost:" + port1 + "?textline=true&sync=false", "Hello World");

assertMockEndpointsSatisfied();
```

214.5. SAMPLE WITH SYNC=TRUE

In the next sample, we have a more common use case where we expose a TCP service on port 6201 also use the textline codec. However, this time we want to return a response, so we set the **sync** option to **true** on the consumer.

```
from("mina2:tcp://localhost:" + port2 + "?textline=true&sync=true").process(new Processor() {
```

```
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody("Bye " + body);
    }
});
```

Then we test the sample by sending some data and retrieving the response using the **template.requestBody()** method. As we know the response is a **String**, we cast it to **String** and can assert that the response is, in fact, something we have dynamically set in our processor code logic.

```
String response = (String)template.requestBody("mina2:tcp://localhost:" + port2 + "?
textline=true&sync=true", "World");
assertEquals("Bye World", response);
```

214.6. SAMPLE WITH SPRING DSL

Spring DSL can, of course, also be used for MINA. In the sample below we expose a TCP server on port 5555:

```
<route>
  <from uri="mina2:tcp://localhost:5555?textline=true"/>
  <to uri="bean:myTCPOrderHandler"/>
</route>
```

In the route above, we expose a TCP server on port 5555 using the textline codec. We let the Spring bean with ID, **myTCPOrderHandler**, handle the request and return a reply. For instance, the handler bean could be implemented as follows:

```
public String handleOrder(String payload) {
    ...
    return "Order: OK"
}
```

214.7. CLOSING SESSION WHEN COMPLETE

When acting as a server you sometimes want to close the session when, for example, a client conversion is finished. To instruct Camel to close the session, you should add a header with the key **CamelMinaCloseSessionWhenComplete** set to a boolean **true** value.

For instance, the example below will close the session after it has written the **bye** message back to the client:

```
    from("mina2:tcp://localhost:8080?sync=true&textline=true").process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            String body = exchange.getIn().getBody(String.class);
            exchange.getOut().setBody("Bye " + body);

exchange.getOut().setHeader(Mina2Constants.MINA_CLOSE_SESSION_WHEN_COMPLETE,
true);
        }
    });
```

214.8. GET THE IOSESSION FOR MESSAGE

You can get the IoSession from the message header with this key **Mina2Constants.MINA_IOSESSION**, and also get the local host address with the key **Mina2Constants.MINA_LOCAL_ADDRESS** and remote host address with the key **Mina2Constants.MINA_REMOTE_ADDRESS**.

214.9. CONFIGURING MINA FILTERS

Filters permit you to use some Mina Filters, such as **SslFilter**. You can also implement some customized filters. Please note that **codec** and **logger** are also implemented as Mina filters of type, **IoFilter**. Any filters you may define are appended to the end of the filter chain; that is, after **codec** and **logger**.

214.10. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Netty

CHAPTER 215. MLLP COMPONENT

Available as of Camel version 2.17

The MLLP component is specifically designed to handle the nuances of the MLLP protocol and provide the functionality required by Healthcare providers to communicate with other systems using the MLLP protocol. The MLLP component provides a simple configuration URI, automated HL7 acknowledgment generation and automatic acknowledgement interrogation.

The MLLP protocol does not typically use a large number of concurrent TCP connections - a single active TCP connection is the normal case. Therefore, the MLLP component uses a simple thread-per-connection model based an standard Java Sockets. This keeps the implementation simple and eliminates the dependencies other than Camel itself.

The component supports the following:

- A Camel consumer using a TCP Server
- A Camel producer using a TCP Client

The MLLP component uses byte[] payloads, and relies on Camel Type Conversion to convert byte[] to other types.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mllp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

215.1. MLLP OPTIONS

The MLLP component supports 5 options which are listed below.

Name	Description	Default	Type
logPhi (advanced)	Set the component to log PHI data.	true	Boolean
logPhiMaxBytes (advanced)	Set the maximum number of bytes of PHI that will be logged in a log entry.	5120	Integer
defaultCharset (advanced)	Set the default character set to use for byte to/from String conversions.	ISO-8859-1	String
configuration (common)	Sets the default configuration to use when creating MLLP endpoints.		MllpConfiguration

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The MLLP endpoint is configured using URI syntax:

```
mllp:hostname:port
```

with the following path and query parameters:

215.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
hostname	**Required** Hostname or IP for connection for the TCP connection. The default value is null, which means any local IP address		String
port	**Required** Port number for the TCP connection		int

215.1.2. Query Parameters (27 parameters):

Name	Description	Default	Type
autoAck (common)	Enable/Disable the automatic generation of a MLLP Acknowledgement MLLP Consumers only	true	boolean
bufferWrites (common)	**Deprecated** Enable/Disable the buffering of HL7 payloads before writing to the socket.	false	boolean
hl7Headers (common)	Enable/Disable the automatic generation of message headers from the HL7 Message MLLP Consumers only	true	boolean
requireEndOfData (common)	Enable/Disable strict compliance to the MLLP standard. The MLLP standard specifies START_OF_BLOCKhl7 payloadEND_OF_BLOCKEND_OF_DATA, however, some systems do not send the final END_OF_DATA byte. This setting controls whether or not the final END_OF_DATA byte is required or optional.	true	boolean

Name	Description	Default	Type
stringPayload (common)	Enable/Disable converting the payload to a String. If enabled, HL7 Payloads received from external systems will be validated converted to a String. If the charsetName property is set, that character set will be used for the conversion. If the charsetName property is not set, the value of MSH-18 will be used to determine th appropriate character set. If MSH-18 is not set, then the default ISO-8859-1 character set will be use.	true	boolean
validatePayload (common)	Enable/Disable the validation of HL7 Payloads If enabled, HL7 Payloads received from external systems will be validated (see Hl7Util.generateInvalidPayloadExceptionMessage for details on the validation). If and invalid payload is detected, a MllpInvalidMessageException (for consumers) or a MllpInvalidAcknowledgementException will be thrown.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to receive incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. If disabled, the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions by logging them at WARN or ERROR level and ignored.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.	InOut	ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used (this component only supports synchronous operations).	true	boolean
backlog (tcp)	The maximum queue length for incoming connection indications (a request to connect) is set to the backlog parameter. If a connection indication arrives when the queue is full, the connection is refused.	5	Integer

Name	Description	Default	Type
lenientBind (tcp)	TCP Server Only - Allow the endpoint to start before the TCP ServerSocket is bound. In some environments, it may be desirable to allow the endpoint to start before the TCP ServerSocket is bound.	false	boolean
maxConcurrentConsumers (tcp)	The maximum number of concurrent MLLP Consumer connections that will be allowed. If a new connection is received and the maximum is number are already established, the new connection will be reset immediately.	5	int
reuseAddress (tcp)	Enable/disable the SO_REUSEADDR socket option.	false	Boolean
acceptTimeout (timeout)	Timeout (in milliseconds) while waiting for a TCP connection TCP Server Only	60000	int
bindRetryInterval (timeout)	TCP Server Only - The number of milliseconds to wait between bind attempts	5000	int
bindTimeout (timeout)	TCP Server Only - The number of milliseconds to retry binding to a server port	30000	int
connectTimeout (timeout)	Timeout (in milliseconds) for establishing for a TCP connection TCP Client only	30000	int
idleTimeout (timeout)	The approximate idle time allowed before the Client TCP Connection will be reset. A null value or a value less than or equal to zero will disable the idle timeout.		Integer
maxReceiveTimeouts (timeout)	**Deprecated** The maximum number of timeouts (specified by receiveTimeout) allowed before the TCP Connection will be reset.		Integer
keepAlive (tcp)	Enable/disable the SO_KEEPALIVE socket option.	true	Boolean
receiveBufferSize (tcp)	Sets the SO_RCVBUF option to the specified value (in bytes)	8192	Integer
sendBufferSize (tcp)	Sets the SO_SNDBUF option to the specified value (in bytes)	8192	Integer
tcpNoDelay (tcp)	Enable/disable the TCP_NODELAY socket option.	true	Boolean
readTimeout (timeout)	The SO_TIMEOUT value (in milliseconds) used after the start of an MLLP frame has been received	5000	int

Name	Description	Default	Type
receiveTimeout (timeout)	The SO_TIMEOUT value (in milliseconds) used when waiting for the start of an MLLP frame	15000	int
charsetName (codec)	Set the CamelCharsetName property on the exchange		String

215.2. MLLP CONSUMER

The MLLP Consumer supports receiving MLLP-framed messages and sending HL7 Acknowledgements. The MLLP Consumer can automatically generate the HL7 Acknowledgement (HL7 Application Acknowledgements only - AA, AE and AR), or the acknowledgement can be specified using the CamelMllpAcknowledgement exchange property. Additionally, the type of acknowledgement that will be generated can be controlled by setting the CamelMllpAcknowledgementType exchange property.

215.3. MESSAGE HEADERS

The MLLP Consumer adds these headers on the Camel message:

Key	Description	
CamelMllpLocalAddress	The local TCP Address of the Socket	
CamelMllpRemoteAddress	The local TCP Address of the Socket	
CamelMllpSendingApplication	MSH-3 value	
CamelMllpSendingFacility	MSH-4 value	
CamelMllpReceivingApplication	MSH-5 value	
CamelMllpReceivingFacility	MSH-6 value	
CamelMllpTimestamp	MSH-7 value	
CamelMllpSecurity	MSH-8 value	
CamelMllpMessageType	MSH-9 value	
CamelMllpEventType	MSH-9-1 value	
CamelMllpTriggerEvent	MSH-9-2 value	

CamelMllpMessageControlId	MSH-10 value	
CamelMllpProcessingId	MSH-11 value	
CamelMllpVersionId	MSH-12 value	
CamelMllpCharset	MSH-18 value	

All headers are String types. If a header value is missing, its value is null.

215.4. EXCHANGE PROPERTIES

The type of acknowledgment the MLLP Consumer generates and state of the TCP Socket can be controlled by these properties on the Camel exchange:

Key	Type	Description
CamelMllpAcknowledgement	byte[]	If present, this property will we sent to client as the MLLP Acknowledgement
CamelMllpAcknowledgementString	String	If present and CamelMllpAcknowledgement is not present, this property will we sent to client as the MLLP Acknowledgement
CamelMllpAcknowledgementMsaText	String	If neither CamelMllpAcknowledgement or CamelMllpAcknowledgementString are present and autoAck is true, this property can be used to specify the the contents of MSA-3 in the generated HL7 acknowledgement
CamelMllpAcknowledgementType	String	If neither CamelMllpAcknowledgement or CamelMllpAcknowledgementString are present and autoAck is true, this property can be used to specify the HL7 acknowledgement type (i.e. AA, AE, AR)
CamelMllpAutoAcknowledge	Boolean	Overrides the autoAck query parameter
CamelMllpCloseConnectionBeforeSend	Boolean	If true, the Socket will be closed before sending data

CamelMllpResetConnectionBeforeSend	Boolean	If true, the Socket will be reset before sending data
CamelMllpCloseConnectionAfterSend	Boolean	If true, the Socket will be closed immediately after sending data
CamelMllpResetConnectionAfterSend	Boolean	If true, the Socket will be reset immediately after sending any data

215.5. MLLP PRODUCER

The MLLP Producer supports sending MLLP-framed messages and receiving HL7 Acknowledgements. The MLLP Producer interrogates the HL7 Acknowledgments and raises exceptions if a negative acknowledgement is received. The received acknowledgement is interrogated and an exception is raised in the event of a negative acknowledgement.

215.6. MESSAGE HEADERS

The MLLP Producer adds these headers on the Camel message:

Key	Description	
CamelMllpLocalAddress	The local TCP Address of the Socket	
CamelMllpRemoteAddress	The remote TCP Address of the Socket	
CamelMllpAcknowledgement	The HL7 Acknowledgment byte[] received	
CamelMllpAcknowledgementString	The HL7 Acknowledgment received, converted to a String	

215.7. EXCHANGE PROPERTIES

The state of the TCP Socket can be controlled by these properties on the Camel exchange:

Key	Type	Description
CamelMllpCloseConnectionBeforeSend	Boolean	If true, the Socket will be closed before sending data

CamelMllpResetConnectionBeforeSend	Boolean	If true, the Socket will be reset before sending data
CamelMllpCloseConnectionAfterSend	Boolean	If true, the Socket will be closed immediately after sending data
CamelMllpResetConnectionAfterSend	Boolean	If true, the Socket will be reset immediately after sending any data

CHAPTER 216. MOCK COMPONENT

Documentation for Mock component is currently unavailable.

CHAPTER 217. MONGODB COMPONENT

Available as of Camel version 2.10

According to Wikipedia: "NoSQL is a movement promoting a loosely defined class of non-relational data stores that break with a long history of relational databases and ACID guarantees." NoSQL solutions have grown in popularity in the last few years, and major extremely-used sites and services such as Facebook, LinkedIn, Twitter, etc. are known to use them extensively to achieve scalability and agility.

Basically, NoSQL solutions differ from traditional RDBMS (Relational Database Management Systems) in that they don't use SQL as their query language and generally don't offer ACID-like transactional behaviour nor relational data. Instead, they are designed around the concept of flexible data structures and schemas (meaning that the traditional concept of a database table with a fixed schema is dropped), extreme scalability on commodity hardware and blazing-fast processing.

MongoDB is a very popular NoSQL solution and the camel-mongodb component integrates Camel with MongoDB allowing you to interact with MongoDB collections both as a producer (performing operations on the collection) and as a consumer (consuming documents from a MongoDB collection).

MongoDB revolves around the concepts of documents (not as is office documents, but rather hierarchical data defined in JSON/BSON) and collections. This component page will assume you are familiar with them. Otherwise, visit http://www.mongodb.org/.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mongodb</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

217.1. URI FORMAT

```
mongodb:connectionBean?
database=databaseName&collection=collectionName&operation=operationName[&moreOptions...]
```

217.2. MONGODB OPTIONS

The MongoDB component has no options.

The MongoDB endpoint is configured using URI syntax:

```
mongodb:connectionBean
```

with the following path and query parameters:

217.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
connectionBean	**Required** Name of com.mongodb.Mongo to use.		String

217.2.2. Query Parameters (23 parameters):

Name	Description	Default	Type
collection (common)	Sets the name of the MongoDB collection to bind to this endpoint		String
collectionIndex (common)	Sets the collection index (JSON FORMAT : field1 : order1, field2 : order2)		String
createCollection (common)	Create collection during initialisation if it doesn't exist. Default is true.	true	boolean
database (common)	Sets the name of the MongoDB database to target		String
operation (common)	Sets the operation this endpoint will execute against MongoDB. For possible values, see MongoDbOperation.		MongoDbOperation
outputType (common)	Convert the output of the producer to the selected type : DBObjectList DBObject or DBCursor. DBObjectList or DBCursor applies to findAll and aggregate. DBObject applies to all other operations.		MongoDbOutputType
writeConcern (common)	Set the WriteConcern for write operations on MongoDB using the standard ones. Resolved from the fields of the WriteConcern class by calling the link WriteConcernvalueOf(String) method.	ACKNOWLEDGED	WriteConcern
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
cursorRegenerationDelay (advanced)	MongoDB tailable cursors will block until new data arrives. If no new data is inserted, after some time the cursor will be automatically freed and closed by the MongoDB server. The client is expected to regenerate the cursor if needed. This value specifies the time to wait before attempting to fetch a new cursor, and if the attempt fails, how long before the next attempt is made. Default value is 1000ms.	1000	long
dynamicity (advanced)	Sets whether this endpoint will attempt to dynamically resolve the target database and collection from the incoming Exchange properties. Can be used to override at runtime the database and collection specified on the otherwise static endpoint URI. It is disabled by default to boost performance. Enabling it will take a minimal performance hit.	false	boolean
readPreference (advanced)	Sets a MongoDB ReadPreference on the Mongo connection. Read preferences set directly on the connection will be overridden by this setting. The link ReadPreferencevalueOf(String) utility method is used to resolve the passed readPreference value. Some examples for the possible values are nearest, primary or secondary etc.		ReadPreference
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
writeResultAsHeader (advanced)	In write operations, it determines whether instead of returning WriteResult as the body of the OUT message, we transfer the IN message to the OUT and attach the WriteResult as a header.	false	boolean
persistentId (tail)	One tail tracking collection can host many trackers for several tailable consumers. To keep them separate, each tracker should have its own unique persistentId.		String

Name	Description	Default	Type
persistentTailTracking (tail)	Enable persistent tail tracking, which is a mechanism to keep track of the last consumed message across system restarts. The next time the system is up, the endpoint will recover the cursor from the point where it last stopped slurping records.	false	boolean
persistRecords (tail)	Sets the number of tailed records after which the tail tracking data is persisted to MongoDB.	-1	int
tailTrackCollection (tail)	Collection where tail tracking information will be persisted. If not specified, link MongoDbTailTrackingConfigDEFAULT_COLLECTION will be used by default.		String
tailTrackDb (tail)	Indicates what database the tail tracking mechanism will persist to. If not specified, the current database will be picked by default. Dynamicity will not be taken into account even if enabled, i.e. the tail tracking database will not vary past endpoint initialisation.		String
tailTrackField (tail)	Field where the last tracked value will be placed. If not specified, link MongoDbTailTrackingConfigDEFAULT_FIELD will be used by default.		String
tailTrackIncreasingField (tail)	Correlation field in the incoming record which is of increasing nature and will be used to position the tailing cursor every time it is generated. The cursor will be (re)created with a query of type: tailTrackIncreasingField lastValue (possibly recovered from persistent tail tracking). Can be of type Integer, Date, String, etc. NOTE: No support for dot notation at the current time, so the field should be at the top level of the document.		String
tailTrackingStrategy (tail)	Sets the strategy used to extract the increasing field value and to create the query to position the tail cursor.	LITERAL	MongoDBTailTracking Enum

217.3. CONFIGURATION OF DATABASE IN SPRING XML

The following Spring XML creates a bean defining the connection to a MongoDB instance.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <bean id="mongoBean" class="com.mongodb.Mongo">
        <constructor-arg name="host" value="${mongodb.host}" />
        <constructor-arg name="port" value="${mongodb.port}" />
    </bean>
</beans>
```

217.4. SAMPLE ROUTE

The following route defined in Spring XML executes the operation **dbStats** on a collection.

Get DB stats for specified collection

```
<route>
  <from uri="direct:start" />
  <!-- using bean 'mongoBean' defined above -->
  <to uri="mongodb:mongoBean?
database=${mongodb.database}&amp;collection=${mongodb.collection}&amp;operation=getDbStats"
/>
  <to uri="direct:result" />
</route>
```

217.5. MONGODB OPERATIONS - PRODUCER ENDPOINTS

217.5.1. Query operations

217.5.1.1. findById

This operation retrieves only one element from the collection whose _id field matches the content of the IN message body. The incoming object can be anything that has an equivalent to a BSON type. See http://bsonspec.org//specification[http://bsonspec.org//specification] and http://www.mongodb.org/display/DOCS/Java+Types.

```
from("direct:findById")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findById")
    .to("mock:resultFindById");
```

TIP

Supports optional parameters. This operation supports specifying a fields filter. See Specifying optional parameters.

217.5.1.2. findOneByQuery

Use this operation to retrieve just one element from the collection that matches a MongoDB query. **The query object is extracted from the IN message body**, i.e. it should be of type **DBObject** or convertible to **DBObject**. It can be a JSON String or a Hashmap. See #Type conversions for more info.

Example with no query (returns any object of the collection):

```
from("direct:findOneByQuery")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

Example with a query (returns one matching result):

```
from("direct:findOneByQuery")
    .setBody().constant("{ \"name\": \"Raul Kripalani\" }")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

TIP

Supports optional parameters. This operation supports specifying a fields filter and/or a sort clause. See Specifying optional parameters.

217.5.1.3. findAll

The **findAll** operation returns all documents matching a query, or none at all, in which case all documents contained in the collection are returned. **The query object is extracted from the IN message body**, i.e. it should be of type **DBObject** or convertible to **DBObject**. It can be a JSON String or a Hashmap. See #Type conversions for more info.

Example with no query (returns all object in the collection):

```
from("direct:findAll")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findAll")
    .to("mock:resultFindAll");
```

Example with a query (returns all matching results):

```
from("direct:findAll")
    .setBody().constant("{ \"name\": \"Raul Kripalani\" }")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=findAll")
    .to("mock:resultFindAll");
```

Paging and efficient retrieval is supported via the following headers:

Header key	Quick constant	Description (extracted from Mongo DB API doc)	Expected type

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbNumToSkip	**MongoDbConstants.NUM_TO_SKIP**	Discards a given number of elements at the beginning of the cursor.	int/Integer
CamelMongoDbLimit	**MongoDbConstants.LIMIT**	Limits the number of elements returned.	int/Integer
CamelMongoDbBatchSize	**MongoDbConstants.BATCH_SIZE**	Limits the number of elements returned in one batch. A cursor typically fetches a batch of result objects and store them	int/Integer

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
		locally. If batchSize is positive, it represents the size of each batch of objects retrieved. It can be adjusted to optimize performance and limit data transfer. If batchSize is negative, it will limit of number objects returned, that fit within the max batch size limit (usually 4MB), and cursor will be closed. For example if batchSize is -10, then	

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
		and as many as can fit in 4MB, then close the cursor. Note that this feature is different from limit() in that documents must fit within a maximum size, and it removes the need to send a request to close the cursor server-side. The batch size can be changed even after a cursor is iterated, in	

Header key	Quick constant	Description (extracted from Mongo DB API doc)	Expected type
		which case the setting will apply on the next batch retrieval.	

You can also "stream" the documents returned from the server into your route by including outputType=DBCursor (Camel 2.16+) as an endpoint option which may prove simpler than setting the above headers. This hands your Exchange the DBCursor from the Mongo driver, just as if you were executing the findAll() within the Mongo shell, allowing your route to iterate over the results. By default and without this option, this component will load the documents from the driver's cursor into a List and return this to your route - which may result in a large number of in-memory objects. Remember, with a DBCursor do not ask for the number of documents matched - see the MongoDB documentation site for details.

Example with option outputType=DBCursor and batch size :

```
from("direct:findAll")
    .setHeader(MongoDbConstants.BATCH_SIZE).constant(10)
    .setBody().constant("{ \"name\": \"Raul Kripalani\" }")
    .to("mongodb:myDb?
database=flights&collection=tickets&operation=findAll&outputType=DBCursor")
    .to("mock:resultFindAll");
```

The **findAll** operation will also return the following OUT headers to enable you to iterate through result pages if you are using paging:

Header key	Quick constant	Description (extracted from Mongo DB API doc)	Data type

Header key	Quick constant	Description (extracted from MongoDB API doc)	Data type
CamelMongoDbResultTotalSize	**MongoDbConstants.RESULT_TOTAL_SIZE**	Number of objects matching the query. This does not take limit/skip into consideration.	int/Integer
CamelMongoDbResultPageSize	**MongoDbConstants.RESULT_PAGE_SIZE**	Number of objects matching the query. This does not take limit/skip into consideration.	int/Integer

TIP

Supports optional parameters. This operation supports specifying a fields filter and/or a sort clause. See Specifying optional parameters.

217.5.1.4. count

Returns the total number of objects in a collection, returning a Long as the OUT message body. The following example will count the number of records in the "dynamicCollectionName" collection. Notice how dynamicity is enabled, and as a result, the operation will not run against the "notableScientists" collection, but against the "dynamicCollectionName" collection.

```
// from("direct:count").to("mongodb:myDb?
database=tickets&collection=flights&operation=count&dynamicity=true");
Long result = template.requestBodyAndHeader("direct:count", "irrelevantBody",
MongoDbConstants.COLLECTION, "dynamicCollectionName");
assertTrue("Result is not of type Long", result instanceof Long);
```

From **Camel 2.14** onwards you can provide a **com.mongodb.DBObject** object in the message body as a query, and operation will return the amount of documents matching this criteria.

```
DBObject query = ...
Long count = template.requestBodyAndHeader("direct:count", query,
MongoDbConstants.COLLECTION, "dynamicCollectionName");
```

217.5.1.5. Specifying a fields filter (projection)

Query operations will, by default, return the matching objects in their entirety (with all their fields). If your documents are large and you only require retrieving a subset of their fields, you can specify a field filter in all query operations, simply by setting the relevant **DBObject** (or type convertible to **DBObject**, such as a JSON String, Map, etc.) on the **CamelMongoDbFieldsFilter** header, constant shortcut: **MongoDbConstants.FIELDS_FILTER**.

Here is an example that uses MongoDB's BasicDBObjectBuilder to simplify the creation of DBObjects. It retrieves all fields except **_id** and **boringField**:

```
// route: from("direct:findAll").to("mongodb:myDb?
database=flights&collection=tickets&operation=findAll")
DBObject fieldFilter = BasicDBObjectBuilder.start().add("_id", 0).add("boringField", 0).get();
Object result = template.requestBodyAndHeader("direct:findAll", (Object) null,
MongoDbConstants.FIELDS_FILTER, fieldFilter);
```

217.5.1.6. Specifying a sort clause

There is a often a requirement to fetch the min/max record from a collection based on sorting by a particular field. In Mongo the operation is performed using syntax similar to:

```
db.collection.find().sort({_id: -1}).limit(1)
// or
db.collection.findOne({$query:{},$orderby:{_id:-1}})
```

In a Camel route the SORT_BY header can be used with the findOneByQuery operation to achieve the same result. If the FIELDS_FILTER header is also specified the operation will return a single field/value pair that can be passed directly to another component (for example, a parameterized MyBatis SELECT query). This example demonstrates fetching the temporally newest document from a collection and reducing the result to a single field, based on the **documentTimestamp** field:

```
.from("direct:someTriggeringEvent")
.setHeader(MongoDbConstants.SORT_BY).constant("{\"documentTimestamp\": -1}")
.setHeader(MongoDbConstants.FIELDS_FILTER).constant("{\"documentTimestamp\": 1}")
.setBody().constant("{}")
```

```
.to("mongodb:myDb?database=local&collection=myDemoCollection&operation=findOneByQuery")
.to("direct:aMyBatisParameterizedSelect")
;
```

217.5.2. Create/update operations

217.5.2.1. insert

Inserts an new object into the MongoDB collection, taken from the IN message body. Type conversion is attempted to turn it into **DBObject** or a **List**.
Two modes are supported: single insert and multiple insert. For multiple insert, the endpoint will expect a List, Array or Collections of objects of any type, as long as they are - or can be converted to - **DBObject**. All objects are inserted at once. The endpoint will intelligently decide which backend operation to invoke (single or multiple insert) depending on the input.

Example:

```
from("direct:insert")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=insert");
```

The operation will return a WriteResult, and depending on the **WriteConcern** or the value of the **invokeGetLastError** option, **getLastError()** would have been called already or not. If you want to access the ultimate result of the write operation, you need to retrieve the **CommandResult** by calling **getLastError()** or **getCachedLastError()** on the **WriteResult**. Then you can verify the result by calling **CommandResult.ok()**, **CommandResult.getErrorMessage()** and/or **CommandResult.getException()**.

Note that the new object's **_id** must be unique in the collection. If you don't specify the value, MongoDB will automatically generate one for you. But if you do specify it and it is not unique, the insert operation will fail (and for Camel to notice, you will need to enable invokeGetLastError or set a WriteConcern that waits for the write result).

This is not a limitation of the component, but it is how things work in MongoDB for higher throughput. If you are using a custom **_id**, you are expected to ensure at the application level that is unique (and this is a good practice too).

Since Camel **2.15**: OID(s) of the inserted record(s) is stored in the message header under **CamelMongoOid** key (**MongoDbConstants.OID** constant). The value stored is **org.bson.types.ObjectId** for single insert or **java.util.List<org.bson.types.ObjectId>** if multiple records have been inserted.

217.5.2.2. save

The save operation is equivalent to an *upsert* (UPdate, inSERT) operation, where the record will be updated, and if it doesn't exist, it will be inserted, all in one atomic operation. MongoDB will perform the matching based on the _id field.

Beware that in case of an update, the object is replaced entirely and the usage of MongoDB's $modifiers is not permitted. Therefore, if you want to manipulate the object if it already exists, you have two options:

1. perform a query to retrieve the entire object first along with all its fields (may not be efficient), alter it inside Camel and then save it.

2. use the update operation with $modifiers, which will execute the update at the server-side instead. You can enable the upsert flag, in which case if an insert is required, MongoDB will apply the $modifiers to the filter query object and insert the result.

For example:

```
from("direct:insert")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=save");
```

217.5.2.3. update

Update one or multiple records on the collection. Requires a List<DBObject> as the IN message body containing exactly 2 elements:

- Element 1 (index 0) ⇒ filter query ⇒ determines what objects will be affected, same as a typical query object
- Element 2 (index 1) ⇒ update rules ⇒ how matched objects will be updated. All modifier operations from MongoDB are supported.

NOTE

Multiupdates . By default, MongoDB will only update 1 object even if multiple objects match the filter query. To instruct MongoDB to update **all** matching records, set the **CamelMongoDbMultiUpdate** IN message header to **true**.

A header with key **CamelMongoDbRecordsAffected** will be returned (**MongoDbConstants.RECORDS_AFFECTED** constant) with the number of records updated (copied from **WriteResult.getN()**).

Supports the following IN message headers:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbMultiUpdate	**MongoDbConstants.MULTIUPDATE**	If the update should be applied to all objects matching. See http://www.mongodb.org/display/DOCS/Atomic+Operations	boolean/Boolean
CamelMongoDbUpsert	**MongoDbConstants.UPSERT**	If the database should create the element if it does not exist	boolean/Boolean

For example, the following will update **all** records whose filterField field equals true by setting the value of the "scientist" field to "Darwin":

```
// route: from("direct:update").to("mongodb:myDb?database=science&collection=notableScientists&operation=update");
DBObject filterField = new BasicDBObject("filterField", true);
DBObject updateObj = new BasicDBObject("$set", new BasicDBObject("scientist", "Darwin"));
Object result = template.requestBodyAndHeader("direct:update", new Object[] {filterField, updateObj},
MongoDbConstants.MULTIUPDATE, true);
```

217.5.3. Delete operations

217.5.3.1. remove

Remove matching records from the collection. The IN message body will act as the removal filter query, and is expected to be of type **DBObject** or a type convertible to it.
The following example will remove all objects whose field 'conditionField' equals true, in the science database, notableScientists collection:

```
// route: from("direct:remove").to("mongodb:myDb?
database=science&collection=notableScientists&operation=remove");
DBObject conditionField = new BasicDBObject("conditionField", true);
Object result = template.requestBody("direct:remove", conditionField);
```

A header with key **CamelMongoDbRecordsAffected** is returned (**MongoDbConstants.RECORDS_AFFECTED** constant) with type **int**, containing the number of records deleted (copied from **WriteResult.getN()**).

217.5.4. Bulk Write Operations

217.5.4.1. bulkWrite

Available as of Camel 2.21

Performs write operations in bulk with controls for order of execution. Requires a **List<WriteModel<DBObject>>** as the IN message body containing commands for insert, update, and delete operations.

The following example will insert a new scientist "Pierre Curie", update record with id "5" by setting the value of the "scientist" field to "Marie Curie" and delete record with id "3" :

```
// route: from("direct:bulkWrite").to("mongodb:myDb?
database=science&collection=notableScientists&operation=bulkWrite");
List<WriteModel<DBObject>> bulkOperations = Arrays.asList(
        new InsertOneModel<>(new BasicDBObject("scientist", "Pierre Curie")),
        new UpdateOneModel<>(new BasicDBObject("_id", "5"),
                new BasicDBObject("$set", new BasicDBObject("scientist", "Marie Curie"))),
        new DeleteOneModel<>(new BasicDBObject("_id", "3")));

BulkWriteResult result = template.requestBody("direct:bulkWrite", bulkOperations,
BulkWriteResult.class);
```

By default, operations are executed in order and interrupted on the first write error without processing any remaining write operations in the list. To instruct MongoDB to continue to process remaining write operations in the list, set the **CamelMongoDbBulkOrdered** IN message header to **false**. Unordered operations are executed in parallel and this behavior is not guaranteed.

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbBulkOrdered	**MongoDbConstants.BULK_ORDERED**	Perform an ordered or unordered operation execution. Defaults to true.	boolean/Boolean

217.5.5. Other operations

217.5.5.1. aggregate

Available as of Camel 2.14

Perform a aggregation with the given pipeline contained in the body. **Aggregations could be long and heavy operations. Use with care.**

```
// route: from("direct:aggregate").to("mongodb:myDb?
database=science&collection=notableScientists&operation=aggregate");
from("direct:aggregate")
    .setBody().constant("[{ $match : {$or : [{\"scientist\" : \"Darwin\"},{\"scientist\" : \"Einstein\"}]}},{
$group: { _id: \"$scientist\", count: { $sum: 1 }} } ]")
    .to("mongodb:myDb?database=science&collection=notableScientists&operation=aggregate")
    .to("mock:resultAggregate");
```

Supports the following IN message headers:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbBatchSize	**MongoDbConstants.BATCH_SIZE**	Sets the number of documents to return per batch.	int/Integer
CamelMongoDbAllowDiskUse	**MongoDbConstants.ALLOW_DISK_USE**	Enable aggregation pipeline stages to write data to temporary files.	boolean/Boolean

Efficient retrieval is supported via outputType=DBCursor.

You can also "stream" the documents returned from the server into your route by including outputType=DBCursor (Camel 2.21+) as an endpoint option which may prove simpler than setting the above headers. This hands your Exchange the DBCursor from the Mongo driver, just as if you were executing the aggregate() within the Mongo shell, allowing your route to iterate over the results. By default and without this option, this component will load the documents from the driver's cursor into a List and return this to your route - which may result in a large number of in-memory objects. Remember, with a DBCursor do not ask for the number of documents matched - see the MongoDB documentation site for details.

Example with option outputType=DBCursor and batch size:

```
// route: from("direct:aggregate").to("mongodb:myDb?
database=science&collection=notableScientists&operation=aggregate");
from("direct:aggregate")
    .setHeader(MongoDbConstants.BATCH_SIZE).constant(10)
    .setBody().constant("[{ $match : {$or : [{\"scientist\" : \"Darwin\"},{\"scientist\" : \"Einstein\"}]}},{
$group: { _id: \"$scientist\", count: { $sum: 1 }} } ]")
```

```
    .to("mongodb:myDb?
database=science&collection=notableScientists&operation=aggregate&outputType=DBCursor")
    .to("mock:resultAggregate");
```

217.5.5.2. getDbStats

Equivalent of running the **db.stats()** command in the MongoDB shell, which displays useful statistic figures about the database.
For example:

```
> db.stats();
{
    "db" : "test",
    "collections" : 7,
    "objects" : 719,
    "avgObjSize" : 59.73296244784423,
    "dataSize" : 42948,
    "storageSize" : 1000058880,
    "numExtents" : 9,
    "indexes" : 4,
    "indexSize" : 32704,
    "fileSize" : 1275068416,
    "nsSizeMB" : 16,
    "ok" : 1
}
```

Usage example:

```
// from("direct:getDbStats").to("mongodb:myDb?
database=flights&collection=tickets&operation=getDbStats");
Object result = template.requestBody("direct:getDbStats", "irrelevantBody");
assertTrue("Result is not of type DBObject", result instanceof DBObject);
```

The operation will return a data structure similar to the one displayed in the shell, in the form of a **DBObject** in the OUT message body.

217.5.5.3. getColStats

Equivalent of running the **db.collection.stats()** command in the MongoDB shell, which displays useful statistic figures about the collection.
For example:

```
> db.camelTest.stats();
{
    "ns" : "test.camelTest",
    "count" : 100,
    "size" : 5792,
    "avgObjSize" : 57.92,
    "storageSize" : 20480,
    "numExtents" : 2,
    "nindexes" : 1,
    "lastExtentSize" : 16384,
    "paddingFactor" : 1,
    "flags" : 1,
```

```
    "totalIndexSize" : 8176,
    "indexSizes" : {
        "_id_" : 8176
    },
    "ok" : 1
}
```

Usage example:

```
// from("direct:getColStats").to("mongodb:myDb?
database=flights&collection=tickets&operation=getColStats");
Object result = template.requestBody("direct:getColStats", "irrelevantBody");
assertTrue("Result is not of type DBObject", result instanceof DBObject);
```

The operation will return a data structure similar to the one displayed in the shell, in the form of a **DBObject** in the OUT message body.

217.5.5.4. command

Available as of Camel 2.15

Run the body as a command on database. Usefull for admin operation as getting host informations, replication or sharding status.

Collection parameter is not use for this operation.

```
// route: from("command").to("mongodb:myDb?database=science&operation=command");
DBObject commandBody = new BasicDBObject("hostInfo", "1");
Object result = template.requestBody("direct:command", commandBody);
```

217.5.6. Dynamic operations

An Exchange can override the endpoint's fixed operation by setting the **CamelMongoDbOperation** header, defined by the **MongoDbConstants.OPERATION_HEADER** constant.
The values supported are determined by the MongoDbOperation enumeration and match the accepted values for the **operation** parameter on the endpoint URI.

For example:

```
// from("direct:insert").to("mongodb:myDb?database=flights&collection=tickets&operation=insert");
Object result = template.requestBodyAndHeader("direct:insert", "irrelevantBody",
MongoDbConstants.OPERATION_HEADER, "count");
assertTrue("Result is not of type Long", result instanceof Long);
```

217.6. TAILABLE CURSOR CONSUMER

MongoDB offers a mechanism to instantaneously consume ongoing data from a collection, by keeping the cursor open just like the **tail -f** command of *nix systems. This mechanism is significantly more efficient than a scheduled poll, due to the fact that the server pushes new data to the client as it becomes available, rather than making the client ping back at scheduled intervals to fetch new data. It also reduces otherwise redundant network traffic.

There is only one requisite to use tailable cursors: the collection must be a "capped collection", meaning

that it will only hold N objects, and when the limit is reached, MongoDB flushes old objects in the same order they were originally inserted. For more information, please refer to: http://www.mongodb.org/display/DOCS/Tailable+Cursors.

The Camel MongoDB component implements a tailable cursor consumer, making this feature available for you to use in your Camel routes. As new objects are inserted, MongoDB will push them as DBObjects in natural order to your tailable cursor consumer, who will transform them to an Exchange and will trigger your route logic.

217.7. HOW THE TAILABLE CURSOR CONSUMER WORKS

To turn a cursor into a tailable cursor, a few special flags are to be signalled to MongoDB when first generating the cursor. Once created, the cursor will then stay open and will block upon calling the **DBCursor.next()** method until new data arrives. However, the MongoDB server reserves itself the right to kill your cursor if new data doesn't appear after an indeterminate period. If you are interested to continue consuming new data, you have to regenerate the cursor. And to do so, you will have to remember the position where you left off or else you will start consuming from the top again.

The Camel MongoDB tailable cursor consumer takes care of all these tasks for you. You will just need to provide the key to some field in your data of increasing nature, which will act as a marker to position your cursor every time it is regenerated, e.g. a timestamp, a sequential ID, etc. It can be of any datatype supported by MongoDB. Date, Strings and Integers are found to work well. We call this mechanism "tail tracking" in the context of this component.

The consumer will remember the last value of this field and whenever the cursor is to be regenerated, it will run the query with a filter like: **increasingField > lastValue**, so that only unread data is consumed.

Setting the increasing field: Set the key of the increasing field on the endpoint URI **tailTrackingIncreasingField** option. In Camel 2.10, it must be a top-level field in your data, as nested navigation for this field is not yet supported. That is, the "timestamp" field is okay, but "nested.timestamp" will not work. Please open a ticket in the Camel JIRA if you do require support for nested increasing fields.

Cursor regeneration delay: One thing to note is that if new data is not already available upon initialisation, MongoDB will kill the cursor instantly. Since we don't want to overwhelm the server in this case, a **cursorRegenerationDelay** option has been introduced (with a default value of 1000ms.), which you can modify to suit your needs.

An example:

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime")
    .id("tailableCursorConsumer1")
    .autoStartup(false)
    .to("mock:test");
```

The above route will consume from the "flights.cancellations" capped collection, using "departureTime" as the increasing field, with a default regeneration cursor delay of 1000ms.

217.8. PERSISTENT TAIL TRACKING

Standard tail tracking is volatile and the last value is only kept in memory. However, in practice you will need to restart your Camel container every now and then, but your last value would then be lost and your tailable cursor consumer would start consuming from the top again, very likely sending duplicate records into your route.

To overcome this situation, you can enable the **persistent tail tracking** feature to keep track of the last consumed increasing value in a special collection inside your MongoDB database too. When the consumer initialises again, it will restore the last tracked value and continue as if nothing happened.

The last read value is persisted on two occasions: every time the cursor is regenerated and when the consumer shuts down. We may consider persisting at regular intervals too in the future (flush every 5 seconds) for added robustness if the demand is there. To request this feature, please open a ticket in the Camel JIRA.

217.9. ENABLING PERSISTENT TAIL TRACKING

To enable this function, set at least the following options on the endpoint URI:

- **persistentTailTracking** option to **true**
- **persistentId** option to a unique identifier for this consumer, so that the same collection can be reused across many consumers

Additionally, you can set the **tailTrackDb**, **tailTrackCollection** and **tailTrackField** options to customise where the runtime information will be stored. Refer to the endpoint options table at the top of this page for descriptions of each option.

For example, the following route will consume from the "flights.cancellations" capped collection, using "departureTime" as the increasing field, with a default regeneration cursor delay of 1000ms, with persistent tail tracking turned on, and persisting under the "cancellationsTracker" id on the "flights.camelTailTracking", storing the last processed value under the "lastTrackingValue" field (**camelTailTracking** and **lastTrackingValue** are defaults).

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime&persistentTailTrack
ng=true" +
    "&persistentId=cancellationsTracker")
  .id("tailableCursorConsumer2")
  .autoStartup(false)
  .to("mock:test");
```

Below is another example identical to the one above, but where the persistent tail tracking runtime information will be stored under the "trackers.camelTrackers" collection, in the "lastProcessedDepartureTime" field:

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime&persistentTailTrack
ng=true" +
    "&persistentId=cancellationsTracker&tailTrackDb=trackers&tailTrackCollection=camelTrackers" +
    "&tailTrackField=lastProcessedDepartureTime")
  .id("tailableCursorConsumer3")
  .autoStartup(false)
  .to("mock:test");
```

217.10. OPLOG TAIL TRACKING

The **oplog** collection tracking feature allows to implement trigger like functionality in MongoDB. In order to activate this collection you will have first to activate a replica set. For more information on this topic please check https://docs.mongodb.com/manual/tutorial/deploy-replica-set/ .

Below you can find an example of a Java DSL based route demonstrating how you can use the component to track the **oplog** collection. In this specific case we are filtering the events which affect a collection **customers** in database **optlog_test**. Note that the **tailTrackIncreasingField** is a timestamp field ('ts') which implies that you have to use the **tailTrackingStrategy** parameter with the **TIMESTAMP** value.

```
import com.mongodb.BasicDBObject;
import com.mongodb.MongoClient;
import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.Processor;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.mongodb.MongoDBTailTrackingEnum;
import org.apache.camel.main.Main;

import java.io.InputStream;

/**
 * For this to work you need to turn on the replica set
 * <p>
 * Commands to create a replica set:
 * <p>
 * rs.initiate( {
 * _id : "rs0",
 * members: [ { _id : 0, host : "localhost:27017" } ]
 * })
 */
public class MongoDbTracker {

    private final String database;

    private final String collection;

    private final String increasingField;

    private MongoDBTailTrackingEnum trackingStrategy;

    private int persistRecords = -1;

    private boolean persistenTailTracking;

    public MongoDbTracker(String database, String collection, String increasingField) {
        this.database = database;
        this.collection = collection;
        this.increasingField = increasingField;
    }

    public static void main(String[] args) throws Exception {
        final MongoDbTracker mongoDbTracker = new MongoDbTracker("local", "oplog.rs", "ts");
        mongoDbTracker.setTrackingStrategy(MongoDBTailTrackingEnum.TIMESTAMP);
        mongoDbTracker.setPersistRecords(5);
        mongoDbTracker.setPersistenTailTracking(true);
        mongoDbTracker.startRouter();
        // run until you terminate the JVM
        System.out.println("Starting Camel. Use ctrl + c to terminate the JVM.\n");
```

```
    }

    public void setTrackingStrategy(MongoDBTailTrackingEnum trackingStrategy) {
        this.trackingStrategy = trackingStrategy;
    }

    public void setPersistRecords(int persistRecords) {
        this.persistRecords = persistRecords;
    }

    public void setPersistenTailTracking(boolean persistenTailTracking) {
        this.persistenTailTracking = persistenTailTracking;
    }

    void startRouter() throws Exception {
        // create a Main instance
        Main main = new Main();
        main.bind(MongoConstants.CONN_NAME, new MongoClient("localhost", 27017));
        main.addRouteBuilder(new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                getContext().getTypeConverterRegistry().addTypeConverter(InputStream.class,
BasicDBObject.class,
                        new MongoToInputStreamConverter());
                from("mongodb://" + MongoConstants.CONN_NAME + "?database=" + database
                        + "&collection=" + collection
                        + "&persistentTailTracking=" + persistenTailTracking
                        + "&persistentId=trackerName" + "&tailTrackDb=local"
                        + "&tailTrackCollection=talendTailTracking"
                        + "&tailTrackField=lastTrackingValue"
                        + "&tailTrackIncreasingField=" + increasingField
                        + "&tailTrackingStrategy=" + trackingStrategy.toString()
                        + "&persistRecords=" + persistRecords
                        + "&cursorRegenerationDelay=1000")
                        .filter().jsonpath("$[?(@.ns=='optlog_test.customers')]")
                        .id("logger")
                        .to("log:logger?level=WARN")
                        .process(new Processor() {
                            public void process(Exchange exchange) throws Exception {
                                Message message = exchange.getIn();
                                System.out.println(message.getBody().toString());
                                exchange.getOut().setBody(message.getBody().toString());
                            }
                        });
            }
        });
        main.run();
    }
}
```

217.11. TYPE CONVERSIONS

The **MongoDbBasicConverters** type converter included with the camel-mongodb component provides the following conversions:

Name	From type	To type	How?
fromMapToDBObject	**Map**	**DBObject**	constructs a new **BasicDBObject** via the **new BasicDBObject(Map m)** constructor
fromBasicDBObjectToMap	**BasicDBObject**	**Map**	**BasicDBObject** already implements **Map**
fromStringToDBObject	**String**	**DBObject**	uses **com.mongodb.util.JSON.parse(String s)**
fromAnyObjectToDBObject	**Object**	**DBObject**	uses the Jackson library to convert the object to a **Map**, which is in turn used to initialise a new **BasicDBObject**

This type converter is auto-discovered, so you don't need to configure anything manually.

217.12. SEE ALSO

- MongoDB website
- NoSQL Wikipedia article
- MongoDB Java driver API docs - current version * Unit tests for more examples of usage

CHAPTER 218. MONGODB GRIDFS COMPONENT

Available as of Camel version 2.18

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mongodb-gridfs</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

218.1. URI FORMAT

```
mongodb-gridfs:connectionBean?database=databaseName&bucket=bucketName[&moreOptions...]
```

218.2. MONGODB GRIDFS OPTIONS

The MongoDB GridFS component has no options.

The MongoDB GridFS endpoint is configured using URI syntax:

```
mongodb-gridfs:connectionBean
```

with the following path and query parameters:

218.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
connectionBean	**Required** Name of com.mongodb.Mongo to use.		String

218.2.2. Query Parameters (17 parameters):

Name	Description	Default	Type
bucket (common)	Sets the name of the GridFS bucket within the database. Default is fs.	fs	String
database (common)	**Required** Sets the name of the MongoDB database to target		String

Name	Description	Default	Type
readPreference (common)	Sets a MongoDB ReadPreference on the Mongo connection. Read preferences set directly on the connection will be overridden by this setting. The link com.mongodb.ReadPreferencevalueOf(String) utility method is used to resolve the passed readPreference value. Some examples for the possible values are nearest, primary or secondary etc.		ReadPreference
writeConcern (common)	Set the WriteConcern for write operations on MongoDB using the standard ones. Resolved from the fields of the WriteConcern class by calling the link WriteConcernvalueOf(String) method.		WriteConcern
writeConcernRef (common)	Set the WriteConcern for write operations on MongoDB, passing in the bean ref to a custom WriteConcern which exists in the Registry. You can also use standard WriteConcerns by passing in their key. See the link setWriteConcern(String) setWriteConcern method.		WriteConcern
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delay (consumer)	Sets the delay between polls within the Consumer. Default is 500ms	500	long
fileAttributeName (consumer)	If the QueryType uses a FileAttribute, this sets the name of the attribute that is used. Default is camel-processed.	camel-processed	String
initialDelay (consumer)	Sets the initialDelay before the consumer will start polling. Default is 1000ms	1000	long
persistentTSCollection (consumer)	If the QueryType uses a persistent timestamp, this sets the name of the collection within the DB to store the timestamp.	camel-timestamps	String
persistentTSObject (consumer)	If the QueryType uses a persistent timestamp, this is the ID of the object in the collection to store the timestamp.	camel-timestamp	String

Name	Description	Default	Type
query (consumer)	Additional query parameters (in JSON) that are used to configure the query used for finding files in the GridFsConsumer		String
queryStrategy (consumer)	Sets the QueryStrategy that is used for polling for new files. Default is Timestamp	TimeStamp	QueryStrategy
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
operation (producer)	Sets the operation this endpoint will execute against GridRS.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

218.3. CONFIGURATION OF DATABASE IN SPRING XML

The following Spring XML creates a bean defining the connection to a MongoDB instance.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="mongoBean" class="com.mongodb.Mongo">
        <constructor-arg name="host" value="${mongodb.host}" />
        <constructor-arg name="port" value="${mongodb.port}" />
    </bean>
</beans>
```

218.4. SAMPLE ROUTE

The following route defined in Spring XML executes the operation **findOne** on a collection.

Get a file from GridFS

```
<route>
  <from uri="direct:start" />
```

```
  <!-- using bean 'mongoBean' defined above -->
  <to uri="mongodb-gridfs:mongoBean?database=${mongodb.database}&amp;operation=findOne" />
  <to uri="direct:result" />
</route>
```

218.5. GRIDFS OPERATIONS - PRODUCER ENDPOINT

218.5.1. count

Returns the total number of file in the collection, returning an Integer as the OUT message body.

```
// from("direct:count").to("mongodb-gridfs?database=tickets&operation=count");
Integer result = template.requestBodyAndHeader("direct:count", "irrelevantBody");
assertTrue("Result is not of type Long", result instanceof Integer);
```

You can provide a filename header to provide a count of files matching that filename.

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
Integer count = template.requestBodyAndHeaders("direct:count", query, headers);
```

218.5.2. listAll

Returns an Reader that lists all the filenames and their IDs in a tab separated stream.

```
// from("direct:listAll").to("mongodb-gridfs?database=tickets&operation=listAll");
Reader result = template.requestBodyAndHeader("direct:listAll", "irrelevantBody");

filename1.txt   1252314321
filename2.txt   2897651254
```

218.5.3. findOne

Finds a file in the GridFS system and sets the body to an InputStream of the content. Also provides the metadata has headers. It uses Exchange.FILE_NAME from the incoming headers to determine the file to find.

```
// from("direct:findOne").to("mongodb-gridfs?database=tickets&operation=findOne");
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
InputStream result = template.requestBodyAndHeaders("direct:findOne", "irrelevantBody", headers);
```

218.5.4. create

Creates a new file in the GridFs database. It uses the Exchange.FILE_NAME from the incoming headers for the name and the body contents (as an InputStream) as the content.

```
// from("direct:create").to("mongodb-gridfs?database=tickets&operation=create");
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
InputStream stream = ... the data for the file ...
template.requestBodyAndHeaders("direct:create", stream, headers);
```

218.5.5. remove

Removes a file from the GridFS database.

```
// from("direct:remove").to("mongodb-gridfs?database=tickets&operation=remove");
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(Exchange.FILE_NAME, "filename.txt");
template.requestBodyAndHeaders("direct:remove", "", headers);
```

218.6. GRIDFS CONSUMER

See also

- MongoDB website
- NoSQL Wikipedia article
- MongoDB Java driver API docs - current version * Unit tests for more examples of usage

CHAPTER 219. MONGODB COMPONENT

Available as of Camel version 2.19

Note: Camel MongoDB3 component Use the Mongo Driver for Java 3.4. If your are looking for previews versions look the Camel MongoDB component

According to Wikipedia: "NoSQL is a movement promoting a loosely defined class of non-relational data stores that break with a long history of relational databases and ACID guarantees." NoSQL solutions have grown in popularity in the last few years, and major extremely-used sites and services such as Facebook, LinkedIn, Twitter, etc. are known to use them extensively to achieve scalability and agility.

Basically, NoSQL solutions differ from traditional RDBMS (Relational Database Management Systems) in that they don't use SQL as their query language and generally don't offer ACID-like transactional behaviour nor relational data. Instead, they are designed around the concept of flexible data structures and schemas (meaning that the traditional concept of a database table with a fixed schema is dropped), extreme scalability on commodity hardware and blazing-fast processing.

MongoDB is a very popular NoSQL solution and the camel-mongodb component integrates Camel with MongoDB allowing you to interact with MongoDB collections both as a producer (performing operations on the collection) and as a consumer (consuming documents from a MongoDB collection).

MongoDB revolves around the concepts of documents (not as is office documents, but rather hierarchical data defined in JSON/BSON) and collections. This component page will assume you are familiar with them. Otherwise, visit http://www.mongodb.org/.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mongodb3</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

219.1. URI FORMAT

```
mongodb3:connectionBean?
database=databaseName&collection=collectionName&operation=operationName[&moreOptions...]
```

219.2. MONGODB OPTIONS

The MongoDB component has no options.

The MongoDB endpoint is configured using URI syntax:

```
mongodb3:connectionBean
```

with the following path and query parameters:

219.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
connectionBean	**Required** Name of com.mongodb.Mongo to use.		String

219.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
collection (common)	Sets the name of the MongoDB collection to bind to this endpoint		String
collectionIndex (common)	Sets the collection index (JSON FORMAT : field1 : order1, field2 : order2)		String
createCollection (common)	Create collection during initialisation if it doesn't exist. Default is true.	true	boolean
database (common)	Sets the name of the MongoDB database to target		String
operation (common)	Sets the operation this endpoint will execute against MongoDB. For possible values, see MongoDbOperation.		MongoDbOperation
outputType (common)	Convert the output of the producer to the selected type : DocumentList Document or MongoIterable. DocumentList or MongoIterable applies to findAll and aggregate. Document applies to all other operations.		MongoDbOutputType
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
cursorRegenerationDelay (advanced)	MongoDB tailable cursors will block until new data arrives. If no new data is inserted, after some time the cursor will be automatically freed and closed by the MongoDB server. The client is expected to regenerate the cursor if needed. This value specifies the time to wait before attempting to fetch a new cursor, and if the attempt fails, how long before the next attempt is made. Default value is 1000ms.	1000	long
dynamicity (advanced)	Sets whether this endpoint will attempt to dynamically resolve the target database and collection from the incoming Exchange properties. Can be used to override at runtime the database and collection specified on the otherwise static endpoint URI. It is disabled by default to boost performance. Enabling it will take a minimal performance hit.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
writeResultAsHeader (advanced)	In write operations, it determines whether instead of returning WriteResult as the body of the OUT message, we transfer the IN message to the OUT and attach the WriteResult as a header.	false	boolean
persistentId (tail)	One tail tracking collection can host many trackers for several tailable consumers. To keep them separate, each tracker should have its own unique persistentId.		String
persistentTailTracking (tail)	Enable persistent tail tracking, which is a mechanism to keep track of the last consumed message across system restarts. The next time the system is up, the endpoint will recover the cursor from the point where it last stopped slurping records.	false	boolean
tailTrackCollection (tail)	Collection where tail tracking information will be persisted. If not specified, link MongoDbTailTrackingConfigDEFAULT_COLLECTION will be used by default.		String

Name	Description	Default	Type
tailTrackDb (tail)	Indicates what database the tail tracking mechanism will persist to. If not specified, the current database will be picked by default. Dynamicity will not be taken into account even if enabled, i.e. the tail tracking database will not vary past endpoint initialisation.		String
tailTrackField (tail)	Field where the last tracked value will be placed. If not specified, link MongoDbTailTrackingConfigDEFAULT_FIELD will be used by default.		String
tailTrackIncreasingField (tail)	Correlation field in the incoming record which is of increasing nature and will be used to position the tailing cursor every time it is generated. The cursor will be (re)created with a query of type: tailTrackIncreasingField lastValue (possibly recovered from persistent tail tracking). Can be of type Integer, Date, String, etc. NOTE: No support for dot notation at the current time, so the field should be at the top level of the document.		String

Note on options of MoongoDB component

writeConcern **Remove in camel 2.19.**See Mongo client options Section 219.3, “Configuration of database in Spring XML”. Set the WriteConcern for write operations on MongoDB using the standard ones. Resolved from the fields of the WriteConcern class by calling the link WriteConcernvalueOf(String) method.

readPreference **Remove in camel 2.19.**See Mongo client options Section 219.3, “Configuration of database in Spring XML”. Sets a MongoDB ReadPreference on the Mongo connection. Read preferences set directly on the connection will be overridden by this setting. The link com.mongodb.ReadPreferencevalueOf(String) utility method is used to resolve the passed readPreference value. Some examples for the possible values are nearest primary or secondary etc.

219.3. CONFIGURATION OF DATABASE IN SPRING XML

The following Spring XML creates a bean defining the connection to a MongoDB instance.

Since mongo java driver 3, the WriteConcern and readPreference options are not dynamically modifiable. They are defined in the mongoClient object

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:mongo="http://www.springframework.org/schema/data/mongo"
xsi:schemaLocation="http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
    http://www.springframework.org/schema/beans
```

```
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <mongo:mongo-client id="mongoBean" host="${mongo.url}" port="${mongo.port}"
credentials="${mongo.user}:${mongo.pass}@${mongo.dbname}">
    <mongo:client-options write-concern="NORMAL" />
  </mongo:mongo-client>
</beans>
```

219.4. SAMPLE ROUTE

The following route defined in Spring XML executes the operation **dbStats** on a collection.

Get DB stats for specified collection

```
<route>
  <from uri="direct:start" />
  <!-- using bean 'mongoBean' defined above -->
  <to uri="mongodb3:mongoBean?
database=${mongodb.database}&amp;collection=${mongodb.collection}&amp;operation=getDbStats"
/>
  <to uri="direct:result" />
</route>
```

219.5. MONGODB OPERATIONS - PRODUCER ENDPOINTS

219.5.1. Query operations

219.5.1.1. findById

This operation retrieves only one element from the collection whose _id field matches the content of the IN message body. The incoming object can be anything that has an equivalent to a **Bson** type. See http://bsonspec.org//specification[http://bsonspec.org//specification] and http://www.mongodb.org/display/DOCS/Java+Types.

```
from("direct:findById")
    .to("mongodb3:myDb?database=flights&collection=tickets&operation=findById")
    .to("mock:resultFindById");
```

TIP

Supports optional parameters. This operation supports specifying a fields filter. See Specifying optional parameters.

219.5.1.2. findOneByQuery

Use this operation to retrieve just one element (the first) from the collection that matches a MongoDB query. **The query object is extracted CamelMongoDbCriteria header**. if the CamelMongoDbCriteria header is null the query object is extracted message body, i.e. it should be of type **Bson** or convertible to **Bson**. It can be a JSON String or a Hashmap. See #Type conversions for more info. you can use the Filters class from MongoDB Driver.

Example with no query (returns any object of the collection):

```
from("direct:findOneByQuery")
    .to("mongodb3:myDb?database=flights&collection=tickets&operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

Example with a query (returns one matching result):

```
from("direct:findOneByQuery")
    .setHeader(MongoDbConstants.CRITERIA, Filters.eq("name", "Raul Kripalani"))
    .to("mongodb3:myDb?database=flights&collection=tickets&operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

TIP

Supports optional parameters. This operation supports specifying a fields projection and/or a sort clause. See Specifying optional parameters.

219.5.1.3. findAll

The **findAll** operation returns all documents matching a query, or none at all, in which case all documents contained in the collection are returned. **The query object is extracted CamelMongoDbCriteria header**. if the CamelMongoDbCriteria header is null the query object is extracted message body, i.e. it should be of type **Bson** or convertible to **Bson**. It can be a JSON String or a Hashmap. See #Type conversions for more info.

Example with no query (returns all object in the collection):

```
from("direct:findAll")
    .to("mongodb3:myDb?database=flights&collection=tickets&operation=findAll")
    .to("mock:resultFindAll");
```

Example with a query (returns all matching results):

```
from("direct:findAll")
    .setHeader(MongoDbConstants.CRITERIA, Filters.eq("name", "Raul Kripalani"))
    .to("mongodb3:myDb?database=flights&collection=tickets&operation=findAll")
    .to("mock:resultFindAll");
```

Paging and efficient retrieval is supported via the following headers:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbNumToSkip	**MongoDbConstants.NUM_TO_SKIP**	Discards a given number of elements at the beginning of the cursor.	int/Integer

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbLimit	**MongoDbConstants.LIMIT**	Limits the number of elements returned.	int/Integer
CamelMongoDbBatchSize	**MongoDbConstants.BATCH_SIZE**	Limits the number of elements returned in one batch. A cursor typically fetches a batch of result objects and store them locally. If batchSize is positive, it represents the size of each batch of objects retrieved. It can be adjusted to optimize performance and limit data transfer. If batchSize is negative, it will limit of number objects returned, that fit within the max batch size limit (usually 4MB), and cursor will be closed. For example if batchSize is -10, then the server will return a maximum of 10 documents and as many as can fit in 4MB, then close the cursor. Note that this feature is different from limit() in that documents must fit within a maximum size, and it removes the need to send a request to close the cursor server-side. The batch size can be changed even after a cursor is iterated, in which case the setting will apply on the next batch retrieval.	int/Integer

Example with option outputType=MongoIterable and batch size :

```
from("direct:findAll")
    .setHeader(MongoDbConstants.BATCH_SIZE).constant(10)
    .setHeader(MongoDbConstants.CRITERIA, Filters.eq("name", "Raul Kripalani"))
    .to("mongodb3:myDb?
database=flights&collection=tickets&operation=findAll&outputType=MongoIterable")
    .to("mock:resultFindAll");
```

The **findAll** operation will also return the following OUT headers to enable you to iterate through result pages if you are using paging:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Data type
CamelMongoDbResultTotalSize	**MongoDbConstants.RESULT_TOTAL_SIZE**	Number of objects matching the query. This does not take limit/skip into consideration.	int/Integer

Header key	Quick constant	Description (extracted from MongoDB API doc)	Data type
CamelMongoDbResultPageSize	**MongoDbConstants.RESULT_PAGE_SIZE**	Number of objects matching the query. This does not take limit/skip into consideration.	int/Integer

TIP

Supports optional parameters. This operation supports specifying a fields projection and/or a sort clause. See Specifying optional parameters.

219.5.1.4. count

Returns the total number of objects in a collection, returning a Long as the OUT message body. The following example will count the number of records in the "dynamicCollectionName" collection. Notice how dynamicity is enabled, and as a result, the operation will not run against the "notableScientists" collection, but against the "dynamicCollectionName" collection.

```
// from("direct:count").to("mongodb3:myDb?
database=tickets&collection=flights&operation=count&dynamicity=true");
Long result = template.requestBodyAndHeader("direct:count", "irrelevantBody",
MongoDbConstants.COLLECTION, "dynamicCollectionName");
assertTrue("Result is not of type Long", result instanceof Long);
```

You can provide a query **The query object is extracted CamelMongoDbCriteria header**. if the CamelMongoDbCriteria header is null the query object is extracted message body, i.e. it should be of type **Bson** or convertible to **Bson**., and operation will return the amount of documents matching this criteria.

```
Document query = ...
Long count = template.requestBodyAndHeader("direct:count", query,
MongoDbConstants.COLLECTION, "dynamicCollectionName");
```

219.5.1.5. Specifying a fields filter (projection)

Query operations will, by default, return the matching objects in their entirety (with all their fields). If your documents are large and you only require retrieving a subset of their fields, you can specify a field filter in all query operations, simply by setting the relevant **Bson** (or type convertible to **Bson**, such as a JSON String, Map, etc.) on the **CamelMongoDbFieldsProjection** header, constant shortcut: **MongoDbConstants.FIELDS_PROJECTION**.

Here is an example that uses MongoDB's **Projections** to simplify the creation of Bson. It retrieves all fields except **_id** and **boringField**:

```
// route: from("direct:findAll").to("mongodb3:myDb?
database=flights&collection=tickets&operation=findAll")
Bson fieldProjection = Projection.exclude("_id", "boringField");
Object result = template.requestBodyAndHeader("direct:findAll", ObjectUtils.NULL,
MongoDbConstants.FIELDS_PROJECTION, fieldProjection);
```

Here is an example that uses MongoDB's **Projections** to simplify the creation of Bson. It retrieves all fields except **_id** and **boringField**:

```
// route: from("direct:findAll").to("mongodb3:myDb?
database=flights&collection=tickets&operation=findAll")
Bson fieldProjection = Projection.exclude("_id", "boringField");
Object result = template.requestBodyAndHeader("direct:findAll", ObjectUtils.NULL,
MongoDbConstants.FIELDS_PROJECTION, fieldProjection);
```

219.5.1.6. Specifying a sort clause

There is a often a requirement to fetch the min/max record from a collection based on sorting by a particular field that uses MongoDB's **Sorts** to simplify the creation of Bson. It retrieves all fields except **_id** and **boringField**:

```
// route: from("direct:findAll").to("mongodb3:myDb?
database=flights&collection=tickets&operation=findAll")
Bson sorts = Sorts.descending("_id");
Object result = template.requestBodyAndHeader("direct:findAll", ObjectUtils.NULL,
MongoDbConstants.SORT_BY, sorts);
```

In a Camel route the SORT_BY header can be used with the findOneByQuery operation to achieve the same result. If the FIELDS_PROJECTION header is also specified the operation will return a single field/value pair that can be passed directly to another component (for example, a parameterized MyBatis SELECT query). This example demonstrates fetching the temporally newest document from a collection and reducing the result to a single field, based on the **documentTimestamp** field:

```
.from("direct:someTriggeringEvent")
.setHeader(MongoDbConstants.SORT_BY).constant(Sorts.descending("documentTimestamp"))
.setHeader(MongoDbConstants.FIELDS_PROJECTION).constant(Projection.include("documentTime
stamp"))
.setBody().constant("{}")
.to("mongodb3:myDb?database=local&collection=myDemoCollection&operation=findOneByQuery")
.to("direct:aMyBatisParameterizedSelect")
;
```

219.5.2. Create/update operations

219.5.2.1. insert

Inserts an new object into the MongoDB collection, taken from the IN message body. Type conversion is attempted to turn it into **Document** or a **List**.
Two modes are supported: single insert and multiple insert. For multiple insert, the endpoint will expect a List, Array or Collections of objects of any type, as long as they are - or can be converted to - **Document**. Example:

```
from("direct:insert")
    .to("mongodb3:myDb?database=flights&collection=tickets&operation=insert");
```

The operation will return a WriteResult, and depending on the **WriteConcern** or the value of the **invokeGetLastError** option, **getLastError()** would have been called already or not. If you want to access the ultimate result of the write operation, you need to retrieve the **CommandResult** by calling **getLastError()** or **getCachedLastError()** on the **WriteResult**. Then you can verify the result by calling **CommandResult.ok()**, **CommandResult.getErrorMessage()** and/or **CommandResult.getException()**.

Note that the new object's **_id** must be unique in the collection. If you don't specify the value, MongoDB will automatically generate one for you. But if you do specify it and it is not unique, the insert operation will fail (and for Camel to notice, you will need to enable invokeGetLastError or set a WriteConcern that waits for the write result).

This is not a limitation of the component, but it is how things work in MongoDB for higher throughput. If you are using a custom **_id**, you are expected to ensure at the application level that is unique (and this is a good practice too).

OID(s) of the inserted record(s) is stored in the message header under **CamelMongoOid** key (**MongoDbConstants.OID** constant). The value stored is **org.bson.types.ObjectId** for single insert or **java.util.List<org.bson.types.ObjectId>** if multiple records have been inserted.

In MongoDB Java Driver 3.x the insertOne and insertMany operation return void. The Camel insert operation return the Document or List of Documents inserted. Note that each Documents are Updated by a new OID if need.

219.5.2.2. save

The save operation is equivalent to an *upsert* (UPdate, inSERT) operation, where the record will be updated, and if it doesn't exist, it will be inserted, all in one atomic operation. MongoDB will perform the matching based on the **_id** field.

Beware that in case of an update, the object is replaced entirely and the usage of MongoDB's $modifiers is not permitted. Therefore, if you want to manipulate the object if it already exists, you have two options:

1. perform a query to retrieve the entire object first along with all its fields (may not be efficient), alter it inside Camel and then save it.

2. use the update operation with $modifiers, which will execute the update at the server-side instead. You can enable the upsert flag, in which case if an insert is required, MongoDB will apply the $modifiers to the filter query object and insert the result.

If the document to be saved does not contain the **_id** attribute, the operation will be an insert, and the new **_id** created will be placed in the **CamelMongoOid** header.

For example:

```
from("direct:insert")
    .to("mongodb3:myDb?database=flights&collection=tickets&operation=save");
```

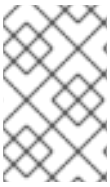
219.5.2.3. update

Update one or multiple records on the collection. Requires a filter query and a update rules.

You can define the filter using MongoDBConstants.CRITERIA header as **Bson** and define the update rules as **Bson** in Body.

The second way Require a List<Bson> as the IN message body containing exactly 2 elements:

- Element 1 (index 0) ⇒ filter query ⇒ determines what objects will be affected, same as a typical query object
- Element 2 (index 1) ⇒ update rules ⇒ how matched objects will be updated. All modifier operations from MongoDB are supported.



NOTE

Multiupdates . By default, MongoDB will only update 1 object even if multiple objects match the filter query. To instruct MongoDB to update **all** matching records, set the **CamelMongoDbMultiUpdate** IN message header to **true**.

A header with key **CamelMongoDbRecordsAffected** will be returned (**MongoDbConstants.RECORDS_AFFECTED** constant) with the number of records updated (copied from **WriteResult.getN()**).

Supports the following IN message headers:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbMultiUpdate	**MongoDbConstants.MULTIUPDATE**	If the update should be applied to all objects matching. See http://www.mongodb.org/display/DOCS/Atomic+Operations	boolean/Boolean

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbUpsert	**MongoDbConstants.UPSERT**	If the database should create the element if it does not exist	boolean/Boolean

For example, the following will update **all** records whose filterField field equals true by setting the value of the "scientist" field to "Darwin":

```
// route: from("direct:update").to("mongodb3:myDb?database=science&collection=notableScientists&operation=update");
Bson filterField = Filters.eq("filterField", true);
String updateObj = Updates.set("scientist", "Darwin");
Object result = template.requestBodyAndHeader("direct:update", new Bson[] {filterField, Document.parse(updateObj)}, MongoDbConstants.MULTIUPDATE, true);
```

```
// route: from("direct:update").to("mongodb3:myDb?database=science&collection=notableScientists&operation=update");
Maps<String, Object> headers = new HashMap<>(2);
headers.add(MongoDbConstants.MULTIUPDATE, true);
headers.add(MongoDbConstants.FIELDS_FILTER, Filters.eq("filterField", true));
String updateObj = Updates.set("scientist", "Darwin");;
Object result = template.requestBodyAndHeaders("direct:update", updateObj, headers);
```

```
// route: from("direct:update").to("mongodb3:myDb?database=science&collection=notableScientists&operation=update");
String updateObj = "[{\"filterField\": true}, {\"$set\", {\"scientist\", \"Darwin\"}}]";
Object result = template.requestBodyAndHeader("direct:update", updateObj, MongoDbConstants.MULTIUPDATE, true);
```

219.5.3. Delete operations

219.5.3.1. remove

Remove matching records from the collection. The IN message body will act as the removal filter query, and is expected to be of type **DBObject** or a type convertible to it.

The following example will remove all objects whose field 'conditionField' equals true, in the science database, notableScientists collection:

```
// route: from("direct:remove").to("mongodb3:myDb?
database=science&collection=notableScientists&operation=remove");
Bson conditionField = Filters.eq("conditionField", true);
Object result = template.requestBody("direct:remove", conditionField);
```

A header with key **CamelMongoDbRecordsAffected** is returned (**MongoDbConstants.RECORDS_AFFECTED** constant) with type **int**, containing the number of records deleted (copied from **WriteResult.getN()**).

219.5.4. Bulk Write Operations

219.5.4.1. bulkWrite

Available as of Camel 2.21

Performs write operations in bulk with controls for order of execution. Requires a **List<WriteModel<Document>>** as the IN message body containing commands for insert, update, and delete operations.

The following example will insert a new scientist "Pierre Curie", update record with id "5" by setting the value of the "scientist" field to "Marie Curie" and delete record with id "3" :

```
// route: from("direct:bulkWrite").to("mongodb:myDb?
database=science&collection=notableScientists&operation=bulkWrite");
List<WriteModel<Document>> bulkOperations = Arrays.asList(
        new InsertOneModel<>(new Document("scientist", "Pierre Curie")),
        new UpdateOneModel<>(new Document("_id", "5"),
                new Document("$set", new Document("scientist", "Marie Curie"))),
        new DeleteOneModel<>(new Document("_id", "3")));

BulkWriteResult result = template.requestBody("direct:bulkWrite", bulkOperations,
BulkWriteResult.class);
```

By default, operations are executed in order and interrupted on the first write error without processing any remaining write operations in the list. To instruct MongoDB to continue to process remaining write operations in the list, set the **CamelMongoDbBulkOrdered** IN message header to **false**. Unordered operations are executed in parallel and this behavior is not guaranteed.

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbBulkOrdered	**MongoDbConstants.BULK_ORDERED**	Perform an ordered or unordered operation execution. Defaults to true.	boolean/Boolean

219.5.5. Other operations

219.5.5.1. aggregate

Perform a aggregation with the given pipeline contained in the body. **Aggregations could be long and heavy operations. Use with care.**

```
// route: from("direct:aggregate").to("mongodb3:myDb?
database=science&collection=notableScientists&operation=aggregate");
List<Bson> aggregate = Arrays.asList(match(or(eq("scientist", "Darwin"), eq("scientist",
        group("$scientist", sum("count", 1)));
from("direct:aggregate")
    .setBody().constant(aggregate)
    .to("mongodb3:myDb?database=science&collection=notableScientists&operation=aggregate")
    .to("mock:resultAggregate");
```

Supports the following IN message headers:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbBatchSize	**MongoDbConstants.BATCH_SIZE**	Sets the number of documents to return per batch.	int/Integer
CamelMongoDbAllowDiskUse	**MongoDbConstants.ALLOW_DISK_USE**	Enable aggregation pipeline stages to write data to temporary files.	boolean/Boolean

Efficient retrieval is supported via outputType=MongoIterable.

You can also "stream" the documents returned from the server into your route by including outputType=DBCursor (Camel 2.21+) as an endpoint option which may prove simpler than setting the above headers. This hands your Exchange the DBCursor from the Mongo driver, just as if you were executing the aggregate() within the Mongo shell, allowing your route to iterate over the results. By default and without this option, this component will load the documents from the driver's cursor into a List and return this to your route - which may result in a large number of in-memory objects. Remember, with a DBCursor do not ask for the number of documents matched - see the MongoDB documentation site for details.

Example with option outputType=MongoIterable and batch size:

```
// route: from("direct:aggregate").to("mongodb3:myDb?
database=science&collection=notableScientists&operation=aggregate&outputType=MongoIterable");
List<Bson> aggregate = Arrays.asList(match(or(eq("scientist", "Darwin"), eq("scientist",
        group("$scientist", sum("count", 1)));
from("direct:aggregate")
    .setHeader(MongoDbConstants.BATCH_SIZE).constant(10)
    .setBody().constant(aggregate)
```

```
    .to("mongodb3:myDb?
database=science&collection=notableScientists&operation=aggregate&outputType=MongoIterable")
    .to("mock:resultAggregate");
```

219.5.5.2. getDbStats

Equivalent of running the **db.stats()** command in the MongoDB shell, which displays useful statistic figures about the database.
For example:

```
> db.stats();
{
    "db" : "test",
    "collections" : 7,
    "objects" : 719,
    "avgObjSize" : 59.73296244784423,
    "dataSize" : 42948,
    "storageSize" : 1000058880,
    "numExtents" : 9,
    "indexes" : 4,
    "indexSize" : 32704,
    "fileSize" : 1275068416,
    "nsSizeMB" : 16,
    "ok" : 1
}
```

Usage example:

```
// from("direct:getDbStats").to("mongodb3:myDb?
database=flights&collection=tickets&operation=getDbStats");
Object result = template.requestBody("direct:getDbStats", "irrelevantBody");
assertTrue("Result is not of type Document", result instanceof Document);
```

The operation will return a data structure similar to the one displayed in the shell, in the form of a **Document** in the OUT message body.

219.5.5.3. getColStats

Equivalent of running the **db.collection.stats()** command in the MongoDB shell, which displays useful statistic figures about the collection.
For example:

```
> db.camelTest.stats();
{
    "ns" : "test.camelTest",
    "count" : 100,
    "size" : 5792,
    "avgObjSize" : 57.92,
    "storageSize" : 20480,
    "numExtents" : 2,
    "nindexes" : 1,
    "lastExtentSize" : 16384,
    "paddingFactor" : 1,
    "flags" : 1,
```

```
    "totalIndexSize" : 8176,
    "indexSizes" : {
        "_id_" : 8176
    },
    "ok" : 1
}
```

Usage example:

```
// from("direct:getColStats").to("mongodb3:myDb?
database=flights&collection=tickets&operation=getColStats");
Object result = template.requestBody("direct:getColStats", "irrelevantBody");
assertTrue("Result is not of type Document", result instanceof Document);
```

The operation will return a data structure similar to the one displayed in the shell, in the form of a **Document** in the OUT message body.

219.5.5.4. command

Run the body as a command on database. Usefull for admin operation as getting host informations, replication or sharding status.

Collection parameter is not use for this operation.

```
// route: from("command").to("mongodb3:myDb?database=science&operation=command");
DBObject commandBody = new BasicDBObject("hostInfo", "1");
Object result = template.requestBody("direct:command", commandBody);
```

219.5.6. Dynamic operations

An Exchange can override the endpoint's fixed operation by setting the **CamelMongoDbOperation** header, defined by the **MongoDbConstants.OPERATION_HEADER** constant.
The values supported are determined by the MongoDbOperation enumeration and match the accepted values for the **operation** parameter on the endpoint URI.

For example:

```
// from("direct:insert").to("mongodb3:myDb?database=flights&collection=tickets&operation=insert");
Object result = template.requestBodyAndHeader("direct:insert", "irrelevantBody",
MongoDbConstants.OPERATION_HEADER, "count");
assertTrue("Result is not of type Long", result instanceof Long);
```

219.6. TAILABLE CURSOR CONSUMER

MongoDB offers a mechanism to instantaneously consume ongoing data from a collection, by keeping the cursor open just like the **tail -f** command of *nix systems. This mechanism is significantly more efficient than a scheduled poll, due to the fact that the server pushes new data to the client as it becomes available, rather than making the client ping back at scheduled intervals to fetch new data. It also reduces otherwise redundant network traffic.

There is only one requisite to use tailable cursors: the collection must be a "capped collection", meaning that it will only hold N objects, and when the limit is reached, MongoDB flushes old objects in the same order they were originally inserted. For more information, please refer to:

http://www.mongodb.org/display/DOCS/Tailable+Cursors.

The Camel MongoDB component implements a tailable cursor consumer, making this feature available for you to use in your Camel routes. As new objects are inserted, MongoDB will push them as **Document** in natural order to your tailable cursor consumer, who will transform them to an Exchange and will trigger your route logic.

219.7. HOW THE TAILABLE CURSOR CONSUMER WORKS

To turn a cursor into a tailable cursor, a few special flags are to be signalled to MongoDB when first generating the cursor. Once created, the cursor will then stay open and will block upon calling the **MongoCursor.next()** method until new data arrives. However, the MongoDB server reserves itself the right to kill your cursor if new data doesn't appear after an indeterminate period. If you are interested to continue consuming new data, you have to regenerate the cursor. And to do so, you will have to remember the position where you left off or else you will start consuming from the top again.

The Camel MongoDB tailable cursor consumer takes care of all these tasks for you. You will just need to provide the key to some field in your data of increasing nature, which will act as a marker to position your cursor every time it is regenerated, e.g. a timestamp, a sequential ID, etc. It can be of any datatype supported by MongoDB. Date, Strings and Integers are found to work well. We call this mechanism "tail tracking" in the context of this component.

The consumer will remember the last value of this field and whenever the cursor is to be regenerated, it will run the query with a filter like: **increasingField > lastValue**, so that only unread data is consumed.

Setting the increasing field: Set the key of the increasing field on the endpoint URI **tailTrackingIncreasingField** option. In Camel 2.10, it must be a top-level field in your data, as nested navigation for this field is not yet supported. That is, the "timestamp" field is okay, but "nested.timestamp" will not work. Please open a ticket in the Camel JIRA if you do require support for nested increasing fields.

Cursor regeneration delay: One thing to note is that if new data is not already available upon initialisation, MongoDB will kill the cursor instantly. Since we don't want to overwhelm the server in this case, a **cursorRegenerationDelay** option has been introduced (with a default value of 1000ms.), which you can modify to suit your needs.

An example:

```
from("mongodb3:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime")
    .id("tailableCursorConsumer1")
    .autoStartup(false)
    .to("mock:test");
```

The above route will consume from the "flights.cancellations" capped collection, using "departureTime" as the increasing field, with a default regeneration cursor delay of 1000ms.

219.8. PERSISTENT TAIL TRACKING

Standard tail tracking is volatile and the last value is only kept in memory. However, in practice you will need to restart your Camel container every now and then, but your last value would then be lost and your tailable cursor consumer would start consuming from the top again, very likely sending duplicate records into your route.

To overcome this situation, you can enable the **persistent tail tracking** feature to keep track of the last consumed increasing value in a special collection inside your MongoDB database too. When the consumer initialises again, it will restore the last tracked value and continue as if nothing happened.

The last read value is persisted on two occasions: every time the cursor is regenerated and when the consumer shuts down. We may consider persisting at regular intervals too in the future (flush every 5 seconds) for added robustness if the demand is there. To request this feature, please open a ticket in the Camel JIRA.

219.9. ENABLING PERSISTENT TAIL TRACKING

To enable this function, set at least the following options on the endpoint URI:

- **persistentTailTracking** option to **true**
- **persistentId** option to a unique identifier for this consumer, so that the same collection can be reused across many consumers

Additionally, you can set the **tailTrackDb**, **tailTrackCollection** and **tailTrackField** options to customise where the runtime information will be stored. Refer to the endpoint options table at the top of this page for descriptions of each option.

For example, the following route will consume from the "flights.cancellations" capped collection, using "departureTime" as the increasing field, with a default regeneration cursor delay of 1000ms, with persistent tail tracking turned on, and persisting under the "cancellationsTracker" id on the "flights.camelTailTracking", storing the last processed value under the "lastTrackingValue" field (**camelTailTracking** and **lastTrackingValue** are defaults).

```
from("mongodb3:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime&persistentTailTrack
ng=true" +
    "&persistentId=cancellationsTracker")
  .id("tailableCursorConsumer2")
  .autoStartup(false)
  .to("mock:test");
```

Below is another example identical to the one above, but where the persistent tail tracking runtime information will be stored under the "trackers.camelTrackers" collection, in the "lastProcessedDepartureTime" field:

```
from("mongodb3:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime&persistentTailTrack
ng=true" +
    "&persistentId=cancellationsTracker&tailTrackDb=trackers&tailTrackCollection=camelTrackers" +
    "&tailTrackField=lastProcessedDepartureTime")
  .id("tailableCursorConsumer3")
  .autoStartup(false)
  .to("mock:test");
```

219.10. TYPE CONVERSIONS

The **MongoDbBasicConverters** type converter included with the camel-mongodb component provides the following conversions:

Name	From type	To type	How?
fromMapToDocument	**Map**	**Document**	constructs a new **Document** via the **new Document(Map m)** constructor.
fromDocumentToMap	**Document**	**Map**	**Document** already implements **Map**.
fromStringToDocument	**String**	**Document**	uses **com.mongodb.Document.parse(String s)**.
fromAnyObjectToDocument	**Object**	**Document**	uses the Jackson library to convert the object to a **Map**, which is in turn used to initialise a new **Document**.
fromStringToList	**String**	**List<Bson>**	uses **org.bson.codecs.configuration.CodecRegistries** to convert to BsonArray then to List<Bson>.

This type converter is auto-discovered, so you don't need to configure anything manually.

219.11. SEE ALSO

- MongoDB website
- NoSQL Wikipedia article
- MongoDB Java driver API docs - current version * Unit tests for more examples of usage

CHAPTER 220. MQTT COMPONENT

Available as of Camel version 2.10

The **mqtt:** component is used for communicating with MQTT compliant message brokers, like Apache ActiveMQ or Mosquitto

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mqtt</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

220.1. URI FORMAT

```
mqtt://name[?options]
```

Where **name** is the name you want to assign the component.

220.2. OPTIONS

The MQTT component supports 4 options which are listed below.

Name	Description	Default	Type
host (common)	The URI of the MQTT broker to connect too - this component also supports SSL - e.g. ssl://127.0.0.1:8883		String
userName (security)	Username to be used for authentication against the MQTT broker		String
password (security)	Password to be used for authentication against the MQTT broker		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The MQTT endpoint is configured using URI syntax:

```
mqtt:name
```

with the following path and query parameters:

220.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** A logical name to use which is not the topic name.		String

220.2.2. Query Parameters (39 parameters):

Name	Description	Default	Type
blockingExecutor (common)	SSL connections perform blocking operations against internal thread pool unless you call the setBlockingExecutor method to configure that executor they will use instead.		Executor
byDefaultRetain (common)	The default retain policy to be used on messages sent to the MQTT broker	false	boolean
cleanSession (common)	Set to false if you want the MQTT server to persist topic subscriptions and ack positions across client sessions. Defaults to true.	false	boolean
clientId (common)	Use to set the client Id of the session. This is what an MQTT server uses to identify a session where setCleanSession(false); is being used. The id must be 23 characters or less. Defaults to auto generated id (based on your socket address, port and timestamp).		String
connectAttemptsMax (common)	The maximum number of reconnect attempts before an error is reported back to the client on the first attempt by the client to connect to a server. Set to -1 to use unlimited attempts. Defaults to -1.	-1	long
connectWaitInSeconds (common)	Delay in seconds the Component will wait for a connection to be established to the MQTT broker	10	int
disconnectWaitInSeconds (common)	The number of seconds the Component will wait for a valid disconnect on stop() from the MQTT broker	5	int
dispatchQueue (common)	A HawtDispatch dispatch queue is used to synchronize access to the connection. If an explicit queue is not configured via the setDispatchQueue method, then a new queue will be created for the connection. Setting an explicit queue might be handy if you want multiple connection to share the same queue for synchronization.		DispatchQueue

Name	Description	Default	Type
host (common)	The URI of the MQTT broker to connect too - this component also supports SSL - e.g. ssl://127.0.0.1:8883	tcp://127.0.0.1:1883	URI
keepAlive (common)	Configures the Keep Alive timer in seconds. Defines the maximum time interval between messages received from a client. It enables the server to detect that the network connection to a client has dropped, without having to wait for the long TCP/IP timeout.		short
localAddress (common)	The local InetAddress and port to use		URI
maxReadRate (common)	Sets the maximum bytes per second that this transport will receive data at. This setting throttles reads so that the rate is not exceeded. Defaults to 0 which disables throttling.		int
maxWriteRate (common)	Sets the maximum bytes per second that this transport will send data at. This setting throttles writes so that the rate is not exceeded. Defaults to 0 which disables throttling.		int
mqttQosPropertyName (common)	The property name to look for on an Exchange for an individual published message. If this is set (one of AtMostOnce, AtLeastOnce or ExactlyOnce) - then that QoS will be set on the message sent to the MQTT message broker.	MQTTQos	String
mqttRetainPropertyName (common)	The property name to look for on an Exchange for an individual published message. If this is set (expects a Boolean value) - then the retain property will be set on the message sent to the MQTT message broker.	MQTTRetain	String
mqttTopicPropertyName (common)	These a properties that are looked for in an Exchange - to publish to	MQTTTopicPropertyName	String
publishTopicName (common)	The default Topic to publish messages on	camel/mqtt/test	String
qualityOfService (common)	Quality of service level to use for topics.	AtLeastOnce	String
receiveBufferSize (common)	Sets the size of the internal socket receive buffer. Defaults to 65536 (64k)	65536	int

Name	Description	Default	Type
reconnectAttemptsMax (common)	The maximum number of reconnect attempts before an error is reported back to the client after a server connection had previously been established. Set to -1 to use unlimited attempts. Defaults to -1.	-1	long
reconnectBackOffMultiplier (common)	The Exponential backoff be used between reconnect attempts. Set to 1 to disable exponential backoff. Defaults to 2.	2.0	double
reconnectDelay (common)	How long to wait in ms before the first reconnect attempt. Defaults to 10.	10	long
reconnectDelayMax (common)	The maximum amount of time in ms to wait between reconnect attempts. Defaults to 30,000.	30000	long
sendBufferSize (common)	Sets the size of the internal socket send buffer. Defaults to 65536 (64k)	65536	int
sendWaitInSeconds (common)	The maximum time the Component will wait for a receipt from the MQTT broker to acknowledge a published message before throwing an exception	5	int
sslContext (common)	To configure security using SSLContext configuration		SSLContext
subscribeTopicName (common)	**Deprecated** These are set on the Endpoint - together with properties inherited from MQTT		String
subscribeTopicNames (common)	A comma-delimited list of Topics to subscribe to for messages. Note that each item of this list can contain MQTT wildcards (and/or), in order to subscribe to topics matching a certain pattern within a hierarchy. For example, is a wildcard for all topics at a level within the hierarchy, so if a broker has topics topics/one and topics/two, then topics/ can be used to subscribe to both. A caveat to consider here is that if the broker adds topics/three, the route would also begin to receive messages from that topic.		String
trafficClass (common)	Sets traffic class or type-of-service octet in the IP header for packets sent from the transport. Defaults to 8 which means the traffic should be optimized for throughput.	8	int
version (common)	Set to 3.1.1 to use MQTT version 3.1.1. Otherwise defaults to the 3.1 protocol version.	3.1	String

Name	Description	Default	Type
willMessage (common)	The Will message to send. Defaults to a zero length message.		String
willQos (common)	Sets the quality of service to use for the Will message. Defaults to AT_MOST_ONCE.	AtMostOnce	QoS
willRetain (common)	Set to true if you want the Will to be published with the retain option.		QoS
willTopic (common)	If set the server will publish the client's Will message to the specified topics if the client has an unexpected disconnection.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
lazySessionCreation (producer)	Sessions can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

220.3. SAMPLES

Sending messages:

```
from("direct:foo").to("mqtt:cheese?publishTopicName=test.mqtt.topic");
```

Consuming messages:

-

```
from("mqtt:bar?
subscribeTopicName=test.mqtt.topic").transform(body().convertToString()).to("mock:result")
```

220.4. ENDPOINTS

Camel supports the Message Endpoint pattern using the Endpoint interface. Endpoints are usually created by a Component and Endpoints are usually referred to in the DSL via their URIs.

From an Endpoint you can use the following methods

- createProducer() will create a Producer for sending message exchanges to the endpoint
- createConsumer() implements the Event Driven Consumer pattern for consuming message exchanges from the endpoint via a Processor when creating a Consumer
- createPollingConsumer() implements the Polling Consumer pattern for consuming message exchanges from the endpoint via a PollingConsumer

220.5. SEE ALSO

- Configuring Camel
- Message Endpoint pattern
- URIs
- Writing Components

CHAPTER 221. MSV COMPONENT

Available as of Camel version 1.1

The MSV component performs XML validation of the message body using the MSV Library and any of the supported XML schema languages, such as XML Schema or RelaxNG XML Syntax.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-msv</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Note that the Jing component also supports RelaxNG Compact Syntax

221.1. URI FORMAT

```
msv:someLocalOrRemoteResource[?options]
```

Where **someLocalOrRemoteResource** is some URL to a local resource on the classpath or a full URL to a remote resource or resource on the file system. For example

```
msv:org/foo/bar.rng
msv:file:../foo/bar.rng
msv:http://acme.com/cheese.rng
```

You can append query options to the URI in the following format, **?option=value&option=value&...**

221.2. OPTIONS

The MSV component supports 3 options which are listed below.

Name	Description	Default	Type
schemaFactory (advanced)	To use the javax.xml.validation.SchemaFactory.		SchemaFactory
resourceResolverFactory (advanced)	To use a custom LSResourceResolver which depends on a dynamic endpoint resource URI		ValidatorResourceResolverFactory
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The MSV endpoint is configured using URI syntax:

```
msv:resourceUri
```

with the following path and query parameters:

221.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** URL to a local resource on the classpath, or a reference to lookup a bean in the Registry, or a full URL to a remote resource or resource on the file system which contains the XSD to validate against.		String

221.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
failOnNullBody (producer)	Whether to fail if no body exists.	true	boolean
failOnNullHeader (producer)	Whether to fail if no header exists when validating against a header.	true	boolean
headerName (producer)	To validate against a header instead of the message body.		String
errorHandler (advanced)	To use a custom org.apache.camel.processor.validation.ValidatorErrorHandler. The default error handler captures the errors and throws an exception.		ValidatorErrorHandler
resourceResolver (advanced)	To use a custom LSResourceResolver. Do not use together with resourceResolverFactory		LSResourceResolver
resourceResolverFactory (advanced)	To use a custom LSResourceResolver which depends on a dynamic endpoint resource URI. The default resource resolver factory resturns a resource resolver which can read files from the class path and file system. Do not use together with resourceResolver.		ValidatorResourceResolverFactory
schemaFactory (advanced)	To use a custom javax.xml.validation.SchemaFactory		SchemaFactory

Name	Description	Default	Type
schemaLanguage (advanced)	Configures the W3C XML Schema Namespace URI.	http://www.w3.org/2001/XMLSchema	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
useDom (advanced)	Whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator.	false	boolean
useSharedSchema (advanced)	Whether the Schema instance should be shared or not. This option is introduced to work around a JDK 1.6.x bug. Xerces should not have this issue.	true	boolean

221.3. EXAMPLE

The following example shows how to configure a route from endpoint **direct:start** which then goes to one of two endpoints, either **mock:valid** or **mock:invalid** based on whether or not the XML matches the given RelaxNG XML Schema (which is supplied on the classpath).

221.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 222. MUSTACHE COMPONENT

Available as of Camel version 2.12

The **mustache:** component allows for processing a message using a Mustache template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-mustache</artifactId>
<version>x.x.x</version> <!-- use the same version as your Camel core version -->
</dependency>
```

222.1. URI FORMAT

```
mustache:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: file://folder/myfile.mustache).

You can append query options to the URI in the following format, **?option=value&option=value&…**

222.2. OPTIONS

The Mustache component supports 2 options which are listed below.

Name	Description	Default	Type
mustacheFactory (advanced)	To use a custom MustacheFactory		MustacheFactory
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Mustache endpoint is configured using URI syntax:

```
mustache:resourceUri
```

with the following path and query parameters:

222.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

222.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
encoding (producer)	Character encoding of the resource content.		String
endDelimiter (producer)	Characters used to mark template code end.	}}	String
startDelimiter (producer)	Characters used to mark template code beginning.	{{	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

222.3. MUSTACHE CONTEXT

Camel will provide exchange information in the Mustache context (just a **Map**). The **Exchange** is transferred as:

key	value
exchange	The **Exchange** itself.
exchange.properties	The **Exchange** properties.

key	value
headers	The headers of the In message.
camelContext	The Camel Context.
request	The In message.
body	The In message body.
response	The Out message (only for InOut message exchange pattern).

222.4. DYNAMIC TEMPLATES

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description	Support Version
MustacheConstants.MUSTACHE_RESOURCE_URI	String	A URI for the template resource to use instead of the endpoint configured.	

Header	Type	Description	Support Version
MustacheConstants.MUSTACHE_TEMPLATE	String	The template to use instead of the endpoint configured.	

222.5. SAMPLES

For example you could use something like:

```
from("activemq:My.Queue").
to("mustache:com/acme/MyResponse.mustache");
```

To use a Mustache template to formulate a response for a message for InOut message exchanges (where there is a **JMSReplyTo** header).

If you want to use InOnly and consume the message and send it to another destination you could use:

```
from("activemq:My.Queue").
to("mustache:com/acme/MyResponse.mustache").
to("activemq:Another.Queue");
```

It's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
setHeader(MustacheConstants.MUSTACHE_RESOURCE_URI).constant("path/to/my/template.mustache").
to("mustache:dummy");
```

222.6. THE EMAIL SAMPLE

In this sample we want to use Mustache templating for an order confirmation email. The email template is laid out in Mustache as:

```
Dear {{headers.lastName}}}, {{headers.firstName}}

Thanks for the order of {{headers.item}}.

Regards Camel Riders Bookstore
{{body}}
```

222.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 223. MVEL COMPONENT

Available as of Camel version 2.12

The **mvel:** component allows you to process a message using an MVEL template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mvel</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

223.1. URI FORMAT

```
mvel:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: file://folder/myfile.mvel).

You can append query options to the URI in the following format, **?option=value&option=value&...**

223.2. OPTIONS

The MVEL component has no options.

The MVEL endpoint is configured using URI syntax:

```
mvel:resourceUri
```

with the following path and query parameters:

223.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

223.2.2. Query Parameters (3 parameters):

Name	Description	Default	Type
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
encoding (producer)	Character encoding of the resource content.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

223.3. MESSAGE HEADERS

The mvel component sets a couple headers on the message.

Header	Description
CamelMvelResourceUri	The **templateName** as a **String** object.

223.4. MVEL CONTEXT

Camel will provide exchange information in the MVEL context (just a **Map**). The **Exchange** is transfered as:

key	value
exchange	The **Exchange** itself.
exchange.properties	The **Exchange** properties.
headers	The headers of the In message.
camelContext	The Camel Context intance.
request	The In message.

key	value
in	The In message.
body	The In message body.
out	The Out message (only for InOut message exchange pattern).
response	The Out message (only for InOut message exchange pattern).

223.5. HOT RELOADING

The mvel template resource is, by default, hot reloadable for both file and classpath resources (expanded jar). If you set **contentCache=true**, Camel will only load the resource once, and thus hot reloading is not possible. This scenario can be used in production, when the resource never changes.

223.6. DYNAMIC TEMPLATES

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description
CamelMvelResourceUri	String	A URI for the template resource to use instead of the endpoint configured.
CamelMvelTemplate	String	The template to use instead of the endpoint configured.

223.7. SAMPLES

For example you could use something like

```
from("activemq:My.Queue").
  to("mvel:com/acme/MyResponse.mvel");
```

To use a MVEL template to formulate a response to a message for InOut message exchanges (where there is a **JMSReplyTo** header).

To specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelMvelResourceUri").constant("path/to/my/template.mvel").
  to("mvel:dummy");
```

To specify a template directly as a header the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelMvelTemplate").constant("@{\"The result is \" + request.body * 3}\" }").
  to("velocity:dummy");
```

223.8. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 224. MVEL LANGUAGE

Available as of Camel version 2.0

Camel allows Mvel to be used as an Expression or Predicate the DSL or Xml Configuration.

You could use Mvel to create an Predicate in a Message Filter or as an Expression for a Recipient List

You can use Mvel dot notation to invoke operations. If you for instance have a body that contains a POJO that has a **getFamiliyName** method then you can construct the syntax as follows:

```
"request.body.familyName"
  // or
"getRequest().getBody().getFamilyName()"
```

224.1. MVEL OPTIONS

The MVEL language supports 1 options which are listed below.

Name	Default	Java Type	Description
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

224.2. VARIABLES

Variable	Type	Description
this	Exchange	the Exchange is the root object
exchange	Exchange	the Exchange object
exception	Throwable	the Exchange exception (if any)
exchangeId	String	the exchange id
fault	Message	the Fault message (if any)
request	Message	the exchange.in message

Variable	Type	Description
response	Message	the exchange.out message (if any)
properties	Map	the exchange properties
property(name)	Object	the property by the given name
property(name, type)	Type	the property by the given name as the given type

224.3. SAMPLES

For example you could use Mvel inside a Message Filter in XML

```
<route>
  <from uri="seda:foo"/>
  <filter>
    <mvel>request.headers.foo == 'bar'</mvel>
    <to uri="seda:bar"/>
  </filter>
</route>
```

And the sample using Java DSL:

```
from("seda:foo").filter().mvel("request.headers.foo == 'bar'").to("seda:bar");
```

224.4. LOADING SCRIPT FROM EXTERNAL RESOURCE

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as **"classpath:"**, **"file:"**, or **"http:"**.
This is done using the following syntax: **"resource:scheme:location"**, eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").mvel("resource:classpath:script.mvel")
```

224.5. DEPENDENCIES

To use Mvel in your camel routes you need to add the a dependency on **camel-mvel** which implements the Mvel language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mvel</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 225. MYBATIS COMPONENT

Available as of Camel version 2.7

The **mybatis:** component allows you to query, poll, insert, update and delete data in a relational database using MyBatis.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mybatis</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

225.1. URI FORMAT

```
mybatis:statementName[?options]
```

Where **statementName** is the statement name in the MyBatis XML mapping file which maps to the query, insert, update or delete operation you wish to evaluate.

You can append query options to the URI in the following format, **?option=value&option=value&...**

This component will by default load the MyBatis SqlMapConfig file from the root of the classpath with the expected name of **SqlMapConfig.xml**.
If the file is located in another location, you will need to configure the **configurationUri** option on the **MyBatisComponent** component.

225.2. OPTIONS

The MyBatis component supports 3 options which are listed below.

Name	Description	Default	Type
sqlSessionFactory (advanced)	To use the SqlSessionFactory		SqlSessionFactory
configurationUri (common)	Location of MyBatis xml configuration file. The default value is: SqlMapConfig.xml loaded from the classpath	SqlMapConfig.xml	String
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The MyBatis endpoint is configured using URI syntax:

```
mybatis:statement
```

with the following path and query parameters:

225.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
statement	**Required** The statement name in the MyBatis XML mapping file which maps to the query, insert, update or delete operation you wish to evaluate.		String

225.2.2. Query Parameters (29 parameters):

Name	Description	Default	Type
outputHeader (common)	Store the query result in a header instead of the message body. By default, outputHeader == null and the query result is stored in the message body, any existing content in the message body is discarded. If outputHeader is set, the value is used as the name of the header to store the query result and the original message body is preserved. Setting outputHeader will also omit populating the default CamelMyBatisResult header since it would be the same as outputHeader all the time.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
inputHeader (consumer)	User the header value for input parameters instead of the message body. By default, inputHeader == null and the input parameters are taken from the message body. If outputHeader is set, the value is used and query parameters will be taken from the header instead of the body.		String

Name	Description	Default	Type
maxMessagesPerPoll (consumer)	This option is intended to split results returned by the database pool into the batches and deliver them in multiple exchanges. This integer defines the maximum messages to deliver in single exchange. By default, no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disable it.	0	int
onConsume (consumer)	Statement to run after data has been processed in the route		String
routeEmptyResultSet (consumer)	Whether allow empty resultset to be routed to the next hop	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
transacted (consumer)	Enables or disables transaction. If enabled then if processing an exchange failed then the consumer break out processing any further exchanges to cause a rollback eager	false	boolean
useIterator (consumer)	Process resultset individually or as a list	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
processingStrategy (consumer)	To use a custom MyBatisProcessingStrategy		MyBatisProcessing Strategy

Name	Description	Default	Type
executorType (producer)	The executor type to be used while executing statements. simple - executor does nothing special. reuse - executor reuses prepared statements. batch - executor reuses statements and batches updates.	SIMPLE	ExecutorType
statementType (producer)	Mandatory to specify for the producer to control which kind of operation to invoke.		StatementType
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service

Name	Description	Default	Type
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

225.3. MESSAGE HEADERS

Camel will populate the result message, either IN or OUT with a header with the statement used:

Header	Type	Description
CamelMyBatisStatementName	**String**	The **statementName** used (for example: insertAccount).
CamelMyBatisResult	**Object**	The **response** returned from MtBatis in any of the operations. For instance an**INSERT** could return the auto-generated key, or number of rows etc.

225.4. MESSAGE BODY

The response from MyBatis will only be set as the body if it's a **SELECT** statement. That means, for example, for **INSERT** statements Camel will not replace the body. This allows you to continue routing and keep the original body. The response from MyBatis is always stored in the header with the key **CamelMyBatisResult**.

225.5. SAMPLES

For example if you wish to consume beans from a JMS queue and insert them into a database you could do the following:

```
from("activemq:queue:newAccount").
  to("mybatis:insertAccount?statementType=Insert");
```

Notice we have to specify the **statementType**, as we need to instruct Camel which kind of operation to invoke.

Where **insertAccount** is the MyBatis ID in the SQL mapping file:

```
<!-- Insert example, using the Account parameter class -->
<insert id="insertAccount" parameterType="Account">
  insert into ACCOUNT (
    ACC_ID,
    ACC_FIRST_NAME,
    ACC_LAST_NAME,
    ACC_EMAIL
  )
  values (
    #{id}, #{firstName}, #{lastName}, #{emailAddress}
  )
</insert>
```

225.6. USING STATEMENTTYPE FOR BETTER CONTROL OF MYBATIS

When routing to an MyBatis endpoint you will want more fine grained control so you can control whether the SQL statement to be executed is a **SELECT**, **UPDATE**, **DELETE** or **INSERT** etc. So for instance if we want to route to an MyBatis endpoint in which the IN body contains parameters to a **SELECT** statement we can do:

In the code above we can invoke the MyBatis statement **selectAccountById** and the IN body should contain the account id we want to retrieve, such as an **Integer** type.

We can do the same for some of the other operations, such as **SelectList**:

And the same for **UPDATE**, where we can send an **Account** object as the IN body to MyBatis:

225.6.1. Using InsertList StatementType

Available as of Camel 2.10

MyBatis allows you to insert multiple rows using its for-each batch driver. To use this, you need to use the <foreach> in the mapper XML file. For example as shown below:

Then you can insert multiple rows, by sending a Camel message to the **mybatis** endpoint which uses the **InsertList** statement type, as shown below:

225.6.2. Using UpdateList StatementType

Available as of Camel 2.11

MyBatis allows you to update multiple rows using its for-each batch driver. To use this, you need to use the <foreach> in the mapper XML file. For example as shown below:

```
<update id="batchUpdateAccount" parameterType="java.util.Map">
    update ACCOUNT set
```

```
    ACC_EMAIL = #{emailAddress}
    where
    ACC_ID in
    <foreach item="Account" collection="list" open="(" close=")" separator=",">
      #{Account.id}
    </foreach>
</update>
```

Then you can update multiple rows, by sending a Camel message to the mybatis endpoint which uses the UpdateList statement type, as shown below:

```
from("direct:start")
    .to("mybatis:batchUpdateAccount?statementType=UpdateList")
    .to("mock:result");
```

225.6.3. Using DeleteList StatementType

Available as of Camel 2.11

MyBatis allows you to delete multiple rows using its for-each batch driver. To use this, you need to use the <foreach> in the mapper XML file. For example as shown below:

```
<delete id="batchDeleteAccountById" parameterType="java.util.List">
    delete from ACCOUNT
    where
    ACC_ID in
    <foreach item="AccountID" collection="list" open="(" close=")" separator=",">
      #{AccountID}
    </foreach>
</delete>
```

Then you can delete multiple rows, by sending a Camel message to the mybatis endpoint which uses the DeleteList statement type, as shown below:

```
from("direct:start")
    .to("mybatis:batchDeleteAccount?statementType=DeleteList")
    .to("mock:result");
```

225.6.4. Notice on InsertList, UpdateList and DeleteList StatementTypes

Parameter of any type (List, Map, etc.) can be passed to mybatis and an end user is responsible for handling it as required
with the help of mybatis dynamic queries capabilities.

225.6.5. Scheduled polling example

This component supports scheduled polling and can therefore be used as a Polling Consumer. For example to poll the database every minute:

```
from("mybatis:selectAllAccounts?delay=60000").to("activemq:queue:allAccounts");
```

See "ScheduledPollConsumer Options" on Polling Consumer for more options.

Alternatively you can use another mechanism for triggering the scheduled polls, such as the Timer or Quartz components. In the sample below we poll the database, every 30 seconds using the Timer component and send the data to the JMS queue:

```
from("timer://pollTheDatabase?
delay=30000").to("mybatis:selectAllAccounts").to("activemq:queue:allAccounts");
```

And the MyBatis SQL mapping file used:

```
<!-- Select with no parameters using the result map for Account class. -->
<select id="selectAllAccounts" resultMap="AccountResult">
  select * from ACCOUNT
</select>
```

225.6.6. Using onConsume

This component supports executing statements **after** data have been consumed and processed by Camel. This allows you to do post updates in the database. Notice all statements must be **UPDATE** statements. Camel supports executing multiple statements whose names should be separated by commas.

The route below illustrates we execute the **consumeAccount** statement data is processed. This allows us to change the status of the row in the database to processed, so we avoid consuming it twice or more.

And the statements in the sqlmap file:

225.6.7. Participating in transactions

Setting up a transaction manager under camel-mybatis can be a little bit fiddly, as it involves externalising the database configuration outside the standard MyBatis **SqlMapConfig.xml** file.

The first part requires the setup of a **DataSource**. This is typically a pool (either DBCP, or c3p0), which needs to be wrapped in a Spring proxy. This proxy enables non-Spring use of the **DataSource** to participate in Spring transactions (the MyBatis **SqlSessionFactory** does just this).

```
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy">
        <constructor-arg>
            <bean class="com.mchange.v2.c3p0.ComboPooledDataSource">
                <property name="driverClass" value="org.postgresql.Driver"/>
                <property name="jdbcUrl" value="jdbc:postgresql://localhost:5432/myDatabase"/>
                <property name="user" value="myUser"/>
                <property name="password" value="myPassword"/>
            </bean>
        </constructor-arg>
    </bean>
```

This has the additional benefit of enabling the database configuration to be externalised using property placeholders.

A transaction manager is then configured to manage the outermost **DataSource**:

```
    <bean id="txManager"
```

```
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

A mybatis-springSqlSessionFactoryBean then wraps that same **DataSource**:

```
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <!-- standard mybatis config file -->
    <property name="configLocation" value="/META-INF/SqlMapConfig.xml"/>
        <!-- externalised mappers -->
    <property name="mapperLocations" value="classpath*:META-INF/mappers/**/*.xml"/>
    </bean>
```

The camel-mybatis component is then configured with that factory:

```
    <bean id="mybatis" class="org.apache.camel.component.mybatis.MyBatisComponent">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
    </bean>
```

Finally, a transaction policy is defined over the top of the transaction manager, which can then be used as usual:

```
    <bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
        <property name="transactionManager" ref="txManager"/>
        <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/>
    </bean>

    <camelContext id="my-model-context" xmlns="http://camel.apache.org/schema/spring">
        <route id="insertModel">
            <from uri="direct:insert"/>
            <transacted ref="PROPAGATION_REQUIRED"/>
            <to uri="mybatis:myModel.insert?statementType=Insert"/>
        </route>
    </camelContext>
```

225.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 226. NAGIOS COMPONENT

Available as of Camel version 2.3

The Nagios component allows you to send passive checks to Nagios.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-nagios</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

226.1. URI FORMAT

```
nagios://host[:port][?Options]
```

Camel provides two abilities with the Nagios component. You can send passive check messages by sending a message to its endpoint.
Camel also provides a EventNotifer which allows you to send notifications to Nagios.

226.2. OPTIONS

The Nagios component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use a shared NagiosConfiguration		NagiosConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Nagios endpoint is configured using URI syntax:

```
nagios:host:port
```

with the following path and query parameters:

226.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** This is the address of the Nagios host where checks should be send.		String
port	**Required** The port number of the host.		int

226.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
connectionTimeout (producer)	Connection timeout in millis.	5000	int
sendSync (producer)	Whether or not to use synchronous when sending a passive check. Setting it to false will allow Camel to continue routing the message and the passive check message will be send asynchronously.	true	boolean
timeout (producer)	Sending timeout in millis.	5000	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
encryption (security)	To specify an encryption method.		Encryption
encryptionMethod (security)	**Deprecated** To specify an encryption method.		NagiosEncryptionMethod
password (security)	Password to be authenticated when sending checks to Nagios.		String

226.3. SENDING MESSAGE EXAMPLES

You can send a message to Nagios where the message payload contains the message. By default it will be **OK** level and use the CamelContext name as the service name. You can overrule these values using headers as shown above.

For example we send the **Hello Nagios** message to Nagios as follows:

```
template.sendBody("direct:start", "Hello Nagios");

from("direct:start").to("nagios:127.0.0.1:5667?password=secret").to("mock:result");
```

To send a **CRITICAL** message you can send the headers such as:

```
Map headers = new HashMap();
headers.put(NagiosConstants.LEVEL, "CRITICAL");
headers.put(NagiosConstants.HOST_NAME, "myHost");
headers.put(NagiosConstants.SERVICE_NAME, "myService");
template.sendBodyAndHeaders("direct:start", "Hello Nagios", headers);
```

226.4. USING NAGIOSEVENTNOTIFER

The Nagios component also provides an EventNotifer which you can use to send events to Nagios. For example we can enable this from Java as follows:

```
NagiosEventNotifier notifier = new NagiosEventNotifier();
notifier.getConfiguration().setHost("localhost");
notifier.getConfiguration().setPort(5667);
notifier.getConfiguration().setPassword("password");

CamelContext context = ...
context.getManagementStrategy().addEventNotifier(notifier);
return context;
```

In Spring XML its just a matter of defining a Spring bean with the type **EventNotifier** and Camel will pick it up as documented here: Advanced configuration of CamelContext using Spring .

226.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 227. NATS COMPONENT

Available as of Camel version 2.17

NATS is a fast and reliable messaging platform.

Maven users will need to add the following dependency to their **pom.xml** for this component.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-nats</artifactId>
  <!-- use the same version as your Camel core version -->
  <version>x.y.z</version>
</dependency>
```

227.1. URI FORMAT

```
nats:servers[?options]
```

Where **servers** represents the list of NATS servers.

227.2. OPTIONS

The Nats component supports 2 options which are listed below.

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Nats endpoint is configured using URI syntax:

```
nats:servers
```

with the following path and query parameters:

227.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
servers	**Required** URLs to one or more NAT servers. Use comma to separate URLs when specifying multiple servers.		String

227.2.2. Query Parameters (22 parameters):

Name	Description	Default	Type
flushConnection (common)	Define if we want to flush connection or not	false	boolean
flushTimeout (common)	Set the flush timeout	1000	int
maxReconnectAttempts (common)	Max reconnection attempts	3	int
noRandomizeServers (common)	Whether or not randomizing the order of servers for the connection attempts	false	boolean
pedantic (common)	Whether or not running in pedantic mode (this affects performace)	false	boolean
pingInterval (common)	Ping interval to be aware if connection is still alive (in milliseconds)	4000	int
reconnect (common)	Whether or not using reconnection feature	true	boolean
reconnectTimeWait (common)	Waiting time before attempts reconnection (in milliseconds)	2000	int
topic (common)	**Required** The name of topic we want to use		String
verbose (common)	Whether or not running in verbose mode	false	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
maxMessages (consumer)	Stop receiving messages from a topic we are subscribing to after maxMessages		String
poolSize (consumer)	Consumer pool size	10	int
queueName (consumer)	The Queue name if we are using nats for a queue configuration		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
replySubject (producer)	the subject to which subscribers should send response		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
secure (security)	Set secure option indicating TLS is required	false	boolean
ssl (security)	Whether or not using SSL	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
tlsDebug (security)	TLS Debug, it will add additional console output	false	boolean

227.3. HEADERS

Name	Type	Description
CamelNatsMessageTimestamp	long	The timestamp of a consumed message.
CamelNatsSubscriptionId	Integer	The subscription ID of a consumer.

Producer example:

```
from("direct:send").to("nats://localhost:4222?topic=test");
```

Consumer example:

```
from("nats://localhost:4222?topic=test&maxMessages=5&queueName=test").to("mock:result");
```

CHAPTER 228. NETTY COMPONENT (DEPRECATED)

Available as of Camel version 2.3

WARNING

This component is deprecated. You should use Netty4.

The **netty** component in Camel is a socket communication component, based on the Netty project.

Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients.
Netty greatly simplifies and streamlines network programming such as TCP and UDP socket server.

This camel component supports both producer and consumer endpoints.

The Netty component has several options and allows fine-grained control of a number of TCP/UDP communication parameters (buffer sizes, keepAlives, tcpNoDelay etc) and facilitates both In-Only and In-Out communication on a Camel route.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

228.1. URI FORMAT

The URI scheme for a netty component is as follows

```
netty:tcp://localhost:99999[?options]
netty:udp://remotehost:99999/[?options]
```

This component supports producer and consumer endpoints for both TCP and UDP.

You can append query options to the URI in the following format, **?option=value&option=value&…**

228.2. OPTIONS

The Netty component supports 4 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the NettyConfiguration as configuration when creating endpoints.		NettyConfiguration
maximumPoolSize (advanced)	The core pool size for the ordered thread pool, if its in use. The default value is 16.	16	int
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Netty endpoint is configured using URI syntax:

```
netty:protocol:host:port
```

with the following path and query parameters:

228.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol to use which can be tcp or udp.		String
host	**Required** The hostname. For the consumer the hostname is localhost or 0.0.0.0 For the producer the hostname is the remote host to connect to		String
port	**Required** The host port number		int

228.2.2. Query Parameters (67 parameters):

Name	Description	Default	Type
disconnect (common)	Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer.	false	boolean

Name	Description	Default	Type
keepAlive (common)	Setting to ensure socket is not closed due to inactivity	true	boolean
reuseAddress (common)	Setting to facilitate socket multiplexing	true	boolean
sync (common)	Setting to set endpoint as one-way or request-response	true	boolean
tcpNoDelay (common)	Setting to improve TCP protocol performance	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
broadcast (consumer)	Setting to choose Multicast over UDP	false	boolean
clientMode (consumer)	If the clientMode is true, netty consumer will connect the address as a TCP client.	false	boolean
backlog (consumer)	Allows to configure a backlog for netty consumer (server). Note the backlog is just a best effort depending on the OS. Setting this option to a value such as 200, 500 or 1000, tells the TCP stack how long the accept queue can be If this option is not configured, then the backlog depends on OS setting.		int
bossCount (consumer)	When netty works on nio mode, it uses default bossCount parameter from Netty, which is 1. User can use this operation to override the default bossCount from Netty	1	int
bossPool (consumer)	To use a explicit org.jboss.netty.channel.socket.nio.BossPool as the boss thread pool. For example to share a thread pool with multiple consumers. By default each consumer has their own boss pool with 1 core thread.		BossPool
channelGroup (consumer)	To use a explicit ChannelGroup.		ChannelGroup

Name	Description	Default	Type
disconnectOnNoReply (consumer)	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
maxChannelMemorySize (consumer)	The maximum total size of the queued events per channel when using orderedThreadPoolExecutor. Specify 0 to disable.	10485760	long
maxTotalMemorySize (consumer)	The maximum total size of the queued events for this pool when using orderedThreadPoolExecutor. Specify 0 to disable.	209715200	long
nettyServerBootstrapFactory (consumer)	To use a custom NettyServerBootstrapFactory		NettyServerBootstrap Factory
networkInterface (consumer)	When using UDP then this option can be used to specify a network interface by its name, such as eth0 to join a multicast group.		String
noReplyLogLevel (consumer)	If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back.	WARN	LoggingLevel
orderedThreadPoolExecutor (consumer)	Whether to use ordered thread pool, to ensure events are processed orderly on the same channel. See details at the netty javadoc of org.jboss.netty.handler.execution.OrderedMemoryAwareThreadPoolExecutor for more details.	true	boolean
serverClosedChannelExceptionCaughtLogLevel (consumer)	If the server (NettyConsumer) catches an java.nio.channels.ClosedChannelException then its logged using this logging level. This is used to avoid logging the closed channel exceptions, as clients can disconnect abruptly and then cause a flood of closed exceptions in the Netty server.	DEBUG	LoggingLevel

Name	Description	Default	Type
serverExceptionCaughtLog Level (consumer)	If the server (NettyConsumer) catches an exception then its logged using this logging level.	WARN	LoggingLevel
serverPipelineFactory (consumer)	To use a custom ServerPipelineFactory		ServerPipelineFactory
workerCount (consumer)	When netty works on nio mode, it uses default workerCount parameter from Netty, which is cpu_core_threads2. User can use this operation to override the default workerCount from Netty		int
workerPool (consumer)	To use a explicit org.jboss.netty.channel.socket.nio.WorkerPool as the worker thread pool. For example to share a thread pool with multiple consumers. By default each consumer has their own worker pool with 2 x cpu count core threads.		WorkerPool
connectTimeout (producer)	Time to wait for a socket connection to be available. Value is in millis.	10000	long
requestTimeout (producer)	Allows to use a timeout for the Netty producer when calling a remote server. By default no timeout is in use. The value is in milli seconds, so eg 30000 is 30 seconds. The requestTimeout is using Netty's ReadTimeoutHandler to trigger the timeout.		long
clientPipelineFactory (producer)	To use a custom ClientPipelineFactory		ClientPipelineFactory
lazyChannelCreation (producer)	Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.	true	boolean
producerPoolEnabled (producer)	Whether producer pool is enabled or not. Important: Do not turn this off, as the pooling is needed for handling concurrency and reliable request/reply.	true	boolean
producerPoolMaxActive (producer)	Sets the cap on the number of objects that can be allocated by the pool (checked out to clients, or idle awaiting checkout) at a given time. Use a negative value for no limit.	-1	int
producerPoolMaxIdle (producer)	Sets the cap on the number of idle instances in the pool.	100	int

Name	Description	Default	Type
producerPoolMinEvictable Idle (producer)	Sets the minimum amount of time (value in millis) an object may sit idle in the pool before it is eligible for eviction by the idle object evictor.	300000	long
producerPoolMinIdle (producer)	Sets the minimum number of instances allowed in the producer pool before the evictor thread (if active) spawns new objects.		int
udpConnectionlessSending (producer)	This option supports connection less udp sending which is a real fire and forget. A connected udp send receive the PortUnreachableException if no one is listen on the receiving port.	false	boolean
useChannelBuffer (producer)	If the useChannelBuffer is true, netty producer will turn the message body into ChannelBuffer before sending it out.	false	boolean
bootstrapConfiguration (advanced)	To use a custom configured NettyServerBootstrapConfiguration for configuring this endpoint.		NettyServerBootstrap Configuration
options (advanced)	Allows to configure additional netty options using option. as prefix. For example option.child.keepAlive=false to set the netty option child.keepAlive=false. See the Netty documentation for possible options that can be used.		Map
receiveBufferSize (advanced)	The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes.	65536	long
receiveBufferSizePredictor (advanced)	Configures the buffer size predictor. See details at Jetty documentation and this mail thread.		int
sendBufferSize (advanced)	The TCP/UDP buffer sizes to be used during outbound communication. Size is bytes.	65536	long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
transferExchange (advanced)	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
allowDefaultCodec (codec)	The netty component installs a default codec if both, encoder/deocder is null and textline is false. Setting allowDefaultCodec to false prevents the netty component from installing a default codec as the first element in the filter chain.	true	boolean
autoAppendDelimiter (codec)	Whether or not to auto append missing end delimiter when sending using the textline codec.	true	boolean
decoder (codec)	**Deprecated** A custom ChannelHandler class that can be used to perform special marshalling of inbound payloads. Must override org.jboss.netty.channel.ChannelUpStreamHandler.		ChannelHandler
decoderMaxLineLength (codec)	The max line length to use for the textline codec.	1024	int
decoders (codec)	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String
delimiter (codec)	The delimiter to use for the textline codec. Possible values are LINE and NULL.	LINE	TextLineDelimiter
encoder (codec)	**Deprecated** A custom ChannelHandler class that can be used to perform special marshalling of outbound payloads. Must override org.jboss.netty.channel.ChannelDownStreamHandler.		ChannelHandler
encoders (codec)	A list of encoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String

Name	Description	Default	Type
encoding (codec)	The encoding (a charset name) to use for the textline codec. If not provided, Camel will use the JVM default Charset.		String
textline (codec)	Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is false, then Object Serialization is assumed over TCP.	false	boolean
enabledProtocols (security)	Which protocols to enable when using SSL	TLSv1, TLSv1.1, TLSv1.2	String
keyStoreFile (security)	Client side certificate keystore to be used for encryption		File
keyStoreFormat (security)	Keystore format to be used for payload encryption. Defaults to JKS if not set	JKS	String
keyStoreResource (security)	Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String
needClientAuth (security)	Configures whether the server needs client authentication when using SSL.	false	boolean
passphrase (security)	Password setting to use in order to encrypt/decrypt payloads sent using SSH		String
securityProvider (security)	Security provider to be used for payload encryption. Defaults to SunX509 if not set.	SunX509	String
ssl (security)	Setting to specify whether SSL encryption is applied to this endpoint	false	boolean
sslClientCertHeaders (security)	When enabled and in SSL mode, then the Netty consumer will enrich the Camel Message with headers having information about the client certificate such as subject name, issuer name, serial number, and the valid date range.	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters

Name	Description	Default	Type
sslHandler (security)	Reference to a class that could be used to return an SSL Handler		SslHandler
trustStoreFile (security)	Server side certificate keystore to be used for encryption		File
trustStoreResource (security)	Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String

228.3. REGISTRY BASED OPTIONS

Codec Handlers and SSL Keystores can be enlisted in the Registry, such as in the Spring XML file. The values that could be passed in, are the following:

Name	Description
passphrase	password setting to use in order to encrypt/decrypt payloads sent using SSH
keyStoreFormat	keystore format to be used for payload encryption. Defaults to "JKS" if not set
securityProvider	Security provider to be used for payload encryption. Defaults to "SunX509" if not set.
keyStoreFile	**deprecated:** Client side certificate keystore to be used for encryption
trustStoreFile	**deprecated:** Server side certificate keystore to be used for encryption
keyStoreResource	**Camel 2.11.1:** Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with **"classpath:"**, **"file:"**, or **"http:"** to load the resource from different systems.
trustStoreResource	**Camel 2.11.1:** Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with **"classpath:"**, **"file:"**, or **"http:"** to load the resource from different systems.

Name	Description
sslHandler	Reference to a class that could be used to return an SSL Handler
encoder	A custom **ChannelHandler** class that can be used to perform special marshalling of outbound payloads. Must override **org.jboss.netty.channel.ChannelDownStreamHandler**.
encorders	A list of encoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.
decoder	A custom **ChannelHandler** class that can be used to perform special marshalling of inbound payloads. Must override **org.jboss.netty.channel.ChannelUpStreamHandler**.
decoders	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.

Important: Read below about using non shareable encoders/decoders.

228.3.1. Using non shareable encoders or decoders

If your encoders or decoders is not shareable (eg they have the @Shareable class annotation), then your encoder/decoder must implement the **org.apache.camel.component.netty.ChannelHandlerFactory** interface, and return a new instance in the **newChannelHandler** method. This is to ensure the encoder/decoder can safely be used. If this is not the case, then the Netty component will log a WARN when
an endpoint is created.

The Netty component offers a **org.apache.camel.component.netty.ChannelHandlerFactories** factory class, that has a number of commonly used methods.

228.4. SENDING MESSAGES TO/FROM A NETTY ENDPOINT

228.4.1. Netty Producer

In Producer mode, the component provides the ability to send payloads to a socket endpoint using either TCP or UDP protocols (with optional SSL support).

The producer mode supports both one-way and request-response based operations.

228.4.2. Netty Consumer

In Consumer mode, the component provides the ability to:

- listen on a specified socket using either TCP or UDP protocols (with optional SSL support),
- receive requests on the socket using text/xml, binary and serialized object based payloads and
- send them along on a route as message exchanges.

The consumer mode supports both one-way and request-response based operations.

228.5. HEADERS

The following headers are filled for the exchanges created by the Netty consumer:

Header key	Class	Description
NettyConstants.NETTY_CHANNEL_HANDLER_CONTEXT / CamelNettyChannelHandlerContext	**org.jboss.netty.channel.ChannelHandlerContext**	`ChannelHandlerContext `instance associated with the connection received by netty.
NettyConstants.NETTY_MESSAGE_EVENT / CamelNettyMessageEvent	**org.jboss.netty.channel.MessageEvent**	`MessageEvent `instance associated with the connection received by netty.
NettyConstants.NETTY_REMOTE_ADDRESS / CamelNettyRemoteAddress	**java.net.SocketAddress**	Remote address of the incoming socket connection.

Header key	Class	Description
NettyConstants.NETTY_LOCAL_ADDRESS / CamelNettyLocalAddress	**java.net.SocketAddress**	Local address of the incoming socket connection.

228.6. USAGE SAMPLES

228.6.1. A UDP Netty endpoint using Request-Reply and serialized object payload

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    from("netty:udp://localhost:5155?sync=true")
      .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
          Poetry poetry = (Poetry) exchange.getIn().getBody();
          poetry.setPoet("Dr. Sarojini Naidu");
          exchange.getOut().setBody(poetry);
        }
      }
    }
};
```

228.6.2. A TCP based Netty consumer endpoint using One-way communication

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
      from("netty:tcp://localhost:5150")
        .to("mock:result");
  }
};
```

228.6.3. An SSL/TCP based Netty consumer endpoint using Request-Reply communication

Using the JSSE Configuration Utility

As of Camel 2.9, the Netty component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate

how to use the utility with the Netty component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

NettyComponent nettyComponent = getContext().getComponent("netty", NettyComponent.class);
nettyComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
     id="sslContextParameters">
    <camel:keyManagers
       keyPassword="keyPassword">
      <camel:keyStore
         resource="/users/home/server/keystore.jks"
         password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="netty:tcp://localhost:5150?
sync=true&ssl=true&sslContextParameters=#sslContextParameters"/>
...
```

Using Basic SSL/TLS configuration on the Jetty Component

```
JndiRegistry registry = new JndiRegistry(createJndiContext());
registry.bind("password", "changeit");
registry.bind("ksf", new File("src/test/resources/keystore.jks"));
registry.bind("tsf", new File("src/test/resources/keystore.jks"));

context.createRegistry(registry);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
        "netty:tcp://localhost:5150?sync=true&ssl=true&passphrase=#password"
        + "&keyStoreFile=#ksf&trustStoreFile=#tsf";
      String return_string =
        "When You Go Home, Tell Them Of Us And Say,"
        + "For Your Tomorrow, We Gave Our Today.";

      from(netty_ssl_endpoint)
       .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
```

```
            exchange.getOut().setBody(return_string);
          }
        }
      }
    });
```

Getting access to SSLSession and the client certificate

Available as of Camel 2.12

You can get access to the **javax.net.ssl.SSLSession** if you eg need to get details about the client certificate. When **ssl=true** then the Netty component will store the **SSLSession** as a header on the Camel Message as shown below:

```
SSLSession session = exchange.getIn().getHeader(NettyConstants.NETTY_SSL_SESSION,
SSLSession.class);
// get the first certificate which is client certificate
javax.security.cert.X509Certificate cert = session.getPeerCertificateChain()[0];
Principal principal = cert.getSubjectDN();
```

Remember to set **needClientAuth=true** to authenticate the client, otherwise **SSLSession** cannot access information about the client certificate, and you may get an exception **javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated**. You may also get this exception if the client certificate is expired or not valid etc.

TIP

The option **sslClientCertHeaders** can be set to **true** which then enriches the Camel Message with headers having details about the client certificate. For example the subject name is readily available in the header **CamelNettySSLClientCertSubjectName**.

228.6.4. Using Multiple Codecs

In certain cases it may be necessary to add chains of encoders and decoders to the netty pipeline. To add multpile codecs to a camel netty endpoint the 'encoders' and 'decoders' uri parameters should be used. Like the 'encoder' and 'decoder' parameters they are used to supply references (to lists of ChannelUpstreamHandlers and ChannelDownstreamHandlers) that should be added to the pipeline. Note that if encoders is specified then the encoder param will be ignored, similarly for decoders and the decoder param.

INFO: Read further above about using non shareable encoders/decoders.

The lists of codecs need to be added to the Camel's registry so they can be resolved when the endpoint is created.

Spring's native collections support can be used to specify the codec lists in an application context

The bean names can then be used in netty endpoint definitions either as a comma separated list or contained in a List e.g.

or via spring.

228.7. CLOSING CHANNEL WHEN COMPLETE

When acting as a server you sometimes want to close the channel when, for example, a client conversion is finished.
You can do this by simply setting the endpoint option **disconnect=true**.

However you can also instruct Camel on a per message basis as follows.
To instruct Camel to close the channel, you should add a header with the key **CamelNettyCloseChannelWhenComplete** set to a boolean **true** value.
For instance, the example below will close the channel after it has written the bye message back to the client:

```
        from("netty:tcp://localhost:8080").process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                String body = exchange.getIn().getBody(String.class);
                exchange.getOut().setBody("Bye " + body);
                // some condition which determines if we should close
                if (close) {

exchange.getOut().setHeader(NettyConstants.NETTY_CLOSE_CHANNEL_WHEN_COMPLETE,
true);
                }
            }
        });
```

228.8. ADDING CUSTOM CHANNEL PIPELINE FACTORIES TO GAIN COMPLETE CONTROL OVER A CREATED PIPELINE

Available as of Camel 2.5

Custom channel pipelines provide complete control to the user over the handler/interceptor chain by inserting custom handler(s), encoder(s) & decoders without having to specify them in the Netty Endpoint URL in a very simple way.

In order to add a custom pipeline, a custom channel pipeline factory must be created and registered with the context via the context registry (JNDIRegistry,or the camel-spring ApplicationContextRegistry etc).

A custom pipeline factory must be constructed as follows

- A Producer linked channel pipeline factory must extend the abstract class **ClientPipelineFactory**.
- A Consumer linked channel pipeline factory must extend the abstract class **ServerPipelineFactory**.
- The classes should override the getPipeline() method in order to insert custom handler(s), encoder(s) and decoder(s). Not overriding the getPipeline() method creates a pipeline with no handlers, encoders or decoders wired to the pipeline.

The example below shows how ServerChannel Pipeline factory may be created

Using custom pipeline factory

```
public class SampleServerChannelPipelineFactory extends ServerPipelineFactory {
    private int maxLineSize = 1024;

    public ChannelPipeline getPipeline() throws Exception {
```

```
        ChannelPipeline channelPipeline = Channels.pipeline();

        channelPipeline.addLast("encoder-SD", new StringEncoder(CharsetUtil.UTF_8));
        channelPipeline.addLast("decoder-DELIM", new DelimiterBasedFrameDecoder(maxLineSize,
true, Delimiters.lineDelimiter()));
        channelPipeline.addLast("decoder-SD", new StringDecoder(CharsetUtil.UTF_8));
        // here we add the default Camel ServerChannelHandler for the consumer, to allow Camel to
route the message etc.
        channelPipeline.addLast("handler", new ServerChannelHandler(consumer));

        return channelPipeline;
    }
}
```

The custom channel pipeline factory can then be added to the registry and instantiated/utilized on a camel route in the following way

```
Registry registry = camelContext.getRegistry();
serverPipelineFactory = new TestServerChannelPipelineFactory();
registry.bind("spf", serverPipelineFactory);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
        "netty:tcp://localhost:5150?serverPipelineFactory=#spf"
      String return_string =
        "When You Go Home, Tell Them Of Us And Say,"
        + "For Your Tomorrow, We Gave Our Today.";

      from(netty_ssl_endpoint)
       .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            exchange.getOut().setBody(return_string);
          }
       }
  }
});
```

228.9. REUSING NETTY BOSS AND WORKER THREAD POOLS

Available as of Camel 2.12

Netty has two kind of thread pools: boss and worker. By default each Netty consumer and producer has their private thread pools. If you want to reuse these thread pools among multiple consumers or producers then the thread pools must be created and enlisted in the Registry.

For example using Spring XML we can create a shared worker thread pool using the **NettyWorkerPoolBuilder** with 2 worker threads as shown below:

```
<!-- use the worker pool builder to help create the shared thread pool -->
<bean id="poolBuilder" class="org.apache.camel.component.netty.NettyWorkerPoolBuilder">
  <property name="workerCount" value="2"/>
</bean>

<!-- the shared worker thread pool -->
```

```
<bean id="sharedPool" class="org.jboss.netty.channel.socket.nio.WorkerPool"
    factory-bean="poolBuilder" factory-method="build" destroy-method="shutdown">
</bean>
```

TIP

For boss thread pool there is a **org.apache.camel.component.netty.NettyServerBossPoolBuilder** builder for Netty consumers, and a **org.apache.camel.component.netty.NettyClientBossPoolBuilder** for the Netty producers.

Then in the Camel routes we can refer to this worker pools by configuring the **workerPool** option in the URI as shown below:

```
<route>
  <from uri="netty:tcp://localhost:5021?
textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;orderedThreadPoolExecutor=false"
/>
  <to uri="log:result"/>
  ...
</route>
```

And if we have another route we can refer to the shared worker pool:

```
<route>
  <from uri="netty:tcp://localhost:5022?
textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;orderedThreadPoolExecutor=false"
/>
  <to uri="log:result"/>
  ...
</route>
```

i. and so forth.

228.10. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Netty HTTP
- MINA

CHAPTER 229. NETTY HTTP COMPONENT (DEPRECATED)

Available as of Camel version 2.12

The **netty-http** component is an extension to Netty component to facilitiate HTTP transport with Netty.

This camel component supports both producer and consumer endpoints.

WARNING

This component is deprecated. You should use Netty4 HTTP.

INFO: **Stream**. Netty is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**. If you find a situation where the message body appears to be empty or you need to access the data multiple times (eg: doing multicasting, or redelivery error handling) you should use Stream caching or convert the message body to a **String** which is safe to be re-read multiple times. Notice Netty4 HTTP reads the entire stream into memory using **io.netty.handler.codec.http.HttpObjectAggregator** to build the entire full http message. But the resulting message is still a stream based message which is readable once.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty-http</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

229.1. URI FORMAT

The URI scheme for a netty component is as follows

```
netty-http:http://localhost:8080[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&...**

INFO: **Query parameters vs endpoint options** You may be wondering how Camel recognizes URI query parameters and endpoint options. For example you might create endpoint URI as follows - **netty-http:http//example.com?myParam=myValue&compression=true** . In this example **myParam** is the HTTP parameter, while **compression** is the Camel endpoint option. The strategy used by Camel in such situations is to resolve available endpoint options and remove them from the URI. It means that for the discussed example, the HTTP request sent by Netty HTTP producer to the endpoint will look as follows - **http//example.com?myParam=myValue** , because **compression** endpoint option will be resolved and removed from the target URL. Keep also in mind that you cannot specify endpoint options using dynamic headers (like **CamelHttpQuery**). Endpoint options can be specified only at the endpoint URI definition level (like **to** or **from** DSL elements).

229.2. HTTP OPTIONS

INFO: **A lot more options. Important:** This component inherits all the options from Netty. So make sure to look at the Netty documentation as well.
Notice that some options from Netty is not applicable when using this Netty HTTP component, such as options related to UDP transport.

The Netty HTTP component supports 7 options which are listed below.

Name	Description	Default	Type
nettyHttpBinding (advanced)	To use a custom org.apache.camel.component.netty.http.NettyHttpBinding for binding to/from Netty and Camel Message API.		NettyHttpBinding
configuration (common)	To use the NettyConfiguration as configuration when creating endpoints.		NettyHttpConfiguration
headerFilterStrategy (advanced)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter headers.		HeaderFilterStrategy
securityConfiguration (security)	Refers to a org.apache.camel.component.netty.http.NettyHttpSecurityConfiguration for configuring secure web resources.		NettyHttpSecurityConfiguration
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
maximumPoolSize (advanced)	The core pool size for the ordered thread pool, if its in use. The default value is 16.	16	int
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Netty HTTP endpoint is configured using URI syntax:

```
netty-http:protocol:host:port/path
```

with the following path and query parameters:

229.2.1. Path Parameters (4 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol to use which is either http or https		String
host	**Required** The local hostname such as localhost, or 0.0.0.0 when being a consumer. The remote HTTP server hostname when using producer.		String
port	The host port number		int
path	Resource path		String

229.2.2. Query Parameters (78 parameters):

Name	Description	Default	Type
bridgeEndpoint (common)	If the option is true, the producer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the producer send all the fault response back. The consumer working in the bridge mode will skip the gzip compression and WWW URL form encoding (by adding the Exchange.SKIP_GZIP_ENCODING and Exchange.SKIP_WWW_FORM_URLENCODED headers to the consumed exchange).	false	boolean
disconnect (common)	Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer.	false	boolean
keepAlive (common)	Setting to ensure socket is not closed due to inactivity	true	boolean
reuseAddress (common)	Setting to facilitate socket multiplexing	true	boolean
sync (common)	Setting to set endpoint as one-way or request-response	true	boolean
tcpNoDelay (common)	Setting to improve TCP protocol performance	true	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
matchOnUriPrefix (consumer)	Whether or not Camel should try to find a target consumer by matching the URI prefix if no exact match is found.	false	boolean
send503whenSuspended (consumer)	Whether to send back HTTP status code 503 when the consumer has been suspended. If the option is false then the Netty Acceptor is unbound when the consumer is suspended, so clients cannot connect anymore.	true	boolean
backlog (consumer)	Allows to configure a backlog for netty consumer (server). Note the backlog is just a best effort depending on the OS. Setting this option to a value such as 200, 500 or 1000, tells the TCP stack how long the accept queue can be If this option is not configured, then the backlog depends on OS setting.		int
bossCount (consumer)	When netty works on nio mode, it uses default bossCount parameter from Netty, which is 1. User can use this operation to override the default bossCount from Netty	1	int
bossPool (consumer)	To use a explicit org.jboss.netty.channel.socket.nio.BossPool as the boss thread pool. For example to share a thread pool with multiple consumers. By default each consumer has their own boss pool with 1 core thread.		BossPool
channelGroup (consumer)	To use a explicit ChannelGroup.		ChannelGroup
chunkedMaxContentLength (consumer)	Value in bytes the max content length per chunked frame received on the Netty HTTP server.	1048576	int
compression (consumer)	Allow using gzip/deflate for compression on the Netty HTTP server if the client supports it from the HTTP headers.	false	boolean

Name	Description	Default	Type
disableStreamCache (consumer)	Determines whether or not the raw input stream from Netty HttpRequestgetContent() is cached or not (Camel will read the stream into a in light-weight memory based Stream caching) cache. By default Camel will cache the Netty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. Mind that if you enable this option, then you cannot read the Netty stream multiple times out of the box, and you would need manually to reset the reader index on the Netty raw stream.	false	boolean
disconnectOnNoReply (consumer)	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
httpMethodRestrict (consumer)	To disable HTTP methods on the Netty HTTP consumer. You can specify multiple separated by comma.		String
mapHeaders (consumer)	If this option is enabled, then during binding from Netty to Camel Message then the headers will be mapped as well (eg added as header to the Camel Message as well). You can turn off this option to disable this. The headers can still be accessed from the org.apache.camel.component.netty.http.NettyHttpMessage message with the method getHttpRequest() that returns the Netty HTTP request org.jboss.netty.handler.codec.http.HttpRequest instance.	true	boolean
maxChannelMemorySize (consumer)	The maximum total size of the queued events per channel when using orderedThreadPoolExecutor. Specify 0 to disable.	10485760	long

Name	Description	Default	Type
maxHeaderSize (consumer)	The maximum length of all headers. If the sum of the length of each header exceeds this value, a TooLongFrameException will be raised.	8192	int
maxTotalMemorySize (consumer)	The maximum total size of the queued events for this pool when using orderedThreadPoolExecutor. Specify 0 to disable.	209715200	long
nettyServerBootstrapFactory (consumer)	To use a custom NettyServerBootstrapFactory		NettyServerBootstrap Factory
nettySharedHttpServer (consumer)	To use a shared Netty HTTP server. See Netty HTTP Server Example for more details.		NettySharedHttpServer
noReplyLogLevel (consumer)	If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back.	WARN	LoggingLevel
orderedThreadPoolExecutor (consumer)	Whether to use ordered thread pool, to ensure events are processed orderly on the same channel. See details at the netty javadoc of org.jboss.netty.handler.execution.OrderedMemoryAwareThreadPoolExecutor for more details.	true	boolean
serverClosedChannelExceptionCaughtLogLevel (consumer)	If the server (NettyConsumer) catches an java.nio.channels.ClosedChannelException then its logged using this logging level. This is used to avoid logging the closed channel exceptions, as clients can disconnect abruptly and then cause a flood of closed exceptions in the Netty server.	DEBUG	LoggingLevel
serverExceptionCaughtLog Level (consumer)	If the server (NettyConsumer) catches an exception then its logged using this logging level.	WARN	LoggingLevel
serverPipelineFactory (consumer)	To use a custom ServerPipelineFactory		ServerPipelineFactory
traceEnabled (consumer)	Specifies whether to enable HTTP TRACE for this Netty HTTP consumer. By default TRACE is turned off.	false	boolean

Name	Description	Default	Type
urlDecodeHeaders (consumer)	If this option is enabled, then during binding from Netty to Camel Message then the header values will be URL decoded (eg %20 will be a space character. Notice this option is used by the default org.apache.camel.component.netty.http.NettyHttpBinding and therefore if you implement a custom org.apache.camel.component.netty.http.NettyHttpBinding then you would need to decode the headers accordingly to this option.	false	boolean
workerCount (consumer)	When netty works on nio mode, it uses default workerCount parameter from Netty, which is cpu_core_threads2. User can use this operation to override the default workerCount from Netty		int
workerPool (consumer)	To use a explicit org.jboss.netty.channel.socket.nio.WorkerPool as the worker thread pool. For example to share a thread pool with multiple consumers. By default each consumer has their own worker pool with 2 x cpu count core threads.		WorkerPool
connectTimeout (producer)	Time to wait for a socket connection to be available. Value is in millis.	10000	long
requestTimeout (producer)	Allows to use a timeout for the Netty producer when calling a remote server. By default no timeout is in use. The value is in milli seconds, so eg 30000 is 30 seconds. The requestTimeout is using Netty's ReadTimeoutHandler to trigger the timeout.		long
throwExceptionOnFailure (producer)	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
clientPipelineFactory (producer)	To use a custom ClientPipelineFactory		ClientPipelineFactory
lazyChannelCreation (producer)	Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.	true	boolean
okStatusCodeRange (producer)	The status codes which are considered a success response. The values are inclusive. Multiple ranges can be defined, separated by comma, e.g. 200-204,209,301-304. Each range must be a single number or from-to with the dash included. The default range is 200-299	200-299	String

Name	Description	Default	Type
producerPoolEnabled (producer)	Whether producer pool is enabled or not. Important: Do not turn this off, as the pooling is needed for handling concurrency and reliable request/reply.	true	boolean
producerPoolMaxActive (producer)	Sets the cap on the number of objects that can be allocated by the pool (checked out to clients, or idle awaiting checkout) at a given time. Use a negative value for no limit.	-1	int
producerPoolMaxIdle (producer)	Sets the cap on the number of idle instances in the pool.	100	int
producerPoolMinEvictable Idle (producer)	Sets the minimum amount of time (value in millis) an object may sit idle in the pool before it is eligible for eviction by the idle object evictor.	300000	long
producerPoolMinIdle (producer)	Sets the minimum number of instances allowed in the producer pool before the evictor thread (if active) spawns new objects.		int
useChannelBuffer (producer)	If the useChannelBuffer is true, netty producer will turn the message body into ChannelBuffer before sending it out.	false	boolean
useRelativePath (producer)	Sets whether to use a relative path in HTTP requests. Some third party backend systems such as IBM Datapower do not support absolute URIs in HTTP POSTs, and setting this option to true can work around this problem.	false	boolean
bootstrapConfiguration (advanced)	To use a custom configured NettyServerBootstrapConfiguration for configuring this endpoint.		NettyServerBootstrap Configuration
configuration (advanced)	To use a custom configured NettyHttpConfiguration for configuring this endpoint.		NettyHttpConfiguration
headerFilterStrategy (advanced)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter headers.		HeaderFilterStrategy
nettyHttpBinding (advanced)	To use a custom org.apache.camel.component.netty.http.NettyHttpBinding for binding to/from Netty and Camel Message API.		NettyHttpBinding

Name	Description	Default	Type
options (advanced)	Allows to configure additional netty options using option. as prefix. For example option.child.keepAlive=false to set the netty option child.keepAlive=false. See the Netty documentation for possible options that can be used.		Map
receiveBufferSize (advanced)	The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes.	65536	long
receiveBufferSizePredictor (advanced)	Configures the buffer size predictor. See details at Jetty documentation and this mail thread.		int
sendBufferSize (advanced)	The TCP/UDP buffer sizes to be used during outbound communication. Size is bytes.	65536	long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
transferException (advanced)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
transferExchange (advanced)	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
decoder (codec)	**Deprecated** To use a single decoder. This options is deprecated use encoders instead.		ChannelHandler
decoders (codec)	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String

Name	Description	Default	Type
encoder (codec)	**Deprecated** To use a single encoder. This options is deprecated use encoders instead.		ChannelHandler
encoders (codec)	A list of encoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String
enabledProtocols (security)	Which protocols to enable when using SSL	TLSv1, TLSv1.1, TLSv1.2	String
keyStoreFile (security)	Client side certificate keystore to be used for encryption		File
keyStoreFormat (security)	Keystore format to be used for payload encryption. Defaults to JKS if not set	JKS	String
keyStoreResource (security)	Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String
needClientAuth (security)	Configures whether the server needs client authentication when using SSL.	false	boolean
passphrase (security)	Password setting to use in order to encrypt/decrypt payloads sent using SSH		String
securityConfiguration (security)	Refers to a org.apache.camel.component.netty.http.NettyHttpSecurityConfiguration for configuring secure web resources.		NettyHttpSecurityConfiguration
securityOptions (security)	To configure NettyHttpSecurityConfiguration using key/value pairs from the map		Map
securityProvider (security)	Security provider to be used for payload encryption. Defaults to SunX509 if not set.	SunX509	String
ssl (security)	Setting to specify whether SSL encryption is applied to this endpoint	false	boolean

Name	Description	Default	Type
sslClientCertHeaders (security)	When enabled and in SSL mode, then the Netty consumer will enrich the Camel Message with headers having information about the client certificate such as subject name, issuer name, serial number, and the valid date range.	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
sslHandler (security)	Reference to a class that could be used to return an SSL Handler		SslHandler
trustStoreFile (security)	Server side certificate keystore to be used for encryption		File
trustStoreResource (security)	Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String

229.3. MESSAGE HEADERS

The following headers can be used on the producer to control the HTTP request.

Name	Type	Description
CamelHttpMethod	**String**	Allow to control what HTTP method to use such as GET, POST, TRACE etc. The type can also be a **org.jboss.netty.handler.codec.http.HttpMethod** instance.
CamelHttpQuery	**String**	Allows to provide URI query parameters as a **String** value that overrides the endpoint configuration. Separate multiple parameters using the & sign. For example: **foo=bar&beer=yes**.
CamelHttpPath	**String**	**Camel 2.13.1/2.12.4:** Allows to provide URI context-path and query parameters as a **String** value that overrides the endpoint configuration. This allows to reuse the same producer for calling same remote http server, but using a dynamic context-path and query parameters.
Content-Type	**String**	To set the content-type of the HTTP body. For example: **text/plain; charset="UTF-8"**.

Name	Type	Description
CamelHttpResponseCode	**int**	Allows to set the HTTP Status code to use. By default 200 is used for success, and 500 for failure.

The following headers is provided as meta-data when a route starts from an Netty HTTP endpoint:

The description in the table takes offset in a route having: **from("netty-http:http:0.0.0.0:8080/myapp") ...**

Name	Type	Description
CamelHttpMethod	**String**	The HTTP method used, such as GET, POST, TRACE etc.
CamelHttpUrl	**String**	The URL including protocol, host and port, etc
CamelHttpUri	**String**	The URI without protocol, host and port, etc
CamelHttpQuery	**String**	Any query parameters, such as **foo=bar&beer=yes**
CamelHttpRawQuery	**String**	Camel 2.13.0: Any query parameters, such as**foo=bar&beer=yes**. Stored in the raw form, as they arrived to the consumer (i.e. before URL decoding).
CamelHttpPath	**String**	Additional context-path. This value is empty if the client called the context-path **/myapp**. If the client calls**/myapp/mystuff**, then this header value is**/mystuff**. In other words its the value after the context-path configured on the route endpoint.
CamelHttpCharacterEncoding	**String**	The charset from the content-type header.

Name	Type	Description
CamelHttpAuthentication	**String**	If the user was authenticated using HTTP Basic then this header is added with the value **Basic**.
Content-Type	**String**	The content type if provided. For example: **text/plain; charset="UTF-8"**.

229.4. ACCESS TO NETTY TYPES

This component uses the **org.apache.camel.component.netty.http.NettyHttpMessage** as the message implementation on the Exchange. This allows end users to get access to the original Netty request/response instances if needed, as shown below. Mind that the original response may not be accessible at all times.

```
org.jboss.netty.handler.codec.http.HttpRequest request =
exchange.getIn(NettyHttpMessage.class).getHttpRequest();
```

229.5. EXAMPLES

In the route below we use Netty HTTP as a HTTP server, which returns back a hardcoded "Bye World" message.

```
    from("netty-http:http://0.0.0.0:8080/foo")
      .transform().constant("Bye World");
```

And we can call this HTTP server using Camel also, with the ProducerTemplate as shown below:

```
    String out = template.requestBody("netty-http:http://localhost:8080/foo", "Hello World",
String.class);
    System.out.println(out);
```

And we get back "Bye World" as the output.

229.6. HOW DO I LET NETTY MATCH WILDCARDS

By default Netty HTTP will only match on exact uri's. But you can instruct Netty to match prefixes. For example

```
from("netty-http:http://0.0.0.0:8123/foo").to("mock:foo");
```

In the route above Netty HTTP will only match if the uri is an exact match, so it will match if you enter **http://0.0.0.0:8123/foo** but not match if you do **http://0.0.0.0:8123/foo/bar**.

So if you want to enable wildcard matching you do as follows:

```
from("netty-http:http://0.0.0.0:8123/foo?matchOnUriPrefix=true").to("mock:foo");
```

So now Netty matches any endpoints with starts with **foo**.

To match **any** endpoint you can do:

```
from("netty-http:http://0.0.0.0:8123?matchOnUriPrefix=true").to("mock:foo");
```

229.7. USING MULTIPLE ROUTES WITH SAME PORT

In the same CamelContext you can have multiple routes from Netty HTTP that shares the same port (eg a **org.jboss.netty.bootstrap.ServerBootstrap** instance). Doing this requires a number of bootstrap options to be identical in the routes, as the routes will share the same **org.jboss.netty.bootstrap.ServerBootstrap** instance. The instance will be configured with the options from the first route created.

The options the routes must be identical configured is all the options defined in the **org.apache.camel.component.netty.NettyServerBootstrapConfiguration** configuration class. If you have configured another route with different options, Camel will throw an exception on startup, indicating the options is not identical. To mitigate this ensure all options is identical.

Here is an example with two routes that share the same port.

Two routes sharing the same port

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

from("netty-http:http://0.0.0.0:{{port}}/bar")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

And here is an example of a mis configured 2nd route that do not have identical **org.apache.camel.component.netty.NettyServerBootstrapConfiguration** option as the 1st route. This will cause Camel to fail on startup.

Two routes sharing the same port, but the 2nd route is misconfigured and will fail on starting

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

// we cannot have a 2nd route on same port with SSL enabled, when the 1st route is NOT
from("netty-http:http://0.0.0.0:{{port}}/bar?ssl=true")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

229.7.1. Reusing same server bootstrap configuration with multiple routes

By configuring the common server bootstrap option in an single instance of a **org.apache.camel.component.netty.NettyServerBootstrapConfiguration** type, we can use the **bootstrapConfiguration** option on the Netty HTTP consumers to refer and reuse the same options

across all consumers.

```
<bean id="nettyHttpBootstrapOptions"
class="org.apache.camel.component.netty.NettyServerBootstrapConfiguration">
  <property name="backlog" value="200"/>
  <property name="connectTimeout" value="20000"/>
  <property name="workerCount" value="16"/>
</bean>
```

And in the routes you refer to this option as shown below

```
<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/foo?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/bar?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/beer?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>
```

229.7.2. Reusing same server bootstrap configuration with multiple routes across multiple bundles in OSGi container

See the Netty HTTP Server Example for more details and example how to do that.

229.8. USING HTTP BASIC AUTHENTICATION

The Netty HTTP consumer supports HTTP basic authentication by specifying the security realm name to use, as shown below

```
<route>
   <from uri="netty-http:http://0.0.0.0:{{port}}/foo?securityConfiguration.realm=karaf"/>
   ...
</route>
```

The realm name is mandatory to enable basic authentication. By default the JAAS based authenticator is used, which will use the realm name specified (karaf in the example above) and use the JAAS realm and the JAAS \{{LoginModule}}s of this realm for authentication.

End user of Apache Karaf / ServiceMix has a karaf realm out of the box, and hence why the example above would work out of the box in these containers.

229.8.1. Specifying ACL on web resources

The **org.apache.camel.component.netty.http.SecurityConstraint** allows to define constrains on web resources. And the **org.apache.camel.component.netty.http.SecurityConstraintMapping** is provided out of the box, allowing to easily define inclusions and exclusions with roles.

For example as shown below in the XML DSL, we define the constraint bean:

```
<bean id="constraint" class="org.apache.camel.component.netty.http.SecurityConstraintMapping">
  <!-- inclusions defines url -> roles restrictions -->
  <!-- a * should be used for any role accepted (or even no roles) -->
  <property name="inclusions">
    <map>
      <entry key="/*" value="*"/>
      <entry key="/admin/*" value="admin"/>
      <entry key="/guest/*" value="admin,guest"/>
    </map>
  </property>
  <!-- exclusions is used to define public urls, which requires no authentication -->
  <property name="exclusions">
    <set>
      <value>/public/*</value>
    </set>
  </property>
</bean>
```

The constraint above is define so that

- access to /* is restricted and any roles is accepted (also if user has no roles)
- access to /admin/* requires the admin role
- access to /guest/* requires the admin or guest role
- access to /public/* is an exclusion which means no authentication is needed, and is therefore public for everyone without logging in

To use this constraint we just need to refer to the bean id as shown below:

```
<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/foo?
matchOnUriPrefix=true&amp;securityConfiguration.realm=karaf&amp;securityConfiguration.securityCon
straint=#constraint"/>
  ...
</route>
```

229.9. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Netty

- Netty HTTP Server Example
- Jetty

CHAPTER 230. NETTY4 COMPONENT

Available as of Camel version 2.14

The **netty4** component in Camel is a socket communication component, based on the Netty project version 4.
Netty is a NIO client server framework which enables quick and easy development of netwServerInitializerFactoryork applications such as protocol servers and clients.
Netty greatly simplifies and streamlines network programming such as TCP and UDP socket server.

This camel component supports both producer and consumer endpoints.

The Netty component has several options and allows fine-grained control of a number of TCP/UDP communication parameters (buffer sizes, keepAlives, tcpNoDelay etc) and facilitates both In-Only and In-Out communication on a Camel route.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

230.1. URI FORMAT

The URI scheme for a netty component is as follows

```
netty4:tcp://localhost:99999[?options]
netty4:udp://remotehost:99999/[?options]
```

This component supports producer and consumer endpoints for both TCP and UDP.

You can append query options to the URI in the following format, **?option=value&option=value&…**

230.2. OPTIONS

The Netty4 component supports 5 options which are listed below.

Name	Description	Default	Type
maximumPoolSize (advanced)	The thread pool size for the EventExecutorGroup if its in use. The default value is 16.	16	int
configuration (advanced)	To use the NettyConfiguration as configuration when creating endpoints.		NettyConfiguration
executorService (advanced)	To use the given EventExecutorGroup		EventExecutorGroup

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Netty4 endpoint is configured using URI syntax:

```
netty4:protocol:host:port
```

with the following path and query parameters:

230.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol to use which can be tcp or udp.		String
host	**Required** The hostname. For the consumer the hostname is localhost or 0.0.0.0 For the producer the hostname is the remote host to connect to		String
port	**Required** The host port number		int

230.2.2. Query Parameters (72 parameters):

Name	Description	Default	Type
disconnect (common)	Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer.	false	boolean
keepAlive (common)	Setting to ensure socket is not closed due to inactivity	true	boolean
reuseAddress (common)	Setting to facilitate socket multiplexing	true	boolean

Name	Description	Default	Type
reuseChannel (common)	This option allows producers and consumers (in client mode) to reuse the same Netty Channel for the lifecycle of processing the Exchange. This is useful if you need to call a server multiple times in a Camel route and want to use the same network connection. When using this the channel is not returned to the connection pool until the Exchange is done; or disconnected if the disconnect option is set to true. The reused Channel is stored on the Exchange as an exchange property with the key link NettyConstantsNETTY_CHANNEL which allows you to obtain the channel during routing and use it as well.	false	boolean
sync (common)	Setting to set endpoint as one-way or request-response	true	boolean
tcpNoDelay (common)	Setting to improve TCP protocol performance	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
broadcast (consumer)	Setting to choose Multicast over UDP	false	boolean
clientMode (consumer)	If the clientMode is true, netty consumer will connect the address as a TCP client.	false	boolean
reconnect (consumer)	Used only in clientMode in consumer, the consumer will attempt to reconnect on disconnection if this is enabled	true	boolean
reconnectInterval (consumer)	Used if reconnect and clientMode is enabled. The interval in milli seconds to attempt reconnection	10000	int
backlog (consumer)	Allows to configure a backlog for netty consumer (server). Note the backlog is just a best effort depending on the OS. Setting this option to a value such as 200, 500 or 1000, tells the TCP stack how long the accept queue can be If this option is not configured, then the backlog depends on OS setting.		int

Name	Description	Default	Type
bossCount (consumer)	When netty works on nio mode, it uses default bossCount parameter from Netty, which is 1. User can use this operation to override the default bossCount from Netty	1	int
bossGroup (consumer)	Set the BossGroup which could be used for handling the new connection of the server side across the NettyEndpoint		EventLoopGroup
disconnectOnNoReply (consumer)	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
nettyServerBootstrapFactory (consumer)	To use a custom NettyServerBootstrapFactory		NettyServerBootstrap Factory
networkInterface (consumer)	When using UDP then this option can be used to specify a network interface by its name, such as eth0 to join a multicast group.		String
noReplyLogLevel (consumer)	If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back.	WARN	LoggingLevel
serverClosedChannelExceptionCaughtLogLevel (consumer)	If the server (NettyConsumer) catches an java.nio.channels.ClosedChannelException then its logged using this logging level. This is used to avoid logging the closed channel exceptions, as clients can disconnect abruptly and then cause a flood of closed exceptions in the Netty server.	DEBUG	LoggingLevel
serverExceptionCaughtLog Level (consumer)	If the server (NettyConsumer) catches an exception then its logged using this logging level.	WARN	LoggingLevel
serverInitializerFactory (consumer)	To use a custom ServerInitializerFactory		ServerInitializer Factory

Name	Description	Default	Type
usingExecutorService (consumer)	Whether to use ordered thread pool, to ensure events are processed orderly on the same channel.	true	boolean
connectTimeout (producer)	Time to wait for a socket connection to be available. Value is in millis.	10000	int
requestTimeout (producer)	Allows to use a timeout for the Netty producer when calling a remote server. By default no timeout is in use. The value is in milli seconds, so eg 30000 is 30 seconds. The requestTimeout is using Netty's ReadTimeoutHandler to trigger the timeout.		long
clientInitializerFactory (producer)	To use a custom ClientInitializerFactory		ClientInitializerFactory
correlationManager (producer)	To use a custom correlation manager to manage how request and reply messages are mapped when using request/reply with the netty producer. This should only be used if you have a way to map requests together with replies such as if there is correlation ids in both the request and reply messages. This can be used if you want to multiplex concurrent messages on the same channel (aka connection) in netty. When doing this you must have a way to correlate the request and reply messages so you can store the right reply on the inflight Camel Exchange before its continued routed. We recommend extending the TimeoutCorrelationManagerSupport when you build custom correlation managers. This provides support for timeout and other complexities you otherwise would need to implement as well. See also the producerPoolEnabled option for more details.		NettyCamelStateCorrelationManager
lazyChannelCreation (producer)	Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.	true	boolean

Name	Description	Default	Type
producerPoolEnabled (producer)	Whether producer pool is enabled or not. Important: If you turn this off then a single shared connection is used for the producer, also if you are doing request/reply. That means there is a potential issue with interleaved responses if replies comes back out-of-order. Therefore you need to have a correlation id in both the request and reply messages so you can properly correlate the replies to the Camel callback that is responsible for continue processing the message in Camel. To do this you need to implement NettyCamelStateCorrelationManager as correlation manager and configure it via the correlationManager option. See also the correlationManager option for more details.	true	boolean
producerPoolMaxActive (producer)	Sets the cap on the number of objects that can be allocated by the pool (checked out to clients, or idle awaiting checkout) at a given time. Use a negative value for no limit.	-1	int
producerPoolMaxIdle (producer)	Sets the cap on the number of idle instances in the pool.	100	int
producerPoolMinEvictableIdle (producer)	Sets the minimum amount of time (value in millis) an object may sit idle in the pool before it is eligible for eviction by the idle object evictor.	300000	long
producerPoolMinIdle (producer)	Sets the minimum number of instances allowed in the producer pool before the evictor thread (if active) spawns new objects.		int
udpConnectionlessSending (producer)	This option supports connection less udp sending which is a real fire and forget. A connected udp send receive the PortUnreachableException if no one is listen on the receiving port.	false	boolean
useByteBuf (producer)	If the useByteBuf is true, netty producer will turn the message body into ByteBuf before sending it out.	false	boolean
allowSerializedHeaders (advanced)	Only used for TCP when transferExchange is true. When set to true, serializable objects in headers and properties will be added to the exchange. Otherwise Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
bootstrapConfiguration (advanced)	To use a custom configured NettyServerBootstrapConfiguration for configuring this endpoint.		NettyServerBootstrap Configuration

Name	Description	Default	Type
channelGroup (advanced)	To use a explicit ChannelGroup.		ChannelGroup
nativeTransport (advanced)	Whether to use native transport instead of NIO. Native transport takes advantage of the host operating system and is only supported on some platforms. You need to add the netty JAR for the host operating system you are using. See more details at: http://netty.io/wiki/native-transports.html	false	boolean
options (advanced)	Allows to configure additional netty options using option. as prefix. For example option.child.keepAlive=false to set the netty option child.keepAlive=false. See the Netty documentation for possible options that can be used.		Map
receiveBufferSize (advanced)	The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes.	65536	int
receiveBufferSizePredictor (advanced)	Configures the buffer size predictor. See details at Jetty documentation and this mail thread.		int
sendBufferSize (advanced)	The TCP/UDP buffer sizes to be used during outbound communication. Size is bytes.	65536	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
transferExchange (advanced)	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
udpByteArrayCodec (advanced)	For UDP only. If enabled the using byte array codec instead of Java serialization protocol.	false	boolean
workerCount (advanced)	When netty works on nio mode, it uses default workerCount parameter from Netty, which is cpu_core_threads2. User can use this operation to override the default workerCount from Netty		int

Name	Description	Default	Type
workerGroup (advanced)	To use a explicit EventLoopGroup as the boss thread pool. For example to share a thread pool with multiple consumers or producers. By default each consumer or producer has their own worker pool with 2 x cpu count core threads.		EventLoopGroup
allowDefaultCodec (codec)	The netty component installs a default codec if both, encoder/deocder is null and textline is false. Setting allowDefaultCodec to false prevents the netty component from installing a default codec as the first element in the filter chain.	true	boolean
autoAppendDelimiter (codec)	Whether or not to auto append missing end delimiter when sending using the textline codec.	true	boolean
decoder (codec)	**Deprecated** A custom ChannelHandler class that can be used to perform special marshalling of inbound payloads.		ChannelHandler
decoderMaxLineLength (codec)	The max line length to use for the textline codec.	1024	int
decoders (codec)	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String
delimiter (codec)	The delimiter to use for the textline codec. Possible values are LINE and NULL.	LINE	TextLineDelimiter
encoder (codec)	**Deprecated** A custom ChannelHandler class that can be used to perform special marshalling of outbound payloads.		ChannelHandler
encoders (codec)	A list of encoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String
encoding (codec)	The encoding (a charset name) to use for the textline codec. If not provided, Camel will use the JVM default Charset.		String

Name	Description	Default	Type
textline (codec)	Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is false, then Object Serialization is assumed over TCP.	false	boolean
enabledProtocols (security)	Which protocols to enable when using SSL	TLSv1, TLSv1.1, TLSv1.2	String
keyStoreFile (security)	Client side certificate keystore to be used for encryption		File
keyStoreFormat (security)	Keystore format to be used for payload encryption. Defaults to JKS if not set		String
keyStoreResource (security)	Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String
needClientAuth (security)	Configures whether the server needs client authentication when using SSL.	false	boolean
passphrase (security)	Password setting to use in order to encrypt/decrypt payloads sent using SSH		String
securityProvider (security)	Security provider to be used for payload encryption. Defaults to SunX509 if not set.		String
ssl (security)	Setting to specify whether SSL encryption is applied to this endpoint	false	boolean
sslClientCertHeaders (security)	When enabled and in SSL mode, then the Netty consumer will enrich the Camel Message with headers having information about the client certificate such as subject name, issuer name, serial number, and the valid date range.	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
sslHandler (security)	Reference to a class that could be used to return an SSL Handler		SslHandler
trustStoreFile (security)	Server side certificate keystore to be used for encryption		File

Name	Description	Default	Type
trustStoreResource (security)	Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String

230.3. REGISTRY BASED OPTIONS

Codec Handlers and SSL Keystores can be enlisted in the Registry, such as in the Spring XML file. The values that could be passed in, are the following:

Name	Description
passphrase	password setting to use in order to encrypt/decrypt payloads sent using SSH
keyStoreFormat	keystore format to be used for payload encryption. Defaults to "JKS" if not set
securityProvider	Security provider to be used for payload encryption. Defaults to "SunX509" if not set.
keyStoreFile	**deprecated:** Client side certificate keystore to be used for encryption
trustStoreFile	**deprecated:** Server side certificate keystore to be used for encryption
keyStoreResource	**Camel 2.11.1:** Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with **"classpath:"**, **"file:"**, or **"http:"** to load the resource from different systems.
trustStoreResource	**Camel 2.11.1:** Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with **"classpath:"**, **"file:"**, or **"http:"** to load the resource from different systems.
sslHandler	Reference to a class that could be used to return an SSL Handler
encoder	A custom **ChannelHandler** class that can be used to perform special marshalling of outbound payloads. Must override io.netty.channel.ChannelInboundHandlerAdapter.

Name	Description
encoders	A list of encoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.
decoder	A custom **ChannelHandler** class that can be used to perform special marshalling of inbound payloads. Must override io.netty.channel.ChannelOutboundHandlerAdapter.
decoders	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.

NOTE

Read below about using non shareable encoders/decoders.

230.3.1. Using non shareable encoders or decoders

If your encoders or decoders is not shareable (eg they have the @Shareable class annotation), then your encoder/decoder must implement the **org.apache.camel.component.netty.ChannelHandlerFactory** interface, and return a new instance in the **newChannelHandler** method. This is to ensure the encoder/decoder can safely be used. If this is not the case, then the Netty component will log a WARN when
an endpoint is created.

The Netty component offers a **org.apache.camel.component.netty.ChannelHandlerFactories** factory class, that has a number of commonly used methods.

230.4. SENDING MESSAGES TO/FROM A NETTY ENDPOINT

230.4.1. Netty Producer

In Producer mode, the component provides the ability to send payloads to a socket endpoint using either TCP or UDP protocols (with optional SSL support).

The producer mode supports both one-way and request-response based operations.

230.4.2. Netty Consumer

In Consumer mode, the component provides the ability to:

- listen on a specified socket using either TCP or UDP protocols (with optional SSL support),
- receive requests on the socket using text/xml, binary and serialized object based payloads and
- send them along on a route as message exchanges.

The consumer mode supports both one-way and request-response based operations.

230.5. EXAMPLES

230.5.1. A UDP Netty endpoint using Request-Reply and serialized object payload

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    from("netty4:udp://localhost:5155?sync=true")
      .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
          Poetry poetry = (Poetry) exchange.getIn().getBody();
          poetry.setPoet("Dr. Sarojini Naidu");
          exchange.getOut().setBody(poetry);
        }
      }
    }
};
```

230.5.2. A TCP based Netty consumer endpoint using One-way communication

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
      from("netty4:tcp://localhost:5150")
          .to("mock:result");
  }
};
```

230.5.3. An SSL/TCP based Netty consumer endpoint using Request-Reply communication

Using the JSSE Configuration Utility

As of Camel 2.9, the Netty component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Netty component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

NettyComponent nettyComponent = getContext().getComponent("netty4", NettyComponent.class);
nettyComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
```

```
  <camel:sslContextParameters
     id="sslContextParameters">
   <camel:keyManagers
      keyPassword="keyPassword">
     <camel:keyStore
        resource="/users/home/server/keystore.jks"
        password="keystorePassword"/>
   </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="netty4:tcp://localhost:5150?
sync=true&ssl=true&sslContextParameters=#sslContextParameters"/>
...
```

[[Netty4-UsingBasicSSL/TLSconfigurationontheJettyComponent]] Using Basic SSL/TLS configuration on the Jetty Component

```
JndiRegistry registry = new JndiRegistry(createJndiContext());
registry.bind("password", "changeit");
registry.bind("ksf", new File("src/test/resources/keystore.jks"));
registry.bind("tsf", new File("src/test/resources/keystore.jks"));

context.createRegistry(registry);
context.addRoutes(new RouteBuilder() {
  public void configure() {
     String netty_ssl_endpoint =
       "netty4:tcp://localhost:5150?sync=true&ssl=true&passphrase=#password"
       + "&keyStoreFile=#ksf&trustStoreFile=#tsf";
     String return_string =
       "When You Go Home, Tell Them Of Us And Say,"
       + "For Your Tomorrow, We Gave Our Today.";

     from(netty_ssl_endpoint)
      .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
          exchange.getOut().setBody(return_string);
        }
      }
  }
});
```

Getting access to SSLSession and the client certificate

You can get access to the **javax.net.ssl.SSLSession** if you eg need to get details about the client certificate. When **ssl=true** then the Netty4 component will store the **SSLSession** as a header on the Camel Message as shown below:

```
SSLSession session = exchange.getIn().getHeader(NettyConstants.NETTY_SSL_SESSION,
SSLSession.class);
// get the first certificate which is client certificate
javax.security.cert.X509Certificate cert = session.getPeerCertificateChain()[0];
Principal principal = cert.getSubjectDN();
```

Remember to set **needClientAuth=true** to authenticate the client, otherwise **SSLSession** cannot access information about the client certificate, and you may get an exception

javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated. You may also get this exception if the client certificate is expired or not valid etc.

TIP

The option **sslClientCertHeaders** can be set to **true** which then enriches the Camel Message with headers having details about the client certificate. For example the subject name is readily available in the header **CamelNettySSLClientCertSubjectName**.

230.5.4. Using Multiple Codecs

In certain cases it may be necessary to add chains of encoders and decoders to the netty pipeline. To add multpile codecs to a camel netty endpoint the 'encoders' and 'decoders' uri parameters should be used. Like the 'encoder' and 'decoder' parameters they are used to supply references (to lists of ChannelUpstreamHandlers and ChannelDownstreamHandlers) that should be added to the pipeline. Note that if encoders is specified then the encoder param will be ignored, similarly for decoders and the decoder param.

NOTE

Read further above about using non shareable encoders/decoders.

The lists of codecs need to be added to the Camel's registry so they can be resolved when the endpoint is created.

```
ChannelHandlerFactory lengthDecoder =
ChannelHandlerFactories.newLengthFieldBasedFrameDecoder(1048576, 0, 4, 0, 4);

StringDecoder stringDecoder = new StringDecoder();
registry.bind("length-decoder", lengthDecoder);
registry.bind("string-decoder", stringDecoder);

LengthFieldPrepender lengthEncoder = new LengthFieldPrepender(4);
StringEncoder stringEncoder = new StringEncoder();
registry.bind("length-encoder", lengthEncoder);
registry.bind("string-encoder", stringEncoder);

List<ChannelHandler> decoders = new ArrayList<ChannelHandler>();
decoders.add(lengthDecoder);
decoders.add(stringDecoder);

List<ChannelHandler> encoders = new ArrayList<ChannelHandler>();
encoders.add(lengthEncoder);
encoders.add(stringEncoder);

registry.bind("encoders", encoders);
registry.bind("decoders", decoders);
```

Spring's native collections support can be used to specify the codec lists in an application context

```
<util:list id="decoders" list-class="java.util.LinkedList">
        <bean class="org.apache.camel.component.netty4.ChannelHandlerFactories" factory-
method="newLengthFieldBasedFrameDecoder">
            <constructor-arg value="1048576"/>
```

```
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
        </bean>
        <bean class="io.netty.handler.codec.string.StringDecoder"/>
    </util:list>

    <util:list id="encoders" list-class="java.util.LinkedList">
        <bean class="io.netty.handler.codec.LengthFieldPrepender">
            <constructor-arg value="4"/>
        </bean>
        <bean class="io.netty.handler.codec.string.StringEncoder"/>
    </util:list>

    <bean id="length-encoder" class="io.netty.handler.codec.LengthFieldPrepender">
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-encoder" class="io.netty.handler.codec.string.StringEncoder"/>

    <bean id="length-decoder" class="org.apache.camel.component.netty4.ChannelHandlerFactories"
factory-method="newLengthFieldBasedFrameDecoder">
        <constructor-arg value="1048576"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-decoder" class="io.netty.handler.codec.string.StringDecoder"/>
```

The bean names can then be used in netty endpoint definitions either as a comma separated list or contained in a List e.g.

```
 from("direct:multiple-codec").to("netty4:tcp://localhost:{{port}}?encoders=#encoders&sync=false");

 from("netty4:tcp://localhost:{{port}}?decoders=#length-decoder,#string-
decoder&sync=false").to("mock:multiple-codec");
```

or via XML.

```
<camelContext id="multiple-netty-codecs-context" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:multiple-codec"/>
        <to uri="netty4:tcp://localhost:5150?encoders=#encoders&amp;sync=false"/>
    </route>
    <route>
        <from uri="netty4:tcp://localhost:5150?decoders=#length-decoder,#string-
decoder&amp;sync=false"/>
        <to uri="mock:multiple-codec"/>
    </route>
</camelContext>
```

230.6. CLOSING CHANNEL WHEN COMPLETE

When acting as a server you sometimes want to close the channel when, for example, a client conversion is finished.
You can do this by simply setting the endpoint option **disconnect=true**.

However you can also instruct Camel on a per message basis as follows.
To instruct Camel to close the channel, you should add a header with the key **CamelNettyCloseChannelWhenComplete** set to a boolean **true** value.
For instance, the example below will close the channel after it has written the bye message back to the client:

```
from("netty4:tcp://localhost:8080").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody("Bye " + body);
        // some condition which determines if we should close
        if (close) {

exchange.getOut().setHeader(NettyConstants.NETTY_CLOSE_CHANNEL_WHEN_COMPLETE,
true);
        }
    }
});
```

Adding custom channel pipeline factories to gain complete control over a

230.7. CUSTOM PIPELINE

Custom channel pipelines provide complete control to the user over the handler/interceptor chain by inserting custom handler(s), encoder(s) & decoders without having to specify them in the Netty Endpoint URL in a very simple way.

In order to add a custom pipeline, a custom channel pipeline factory must be created and registered with the context via the context registry (JNDIRegistry, or the camel-spring ApplicationContextRegistry etc).

A custom pipeline factory must be constructed as follows

- A Producer linked channel pipeline factory must extend the abstract class **ClientPipelineFactory**.
- A Consumer linked channel pipeline factory must extend the abstract class **ServerInitializerFactory**.
- The classes should override the initChannel() method in order to insert custom handler(s), encoder(s) and decoder(s). Not overriding the **initChannel()** method creates a pipeline with no handlers, encoders or decoders wired to the pipeline.

The example below shows how ServerInitializerFactory factory may be created

230.7.1. Using custom pipeline factory

```
public class SampleServerInitializerFactory extends ServerInitializerFactory {
    private int maxLineSize = 1024;

    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline channelPipeline = ch.pipeline();
```

```
        channelPipeline.addLast("encoder-SD", new StringEncoder(CharsetUtil.UTF_8));
        channelPipeline.addLast("decoder-DELIM", new DelimiterBasedFrameDecoder(maxLineSize,
true, Delimiters.lineDelimiter()));
        channelPipeline.addLast("decoder-SD", new StringDecoder(CharsetUtil.UTF_8));
        // here we add the default Camel ServerChannelHandler for the consumer, to allow Camel to
route the message etc.
        channelPipeline.addLast("handler", new ServerChannelHandler(consumer));
    }
}
```

The custom channel pipeline factory can then be added to the registry and instantiated/utilized on a camel route in the following way

```
Registry registry = camelContext.getRegistry();
ServerInitializerFactory factory = new TestServerInitializerFactory();
registry.bind("spf", factory);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
        "netty4:tcp://localhost:5150?serverInitializerFactory=#spf"
      String return_string =
        "When You Go Home, Tell Them Of Us And Say,"
        + "For Your Tomorrow, We Gave Our Today.";

      from(netty_ssl_endpoint)
       .process(new Processor() {
         public void process(Exchange exchange) throws Exception {
           exchange.getOut().setBody(return_string);
         }
       }
  }
});
```

230.8. REUSING NETTY BOSS AND WORKER THREAD POOLS

Netty has two kind of thread pools: boss and worker. By default each Netty consumer and producer has their private thread pools. If you want to reuse these thread pools among multiple consumers or producers then the thread pools must be created and enlisted in the Registry.

For example using Spring XML we can create a shared worker thread pool using the **NettyWorkerPoolBuilder** with 2 worker threads as shown below:

```
<!-- use the worker pool builder to help create the shared thread pool -->
<bean id="poolBuilder" class="org.apache.camel.component.netty.NettyWorkerPoolBuilder">
  <property name="workerCount" value="2"/>
</bean>

<!-- the shared worker thread pool -->
<bean id="sharedPool" class="org.jboss.netty.channel.socket.nio.WorkerPool"
      factory-bean="poolBuilder" factory-method="build" destroy-method="shutdown">
</bean>
```

TIP

For boss thread pool there is a **org.apache.camel.component.netty4.NettyServerBossPoolBuilder** builder for Netty consumers, and a **org.apache.camel.component.netty4.NettyClientBossPoolBuilder** for the Netty producers.

Then in the Camel routes we can refer to this worker pools by configuring the **workerPool** option in the URI as shown below:

```
<route>
  <from uri="netty4:tcp://localhost:5021?
textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;usingExecutorService=false"/>
  <to uri="log:result"/>
  ...
</route>
```

And if we have another route we can refer to the shared worker pool:

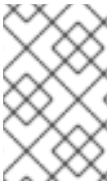
```
<route>
  <from uri="netty4:tcp://localhost:5022?
textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;usingExecutorService=false"/>
  <to uri="log:result"/>
  ...
</route>
```

and so forth.

230.9. MULTIPLEXING CONCURRENT MESSAGES OVER A SINGLE CONNECTION WITH REQUEST/REPLY

When using Netty for request/reply messaging via the netty producer then by default each message is sent via a non-shared connection (pooled). This ensures that replies are automatic being able to map to the correct request thread for further routing in Camel. In other words correlation between request/reply messages happens out-of-the-box because the replies comes back on the same connection that was used for sending the request; and this connection is not shared with others. When the response comes back, the connection is returned back to the connection pool, where it can be reused by others.

However if you want to multiplex concurrent request/responses on a single shared connection, then you need to turn off the connection pooling by setting **producerPoolEnabled=false**. Now this means there is a potential issue with interleaved responses if replies comes back out-of-order. Therefore you need to have a correlation id in both the request and reply messages so you can properly correlate the replies to the Camel callback that is responsible for continue processing the message in Camel. To do this you need to implement **NettyCamelStateCorrelationManager** as correlation manager and configure it via the **correlationManager=#myManager** option.



NOTE

We recommend extending the **TimeoutCorrelationManagerSupport** when you build custom correlation managers. This provides support for timeout and other complexities you otherwise would need to implement as well.

230.10. SEE ALSO

- Netty HTTP
- MINA

CHAPTER 231. NETTY4 HTTP COMPONENT

Available as of Camel version 2.14

The **netty4-http** component is an extension to Netty4 component to facilitiate HTTP transport with Netty4.

This camel component supports both producer and consumer endpoints.

INFO: **Stream**. Netty is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**. If you find a situation where the message body appears to be empty or you need to access the data multiple times (eg: doing multicasting, or redelivery error handling) you should use Stream caching or convert the message body to a **String** which is safe to be re-read multiple times. Notice Netty4 HTTP reads the entire stream into memory using **io.netty.handler.codec.http.HttpObjectAggregator** to build the entire full http message. But the resulting message is still a stream based message which is readable once.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4-http</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

231.1. URI FORMAT

The URI scheme for a netty component is as follows

```
netty4-http:http://localhost:8080[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

INFO: **Query parameters vs endpoint options** You may be wondering how Camel recognizes URI query parameters and endpoint options. For example you might create endpoint URI as follows - **netty4-http:http//example.com?myParam=myValue&compression=true** . In this example **myParam** is the HTTP parameter, while **compression** is the Camel endpoint option. The strategy used by Camel in such situations is to resolve available endpoint options and remove them from the URI. It means that for the discussed example, the HTTP request sent by Netty HTTP producer to the endpoint will look as follows - **http//example.com?myParam=myValue** , because **compression** endpoint option will be resolved and removed from the target URL. Keep also in mind that you cannot specify endpoint options using dynamic headers (like **CamelHttpQuery**). Endpoint options can be specified only at the endpoint URI definition level (like **to** or **from** DSL elements).

231.2. HTTP OPTIONS

INFO: **A lot more options**. **Important:** This component inherits all the options from Netty4. So make sure to look at the Netty4 documentation as well.
Notice that some options from Netty4 is not applicable when using this Netty4 HTTP component, such as options related to UDP transport.

The Netty4 HTTP component supports 8 options which are listed below.

Name	Description	Default	Type
nettyHttpBinding (advanced)	To use a custom org.apache.camel.component.netty4.http.NettyHttpBinding for binding to/from Netty and Camel Message API.		NettyHttpBinding
configuration (common)	To use the NettyConfiguration as configuration when creating endpoints.		NettyHttpConfiguration
headerFilterStrategy (advanced)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter headers.		HeaderFilterStrategy
securityConfiguration (security)	Refers to a org.apache.camel.component.netty4.http.NettyHttpSecurityConfiguration for configuring secure web resources.		NettyHttpSecurityConfiguration
useGlobalSslContextParameters (security)	Enable usage of global SSL context parameters.	false	boolean
maximumPoolSize (advanced)	The thread pool size for the EventExecutorGroup if its in use. The default value is 16.	16	int
executorService (advanced)	To use the given EventExecutorGroup		EventExecutorGroup
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Netty4 HTTP endpoint is configured using URI syntax:

```
netty4-http:protocol:host:port/path
```

with the following path and query parameters:

231.2.1. Path Parameters (4 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol to use which is either http or https		String

Name	Description	Default	Type
host	**Required** The local hostname such as localhost, or 0.0.0.0 when being a consumer. The remote HTTP server hostname when using producer.		String
port	The host port number		int
path	Resource path		String

231.2.2. Query Parameters (79 parameters):

Name	Description	Default	Type
bridgeEndpoint (common)	If the option is true, the producer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the producer send all the fault response back. The consumer working in the bridge mode will skip the gzip compression and WWW URL form encoding (by adding the Exchange.SKIP_GZIP_ENCODING and Exchange.SKIP_WWW_FORM_URLENCODED headers to the consumed exchange).	false	boolean
disconnect (common)	Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer.	false	boolean
keepAlive (common)	Setting to ensure socket is not closed due to inactivity	true	boolean
reuseAddress (common)	Setting to facilitate socket multiplexing	true	boolean
reuseChannel (common)	This option allows producers and consumers (in client mode) to reuse the same Netty Channel for the lifecycle of processing the Exchange. This is useful if you need to call a server multiple times in a Camel route and want to use the same network connection. When using this the channel is not returned to the connection pool until the Exchange is done; or disconnected if the disconnect option is set to true. The reused Channel is stored on the Exchange as an exchange property with the key link NettyConstantsNETTY_CHANNEL which allows you to obtain the channel during routing and use it as well.	false	boolean

Name	Description	Default	Type
sync (common)	Setting to set endpoint as one-way or request-response	true	boolean
tcpNoDelay (common)	Setting to improve TCP protocol performance	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
matchOnUriPrefix (consumer)	Whether or not Camel should try to find a target consumer by matching the URI prefix if no exact match is found.	false	boolean
send503whenSuspended (consumer)	Whether to send back HTTP status code 503 when the consumer has been suspended. If the option is false then the Netty Acceptor is unbound when the consumer is suspended, so clients cannot connect anymore.	true	boolean
backlog (consumer)	Allows to configure a backlog for netty consumer (server). Note the backlog is just a best effort depending on the OS. Setting this option to a value such as 200, 500 or 1000, tells the TCP stack how long the accept queue can be If this option is not configured, then the backlog depends on OS setting.		int
bossCount (consumer)	When netty works on nio mode, it uses default bossCount parameter from Netty, which is 1. User can use this operation to override the default bossCount from Netty	1	int
bossGroup (consumer)	Set the BossGroup which could be used for handling the new connection of the server side across the NettyEndpoint		EventLoopGroup
chunkedMaxContentLength (consumer)	Value in bytes the max content length per chunked frame received on the Netty HTTP server.	1048576	int

Name	Description	Default	Type
compression (consumer)	Allow using gzip/deflate for compression on the Netty HTTP server if the client supports it from the HTTP headers.	false	boolean
disconnectOnNoReply (consumer)	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
httpMethodRestrict (consumer)	To disable HTTP methods on the Netty HTTP consumer. You can specify multiple separated by comma.		String
mapHeaders (consumer)	If this option is enabled, then during binding from Netty to Camel Message then the headers will be mapped as well (eg added as header to the Camel Message as well). You can turn off this option to disable this. The headers can still be accessed from the org.apache.camel.component.netty.http.NettyHttpMessage message with the method getHttpRequest() that returns the Netty HTTP request io.netty.handler.codec.http.HttpRequest instance.	true	boolean
maxHeaderSize (consumer)	The maximum length of all headers. If the sum of the length of each header exceeds this value, a io.netty.handler.codec.TooLongFrameException will be raised.	8192	int
nettyServerBootstrapFactory (consumer)	To use a custom NettyServerBootstrapFactory		NettyServerBootstrap Factory
nettySharedHttpServer (consumer)	To use a shared Netty HTTP server. See Netty HTTP Server Example for more details.		NettySharedHttpServer

Name	Description	Default	Type
noReplyLogLevel (consumer)	If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back.	WARN	LoggingLevel
serverClosedChannelExceptionCaughtLogLevel (consumer)	If the server (NettyConsumer) catches an java.nio.channels.ClosedChannelException then its logged using this logging level. This is used to avoid logging the closed channel exceptions, as clients can disconnect abruptly and then cause a flood of closed exceptions in the Netty server.	DEBUG	LoggingLevel
serverExceptionCaughtLog Level (consumer)	If the server (NettyConsumer) catches an exception then its logged using this logging level.	WARN	LoggingLevel
serverInitializerFactory (consumer)	To use a custom ServerInitializerFactory		ServerInitializerFactory
traceEnabled (consumer)	Specifies whether to enable HTTP TRACE for this Netty HTTP consumer. By default TRACE is turned off.	false	boolean
urlDecodeHeaders (consumer)	If this option is enabled, then during binding from Netty to Camel Message then the header values will be URL decoded (eg %20 will be a space character. Notice this option is used by the default org.apache.camel.component.netty.http.NettyHttpBinding and therefore if you implement a custom org.apache.camel.component.netty4.http.NettyHttpBinding then you would need to decode the headers accordingly to this option.	false	boolean
usingExecutorService (consumer)	Whether to use ordered thread pool, to ensure events are processed orderly on the same channel.	true	boolean
connectTimeout (producer)	Time to wait for a socket connection to be available. Value is in millis.	10000	int
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
requestTimeout (producer)	Allows to use a timeout for the Netty producer when calling a remote server. By default no timeout is in use. The value is in milli seconds, so eg 30000 is 30 seconds. The requestTimeout is using Netty's ReadTimeoutHandler to trigger the timeout.		long

Name	Description	Default	Type
throwExceptionOnFailure (producer)	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	boolean
clientInitializerFactory (producer)	To use a custom ClientInitializerFactory		ClientInitializerFactory
lazyChannelCreation (producer)	Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.	true	boolean
okStatusCodeRange (producer)	The status codes which are considered a success response. The values are inclusive. Multiple ranges can be defined, separated by comma, e.g. 200-204,209,301-304. Each range must be a single number or from-to with the dash included. The default range is 200-299	200-299	String
producerPoolEnabled (producer)	Whether producer pool is enabled or not. Important: If you turn this off then a single shared connection is used for the producer, also if you are doing request/reply. That means there is a potential issue with interleaved responses if replies comes back out-of-order. Therefore you need to have a correlation id in both the request and reply messages so you can properly correlate the replies to the Camel callback that is responsible for continue processing the message in Camel. To do this you need to implement NettyCamelStateCorrelationManager as correlation manager and configure it via the correlationManager option. See also the correlationManager option for more details.	true	boolean
producerPoolMaxActive (producer)	Sets the cap on the number of objects that can be allocated by the pool (checked out to clients, or idle awaiting checkout) at a given time. Use a negative value for no limit.	-1	int
producerPoolMaxIdle (producer)	Sets the cap on the number of idle instances in the pool.	100	int
producerPoolMinEvictable Idle (producer)	Sets the minimum amount of time (value in millis) an object may sit idle in the pool before it is eligible for eviction by the idle object evictor.	300000	long

Name	Description	Default	Type
producerPoolMinIdle (producer)	Sets the minimum number of instances allowed in the producer pool before the evictor thread (if active) spawns new objects.		int
useRelativePath (producer)	Sets whether to use a relative path in HTTP requests.	false	boolean
allowSerializedHeaders (advanced)	Only used for TCP when transferExchange is true. When set to true, serializable objects in headers and properties will be added to the exchange. Otherwise Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
bootstrapConfiguration (advanced)	To use a custom configured NettyServerBootstrapConfiguration for configuring this endpoint.		NettyServerBootstrap Configuration
channelGroup (advanced)	To use a explicit ChannelGroup.		ChannelGroup
configuration (advanced)	To use a custom configured NettyHttpConfiguration for configuring this endpoint.		NettyHttpConfiguration
disableStreamCache (advanced)	Determines whether or not the raw input stream from Netty HttpRequestgetContent() or HttpResponsetgetContent() is cached or not (Camel will read the stream into a in light-weight memory based Stream caching) cache. By default Camel will cache the Netty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. Mind that if you enable this option, then you cannot read the Netty stream multiple times out of the box, and you would need manually to reset the reader index on the Netty raw stream. Also Netty will auto-close the Netty stream when the Netty HTTP server/HTTP client is done processing, which means that if the asynchronous routing engine is in use then any asynchronous thread that may continue routing the org.apache.camel.Exchange may not be able to read the Netty stream, because Netty has closed it.	false	boolean
headerFilterStrategy (advanced)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter headers.		HeaderFilterStrategy

Name	Description	Default	Type
nativeTransport (advanced)	Whether to use native transport instead of NIO. Native transport takes advantage of the host operating system and is only supported on some platforms. You need to add the netty JAR for the host operating system you are using. See more details at: http://netty.io/wiki/native-transports.html	false	boolean
nettyHttpBinding (advanced)	To use a custom org.apache.camel.component.netty4.http.NettyHttpBinding for binding to/from Netty and Camel Message API.		NettyHttpBinding
options (advanced)	Allows to configure additional netty options using option. as prefix. For example option.child.keepAlive=false to set the netty option child.keepAlive=false. See the Netty documentation for possible options that can be used.		Map
receiveBufferSize (advanced)	The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes.	65536	int
receiveBufferSizePredictor (advanced)	Configures the buffer size predictor. See details at Jetty documentation and this mail thread.		int
sendBufferSize (advanced)	The TCP/UDP buffer sizes to be used during outbound communication. Size is bytes.	65536	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
transferException (advanced)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean

Name	Description	Default	Type
transferExchange (advanced)	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.	false	boolean
workerCount (advanced)	When netty works on nio mode, it uses default workerCount parameter from Netty, which is cpu_core_threads2. User can use this operation to override the default workerCount from Netty		int
workerGroup (advanced)	To use a explicit EventLoopGroup as the boss thread pool. For example to share a thread pool with multiple consumers or producers. By default each consumer or producer has their own worker pool with 2 x cpu count core threads.		EventLoopGroup
decoder (codec)	**Deprecated** To use a single decoder. This options is deprecated use encoders instead.		ChannelHandler
decoders (codec)	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String
encoder (codec)	**Deprecated** To use a single encoder. This options is deprecated use encoders instead.		ChannelHandler
encoders (codec)	A list of encoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with so Camel knows it should lookup.		String
enabledProtocols (security)	Which protocols to enable when using SSL	TLSv1, TLSv1.1, TLSv1.2	String
keyStoreFile (security)	Client side certificate keystore to be used for encryption		File
keyStoreFormat (security)	Keystore format to be used for payload encryption. Defaults to JKS if not set		String

Name	Description	Default	Type
keyStoreResource (security)	Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String
needClientAuth (security)	Configures whether the server needs client authentication when using SSL.	false	boolean
passphrase (security)	Password setting to use in order to encrypt/decrypt payloads sent using SSH		String
securityConfiguration (security)	Refers to a org.apache.camel.component.netty4.http.NettyHttpSecurityConfiguration for configuring secure web resources.		NettyHttpSecurityConfiguration
securityOptions (security)	To configure NettyHttpSecurityConfiguration using key/value pairs from the map		Map
securityProvider (security)	Security provider to be used for payload encryption. Defaults to SunX509 if not set.		String
ssl (security)	Setting to specify whether SSL encryption is applied to this endpoint	false	boolean
sslClientCertHeaders (security)	When enabled and in SSL mode, then the Netty consumer will enrich the Camel Message with headers having information about the client certificate such as subject name, issuer name, serial number, and the valid date range.	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
sslHandler (security)	Reference to a class that could be used to return an SSL Handler		SslHandler
trustStoreFile (security)	Server side certificate keystore to be used for encryption		File
trustStoreResource (security)	Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with classpath:, file:, or http: to load the resource from different systems.		String

231.3. MESSAGE HEADERS

The following headers can be used on the producer to control the HTTP request.

Name	Type	Description
CamelHttpMethod	**String**	Allow to control what HTTP method to use such as GET, POST, TRACE etc. The type can also be a **io.netty.handler.codec.http.HttpMethod** instance.
CamelHttpQuery	**String**	Allows to provide URI query parameters as a **String** value that overrides the endpoint configuration. Separate multiple parameters using the & sign. For example: **foo=bar&beer=yes**.
CamelHttpPath	**String**	Allows to provide URI context-path and query parameters as a **String** value that overrides the endpoint configuration. This allows to reuse the same producer for calling same remote http server, but using a dynamic context-path and query parameters.
Content-Type	**String**	To set the content-type of the HTTP body. For example: **text/plain; charset="UTF-8"**.
CamelHttpResponseCode	**int**	Allows to set the HTTP Status code to use. By default 200 is used for success, and 500 for failure.

The following headers is provided as meta-data when a route starts from an Netty4 HTTP endpoint:

The description in the table takes offset in a route having: **from("netty4-http:http:0.0.0.0:8080/myapp") ...**

Name	Type	Description
CamelHttpMethod	**String**	The HTTP method used, such as GET, POST, TRACE etc.
CamelHttpUrl	**String**	The URL including protocol, host and port, etc: **http://0.0.0.0:8080/myapp**
CamelHttpUri	**String**	The URI without protocol, host and port, etc: **/myapp**
CamelHttpQuery	**String**	Any query parameters, such as **foo=bar&beer=yes**

Name	Type	Description
CamelHttpRawQuery	**String**	Any query parameters, such as **foo=bar&beer=yes**. Stored in the raw form, as they arrived to the consumer (i.e. before URL decoding).
CamelHttpPath	**String**	Additional context-path. This value is empty if the client called the context-path **/myapp**. If the client calls**/myapp/mystuff**, then this header value is**/mystuff**. In other words its the value after the context-path configured on the route endpoint.
CamelHttpCharacterEncoding	**String**	The charset from the content-type header.
CamelHttpAuthentication	**String**	If the user was authenticated using HTTP Basic then this header is added with the value **Basic**.
Content-Type	**String**	The content type if provided. For example: **text/plain; charset="UTF-8"**.

231.4. ACCESS TO NETTY TYPES

This component uses the **org.apache.camel.component.netty4.http.NettyHttpMessage** as the message implementation on the Exchange. This allows end users to get access to the original Netty request/response instances if needed, as shown below. Mind that the original response may not be accessible at all times.

```
io.netty.handler.codec.http.HttpRequest request =
exchange.getIn(NettyHttpMessage.class).getHttpRequest();
```

231.5. EXAMPLES

In the route below we use Netty4 HTTP as a HTTP server, which returns back a hardcoded "Bye World" message.

```
from("netty4-http:http://0.0.0.0:8080/foo")
  .transform().constant("Bye World");
```

And we can call this HTTP server using Camel also, with the ProducerTemplate as shown below:

```
    String out = template.requestBody("netty4-http:http://localhost:8080/foo", "Hello World",
String.class);
    System.out.println(out);
```

And we get back "Bye World" as the output.

231.6. HOW DO I LET NETTY MATCH WILDCARDS

By default Netty4 HTTP will only match on exact uri's. But you can instruct Netty to match prefixes. For example

```
from("netty4-http:http://0.0.0.0:8123/foo").to("mock:foo");
```

In the route above Netty4 HTTP will only match if the uri is an exact match, so it will match if you enter **http://0.0.0.0:8123/foo** but not match if you do **http://0.0.0.0:8123/foo/bar**.

So if you want to enable wildcard matching you do as follows:

```
from("netty4-http:http://0.0.0.0:8123/foo?matchOnUriPrefix=true").to("mock:foo");
```

So now Netty matches any endpoints with starts with **foo**.

To match **any** endpoint you can do:

```
from("netty4-http:http://0.0.0.0:8123?matchOnUriPrefix=true").to("mock:foo");
```

231.7. USING MULTIPLE ROUTES WITH SAME PORT

In the same CamelContext you can have multiple routes from Netty4 HTTP that shares the same port (eg a **io.netty.bootstrap.ServerBootstrap** instance). Doing this requires a number of bootstrap options to be identical in the routes, as the routes will share the same **io.netty.bootstrap.ServerBootstrap** instance. The instance will be configured with the options from the first route created.

The options the routes must be identical configured is all the options defined in the **org.apache.camel.component.netty4.NettyServerBootstrapConfiguration** configuration class. If you have configured another route with different options, Camel will throw an exception on startup, indicating the options is not identical. To mitigate this ensure all options is identical.

Here is an example with two routes that share the same port.

Two routes sharing the same port

```
from("netty4-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

from("netty4-http:http://0.0.0.0:{{port}}/bar")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

And here is an example of a mis configured 2nd route that do not have identical **org.apache.camel.component.netty4.NettyServerBootstrapConfiguration** option as the 1st route. This will cause Camel to fail on startup.

Two routes sharing the same port, but the 2nd route is misconfigured and will fail on starting

```
from("netty4-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

// we cannot have a 2nd route on same port with SSL enabled, when the 1st route is NOT
from("netty4-http:http://0.0.0.0:{{port}}/bar?ssl=true")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

231.7.1. Reusing same server bootstrap configuration with multiple routes

By configuring the common server bootstrap option in an single instance of a **org.apache.camel.component.netty4.NettyServerBootstrapConfiguration** type, we can use the **bootstrapConfiguration** option on the Netty4 HTTP consumers to refer and reuse the same options across all consumers.

```
<bean id="nettyHttpBootstrapOptions"
class="org.apache.camel.component.netty4.NettyServerBootstrapConfiguration">
  <property name="backlog" value="200"/>
  <property name="connectionTimeout" value="20000"/>
  <property name="workerCount" value="16"/>
</bean>
```

And in the routes you refer to this option as shown below

```
<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/foo?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/bar?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/beer?
bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>
```

231.7.2. Reusing same server bootstrap configuration with multiple routes across multiple bundles in OSGi container

See the Netty HTTP Server Example for more details and example how to do that.

231.8. USING HTTP BASIC AUTHENTICATION

The Netty HTTP consumer supports HTTP basic authentication by specifying the security realm name to use, as shown below

```
<route>
  <from uri="netty4-http:http://0.0.0.0:{{port}}/foo?securityConfiguration.realm=karaf"/>
  ...
</route>
```

The realm name is mandatory to enable basic authentication. By default the JAAS based authenticator is used, which will use the realm name specified (karaf in the example above) and use the JAAS realm and the JAAS \{{LoginModule}}s of this realm for authentication.

End user of Apache Karaf / ServiceMix has a karaf realm out of the box, and hence why the example above would work out of the box in these containers.

231.8.1. Specifying ACL on web resources

The **org.apache.camel.component.netty4.http.SecurityConstraint** allows to define constrains on web resources. And the **org.apache.camel.component.netty.http.SecurityConstraintMapping** is provided out of the box, allowing to easily define inclusions and exclusions with roles.

For example as shown below in the XML DSL, we define the constraint bean:

```
<bean id="constraint" class="org.apache.camel.component.netty4.http.SecurityConstraintMapping">
  <!-- inclusions defines url -> roles restrictions -->
  <!-- a * should be used for any role accepted (or even no roles) -->
  <property name="inclusions">
    <map>
      <entry key="/*" value="*"/>
      <entry key="/admin/*" value="admin"/>
      <entry key="/guest/*" value="admin,guest"/>
    </map>
  </property>
  <!-- exclusions is used to define public urls, which requires no authentication -->
  <property name="exclusions">
    <set>
      <value>/public/*</value>
    </set>
  </property>
</bean>
```

The constraint above is define so that

- access to /* is restricted and any roles is accepted (also if user has no roles)
- access to /admin/* requires the admin role
- access to /guest/* requires the admin or guest role
- access to /public/* is an exclusion which means no authentication is needed, and is therefore public for everyone without logging in

To use this constraint we just need to refer to the bean id as shown below:

```
<route>
```

```
  <from uri="netty4-http:http://0.0.0.0:{{port}}/foo?
matchOnUriPrefix=true&amp;securityConfiguration.realm=karaf&amp;securityConfiguration.securityCon
straint=#constraint"/>
  ...
</route>
```

231.9. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Netty
- Netty HTTP Server Example
- Jetty

CHAPTER 232. OGNL LANGUAGE

Available as of Camel version 1.1

Camel allows OGNL to be used as an Expression or Predicate the DSL or Xml Configuration.

You could use OGNL to create an Predicate in a Message Filter or as an Expression for a Recipient List

You can use OGNL dot notation to invoke operations. If you for instance have a body that contains a POJO that has a **getFamilyName** method then you can construct the syntax as follows:

```
"request.body.familyName"
  // or
"getRequest().getBody().getFamilyName()"
```

232.1. OGNL OPTIONS

The OGNL language supports 1 options which are listed below.

Name	Default	Java Type	Description
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

232.2. VARIABLES

Variable	Type	Description
this	Exchange	the Exchange is the root object
exchange	Exchange	the Exchange object
exception	Throwable	the Exchange exception (if any)
exchangeId	String	the exchange id
fault	Message	the Fault message (if any)
request	Message	the exchange.in message

Variable	Type	Description
response	Message	the exchange.out message (if any)
properties	Map	the exchange properties
property(name)	Object	the property by the given name
property(name, type)	Type	the property by the given name as the given type

232.3. SAMPLES

For example you could use OGNL inside a Message Filter in XML

```
<route>
  <from uri="seda:foo"/>
  <filter>
    <ognl>request.headers.foo == 'bar'</ognl>
    <to uri="seda:bar"/>
  </filter>
</route>
```

And the sample using Java DSL:

```
from("seda:foo").filter().ognl("request.headers.foo == 'bar'").to("seda:bar");
```

232.4. LOADING SCRIPT FROM EXTERNAL RESOURCE

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as **"classpath:"**, **"file:"**, or **"http:"**.
This is done using the following syntax: **"resource:scheme:location"**, eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").ognl("resource:classpath:myognl.txt")
```

232.5. DEPENDENCIES

To use OGNL in your camel routes you need to add the a dependency on **camel-ognl** which implements the OGNL language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ognl</artifactId>
  <version>x.x.x</version>
</dependency>
```

Otherwise, you'll also need OGNL

CHAPTER 233. OLINGO2 COMPONENT

Available as of Camel version 2.14

The Olingo2 component utilizes Apache Olingo version 2.0 APIs to interact with OData 2.0 compliant services. A number of popular commercial and enterprise vendors and products support the OData protocol. A sample list of supporting products can be found on the OData website.

The Olingo2 component supports reading feeds, delta feeds, entities, simple and complex properties, links, counts, using custom and OData system query parameters. It supports updating entities, properties, and association links. It also supports submitting queries and change requests as a single OData batch operation.

The component supports configuring HTTP connection parameters and headers for OData service connection. This allows configuring use of SSL, OAuth2.0, etc. as required by the target OData service.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-olingo2</artifactId>
    <version>${camel-version}</version>
</dependency>
```

233.1. URI FORMAT

```
olingo2://endpoint/<resource-path>?[options]
```

233.2. OLINGO2 OPTIONS

The Olingo2 component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		Olingo2Configuration
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Olingo2 endpoint is configured using URI syntax:

```
olingo2:apiName/methodName
```

with the following path and query parameters:

233.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		Olingo2ApiName
methodName	**Required** What sub operation to use for the selected operation		String

233.2.2. Query Parameters (14 parameters):

Name	Description	Default	Type
connectTimeout (common)	HTTP connection creation timeout in milliseconds, defaults to 30,000 (30 seconds)	30000	int
contentType (common)	Content-Type header value can be used to specify JSON or XML message format, defaults to application/json;charset=utf-8	application/json;charset=utf-8	String
httpAsyncClientBuilder (common)	Custom HTTP async client builder for more complex HTTP client configuration, overrides connectionTimeout, socketTimeout, proxy and sslContext. Note that a socketTimeout MUST be specified in the builder, otherwise OData requests could block indefinitely		HttpAsyncClientBuilder
httpClientBuilder (common)	Custom HTTP client builder for more complex HTTP client configuration, overrides connectionTimeout, socketTimeout, proxy and sslContext. Note that a socketTimeout MUST be specified in the builder, otherwise OData requests could block indefinitely		HttpClientBuilder
httpHeaders (common)	Custom HTTP headers to inject into every request, this could include OAuth tokens, etc.		Map
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
proxy (common)	HTTP proxy server configuration		HttpHost
serviceUri (common)	Target OData service base URI, e.g. http://services.odata.org/OData/OData.svc		String

Name	Description	Default	Type
socketTimeout (common)	HTTP request timeout in milliseconds, defaults to 30,000 (30 seconds)	30000	int
sslContextParameters (common)	To configure security using SSLContextParameters		SSLContextParameters
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

233.3. PRODUCER ENDPOINTS

Producer endpoints can use endpoint names and options listed next. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message. The **inBody** option defaults to **data** for endpoints that take that option.

233.4. ENDPOINT OPTIONS

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelOlingo2.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelOlingo2.option** header. In addition, query parameters can be specified.

Name	Type	Description
data	Object	Data with appropriate type used to create or modify the OData resource

Name	Type	Description
keyPredicate	String	Key predicate to create a parameterized OData resource endpoint. Useful for create/update operations where the key predicate value is dynamically provided in a header
queryParams	java.util.Map<String,String>	OData system options and custom query options. For more information see OData 2.0 URI Conventions
resourcePath	String	OData resource path, may or may not contain key predicate
endpointHttpHeaders	java.util.Map<String, String>	Dynamic HTTP Headers to be sent to the endpoint
responseHttpHeaders	java.util.Map<String, String>	Dynamic HTTP Response Headers from the endpoint

Note that the resourcePath option can either in specified in the URI as a part of the URI path, as an endpoint option ?resourcePath=<resource-path> or as a header value CamelOlingo2.resourcePath. The OData entity key predicate can either be a part of the resource path, e.g. *Manufacturers('1')*, where *'__1'* is the key predicate, or be specified separately with resource path *Manufacturers* and keyPredicate option *'1'*.

Endpoint	Options	HTTP Method	Result Body Type
batch	data, endpointHttpHeaders	POST with multipart/mixed batch request	java.util.List<org.apache.camel.component.olingo2.api.batch.Olingo2BatchResponse>
create	data, resourcePath, endpointHttpHeaders	POST	org.apache.olingo.odata2.api.ep.entry.ODataEntry for new entries org.apache.olingo.odata2.api.commons.HttpStatusCodes for other OData resources

Endpoint	Options	HTTP Method	Result Body Type
delete	resourcePath, endpointHttpHeaders	DELETE	org.apache.olingo.odata2.api.commons.HttpStatusCodes
merge	data, resourcePath, endpointHttpHeaders	MERGE	org.apache.olingo.odata2.api.commons.HttpStatusCodes
patch	data, resourcePath, endpointHttpHeaders	PATCH	org.apache.olingo.odata2.api.commons.HttpStatusCodes
read	queryParams, resourcePath, endpointHttpHeaders	GET	Depends on OData resource being queried as described next
update	data, resourcePath, endpointHttpHeaders	PUT	org.apache.olingo.odata2.api.commons.HttpStatusCodes

233.5. ENDPOINT HTTP HEADERS (SINCE 2.20)

The component level configuration property **httpHeaders** supplies static HTTP header information. However, some systems requires dynamic header information to be passed to and received from the endpoint. A sample use case would be systems that require dynamic security tokens. The **endpointHttpHeaders** and **responseHttpHeaders** endpoint properties provides this capability. Set

headers that need to be passed to the endpoint in the **CamelOlingo2.endpointHttpHeaders** property and the response headers will be returned in a **CamelOlingo2.responseHttpHeaders** property. Both properties are of the type **java.util.Map<String, String>**.

233.6. ODATA RESOURCE TYPE MAPPING

The result of **read** endpoint and data type of **data** option depends on the OData resource being queried, created or modified.

OData Resource Type	Resource URI from resourcePath and keyPredicate	In or Out Body Type
Entity data model	$metadata	org.apache.olingo.odata2.api.edm.Edm
Service document	/	org.apache.olingo.odata2.api.servicedocument.ServiceDocument
OData feed	<entity-set>	org.apache.olingo.odata2.api.ep.feed.ODataFeed
OData entry	<entity-set>(<key-predicate>)	org.apache.olingo.odata2.api.ep.entry.ODataEntry for Out body (response) java.util.Map<String, Object> for In body (request)
Simple property	<entity-set>(<key-predicate>)/<simple-property>	Appropriate Java data type as described by Olingo EdmProperty

OData Resource Type	Resource URI from resourcePath and keyPredicate	In or Out Body Type
Simple property value	<entity-set>(<key-predicate>)/<simple-property>/$value	Appropriate Java data type as described by Olingo EdmProperty
Complex property	<entity-set>(<key-predicate>)/<complex-property>	java.util.Map<String, Object>
Zero or one association link	<entity-set>(<key-predicate>/$link/<one-to-one-entity-set-property>	String for response java.util.Map<String, Object> with key property names and values for request

OData Resource Type	Resource URI from resourcePath and keyPredicate	In or Out Body Type
Zero or many association links	<entity-set>(<key-predicate>/$link/<one-to-many-entity-set-property>	java.util.List<String> for response java.util.List<java.util.Map<String, Object>> containing list of key property names and values for request
Count	<resource-uri>/$count	java.lang.Long

233.7. CONSUMER ENDPOINTS

Only the **read** endpoint can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. By default consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange. This behavior can be disabled by setting the endpoint property **consumer.splitResult=false**.

233.8. MESSAGE HEADERS

Any URI option can be provided in a message header for producer endpoints with a **CamelOlingo2.** prefix.

233.9. MESSAGE BODY

All result message bodies utilize objects provided by the underlying Apache Olingo 2.0 API used by the Olingo2Component. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint URI parameter. For endpoints that return an array or collection, a consumer endpoint will map every element to distinct messages, unless **consumer.splitResult** is set to **false**.

233.10. USE CASES

The following route reads top 5 entries from the Manufacturer feed ordered by ascending Name property.

```
from("direct:...")
    .setHeader("CamelOlingo2.$top", "5");
    .to("olingo2://read/Manufacturers?orderBy=Name%20asc");
```

The following route reads Manufacturer entry using the key property value in incoming **id** header.

```
from("direct:...")
    .setHeader("CamelOlingo2.keyPredicate", header("id"))
    .to("olingo2://read/Manufacturers");
```

The following route creates Manufacturer entry using the **java.util.Map<String, Object>** in body message.

```
from("direct:...")
    .to("olingo2://create/Manufacturers");
```

The following route polls Manufacturer delta feed every 30 seconds. The bean **blah** updates the bean **paramsBean** to add an updated **!deltatoken** property with the value returned in the **ODataDeltaFeed** result. Since the initial delta token is not known, the consumer endpoint will produce an **ODataFeed** value the first time, and **ODataDeltaFeed** on subsequent polls.

```
from("olingo2://read/Manufacturers?
queryParams=#paramsBean&consumer.timeUnit=SECONDS&consumer.delay=30")
    .to("bean:blah");
```

CHAPTER 234. OLINGO4 COMPONENT

Available as of Camel version 2.19

The Olingo4 component utilizes Apache Olingo version 4.0 APIs to interact with OData 4.0 compliant service. Since version 4.0 OData is OASIS standard and number of popular open source and commercial vendors and products support this protocol. A sample list of supporting products can be found on the OData website.

The Olingo4 component supports reading entity sets, entities, simple and complex properties, counts, using custom and OData system query parameters. It supports updating entities and properties. It also supports submitting queries and change requests as a single OData batch operation.

The component supports configuring HTTP connection parameters and headers for OData service connection. This allows configuring use of SSL, OAuth2.0, etc. as required by the target OData service.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-olingo4</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

234.1. URI FORMAT

```
olingo4://endpoint/<resource-path>?[options]
```

234.2. OLINGO4 OPTIONS

The Olingo4 component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		Olingo4Configuration
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Olingo4 endpoint is configured using URI syntax:

```
olingo4:apiName/methodName
```

with the following path and query parameters:

234.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		Olingo4ApiName
methodName	**Required** What sub operation to use for the selected operation		String

234.2.2. Query Parameters (14 parameters):

Name	Description	Default	Type
connectTimeout (common)	HTTP connection creation timeout in milliseconds, defaults to 30,000 (30 seconds)	30000	int
contentType (common)	Content-Type header value can be used to specify JSON or XML message format, defaults to application/json;charset=utf-8	application/json;charset=utf-8	String
httpAsyncClientBuilder (common)	Custom HTTP async client builder for more complex HTTP client configuration, overrides connectionTimeout, socketTimeout, proxy and sslContext. Note that a socketTimeout MUST be specified in the builder, otherwise OData requests could block indefinitely		HttpAsyncClientBuilder
httpClientBuilder (common)	Custom HTTP client builder for more complex HTTP client configuration, overrides connectionTimeout, socketTimeout, proxy and sslContext. Note that a socketTimeout MUST be specified in the builder, otherwise OData requests could block indefinitely		HttpClientBuilder
httpHeaders (common)	Custom HTTP headers to inject into every request, this could include OAuth tokens, etc.		Map
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
proxy (common)	HTTP proxy server configuration		HttpHost

Name	Description	Default	Type
serviceUri (common)	Target OData service base URI, e.g. http://services.odata.org/OData/OData.svc		String
socketTimeout (common)	HTTP request timeout in milliseconds, defaults to 30,000 (30 seconds)	30000	int
sslContextParameters (common)	To configure security using SSLContextParameters		SSLContextParameters
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

234.3. PRODUCER ENDPOINTS

Producer endpoints can use endpoint names and options listed next. Producer endpoints can also use a special option **inBody** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message. The **inBody** option defaults to **data** for endpoints that take that option.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelOlingo4.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override a **CamelOlingo4.option** header. In addition, query parameters can be specified

Note that the resourcePath option can either in specified in the URI as a part of the URI path, as an endpoint option ?resourcePath=<resource-path> or as a header value CamelOlingo4.resourcePath. The OData entity key predicate can either be a part of the resource path, e.g. *Manufacturers('1')*, where *'__1'* is the key predicate, or be specified separately with resource path *Manufacturers* and keyPredicate option *'1'*.

Endpoint	Options	HTTP Method	Result Body Type
batch	data, endpointHttpHeaders	POST with multipart/mixed batch request	java.util.List<org.apache.camel.component.olingo4.api.batch.Olingo4BatchResponse>
create	data, resourcePath, endpointHttpHeaders	POST	org.apache.olingo.client.api.domain.ClientEntity for new entries org.apache.olingo.commons.api.http.HttpStatusCode for other OData resources
delete	resourcePath, endpointHttpHeaders	DELETE	org.apache.olingo.commons.api.http.HttpStatusCode
merge	data, resourcePath, endpointHttpHeaders	MERGE	org.apache.olingo.commons.api.http.HttpStatusCode
patch	data, resourcePath, endpointHttpHeaders	PATCH	org.apache.olingo.commons.api.http.HttpStatusCode
read	queryParams, resourcePath, endpointHttpHeaders	GET	Depends on OData resource being queried as described next

Endpoint	Options	HTTP Method	Result Body Type
update	data, resourcePath, endpointHttpHeaders	PUT	org.apache.olingo.commons.api.http.HttpStatusCode

234.4. ENDPOINT HTTP HEADERS (SINCE CAMEL 2.20)

The component level configuration property **httpHeaders** supplies static HTTP header information. However, some systems requires dynamic header information to be passed to and received from the endpoint. A sample use case would be systems that require dynamic security tokens. The **endpointHttpHeaders** and **responseHttpHeaders** endpoint properties provides this capability. Set headers that need to be passed to the endpoint in the **CamelOlingo4.endpointHttpHeaders** property and the response headers will be returned in a **CamelOlingo4.responseHttpHeaders** property. Both properties are of the type **java.util.Map<String, String>**.

234.5. ODATA RESOURCE TYPE MAPPING

The result of **read** endpoint and data type of **data** option depends on the OData resource being queried, created or modified.

OData Resource Type	Resource URI from resourcePath and keyPredicate	In or Out Body Type
Entity data model	$metadata	org.apache.olingo.commons.api.edm.Edm
Service document	/	org.apache.olingo.client.api.domain.ClientServiceDocument
OData entity set	<entity-set>	org.apache.olingo.client.api.domain.ClientEntitySet

OData Resource Type	Resource URI from resourcePath and keyPredicate	In or Out Body Type
OData entity	<entity-set>(<key-predicate>)	org.apache.olingo.client.api.domain.ClientEntity for Out body (response) java.util.Map<String, Object> for In body (request)
Simple property	<entity-set>(<key-predicate>)/<simple-property>	org.apache.olingo.client.api.domain.ClientPrimitiveValue
Simple property value	<entity-set>(<key-predicate>)/<simple-property>/$value	org.apache.olingo.client.api.domain.ClientPrimitiveValue
Complex property	<entity-set>(<key-predicate>)/<complex-property>	org.apache.olingo.client.api.domain.ClientComplexValue
Count	<resource-uri>/$count	java.lang.Long

234.6. CONSUMER ENDPOINTS

Only the **read** endpoint can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. By default consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange. This behavior can be disabled by setting the endpoint property **consumer.splitResult=false**.

234.7. MESSAGE HEADERS

Any URI option can be provided in a message header for producer endpoints with a **CamelOlingo4.** prefix.

234.8. MESSAGE BODY

All result message bodies utilize objects provided by the underlying Apache Olingo 4.0 API used by the Olingo4Component. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint URI parameter. For endpoints that return an array or collection, a consumer endpoint will map every element to distinct messages, unless **consumer.splitResult** is set to **false**.

234.9. USE CASES

The following route reads top 5 entries from the People entity ordered by ascending FirstName property.

```
from("direct:...")
    .setHeader("CamelOlingo4.$top", "5");
    .to("olingo4://read/People?orderBy=FirstName%20asc");
```

The following route reads Airports entity using the key property value in incoming **id** header.

```
from("direct:...")
    .setHeader("CamelOlingo4.keyPredicate", header("id"))
    .to("olingo4://read/Airports");
```

The following route creates People entity using the **ClientEntity** in body message.

```
from("direct:...")
    .to("olingo4://create/People");
```

CHAPTER 235. OPENSHIFT COMPONENT (DEPRECATED)

Available as of Camel version 2.14

The **openshift** component is a component for managing your OpenShift applications.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-openshift</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

235.1. URI FORMAT

```
openshift:clientId[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&...**

235.2. OPTIONS

The OpenShift component supports 5 options which are listed below.

Name	Description	Default	Type
username (security)	The username to login to openshift server.		String
password (security)	The password for login to openshift server.		String
domain (common)	Domain name. If not specified then the default domain is used.		String
server (common)	Url to the openshift server. If not specified then the default value from the local openshift configuration file /.openshift/express.conf is used. And if that fails as well then openshift.redhat.com is used.		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The OpenShift endpoint is configured using URI syntax:

```
openshift:clientId
```

with the following path and query parameters:

235.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
clientId	**Required** The client id		String

235.2.2. Query Parameters (26 parameters):

Name	Description	Default	Type
domain (common)	Domain name. If not specified then the default domain is used.		String
password (common)	**Required** The password for login to openshift server.		String
server (common)	Url to the openshift server. If not specified then the default value from the local openshift configuration file /.openshift/express.conf is used. And if that fails as well then openshift.redhat.com is used.		String
username (common)	**Required** The username to login to openshift server.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
application (producer)	The application name to start, stop, restart, or get the state.		String
mode (producer)	Whether to output the message body as a pojo or json. For pojo the message is a List type.		String
operation (producer)	The operation to perform which can be: list, start, stop, restart, and state. The list operation returns information about all the applications in json format. The state operation returns the state such as: started, stopped etc. The other operations does not return any value.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long

Name	Description	Default	Type
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

235.3. EXAMPLES

235.3.1. Listing all applications

```
// sending route
from("direct:apps")
    .to("openshift:myClient?username=foo&password=secret&operation=list");
    .to("log:apps");
```

In this case the information about all the applications is returned as pojo. If you want a json response, then set mode=json.

235.3.2. Stopping an application

```
// stopping the foobar application
from("direct:control")
    .to("openshift:myClient?username=foo&password=secret&operation=stop&application=foobar");
```

In the example above we stop the application named foobar.

Polling for gear state changes

The consumer is used for polling state changes in gears. Such as when a new gear is added/removed/ or its lifecycle is changed, eg started, or stopped etc.

```
// trigger when state changes on our gears
from("openshift:myClient?username=foo&password=secret&delay=30s")
    .log("Event ${header.CamelOpenShiftEventType} on application ${body.name} changed state to
${header.CamelOpenShiftEventNewState}");
```

When the consumer emits an Exchange then the body contains the **com.openshift.client.IApplication** as the message body. And the following headers is included.

Header	May be null	Description
CamelOpenShiftEventType	No	The type of the event which can be one of: added, removed or changed.
CamelOpenShiftEventOldState	Yes	The old state, when the event type is changed.
CamelOpenShiftEventNewState	No	The new state, for any of the event types

235.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint

- Getting Started

CHAPTER 236. OPENSHIFT BUILD CONFIG COMPONENT

Available as of Camel version 2.17

The **OpenShift Build Config** component is one of Kubernetes Components which provides a producer to execute kubernetes build config operations.

236.1. COMPONENT OPTIONS

The Openshift Build Config component has no options.

236.2. ENDPOINT OPTIONS

The Openshift Build Config endpoint is configured using URI syntax:

```
openshift-build-configs:masterUrl
```

with the following path and query parameters:

236.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

236.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

CHAPTER 237. OPENSHIFT BUILDS COMPONENT

Available as of Camel version 2.17

The **Kubernetes Builds** component is one of Kubernetes Components which provides a producer to execute kubernetes build operations.

237.1. COMPONENT OPTIONS

The Openshift Builds component has no options.

237.2. ENDPOINT OPTIONS

The Openshift Builds endpoint is configured using URI syntax:

```
openshift-builds:masterUrl
```

with the following path and query parameters:

237.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
masterUrl	**Required** Kubernetes Master url		String

237.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
apiVersion (producer)	The Kubernetes API Version to use		String
dnsDomain (producer)	The dns domain, used for ServiceCall EIP		String
kubernetesClient (producer)	Default KubernetesClient to use if provided		KubernetesClient
operation (producer)	Producer operation to do on Kubernetes		String
portName (producer)	The port name, used for ServiceCall EIP		String
connectionTimeout (advanced)	Connection timeout in milliseconds to use when making requests to the Kubernetes API server.		Integer

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
caCertData (security)	The CA Cert Data		String
caCertFile (security)	The CA Cert File		String
clientCertData (security)	The Client Cert Data		String
clientCertFile (security)	The Client Cert File		String
clientKeyAlgo (security)	The Key Algorithm used by the client		String
clientKeyData (security)	The Client Key data		String
clientKeyFile (security)	The Client Key file		String
clientKeyPassphrase (security)	The Client Key Passphrase		String
oauthToken (security)	The Auth Token		String
password (security)	Password to connect to Kubernetes		String
trustCerts (security)	Define if the certs we used are trusted anyway or not		Boolean
username (security)	Username to connect to Kubernetes		String

237.3. OPENSTACK COMPONENT

Available as of Camel 2.19

The **openstack** component is a component for managing your OpenStack applications.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-openstack</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

OpenStack service	Camel Component	Description
OpenStack Cinder	openstack-cinder	Component to maintain OpenStack cinder.
OpenStack Glance	openstack-glance	Component to maintain OpenStack glance.
OpenStack Keystone	openstack-keystone	Component to maintain OpenStack keystone.
OpenStack Neutron	openstack-neutron	Component to maintain OpenStack neutron.
OpenStack Nova	openstack-nova	Component to maintain OpenStack nova.
OpenStack Swift	openstack-swift	Component to maintain OpenStack swift.

CHAPTER 238. OPENSTACK CINDER COMPONENT

Available as of Camel version 2.19

The openstack-cinder component allows messages to be sent to an OpenStack block storage services.

238.1. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-openstack</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel.

238.2. URI FORMAT

```
openstack-cinder://hosturl[?options]
```

You can append query options to the URI in the following format **?options=value&option2=value&…**

238.3. URI OPTIONS

The OpenStack Cinder component has no options.

The OpenStack Cinder endpoint is configured using URI syntax:

```
openstack-cinder:host
```

with the following path and query parameters:

238.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** OpenStack host url		String

238.3.2. Query Parameters (9 parameters):

Name	Description	Default	Type
apiVersion (producer)	OpenStack API version	V3	String
config (producer)	OpenStack configuration		Config
domain (producer)	Authentication domain	default	String
operation (producer)	The operation to do		String
password (producer)	**Required** OpenStack password		String
project (producer)	**Required** The project ID		String
subsystem (producer)	**Required** OpenStack Cinder subsystem		String
username (producer)	**Required** OpenStack username		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

238.4. USAGE

You can use following settings for each subsystem:

238.5. VOLUMES

238.5.1. Operations you can perform with the Volume producer

Operation	Description
create	Create new volume.
get	Get the volume.
getAll	Get all volumes.
getAllTypes	Get volume types.

Operation	Description
update	Update the volume.
delete	Delete the volume.

238.5.2. Message headers evaluated by the Volume producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the volume.
name	**String**	The volume name.
description	**String**	Volume description.
size	**Integer**	Size of volume.
volumeType	**String**	Volume type.
imageRef	**String**	ID of image.
snapshotId	**String**	ID of snapshot.
isBootable	**Boolean**	Is bootable.

If you need more precise volume settings you can create new object of the type **org.openstack4j.model.storage.block.Volume** and send in the message body.

238.6. SNAPSHOTS

238.6.1. Operations you can perform with the Snapshot producer

Operation	Description
create	Create new snapshot.

Operation	Description
get	Get the snapshot.
getAll	Get all snapshots.
update	Get update the snapshot.
delete	Delete the snapshot.

238.6.2. Message headers evaluated by the Snapshot producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the server.
name	**String**	The server name.
description	**String**	The snapshot description.
VolumeId	**String**	The Volume ID.
force	**Boolean**	Force.

If you need more precise server settings you can create new object of the type **org.openstack4j.model.storage.block.VolumeSnapshot** and send in the message body.

238.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- openstack Component

CHAPTER 239. OPENSTACK GLANCE COMPONENT

Available as of Camel version 2.19

The openstack-glance component allows messages to be sent to an OpenStack image services.

239.1. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-openstack</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel.

239.2. URI FORMAT

```
openstack-glance://hosturl[?options]
```

You can append query options to the URI in the following format **?options=value&option2=value&...**

239.3. URI OPTIONS

The OpenStack Glance component has no options.

The OpenStack Glance endpoint is configured using URI syntax:

```
openstack-glance:host
```

with the following path and query parameters:

239.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** OpenStack host url		String

239.3.2. Query Parameters (8 parameters):

Name	Description	Default	Type
apiVersion (producer)	OpenStack API version	V3	String
config (producer)	OpenStack configuration		Config
domain (producer)	Authentication domain	default	String
operation (producer)	The operation to do		String
password (producer)	**Required** OpenStack password		String
project (producer)	**Required** The project ID		String
username (producer)	**Required** OpenStack username		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

239.4. USAGE

Operation	Description
reserve	Reserve image.
create	Create new image.
update	Update image.
upload	Upload image.
get	Get the image.
getAll	Get all image.
delete	Delete the image.

239.4.1. Message headers evaluated by the Glance producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the flavor.
name	**String**	The flavor name.
diskFormat	**org.openstack4j.model.image.DiskFormat**	The number of flavor VCPU.
containerFormat	**org.openstack4j.model.image.ContainerFormat**	Size of RAM.
owner	**String**	Image owner.
isPublic	**Boolean**	Is public.
minRam	**Long**	Minimum ram.
minDisk	**Long**	Minimum disk.
size	**Long**	Size.
checksum	**String**	Checksum.
properties	**Map**	Image properties.

239.5. SEE ALSO

- Configuring Camel
- Component

- Endpoint
- Getting Started
- openstack Component

CHAPTER 240. OPENSTACK KEYSTONE COMPONENT

Available as of Camel version 2.19

The openstack-keystone component allows messages to be sent to an OpenStack identity services.

The openstack-keystone component supports only Identity API v3!

240.1. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-openstack</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version**} must be replaced by the actual version of Camel.

240.2. URI FORMAT

```
openstack-keystone://hosturl[?options]
```

You can append query options to the URI in the following format **?options=value&option2=value&...**

240.3. URI OPTIONS

The OpenStack Keystone component has no options.

The OpenStack Keystone endpoint is configured using URI syntax:

```
openstack-keystone:host
```

with the following path and query parameters:

240.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** OpenStack host url		String

240.3.2. Query Parameters (8 parameters):

Name	Description	Default	Type
config (producer)	OpenStack configuration		Config
domain (producer)	Authentication domain	default	String
operation (producer)	The operation to do		String
password (producer)	**Required** OpenStack password		String
project (producer)	**Required** The project ID		String
subsystem (producer)	**Required** OpenStack Keystone subsystem		String
username (producer)	**Required** OpenStack username		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

240.4. USAGE

You can use following settings for each subsystem:

240.5. DOMAINS

240.5.1. Operations you can perform with the Domain producer

Operation	Description
create	Create new domain.
get	Get the domain.
getAll	Get all domains.
update	Update the domain.
delete	Delete the domain.

240.5.2. Message headers evaluated by the Domain producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the domain.
name	**String**	The domain name.
description	**String**	Domain description.

If you need more precise domain settings you can create new object of the type **org.openstack4j.model.identity.v3.Domain** and send in the message body.

240.6. GROUPS

240.6.1. Operations you can perform with the Group producer

Operation	Description
create	Create new group.
get	Get the group.
getAll	Get all groups.
update	Update the group.
delete	Delete the group.
addUserToGroup	Add the user to the group.
checkUserGroup	Check whether is the user in the group.
removeUserFromGroup	Remove the user from the group.

240.6.2. Message headers evaluated by the Group producer

Header	Type	Description
operation	**String**	The operation to perform.
groupId	**String**	ID of the group.
name	**String**	The group name.
userId	**String**	ID of the user.
domainId	**String**	ID of the domain.
description	**String**	Group description.

If you need more precise group settings you can create new object of the type **org.openstack4j.model.identity.v3.Group** and send in the message body.

240.7. PROJECTS

240.7.1. Operations you can perform with the Project producer

Operation	Description
create	Create new project.
get	Get the project.
getAll	Get all projects.
update	Update the project.
delete	Delete the project.

240.7.2. Message headers evaluated by the Project producer

Header	Type	Description
operation	**String**	The operation to perform.

Header	Type	Description
ID	**String**	ID of the project.
name	**String**	The project name.
description	**String**	Project description.
domainId	**String**	ID of the domain.
parentId	**String**	The parent project ID.

If you need more precise project settings you can create new object of the type **org.openstack4j.model.identity.v3.Project** and send in the message body.

240.8. REGIONS

240.8.1. Operations you can perform with the Region producer

Operation	Description
create	Create new region.
get	Get the region.
getAll	Get all regions.
update	Update the region.
delete	Delete the region.

240.8.2. Message headers evaluated by the Region producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the region.
description	**String**	Region description.

If you need more precise region settings you can create new object of the type **org.openstack4j.model.identity.v3.Region** and send in the message body.

240.9. USERS

240.9.1. Operations you can perform with the User producer

Operation	Description
create	Create new user.
get	Get the user.
getAll	Get all users.
update	Update the user.
delete	Delete the user.

240.9.2. Message headers evaluated by the User producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the user.
name	**String**	The user name.
description	**String**	User description.
domainId	**String**	ID of the domain.
password	**String**	User's password.
email	**String**	User's email.

If you need more precise user settings you can create new object of the type **org.openstack4j.model.identity.v3.User** and send in the message body.

240.10. SEE ALSO

- Configuring Camel

- Component
- Endpoint
- Getting Started
- openstack Component

CHAPTER 241. OPENSTACK NEUTRON COMPONENT

Available as of Camel version 2.19

The openstack-neutron component allows messages to be sent to an OpenStack network services.

241.1. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-openstack</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel.

241.2. URI FORMAT

```
openstack-neutron://hosturl[?options]
```

You can append query options to the URI in the following format **?options=value&option2=value&...**

241.3. URI OPTIONS

The OpenStack Neutron component has no options.

The OpenStack Neutron endpoint is configured using URI syntax:

```
openstack-neutron:host
```

with the following path and query parameters:

241.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** OpenStack host url		String

241.3.2. Query Parameters (9 parameters):

Name	Description	Default	Type
apiVersion (producer)	OpenStack API version	V3	String
config (producer)	OpenStack configuration		Config
domain (producer)	Authentication domain	default	String
operation (producer)	The operation to do		String
password (producer)	**Required** OpenStack password		String
project (producer)	**Required** The project ID		String
subsystem (producer)	**Required** OpenStack Neutron subsystem		String
username (producer)	**Required** OpenStack username		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

241.4. USAGE

You can use following settings for each subsystem:

241.5. NETWORKS

241.5.1. Operations you can perform with the Network producer

Operation	Description
create	Create new network.
get	Get the network.
getAll	Get all networks.
delete	Delete the network.

241.5.2. Message headers evaluated by the Network producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the network.
name	**String**	The network name.
tenantId	**String**	Tenant ID.
adminStateUp	**Boolean**	AdminStateUp header.
networkType	**org.openstack4j.model.network.NetworkType**	Network type.
physicalNetwork	**String**	Physical network.
segmentId	**String**	Segment ID.
isShared	**Boolean**	Is shared.
isRouterExternal	**Boolean**	Is router external.

If you need more precise network settings you can create new object of the type **org.openstack4j.model.network.Network** and send in the message body.

241.6. SUBNETS

241.6.1. Operations you can perform with the Subnet producer

Operation	Description
create	Create new subnet.
get	Get the subnet.
getAll	Get all subnets.
delete	Delete the subnet.
action	Perform an action on the subnet.

241.6.2. Message headers evaluated by the Subnet producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the subnet.
name	**String**	The subnet name.
networkId	**String**	Network ID.
enableDHCP	**Boolean**	Enable DHCP.
gateway	**String**	Gateway.

If you need more precise subnet settings you can create new object of the type **org.openstack4j.model.network.Subnet** and send in the message body.

241.7. PORTS

241.7.1. Operations you can perform with the Port producer

Operation	Description
create	Create new port.
get	Get the port.

Operation	Description
getAll	Get all ports.
update	Update the port.
delete	Delete the port.

241.7.2. Message headers evaluated by the Port producer

Header	Type	Description
operation	**String**	The operation to perform.
name	**String**	The port name.
networkId	**String**	Network ID.
tenantId	**String**	Tenant ID.
deviceId	**String**	Device ID.
macAddress	**String**	MAC address.

241.8. ROUTERS

241.8.1. Operations you can perform with the Router producer

Operation	Description
create	Create new router.
get	Get the router.
getAll	Get all routers.
update	Update the router.
delete	Delete the router.

Operation	Description
attachInterface	Attach an interface.
detachInterface	Detach an interface.

241.8.2. Message headers evaluated by the Port producer

Header	Type	Description
operation	**String**	The operation to perform.
name	**String**	The router name.
routerId	**String**	Router ID.
subnetId	**String**	Subnet ID.
portId	**String**	Port ID.
interfaceType	**org.openstack4j.model.network.AttachInterfaceType**	Interface type.
tenantId	**String**	Tenant ID.

241.9. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- openstack Component

CHAPTER 242. OPENSTACK NOVA COMPONENT

Available as of Camel version 2.19

The openstack-nova component allows messages to be sent to an OpenStack compute services.

242.1. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-openstack</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel.

242.2. URI FORMAT

```
openstack-nova://hosturl[?options]
```

You can append query options to the URI in the following format **?options=value&option2=value&...**

242.3. URI OPTIONS

The OpenStack Nova component has no options.

The OpenStack Nova endpoint is configured using URI syntax:

```
openstack-nova:host
```

with the following path and query parameters:

242.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** OpenStack host url		String

242.3.2. Query Parameters (9 parameters):

Name	Description	Default	Type
apiVersion (producer)	OpenStack API version	V3	String
config (producer)	OpenStack configuration		Config
domain (producer)	Authentication domain	default	String
operation (producer)	The operation to do		String
password (producer)	**Required** OpenStack password		String
project (producer)	**Required** The project ID		String
subsystem (producer)	**Required** OpenStack Nova subsystem		String
username (producer)	**Required** OpenStack username		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

242.4. USAGE

You can use following settings for each subsystem:

242.5. FLAVORS

242.5.1. Operations you can perform with the Flavor producer

Operation	Description
create	Create new flavor.
get	Get the flavor.
getAll	Get all flavors.
delete	Delete the flavor.

242.5.2. Message headers evaluated by the Flavor producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the flavor.
name	**String**	The flavor name.
VCPU	**Integer**	The number of flavor VCPU.
ram	**Integer**	Size of RAM.
disk	**Integer**	Size of disk.
swap	**Integer**	Size of swap.
rxtxFactor	**Integer**	Rxtx Factor.

If you need more precise flavor settings you can create new object of the type **org.openstack4j.model.compute.Flavor** and send in the message body.

242.6. SERVERS

242.6.1. Operations you can perform with the Server producer

Operation	Description
create	Create new server.
createSnapshot	Create snapshot of the server.
get	Get the server.
getAll	Get all servers.
delete	Delete the server.
action	Perform an action on the server.

242.6.2. Message headers evaluated by the Server producer

Header	Type	Description
operation	**String**	The operation to perform.
ID	**String**	ID of the server.
name	**String**	The server name.
ImageId	**String**	The Image ID.
FlavorId	**String**	The ID of flavor which will be used.
KeypairName	**String**	The Keypair name.
NetworkId	**String**	The network ID.
AdminPassword	**String**	Admin password of the new server.
action	**org.openstack4j.model.compute.Action**	An action to perform.

If you need more precise server settings you can create new object of the type **org.openstack4j.model.compute.ServerCreate** and send in the message body.

242.7. KEYPAIRS

242.7.1. Operations you can perform with the Keypair producer

Operation	Description
create	Create new keypair.
get	Get the keypair.

Operation	Description
getAll	Get all keypairs.
delete	Delete the keypair.

242.7.2. Message headers evaluated by the Keypair producer

Header	Type	Description
operation	**String**	The operation to perform.
name	**String**	The keypair name.

242.8. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- openstack Component

CHAPTER 243. OPENSTACK SWIFT COMPONENT

Available as of Camel version 2.19

The openstack-swift component allows messages to be sent to an OpenStack object storage services.

243.1. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-openstack</artifactId>
    <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel.

243.2. URI FORMAT

```
openstack-swift://hosturl[?options]
```

You can append query options to the URI in the following format **?options=value&option2=value&…**

243.3. URI OPTIONS

The OpenStack Swift component has no options.

The OpenStack Swift endpoint is configured using URI syntax:

```
openstack-swift:host
```

with the following path and query parameters:

243.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
host	**Required** OpenStack host url		String

243.3.2. Query Parameters (9 parameters):

Name	Description	Default	Type
apiVersion (producer)	OpenStack API version	V3	String
config (producer)	OpenStack configuration		Config
domain (producer)	Authentication domain	default	String
operation (producer)	The operation to do		String
password (producer)	**Required** OpenStack password		String
project (producer)	**Required** The project ID		String
subsystem (producer)	**Required** OpenStack Swift subsystem		String
username (producer)	**Required** OpenStack username		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

243.4. USAGE

You can use following settings for each subsystem:

243.5. CONTAINERS

243.5.1. Operations you can perform with the Container producer

Operation	Description
create	Create new container.
get	Get the container.
getAll	Get all containers.
update	Update the container.

Operation	Description
delete	Delete the container.
getMetadata	Get metadata.
createUpdateMetadata	Create/update metadata.
deleteMetadata	Delete metadata.

243.5.2. Message headers evaluated by the Volume producer

Header	Type	Description
operation	**String**	The operation to perform.
name	**String**	The container name.
X-Container-Meta-	**Map**	Container metadata prefix.
X-Versions-Location	**String**	Versions location.
X-Container-Read	**String**	ACL - container read.
X-Container-Write	**String**	ACL - container write.
limit	**Integer**	List options - limit.
marker	**String**	List options - marker.
end_marker	**String**	List options - end marker.
delimiter	**Character**	List options - delimiter.
path	**String**	List options - path.

If you need more precise container settings you can create new object of the type **org.openstack4j.model.storage.object.options.CreateUpdateContainerOptions** (in case of create or update operation) or **org.openstack4j.model.storage.object.options.ContainerListOptions** for listing containers and send in the message body.

243.6. OBJECTS

243.6.1. Operations you can perform with the Object producer

Operation	Description
create	Create new object.
get	Get the object.
getAll	Get all objects.
update	Get update the object.
delete	Delete the object.
getMetadata	Get metadata.
createUpdateMetadata	Create/update metadata.

243.6.2. Message headers evaluated by the Object producer

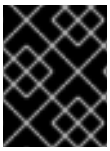
Header	Type	Description
operation	**String**	The operation to perform.
containerName	**String**	The container name.
objectName	**String**	The object name.

243.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- openstack Component

CHAPTER 244. OPENTRACING COMPONENT

Available as of Camel 2.19



IMPORTANT

Starting with Camel 2.21, it will be necessary to use an OpenTracing complaint tracer that is compatible with OpenTracing Java API version 0.31 or higher.

The camel-opentracing component is used for tracing and timing incoming and outgoing Camel messages using OpenTracing.

Events (spans) are captured for incoming and outgoing messages being sent to/from Camel.

See the OpenTracing website for a list of supported tracers.

244.1. CONFIGURATION

The configuration properties for the OpenTracing tracer are:

Option	Default	Description
excludePatterns		Sets exclude pattern(s) that will disable tracing for Camel messages that matches the pattern. The content is a Set<String> where the key is a pattern. The pattern uses the rules from Intercept.

There are three ways in which an OpenTracing tracer can be configured to provide distributed tracing for a Camel application:

244.1.1. Explicit

Include the **camel-opentracing** component in your POM, along with any specific dependencies associated with the chosen OpenTracing compliant Tracer.

To explicitly configure OpenTracing support, instantiate the **OpenTracingTracer** and initialize the camel context. You can optionally specify a **Tracer**, or alternatively it can be implicitly discovered using the **Registry** or **ServiceLoader**.

```
OpenTracingTracer ottracer = new OpenTracingTracer();
// By default it uses a Noop Tracer, but you can override it with a specific OpenTracing implementation.
ottracer.setTracer(...);
// And then initialize the context
ottracer.init(camelContext);
```

To use OpenTracingTracer in XML, all you need to do is to define the OpenTracing tracer beans. Camel will automatically discover and use them.

```
<bean id="tracer" class="..."/>
<bean id="ottracer" class="org.apache.camel.opentracing.OpenTracingTracer">
  <property name="tracer" ref="tracer"/>
</bean>
```

244.1.2. Spring Boot

If you are using Spring Boot then you can add the **camel-opentracing-starter** dependency, and turn on OpenTracing by annotating the main class with **@CamelOpenTracing**.

The **Tracer** will be implicitly obtained from the camel context's **Registry**, or the **ServiceLoader**, unless a **Tracer** bean has been defined by the application.

244.1.3. Java Agent

The third approach is to use a Java Agent to automatically configure the OpenTracing support.

Include the **camel-opentracing** component in your POM, along with any specific dependencies associated with the chosen OpenTracing compliant Tracer.

The OpenTracing Java Agent is associated with the following dependency:

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-agent</artifactId>
</dependency>
```

The **Tracer** used will be implicitly loaded from the camel context **Registry** or using the **ServiceLoader**.

How this agent is used will be specific to how you execute your application. *Service2* in the camel-example-opentracing downloads the agent into a local folder and then uses the **exec-maven-plugin** to launch the service with the **-javaagent** command line option.

244.2. EXAMPLE

You can find an example demonstrating the three ways to configure OpenTracing here: camel-example-opentracing

CHAPTER 245. OPTAPLANNER COMPONENT

Available as of Camel version 2.13

The **optaplanner:** component solves the planning problem contained in a message with OptaPlanner. For example: feed it an unsolved Vehicle Routing problem and it solves it.

The component supports consumer as BestSolutionChangedEvent listener and producer for processing Solution and ProblemFactChange

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-optaplanner</artifactId>
  <version>x.x.x</version><!-- use the same version as your Camel core version -->
</dependency>
```

245.1. URI FORMAT

```
optaplanner:solverConfig[?options]
```

The **solverConfig** is the classpath-local URI of the solverConfig, for example **/org/foo/barSolverConfig.xml**.

You can append query options to the URI in the following format, **?option=value&option=value&…**

245.2. OPTAPLANNER OPTIONS

The OptaPlanner component has no options.

The OptaPlanner endpoint is configured using URI syntax:

```
optaplanner:configFile
```

with the following path and query parameters:

245.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
configFile	**Required** Specifies the location to the solver file		String

245.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
solverId (common)	Specifies the solverId to user for the solver instance key	DEFAULT_SOLVER	String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
async (producer)	Specifies to perform operations in async mode	false	boolean
threadPoolSize (producer)	Specifies the thread pool size to use when async is true	10	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

245.3. MESSAGE HEADERS

Name	Default Value	Type	Context	Description
CamelOptaPlannerSolverId	null	String	Shared	Specifies the solverId to use

Name	Default Value	Type	Context	Description
Camel OptaPlannerIsAsync	PUT	String	Producer	Specify whether to use another thread for submitting Solution instances rather than blocking the current thread.

245.4. MESSAGE BODY

Camel takes the planning problem for the IN body, solves it and returns it on the OUT body. (since v 2.16) The IN body object supports the following use cases:

- If the body is instance of Solution, then it will be solved using the solver identified by solverId and either synchronously or asynchronously.
- If the body is instance of ProblemFactChange, then it will trigger addProblemFactChange. If the processing is asynchronously, then it will wait till isEveryProblemFactChangeProcessed before returning result.
- If the body is none of the above types, then the producer will return the best result from the solver identified by solverId

245.5. TERMINATION

The solving will take as long as specified in the **solverConfig**.

```
<solver>
  ...
  <termination>
    <!-- Terminate after 10 seconds, unless it's not feasible by then yet -->
    <terminationCompositionStyle>AND</terminationCompositionStyle>
    <secondsSpentLimit>10</secondsSpentLimit>
    <bestScoreLimit>-1hard/0soft</bestScoreLimit>
  </termination>
  ...
<solver>
```

245.5.1. Samples

Solve an planning problem that's on the ActiveMQ queue with OptaPlanner:

```
from("activemq:My.Queue").
  .to("optaplanner:/org/foo/barSolverConfig.xml");
```

Expose OptaPlanner as a REST service:

```
from("cxfrs:bean:rsServer?bindingStyle=SimpleConsumer")
  .to("optaplanner:/org/foo/barSolverConfig.xml");
```

245.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 246. PAHO COMPONENT

Available as of Camel version 2.16

Paho component provides connector for the MQTT messaging protocol using the Eclipse Paho library. Paho is one of the most popular MQTT libraries, so if you would like to integrate it with your Java project - Camel Paho connector is a way to go.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-paho</artifactId>
  <version>x.y.z</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Keep in mind that Paho artifacts are not hosted in the Maven Central, so you need to add Eclipse Paho repository to your POM xml file:

```
<repositories>
 <repository>
  <id>eclipse-paho</id>
  <url>https://repo.eclipse.org/content/repositories/paho-releases</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
 </repository>
</repositories>
```

246.1. URI FORMAT

```
paho:topic[?options]
```

Where **topic** is the name of the topic.

246.2. OPTIONS

The Paho component supports 4 options which are listed below.

Name	Description	Default	Type
brokerUrl (common)	The URL of the MQTT broker.		String
clientId (common)	MQTT client identifier.		String
connectOptions (advanced)	Client connection options		MqttConnectOptions

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Paho endpoint is configured using URI syntax:

```
paho:topic
```

with the following path and query parameters:

246.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
topic	**Required** Name of the topic		String

246.2.2. Query Parameters (14 parameters):

Name	Description	Default	Type
autoReconnect (common)	Client will automatically attempt to reconnect to the server if the connection is lost	true	boolean
brokerUrl (common)	The URL of the MQTT broker.	tcp://localhost:1883	String
clientId (common)	MQTT client identifier.		String
connectOptions (common)	Client connection options		MqttConnectOptions
filePersistenceDirectory (common)	Base directory used by the file persistence provider.		String
password (common)	Password to be used for authentication against the MQTT broker		String
persistence (common)	Client persistence to be used - memory or file.	MEMORY	PahoPersistence

Name	Description	Default	Type
qos (common)	Client quality of service level (0-2).	2	int
retained (common)	Retain option	false	boolean
userName (common)	Username to be used for authentication against the MQTT broker		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

246.3. HEADERS

The following headers are recognized by the Paho component:

Header	Java constant	Endpoint type	Value type	Description
CamelMqttTopic	PahoConstants.MQTT_TOPIC	Consumer	String	The name of the topic

Header	Java constant	Endpoint type	Value type	Description
CamelPahoOverrideTopic	PahoConstants.CAMEL_PAHO_OVERRIDE_TOPIC	Producer	String	Name of topic to override and send to instead of topic specified on endpoint

246.4. DEFAULT PAYLOAD TYPE

By default Camel Paho component operates on the binary payloads extracted out of (or put into) the MQTT message:

```
// Receive payload
byte[] payload = (byte[]) consumerTemplate.receiveBody("paho:topic");

// Send payload
byte[] payload = "message".getBytes();
producerTemplate.sendBody("paho:topic", payload);
```

But of course Camel build-in type conversion API can perform the automatic data type transformations for you. In the example below Camel automatically converts binary payload into **String** (and conversely):

```
// Receive payload
String payload = consumerTemplate.receiveBody("paho:topic", String.class);

// Send payload
String payload = "message";
producerTemplate.sendBody("paho:topic", payload);
```

246.5. SAMPLES

For example the following snippet reads messages from the MQTT broker installed on the same host as the Camel router:

```
from("paho:some/queue")
    .to("mock:test");
```

While the snippet below sends message to the MQTT broker:

```
from("direct:test")
    .to("paho:some/target/queue");
```

For example this is how to read messages from the remote MQTT broker:

```
from("paho:some/queue?brokerUrl=tcp://iot.eclipse.org:1883")
    .to("mock:test");
```

And here we override the default topic and set to a dynamic topic

```
from("direct:test")
    .setHeader(PahoConstants.CAMEL_PAHO_OVERRIDE_TOPIC, simple("${header.customerId}"))
    .to("paho:some/target/queue");
```

CHAPTER 247. OSGI PAX LOGGING COMPONENT

Available as of Camel version 2.6

The **paxlogging** component can be used in an OSGi environment to receive PaxLogging events and process them.

247.1. DEPENDENCIES

Maven users need to add the following dependency to their **pom.xml**

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-paxlogging</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where **${camel-version}** must be replaced by the actual version of Camel (2.6.0 or higher).

247.2. URI FORMAT

```
paxlogging:appender[?options]
```

where **appender** is the name of the pax appender that need to be configured in the PaxLogging service configuration.

247.3. URI OPTIONS

The OSGi PAX Logging component supports 2 options which are listed below.

Name	Description	Default	Type
bundleContext (consumer)	The OSGi BundleContext is automatic injected by Camel		BundleContext
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The OSGi PAX Logging endpoint is configured using URI syntax:

```
paxlogging:appender
```

with the following path and query parameters:

247.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
appender	**Required** Appender is the name of the pax appender that need to be configured in the PaxLogging service configuration.		String

247.3.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

247.4. MESSAGE BODY

The **in** message body will be set to the received PaxLoggingEvent.

247.5. EXAMPLE USAGE

```
<route>
  <from uri="paxlogging:camel"/>
  <to uri="stream:out"/>
</route>
```

Configuration:

```
log4j.rootLogger=INFO, out, osgi:VmLogAppender, osgi:camel
```

CHAPTER 248. PDF COMPONENT

Available as of Camel version 2.16

The **PDF**: components provides the ability to create, modify or extract content from PDF documents. This component uses Apache PDFBox as underlying library to work with PDF documents.

In order to use the PDF component, Maven users will need to add the following dependency to their **pom.xml**:

pom.xml

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-pdf</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

248.1. URI FORMAT

The PDF component only supports producer endpoints.

```
pdf:operation[?options]
```

248.2. OPTIONS

The PDF component has no options.

The PDF endpoint is configured using URI syntax:

```
pdf:operation
```

with the following path and query parameters:

248.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
operation	**Required** Operation type		PdfOperation

248.2.2. Query Parameters (9 parameters):

Name	Description	Default	Type
font (producer)	Font	Helvetica	PDFont

Name	Description	Default	Type
fontSize (producer)	Font size in pixels	14	float
marginBottom (producer)	Margin bottom in pixels	20	int
marginLeft (producer)	Margin left in pixels	20	int
marginRight (producer)	Margin right in pixels	40	int
marginTop (producer)	Margin top in pixels	20	int
pageSize (producer)	Page size	A4	PDRectangle
textProcessingFactory (producer)	Text processing to use. autoFormatting: Text is getting sliced by words, then max amount of words that fits in the line will be written into pdf document. With this strategy all words that doesn't fit in the line will be moved to the new line. lineTermination: Builds set of classes for line-termination writing strategy. Text getting sliced by line termination symbol and then it will be written regardless it fits in the line or not.	lineTermination	TextProcessingFactory
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

248.3. HEADERS

Header	Description
pdf-document	**Mandatory** header for **append** operation and ignored in all other operations. Expected type is PDDocument. Stores PDF document which will be used for append operation.
protection-policy	Expected type ishttps://pdfbox.apache.org/docs/1.8.10/javadocs/org/apache/pdfbox/pdmodel/encryption/ProtectionPolicy.html[ProtectionPolicy]. If specified then PDF document will be encrypted with it.

Header	Description
decryption-material	Expected type ishttps://pdfbox.apache.org/docs/1.8.10/javadocs/org/apache/pdfbox/pdmodel/encryption/DecryptionMaterial.html[DecryptionMaterial]. **Mandatory** header if PDF document is encrypted.

248.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

- -

CHAPTER 249. POSTGRESSQL EVENT COMPONENT

Available as of Camel version 2.15

This is a component for Apache Camel which allows for Producing/Consuming PostgreSQL events related to the LISTEN/NOTIFY commands added since PostgreSQL 8.3.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-pgevent</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI format

The pgevent component uses the following two styles of endpoint URI notation:

```
pgevent:datasource[?parameters]
pgevent://host:port/database/channel[?parameters]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

249.1. OPTIONS

The PostgresSQL Event component has no options.

The PostgresSQL Event endpoint is configured using URI syntax:

```
pgevent:host:port/database/channel
```

with the following path and query parameters:

249.1.1. Path Parameters (4 parameters):

Name	Description	Default	Type
host	To connect using hostname and port to the database.	localhost	String
port	To connect using hostname and port to the database.	5432	Integer
database	**Required** The database name		String
channel	**Required** The channel name		String

249.1.2. Query Parameters (7 parameters):

Name	Description	Default	Type
datasource (common)	To connect using the given javax.sql.DataSource instead of using hostname and port.		DataSource
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
pass (security)	Password for login		String
user (security)	Username for login	postgres	String

249.2. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 250. PGP DATAFORMAT

Available as of Camel version 2.9

The PGP Data Format integrates the Java Cryptographic Extension into Camel, allowing simple and flexible encryption and decryption of messages using Camel's familiar marshall and unmarshal formatting mechanism. It assumes marshalling to mean encryption to cyphertext and unmarshalling to mean decryption back to the original plaintext. This data format implements only symmetric (shared-key) encryption and decyption.

250.1. PGPDATAFORMAT OPTIONS

The PGP dataformat supports 15 options which are listed below.

Name	Default	Java Type	Description
keyUserid		**String**	The user ID of the key in the PGP keyring used during encryption. Can also be only a part of a user ID. For example, if the user ID is Test User then you can use the part Test User or to address the user ID.
signatureKeyUserid		**String**	User ID of the key in the PGP keyring used for signing (during encryption) or signature verification (during decryption). During the signature verification process the specified User ID restricts the public keys from the public keyring which can be used for the verification. If no User ID is specified for the signature verficiation then any public key in the public keyring can be used for the verification. Can also be only a part of a user ID. For example, if the user ID is Test User then you can use the part Test User or to address the User ID.
password		**String**	Password used when opening the private key (not used for encryption).
signaturePassword		**String**	Password used when opening the private key used for signing (during encryption).
keyFileName		**String**	Filename of the keyring; must be accessible as a classpath resource (but you can specify a location in the file system by using the file: prefix).
signatureKeyFileName		**String**	Filename of the keyring to use for signing (during encryption) or for signature verification (during decryption); must be accessible as a classpath resource (but you can specify a location in the file system by using the file: prefix).
signatureKeyRing		**String**	Keyring used for signing/verifying as byte array. You can not set the signatureKeyFileName and signatureKeyRing at the same time.

Name	Default	Java Type	Description
armored	false	Boolean	This option will cause PGP to base64 encode the encrypted text, making it available for copy/paste, etc.
integrity	true	Boolean	Adds an integrity check/sign into the encryption file. The default value is true.
provider		String	Java Cryptography Extension (JCE) provider, default is Bouncy Castle (BC). Alternatively you can use, for example, the IAIK JCE provider; in this case the provider must be registered beforehand and the Bouncy Castle provider must not be registered beforehand. The Sun JCE provider does not work.
algorithm		Integer	Symmetric key encryption algorithm; possible values are defined in org.bouncycastle.bcpg.SymmetricKeyAlgorithmTags; for example 2 (= TRIPLE DES), 3 (= CAST5), 4 (= BLOWFISH), 6 (= DES), 7 (= AES_128). Only relevant for encrypting.
compressionAlgorithm		Integer	Compression algorithm; possible values are defined in org.bouncycastle.bcpg.CompressionAlgorithmTags; for example 0 (= UNCOMPRESSED), 1 (= ZIP), 2 (= ZLIB), 3 (= BZIP2). Only relevant for encrypting.
hashAlgorithm		Integer	Signature hash algorithm; possible values are defined in org.bouncycastle.bcpg.HashAlgorithmTags; for example 2 (= SHA1), 8 (= SHA256), 9 (= SHA384), 10 (= SHA512), 11 (=SHA224). Only relevant for signing.
signatureVerificationOption		String	Controls the behavior for verifying the signature during unmarshaling. There are 4 values possible: optional: The PGP message may or may not contain signatures; if it does contain signatures, then a signature verification is executed. required: The PGP message must contain at least one signature; if this is not the case an exception (PGPException) is thrown. A signature verification is executed. ignore: Contained signatures in the PGP message are ignored; no signature verification is executed. no_signature_allowed: The PGP message must not contain a signature; otherwise an exception (PGPException) is thrown.
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

250.2. PGPDATAFORMAT MESSAGE HEADERS

You can override the PGPDataFormat options by applying below headers into message dynamically.

Name	Type	Description
CamelPGPDataFormatKeyFileName	**String**	**Since Camel 2.11.0**; filename of the keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatEncryptionKeyRing	**byte[]**	**Since Camel 2.12.1**; the encryption keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatKeyUserid	**String**	**Since Camel 2.11.0**; the User ID of the key in the PGP keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatKeyUserids	**List<String>**	**Since camel 2.12.2**: the User IDs of the key in the PGP keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatKeyPassword	**String**	**Since Camel 2.11.0**; password used when opening the private key; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureKeyFileName	**String**	**Since Camel 2.11.0**; filename of the signature keyring; will override existing setting directly on the PGPDataFormat.

Name	Type	Description
CamelPGPDataFormatSignatureKeyRing	**byte[]**	**Since Camel 2.12.1**; the signature keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureKeyUserid	**String**	**Since Camel 2.11.0**; the User ID of the signature key in the PGP keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureKeyUserids	**List<String>**	**Since Camel 2.12.3**; the User IDs of the signature keys in the PGP keyring; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureKeyPassword	**String**	**Since Camel 2.11.0**; password used when opening the signature private key; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatEncryptionAlgorithm	**int**	**Since Camel 2.12.2**; symmetric key encryption algorithm; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatSignatureHashAlgorithm	**int**	**Since Camel 2.12.2**; signature hash algorithm; will override existing setting directly on the PGPDataFormat.

Name	Type	Description
CamelPGPDataFormatCompressionAlgorithm	**int**	**Since Camel 2.12.2**; compression algorithm; will override existing setting directly on the PGPDataFormat.
CamelPGPDataFormatNumberOfEncryptionKeys	**Integer**	**Since** *Camel 2.12.3; *number of public keys used for encrypting the symmectric key, set by PGPDataFormat during encryptiion process
CamelPGPDataFormatNumberOfSigningKeys	**Integer**	**Since** *Camel 2.12.3; *number of private keys used for creating signatures, set by PGPDataFormat during signing process

250.3. ENCRYPTING WITH PGPDATAFORMAT

The following sample uses the popular PGP format for encrypting/decrypting files using the Bouncy Castle Java libraries:

The following sample performs signing + encryption, and then signature verification + decryption. It uses the same keyring for both signing and encryption, but you can obviously use different keys:

Or using Spring:

250.3.1. To work with the previous example you need the following

- A public keyring file which contains the public keys used to encrypt the data
- A private keyring file which contains the keys used to decrypt the data
- The keyring password

250.3.2. Managing your keyring

To manage the keyring, I use the command line tools, I find this to be the simplest approach in managing the keys. There are also Java libraries available from http://www.bouncycastle.org/java.html if you would prefer to do it that way.

Install the command line utilities on linux

```
apt-get install gnupg
```

Create your keyring, entering a secure password

```
gpg --gen-key
```

If you need to import someone elses public key so that you can encrypt a file for them.

```
gpg --import <filename.key
```

The following files should now exist and can be used to run the example

```
ls -l ~/.gnupg/pubring.gpg ~/.gnupg/secring.gpg
```

[[Crypto-PGPDecrypting/VerifyingofMessagesEncrypted/SignedbyDifferentPrivate/PublicKeys]] PGP Decrypting/Verifying of Messages Encrypted/Signed by Different # Private/Public Keys

Since **Camel 2.12.2**.

A PGP Data Formater can decrypt/verify messages which have been encrypted by different public keys or signed by different private keys. Just, provide the corresponding private keys in the secret keyring, the corresponding public keys in the public keyring, and the passphrases in the passphrase accessor.

```
Map<String, String> userId2Passphrase = new HashMap<String, String>(2);
// add passphrases of several private keys whose corresponding public keys have been used to
encrypt the messages
userId2Passphrase.put("UserIdOfKey1","passphrase1"); // you must specify the exact User ID!
userId2Passphrase.put("UserIdOfKey2","passphrase2");
PGPPassphraseAccessor passphraseAccessor = new
PGPPassphraseAccessorDefault(userId2Passphrase);

PGPDataFormat pgpVerifyAndDecrypt = new PGPDataFormat();
pgpVerifyAndDecrypt.setPassphraseAccessor(passphraseAccessor);
// the method getSecKeyRing() provides the secret keyring as byte array containing the private keys
pgpVerifyAndDecrypt.setEncryptionKeyRing(getSecKeyRing()); // alternatively you can use
setKeyFileName(keyfileName)
// the method getPublicKeyRing() provides the public keyring as byte array containing the public keys
pgpVerifyAndDecrypt.setSignatureKeyRing((getPublicKeyRing()); // alternatively you can use
setSignatureKeyFileName(signatgureKeyfileName)
// it is not necessary to specify the encryption or signer  User Id

from("direct:start")
     ...
    .unmarshal(pgpVerifyAndDecrypt) // can decrypt/verify messages encrypted/signed by different
private/public keys
    ...
```

- The functionality is especially useful to support the key exchange. If you want to exchange the private key for decrypting you can accept for a period of time messages which are either encrypted with the old or new corresponding public key. Or if the sender wants to exchange his signer private key, you can accept for a period of time, the old or new signer key.

- Technical background: The PGP encrypted data contains a Key ID of the public key which was used to encrypt the data. This Key ID can be used to locate the private key in the secret keyring to decrypt the data. The same mechanism is also used to locate the public key for verifying a signature. Therefore you no longer must specify User IDs for the unmarshaling.

250.4. RESTRICTING THE SIGNER IDENTITIES DURING PGP SIGNATURE VERIFICATION

Since **Camel 2.12.3.**

If you verify a signature you not only want to verify the correctness of the signature but you also want check that the signature comes from a certain identity or a specific set of identities. Therefore it is possible to restrict the number of public keys from the public keyring which can be used for the verification of a signature.

Signature User IDs

```
// specify the User IDs of the expected signer identities
List<String> expectedSigUserIds = new ArrayList<String>();
expectedSigUserIds.add("Trusted company1");
expectedSigUserIds.add("Trusted company2");

PGPDataFormat pgpVerifyWithSpecificKeysAndDecrypt = new PGPDataFormat();
pgpVerifyWithSpecificKeysAndDecrypt.setPassword("my password"); // for decrypting with private key
pgpVerifyWithSpecificKeysAndDecrypt.setKeyFileName(keyfileName);
pgpVerifyWithSpecificKeysAndDecrypt.setSignatureKeyFileName(signatgureKeyfileName);
pgpVerifyWithSpecificKeysAndDecrypt.setSignatureKeyUserids(expectedSigUserIds); // if you have only one signer identity then you can also use setSignatureKeyUserid("expected Signer")

from("direct:start")
    ...
    .unmarshal(pgpVerifyWithSpecificKeysAndDecrypt)
    ...
```

- If the PGP content has several signatures the verification is successful as soon as one signature can be verified.

- If you do not want to restrict the signer identities for verification then do not specify the signature key User IDs. In this case all public keys in the public keyring are taken into account.

250.5. SEVERAL SIGNATURES IN ONE PGP DATA FORMAT

Since **Camel 2.12.3.**

The PGP specification allows that one PGP data format can contain several signatures from different keys. Since Camel 2.13.3 it is possible to create such kind of PGP content via specifying signature User IDs which relate to several private keys in the secret keyring.

Several Signatures

```
PGPDataFormat pgpSignAndEncryptSeveralSignerKeys = new PGPDataFormat();
pgpSignAndEncryptSeveralSignerKeys.setKeyUserid(keyUserid); // for encrypting, you can also use setKeyUserids if you want to encrypt with several keys
```

```
pgpSignAndEncryptSeveralSignerKeys.setKeyFileName(keyfileName);
pgpSignAndEncryptSeveralSignerKeys.setSignatureKeyFileName(signatgureKeyfileName);
pgpSignAndEncryptSeveralSignerKeys.setSignaturePassword("sdude"); // here we assume that all
private keys have the same password, if this is not the case then you can use
setPassphraseAccessor

List<String> signerUserIds = new ArrayList<String>();
signerUserIds.add("company old key");
signerUserIds.add("company new key");
pgpSignAndEncryptSeveralSignerKeys.setSignatureKeyUserids(signerUserIds);

from("direct:start")
        ...
       .marshal(pgpSignAndEncryptSeveralSignerKeys)
       ...
```

250.6. SUPPORT OF SUB-KEYS AND KEY FLAGS IN PGP DATA FORMAT MARSHALER

Since *Camel 2.12.3.
*An OpenPGP V4 key can have a primary key and sub-keys. The usage of the keys is indicated by the so called Key Flags. For example, you can have a primary key with two sub-keys; the primary key shall only be used for certifying other keys (Key Flag 0x01), the first sub-key shall only be used for signing (Key Flag 0x02), and the second sub-key shall only be used for encryption (Key Flag 0x04 or 0x08). The PGP Data Format marshaler takes into account these Key Flags of the primary key and sub-keys in order to determine the right key for signing and encryption. This is necessary because the primary key and its sub-keys have the same User IDs.

250.7. SUPPORT OF CUSTOM KEY ACCESSORS

Since *Camel 2.13.0.
*You can implement custom key accessors for encryption/signing. The above PGPDataFormat class selects in a certain predefined way the keys which should be used for signing/encryption or verifying/decryption. If you have special requirements how your keys should be selected you should use the PGPKeyAccessDataFormat class instead and implement the interfaces PGPPublicKeyAccessor and PGPSecretKeyAccessor as beans. There are default implementations DefaultPGPPublicKeyAccessor and DefaultPGPSecretKeyAccessor which cache the keys, so that not every time the keyring is parsed when the processor is called.

PGPKeyAccessDataFormat has the same options as PGPDataFormat except password, keyFileName, encryptionKeyRing, signaturePassword, signatureKeyFileName, and signatureKeyRing.

250.8. DEPENDENCIES

To use the PGP dataformat in your camel routes you need to add the following dependency to your pom.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-crypto</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

250.9. SEE ALSO

- Data Format
- Crypto (Digital Signatures)
- http://www.bouncycastle.org/java.html

CHAPTER 251. PROPERTIES COMPONENT

Available as of Camel version 2.3

251.1. URI FORMAT

```
properties:key[?options]
```

Where **key** is the key for the property to lookup

251.2. OPTIONS

The Properties component supports 17 options which are listed below.

Name	Description	Default	Type
locations (common)	A list of locations to load properties. This option will override any default locations and only use the locations from this option.		List
location (common)	A list of locations to load properties. You can use comma to separate multiple locations. This option will override any default locations and only use the locations from this option.		String
encoding (common)	Encoding to use when loading properties file from the file system or classpath. If no encoding has been set, then the properties files is loaded using ISO-8859-1 encoding (latin-1) as documented by link java.util.Propertiesload(java.io.InputStream)		String
propertiesResolver (common)	To use a custom PropertiesResolver		PropertiesResolver
propertiesParser (common)	To use a custom PropertiesParser		PropertiesParser
cache (common)	Whether or not to cache loaded properties. The default value is true.	true	boolean
propertyPrefix (advanced)	Optional prefix prepended to property names before resolution.		String
propertySuffix (advanced)	Optional suffix appended to property names before resolution.		String

Name	Description	Default	Type
fallbackToUnaugmented Property (advanced)	If true, first attempt resolution of property name augmented with propertyPrefix and propertySuffix before falling back the plain property name specified. If false, only the augmented property name is searched.	true	boolean
defaultFallbackEnabled (common)	If false, the component does not attempt to find a default for the key by looking after the colon separator.	true	boolean
ignoreMissingLocation (common)	Whether to silently ignore if a location cannot be located, such as a properties file not found.	false	boolean
prefixToken (advanced)	Sets the value of the prefix token used to identify properties to replace. Setting a value of null restores the default token (link link DEFAULT_PREFIX_TOKEN).	{{	String
suffixToken (advanced)	Sets the value of the suffix token used to identify properties to replace. Setting a value of null restores the default token (link link DEFAULT_SUFFIX_TOKEN).	}}	String
initialProperties (advanced)	Sets initial properties which will be used before any locations are resolved.		Properties
overrideProperties (advanced)	Sets a special list of override properties that take precedence and will use first, if a property exist.		Properties
systemPropertiesMode (common)	Sets the system property mode.	2	int
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Properties endpoint is configured using URI syntax:

```
properties:key
```

with the following path and query parameters:

251.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
key	**Required** Property key to use as placeholder		String

251.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
ignoreMissingLocation (common)	Whether to silently ignore if a location cannot be located, such as a properties file not found.	false	boolean
locations (common)	A list of locations to load properties. You can use comma to separate multiple locations. This option will override any default locations and only use the locations from this option.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

TIP

Resolving property from Java code
You can use the method **resolvePropertyPlaceholders** on the **CamelContext** to resolve a property from any Java code.

251.3. USING PROPERTYPLACEHOLDER

Available as of Camel 2.3

Camel now provides a new **PropertiesComponent** in **camel-core** which allows you to use property placeholders when defining Camel Endpoint URIs.

This works much like you would do if using Spring's **<property-placeholder>** tag. However Spring have a limitation which prevents 3rd party frameworks to leverage Spring property placeholders to the fullest. See more at How do I use Spring Property Placeholder with Camel XML .

TIP

Bridging Spring and Camel property placeholders
From Camel 2.10 onwards, you can bridge the Spring property placeholder with Camel, see further below for more details.

The property placeholder is generally in use when doing:

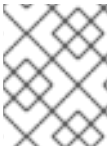
- lookup or creating endpoints
- lookup of beans in the Registry
- additional supported in Spring XML (see below in examples)
- using Blueprint PropertyPlaceholder with Camel Properties component
- using **@PropertyInject** to inject a property in a POJO
- **Camel 2.14.1** Using default value if a property does not exists
- **Camel 2.14.1** Include out of the box functions, to lookup property values from OS environment variables, JVM system properties, or the service idiom.
- **Camel 2.14.1** Using custom functions, which can be plugged into the property component.

251.4. SYNTAX

The syntax to use Camel's property placeholder is to use **{{key}}** for example **{{file.uri}}** where **file.uri** is the property key.

You can use property placeholders in parts of the endpoint URI's which for example you can use placeholders for parameters in the URIs.

From **Camel 2.14.1** onwards you can specify a default value to use if a property with the key does not exists, eg **file.url:/some/path** where the default value is the text after the colon (eg /some/path).



NOTE

Do not use colon in the property key. The colon is used as a separator token when you are providing a default value, which is supported from **Camel 2.14.1** onwards.

251.5. PROPERTYRESOLVER

Camel provides a pluggable mechanism which allows 3rd part to provide their own resolver to lookup properties. Camel provides a default implementation **org.apache.camel.component.properties.DefaultPropertiesResolver** which is capable of loading properties from the file system, classpath or Registry. You can prefix the locations with either:

- **ref: Camel 2.4:** to lookup in the Registry
- **file:** to load the from file system
- **classpath:** to load from classpath (this is also the default if no prefix is provided)
- **blueprint: Camel 2.7:** to use a specific OSGi blueprint placeholder service

251.6. DEFINING LOCATION

The **PropertiesResolver** need to know a location(s) where to resolve the properties. You can define 1 to many locations. If you define the location in a single String property you can separate multiple locations with comma such as:

```
pc.setLocation("com/mycompany/myprop.properties,com/mycompany/other.properties");
```

Available as of Camel 2.19.0

You can set which location can be discarded if missing by by setting the **optional** attribute, which is false by default, i.e:

```
pc.setLocations(
   "com/mycompany/override.properties;optional=true"
   "com/mycompany/defaults.properties");
```

251.7. USING SYSTEM AND ENVIRONMENT VARIABLES IN LOCATIONS

Available as of Camel 2.7

The location now supports using placeholders for JVM system properties and OS environments variables.

For example:

```
location=file:${karaf.home}/etc/foo.properties
```

In the location above we defined a location using the file scheme using the JVM system property with key **karaf.home**.

To use an OS environment variable instead you would have to prefix with env:

```
location=file:${env:APP_HOME}/etc/foo.properties
```

Where **APP_HOME** is an OS environment.

You can have multiple placeholders in the same location, such as:

```
location=file:${env:APP_HOME}/etc/${prop.name}.properties
```

#=== Using system and environment variables to configure property prefixes and suffixes

Available as of Camel 2.12.5, 2.13.3, 2.14.0

propertyPrefix, **propertySuffix** configuration properties support using placeholders for JVM system properties and OS environments variables.

For example. if **PropertiesComponent** is configured with the following properties file:

```
dev.endpoint = result1
test.endpoint = result2
```

Then with the following route definition:

```
PropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class);
pc.setPropertyPrefix("${stage}.");
// ...
context.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").to("properties:mock:{{endpoint}}");
    }
});
```

it is possible to change the target endpoint by changing system property **stage** either to **dev** (the message will be routed to **mock:result1**) or **test** (the message will be routed to **mock:result2**).

251.8. CONFIGURING IN JAVA DSL

You have to create and register the **PropertiesComponent** under the name **properties** such as:

```
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation("classpath:com/mycompany/myprop.properties");
context.addComponent("properties", pc);
```

251.9. CONFIGURING IN SPRING XML

Spring XML offers two variations to configure. You can define a spring bean as a **PropertiesComponent** which resembles the way done in Java DSL. Or you can use the **<propertyPlaceholder>** tag.

```
<bean id="properties" class="org.apache.camel.component.properties.PropertiesComponent">
    <property name="location" value="classpath:com/mycompany/myprop.properties"/>
</bean>
```

Using the **<propertyPlaceholder>** tag makes the configuration a bit more fresh such as:

```
<camelContext ...>
  <propertyPlaceholder id="properties" location="com/mycompany/myprop.properties"/>
</camelContext>
```

Setting the properties location through the location tag works just fine but sometime you have a number of resources to take into account and starting from **Camel 2.19.0** you can set the properties location with a dedicated propertiesLocation:

```
<camelContext ...>
  <propertyPlaceholder id="myPropertyPlaceholder">
```

```
    <propertiesLocation
      resolver = "classpath"
      path     = "com/my/company/something/my-properties-1.properties"
      optional = "false"/>
    <propertiesLocation
      resolver = "classpath"
      path     = "com/my/company/something/my-properties-2.properties"
      optional = "false"/>
    <propertiesLocation
      resolver = "file"
      path     = "${karaf.home}/etc/my-override.properties"
      optional = "true"/>
   </propertyPlaceholder>
</camelContext>
```

TIP

Specifying the cache option inside XML
Camel 2.10 onwards supports specifying a value for the cache option both inside the Spring as well as the Blueprint XML.

251.10. USING A PROPERTIES FROM THE REGISTRY

Available as of Camel 2.4
For example in OSGi you may want to expose a service which returns the properties as a **java.util.Properties** object.

Then you could setup the Properties component as follows:

```
<propertyPlaceholder id="properties" location="ref:myProperties"/>
```

Where **myProperties** is the id to use for lookup in the OSGi registry. Notice we use the **ref:** prefix to tell Camel that it should lookup the properties for the Registry.

251.11. EXAMPLES USING PROPERTIES COMPONENT

When using property placeholders in the endpoint URIs you can either use the **properties:** component or define the placeholders directly in the URI. We will show example of both cases, starting with the former.

```
// properties
cool.end=mock:result

// route
from("direct:start").to("properties:{{cool.end}}");
```

You can also use placeholders as a part of the endpoint uri:

```
// properties
cool.foo=result

// route
from("direct:start").to("properties:mock:{{cool.foo}}");
```

In the example above the to endpoint will be resolved to **mock:result**.

You can also have properties with refer to each other such as:

```
// properties
cool.foo=result
cool.concat=mock:{{cool.foo}}

// route
from("direct:start").to("properties:mock:{{cool.concat}}");
```

Notice how **cool.concat** refer to another property.

The **properties:** component also offers you to override and provide a location in the given uri using the **locations** option:

```
    from("direct:start").to("properties:bar.end?locations=com/mycompany/bar.properties");
```

251.12. EXAMPLES

You can also use property placeholders directly in the endpoint uris without having to use **properties:**.

```
// properties
cool.foo=result

// route
from("direct:start").to("mock:{{cool.foo}}");
```

And you can use them in multiple wherever you want them:

```
// properties
cool.start=direct:start
cool.showid=true
cool.result=result

// route
from("{{cool.start}}")
    .to("log:{{cool.start}}?showBodyType=false&showExchangeId={{cool.showid}}")
    .to("mock:{{cool.result}}");
```

You can also your property placeholders when using ProducerTemplate for example:

```
template.sendBody("{{cool.start}}", "Hello World");
```

251.13. EXAMPLE WITH SIMPLE LANGUAGE

The Simple language now also support using property placeholders, for example in the route below:

```
// properties
cheese.quote=Camel rocks
```

```
// route
from("direct:start")
    .transform().simple("Hi ${body} do you think ${properties:cheese.quote}?");
```

You can also specify the location in the Simple language for example:

```
// bar.properties
bar.quote=Beer tastes good

// route
from("direct:start")
    .transform().simple("Hi ${body}. ${properties:com/mycompany/bar.properties:bar.quote}.");
```

251.14. ADDITIONAL PROPERTY PLACEHOLDER SUPPORTED IN SPRING XML

The property placeholders is also supported in many of the Camel Spring XML tags such as **<package>, <packageScan>, <contextScan>, <jmxAgent>, <endpoint>, <routeBuilder>, <proxy>** and the others.

The example below has property placeholder in the **<jmxAgent>** tag:

You can also define property placeholders in the various attributes on the **<camelContext>** tag such as **trace** as shown here:

251.15. OVERRIDING A PROPERTY SETTING USING A JVM SYSTEM PROPERTY

Available as of Camel 2.5

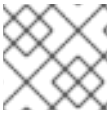
It is possible to override a property value at runtime using a JVM System property without the need to restart the application to pick up the change. This may also be accomplished from the command line by creating a JVM System property of the same name as the property it replaces with a new value. An example of this is given below

```
PropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class);
pc.setCache(false);

System.setProperty("cool.end", "mock:override");
System.setProperty("cool.result", "override");

context.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").to("properties:cool.end");
        from("direct:foo").to("properties:mock:{{cool.result}}");
    }
});
context.start();

getMockEndpoint("mock:override").expectedMessageCount(2);

template.sendBody("direct:start", "Hello World");
template.sendBody("direct:foo", "Hello Foo");
```

```
System.clearProperty("cool.end");
System.clearProperty("cool.result");

assertMockEndpointsSatisfied();
```

251.16. USING PROPERTY PLACEHOLDERS FOR ANY KIND OF ATTRIBUTE IN THE XML DSL

Available as of Camel 2.7



NOTE

If you use OSGi Blueprint then this only works from **2.11.1** or **2.10.5** onwards.

Previously it was only the **xs:string** type attributes in the XML DSL that support placeholders. For example often a timeout attribute would be a **xs:int** type and thus you cannot set a string value as the placeholder key. This is now possible from Camel 2.7 onwards using a special placeholder namespace.

In the example below we use the **prop** prefix for the namespace **http://camel.apache.org/schema/placeholder** by which we can use the **prop** prefix in the attributes in the XML DSLs. Notice how we use that in the Multicast to indicate that the option **stopOnException** should be the value of the placeholder with the key "stop".

In our properties file we have the value defined as

```
stop=true
```

251.17. USING BLUEPRINT PROPERTY PLACEHOLDER WITH CAMEL ROUTES

Available as of Camel 2.7

Camel supports Blueprint which also offers a property placeholder service. Camel supports convention over configuration, so all you have to do is to define the OSGi Blueprint property placeholder in the XML file as shown below:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
           xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

   <!-- OSGI blueprint property placeholder -->
   <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint">
      <!-- list some properties as needed -->
      <cm:default-properties>
         <cm:property name="result" value="mock:result"/>
      </cm:default-properties>
   </cm:property-placeholder>
```

```
    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <!-- in the route we can use {{ }} placeholders which will lookup in blueprint
             as Camel will auto detect the OSGi blueprint property placeholder and use it -->
        <route>
            <from uri="direct:start"/>
            <to uri="mock:foo"/>
            <to uri="{{result}}"/>
        </route>
    </camelContext>
</blueprint>
```

251.17.1. Using OSGi blueprint property placeholders in Camel routes

By default Camel detects and uses OSGi blueprint property placeholder service. You can disable this by setting the attribute **useBlueprintPropertyResolver** to false on the **<camelContext>** definition.

251.17.2. About placeholder syntax

Notice how we can use the Camel syntax for placeholders **{{** and **}}** in the Camel route, which will lookup the value from OSGi blueprint.

The blueprint syntax for placeholders is **${ }**. So outside the **<camelContext>** you must use the **${ }** syntax. Where as inside **<camelContext>** you must use **{{** and **}}** syntax.

OSGi blueprint allows you to configure the syntax, so you can actually align those if you want.

You can also explicit refer to a specific OSGi blueprint property placeholder by its id. For that you need to use the Camel's **<propertyPlaceholder>** as shown in the example below:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
           xsi:schemaLocation="
           http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <!-- OSGI blueprint property placeholder -->
    <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint">
        <!-- list some properties as needed -->
        <cm:default-properties>
            <cm:property name="prefix.result" value="mock:result"/>
        </cm:default-properties>
    </cm:property-placeholder>

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <!-- using Camel properties component and refer to the blueprint property placeholder by its id --
>
        <propertyPlaceholder id="properties" location="blueprint:myblueprint.placeholder"
                             prefixToken="[[" suffixToken="]]"
                             propertyPrefix="prefix."/>

        <!-- in the route we can use {{ }} placeholders which will lookup in blueprint -->
        <route>
            <from uri="direct:start"/>
            <to uri="mock:foo"/>
```

```
        <to uri="[[result]]"/>
      </route>
    </camelContext>
  </blueprint>
```

251.18. EXPLICIT REFERRING TO A OSGI BLUEPRINT PLACEHOLDER IN CAMEL

Notice how we use the **blueprint** scheme to refer to the OSGi blueprint placeholder by its id. This allows you to mix and match, for example you can also have additional schemes in the location. For example to load a file from the classpath you can do:

```
location="blueprint:myblueprint.placeholder,classpath:myproperties.properties"
```

Each location is separated by comma.

251.19. OVERRIDING BLUEPRINT PROPERTY PLACEHOLDERS OUTSIDE CAMELCONTEXT

Available as of Camel 2.10.4

When using Blueprint property placeholder in the Blueprint XML file, you can declare the properties directly in the XML file as shown below:

Notice that we have a **<bean>** which refers to one of the properties. And in the Camel route we refer to the other using the **{{** and **}}** notation.

Now if you want to override these Blueprint properties from an unit test, you can do this as shown below:

To do this we override and implement the **useOverridePropertiesWithConfigAdmin** method. We can then put the properties we want to override on the given props parameter. And the return value **must** be the **persistence-id** of the **<cm:property-placeholder>** tag, which you define in the blueprint XML file.

251.20. USING .CFG OR .PROPERTIES FILE FOR BLUEPRINT PROPERTY PLACEHOLDERS

Available as of Camel 2.10.4

When using Blueprint property placeholder in the Blueprint XML file, you can declare the properties in a **.properties** or **.cfg** file. If you use Apache ServieMix / Karaf then this container has a convention that it loads the properties from a file in the etc directory with the naming **etc/pid.cfg**, where **pid** is the **persistence-id**.

For example in the blueprint XML file we have the **persistence-id="stuff"**, which mean it will load the configuration file as **etc/stuff.cfg**.

Now if you want to unit test this blueprint XML file, then you can override the **loadConfigAdminConfigurationFile** and tell Camel which file to load as shown below:

Notice that this method requires to return a **String[]** with 2 values. The 1st value is the path for the configuration file to load. The 2nd value is the **persistence-id** of the **<cm:property-placeholder>** tag.

The **stuff.cfg** file is just a plain properties file with the property placeholders such as:

```
== this is a comment
greeting=Bye
```

251.21. USING .CFG FILE AND OVERRIDING PROPERTIES FOR BLUEPRINT PROPERTY PLACEHOLDERS

You can do both as well. Here is a complete example. First we have the Blueprint XML file:

And in the unit test class we do as follows:

And the **etc/stuff.cfg** configuration file contains

```
greeting=Bye
echo=Yay
destination=mock:result
```

251.22. BRIDGING SPRING AND CAMEL PROPERTY PLACEHOLDERS

Available as of Camel 2.10

The Spring Framework does not allow 3rd party frameworks such as Apache Camel to seamless hook into the Spring property placeholder mechanism. However you can easily bridge Spring and Camel by declaring a Spring bean with the type **org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer**, which is a Spring **org.springframework.beans.factory.config.PropertyPlaceholderConfigurer** type.

To bridge Spring and Camel you must define a single bean as shown below:

Bridging Spring and Camel property placeholders

You **must not** use the spring <context:property-placeholder> namespace at the same time; this is not possible.

After declaring this bean, you can define property placeholders using both the Spring style, and the Camel style within the <camelContext> tag as shown below:

Using bridge property placeholders

Notice how the hello bean is using pure Spring property placeholders using the **${ }** notation. And in the Camel routes we use the Camel placeholder notation with **{{** and **}}**.

251.23. CLASHING SPRING PROPERTY PLACEHOLDERS WITH CAMELS SIMPLE LANGUAGE

Take notice when using Spring bridging placeholder then the spring **${ }** syntax clashes with the Simple in Camel, and therefore take care. For example:

```
<setHeader headerName="Exchange.FILE_NAME">
  <simple>{{file.rootdir}}/${in.header.CamelFileName}</simple>
</setHeader>
```

clashes with Spring property placeholders, and you should use **$simple{ }** to indicate using the Simple language in Camel.

```
<setHeader headerName="Exchange.FILE_NAME">
  <simple>{{file.rootdir}}/$simple{in.header.CamelFileName}</simple>
</setHeader>
```

An alternative is to configure the **PropertyPlaceholderConfigurer** with **ignoreUnresolvablePlaceholders** option to **true**.

251.24. OVERRIDING PROPERTIES FROM CAMEL TEST KIT

Available as of Camel 2.10

When Testing with Camel and using the Properties component, you may want to be able to provide the properties to be used from directly within the unit test source code.
This is now possible from Camel 2.10 onwards, as the Camel test kits, eg **CamelTestSupport** class offers the following methods

- **useOverridePropertiesWithPropertiesComponent**
- **ignoreMissingLocationWithPropertiesComponent**

So for example in your unit test classes, you can override the **useOverridePropertiesWithPropertiesComponent** method and return a **java.util.Properties** that contains the properties which should be preferred to be used.

251.24.1. Providing properties from within unit test source

This can be done from any of the Camel Test kits, such as camel-test, camel-test-spring, and camel-test-blueprint.

The **ignoreMissingLocationWithPropertiesComponent** can be used to instruct Camel to ignore any locations which was not discoverable, for example if you run the unit test, in an environment that does not have access to the location of the properties.

251.25. USING @PROPERTYINJECT

Available as of Camel 2.12

Camel allows to inject property placeholders in POJOs using the **@PropertyInject** annotation which can be set on fields and setter methods.

For example you can use that with **RouteBuilder** classes, such as shown below:

```
public class MyRouteBuilder extends RouteBuilder {

    @PropertyInject("hello")
    private String greeting;

    @Override
    public void configure() throws Exception {
        from("direct:start")
            .transform().constant(greeting)
            .to("{{result}}");
```

```
    }

}
```

Notice we have annotated the greeting field with **@PropertyInject** and define it to use the key **"hello"**. Camel will then lookup the property with this key and inject its value, converted to a String type.

You can also use multiple placeholders and text in the key, for example we can do:

```
@PropertyInject("Hello {{name}} how are you?")
private String greeting;
```

This will lookup the placeholder with they key **"name"**.

You can also add a default value if the key does not exists, such as:

```
@PropertyInject(value = "myTimeout", defaultValue = "5000")
private int timeout;
```

251.26. USING OUT OF THE BOX FUNCTIONS

Available as of Camel 2.14.1

The Properties component includes the following functions out of the box

- **env** - A function to lookup the property from OS environment variables
- **sys** - A function to lookup the property from Java JVM system properties
- **service** - A function to lookup the property from OS environment variables using the service naming idiom
- **service.name** - **Camel 2.16.1:** A function to lookup the property from OS environment variables using the service naming idiom returning the hostname part only
- **service.port** - **Camel 2.16.1:** A function to lookup the property from OS environment variables using the service naming idiom returning the port part only

As you can see these functions is intended to make it easy to lookup values from the environment. As they are provided out of the box, they can easily be used as shown below:

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">

  <route>
    <from uri="direct:start"/>
    <to uri="{`{env:SOMENAME}`}"/>
    <to uri="{`{sys:MyJvmPropertyName}`}"/>
  </route>
</camelContext>
```

You can use default values as well, so if the property does not exists, you can define a default value as shown below, where the default value is a **log:foo** and **log:bar** value.

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
```

```
  <route>
    <from uri="direct:start"/>
    <to uri="{`{env:SOMENAME:log:foo}`}"/>
    <to uri="{`{sys:MyJvmPropertyName:log:bar}`}"/>
  </route>
</camelContext>
```

The service function is for looking up a service which is defined using OS environment variables using the service naming idiom, to refer to a service location using **hostname : port**

- *NAME*_SERVICE_HOST
- *NAME*_SERVICE_PORT

in other words the service uses **_SERVICE_HOST** and **_SERVICE_PORT** as prefix. So if the service is named FOO, then the OS environment variables should be set as

```
export $FOO_SERVICE_HOST=myserver
export $FOO_SERVICE_PORT=8888
```

For example if the FOO service a remote HTTP service, then we can refer to the service in the Camel endpoint uri, and use the HTTP component to make the HTTP call:

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <route>
    <from uri="direct:start"/>
    <to uri="http://{`{service:FOO}`}/myapp"/>
  </route>
</camelContext>
```

And we can use default values if the service has not been defined, for example to call a service on localhost, maybe for unit testing etc

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <route>
    <from uri="direct:start"/>
    <to uri="http://{`{service:FOO:localhost:8080}`}/myapp"/>
  </route>
</camelContext>
```

251.27. USING CUSTOM FUNCTIONS

Available as of Camel 2.14.1

The Properties component allow to plugin 3rd party functions which can be used during parsing of the property placeholders. These functions are then able to do custom logic to resolve the placeholders, such as looking up in databases, do custom computations, or whatnot. The name of the function becomes the prefix used in the placeholder. This is best illustrated in the example code below

```
<bean id="beerFunction" class="MyBeerFunction"/>
```

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <propertyPlaceholder id="properties">
    <propertiesFunction ref="beerFunction"/>
  </propertyPlaceholder>

  <route>
    <from uri="direct:start"/>
    <to uri="{`{beer:FOO}`}"/>
    <to uri="{`{beer:BAR}`}"/>
  </route>
</camelContext>
```

NOTE

from **camel 2.19.0** the location attribute (on propertyPlaceholder tag) is not more mandatory

Here we have a Camel XML route where we have defined the **<propertyPlaceholder>** to use a custom function, which we refer to be the bean id - eg the **beerFunction**. As the beer function uses **"beer"** as its name, then the placeholder syntax can trigger the beer function by starting with **beer:value**.

The implementation of the function is only two methods as shown below:

```
public static final class MyBeerFunction implements PropertiesFunction {

    @Override
    public String getName() {
        return "beer";
    }

    @Override
    public String apply(String remainder) {
        return "mock:" + remainder.toLowerCase();
    }
}
```

The function must implement the **org.apache.camel.component.properties.PropertiesFunction** interface. The method **getName** is the name of the function, eg beer. And the **apply** method is where we implement the custom logic to do. As the sample code is from an unit test, it just returns a value to refer to a mock endpoint.

To register a custom function from Java code is as shown below:

```
PropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class);
pc.addFunction(new MyBeerFunction());
```

251.28. SEE ALSO

- Properties component
- Jasypt for using encrypted values (eg passwords) in the properties

CHAPTER 252. PROTOBUF DATAFORMAT

Available as of Camel version 2.2.0

CHAPTER 253. PROTOBUF - PROTOCOL BUFFERS

"Protocol Buffers - Google's data interchange format"

Camel provides a Data Format to serialize between Java and the Protocol Buffer protocol. The project's site details why you may wish to choose this format over xml . Protocol Buffer is language-neutral and platform-neutral, so messages produced by your Camel routes may be consumed by other language implementations.

API Site
Protobuf Implementation

Protobuf Java Tutorial

253.1. PROTOBUF OPTIONS

The Protobuf dataformat supports 3 options which are listed below.

Name	Default	Java Type	Description
instanceClass		**String**	Name of class to use when unarmshalling
contentTypeFormat	**native**	**String**	Defines a content type format in which protobuf message will be serialized/deserialized from(to) the Java been. The format can either be native or json for either native protobuf or json fields representation. The default value is native.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

253.2. CONTENT TYPE FORMAT (STARTING FROM CAMEL 2.19)

It's possible to parse JSON message to convert it to the protobuf format and unparse it back using native util converter. To use this option, set contentTypeFormat value to 'json' or call protobuf with second parameter. If default instance is not specified, always use native protobuf format. The sample code shows below:

```
from("direct:marshal")
    .unmarshal()
    .protobuf("org.apache.camel.dataformat.protobuf.generated.AddressBookProtos$Person", "json")
    .to("mock:reverse");
```

253.3. PROTOBUF OVERVIEW

This quick overview of how to use Protobuf. For more detail see the complete tutorial

253.4. DEFINING THE PROTO FORMAT

The first step is to define the format for the body of your exchange. This is defined in a .proto file as so:

addressbook.proto

```
syntax = "proto2";

package org.apache.camel.component.protobuf;

option java_package = "org.apache.camel.component.protobuf";
option java_outer_classname = "AddressBookProtos";

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

253.5. GENERATING JAVA CLASSES

The Protobuf SDK provides a compiler which will generate the Java classes for the format we defined in our .proto file. If your operating system is supporting by Protobuf Java code generator maven plugin , you can automate protobuf Java code generating by adding following configurations to your pom.xml:

Insert operating system and CPU architecture detection extension inside **<build>** tag of the project pom.xml or set ${os.detected.classifier} parameter manually

```
<extensions>
  <extension>
    <groupId>kr.motd.maven</groupId>
    <artifactId>os-maven-plugin</artifactId>
    <version>1.4.1.Final</version>
  </extension>
</extensions>
```

Insert gRPC and protobuf Java code generator plugin **<plugins>** tag of the project pom.xml

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.5.0</version>
  <extensions>true</extensions>
  <executions>
    <execution>
      <goals>
        <goal>test-compile</goal>
        <goal>compile</goal>
      </goals>
      <configuration>
        <protocArtifact>com.google.protobuf:protoc:${protobuf-
version}:exe:${os.detected.classifier}</protocArtifact>
      </configuration>
    </execution>
  </executions>
</plugin>
```

You can also run the compiler for any additional supported languages you require manually.

protoc --java_out=. ./proto/addressbook.proto

This will generate a single Java class named AddressBookProtos which contains inner classes for Person and AddressBook. Builders are also implemented for you. The generated classes implement com.google.protobuf.Message which is required by the serialization mechanism. For this reason it important that only these classes are used in the body of your exchanges. Camel will throw an exception on route creation if you attempt to tell the Data Format to use a class that does not implement com.google.protobuf.Message. Use the generated builders to translate the data from any of your existing domain classes.

253.6. JAVA DSL

You can use create the ProtobufDataFormat instance and pass it to Camel DataFormat marshal and unmarshal API like this.

```
ProtobufDataFormat format = new ProtobufDataFormat(Person.getDefaultInstance());

from("direct:in").marshal(format);
from("direct:back").unmarshal(format).to("mock:reverse");
```

Or use the DSL protobuf() passing the unmarshal default instance or default instance class name like this.

```
// You don't need to specify the default instance for protobuf marshaling
from("direct:marshal").marshal().protobuf();
from("direct:unmarshalA").unmarshal()
    .protobuf("org.apache.camel.dataformat.protobuf.generated.AddressBookProtos$Person")
    .to("mock:reverse");

from("direct:unmarshalB").unmarshal().protobuf(Person.getDefaultInstance()).to("mock:reverse");
```

253.7. SPRING DSL

The following example shows how to use Protobuf to unmarshal using Spring configuring the protobuf data type

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <unmarshal>
      <protobuf
instanceClass="org.apache.camel.dataformat.protobuf.generated.AddressBookProtos$Person" />
    </unmarshal>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

253.8. DEPENDENCIES

To use Protobuf in your camel routes you need to add the a dependency on **camel-protobuf** which implements this data format.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-protobuf</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

253.9. SEE ALSO

- Camel gRPC component

CHAPTER 254. PUBNUB COMPONENT

Available as of Camel version 2.19

Camel PubNub component can be used to communicate with the PubNub data stream network for connected devices. This component uses pubnub java library.

Use cases includs:

- Chat rooms: Sending and receiving messages
- Locations and Connected cars: dispatching taxi cabs
- Smart sensors: Receiving data from a sensor for data visualizations
- Health: Monitoring heart rate from a patient's wearable device
- Multiplayer gamings
- Interactive media: audience-participating voting system

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-pubnub</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

254.1. URI FORMAT

```
pubnub:channel[?options]
```

Where **channel** is the PubNub channel to publish or subscribe to.

254.2. OPTIONS

The PubNub component has no options.

The PubNub endpoint is configured using URI syntax:

```
pubnub:channel
```

with the following path and query parameters:

254.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
channel	**Required** The channel used for subscribing/publishing events		String

254.2.2. Query Parameters (14 parameters):

Name	Description	Default	Type
uuid (common)	UUID to be used as a device identifier, a default UUID is generated if not passed.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
withPresence (consumer)	Also subscribe to related presence information	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
operation (producer)	The operation to perform. PUBLISH: Default. Send a message to all subscribers of a channel. FIRE: allows the client to send a message to BLOCKS Event Handlers. These messages will go directly to any Event Handlers registered on the channel. HERENOW: Obtain information about the current state of a channel including a list of unique user-ids currently subscribed to the channel and the total occupancy count. WHERENOW: Obtain information about the current list of channels to which a uuid is subscribed to. GETSTATE: Used to get key/value pairs specific to a subscriber uuid. State information is supplied as a JSON object of key/value pairs SETSTATE: Used to set key/value pairs specific to a subscriber uuid GETHISTORY: Fetches historical messages of a channel.		String
pubnub (advanced)	Reference to a Pubnub client in the registry.		PubNub
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
authKey (security)	If Access Manager is utilized, client will use this authKey in all restricted requests.		String
cipherKey (security)	If cipher is passed, all communicatons to/from PubNub will be encrypted.		String
publishKey (security)	The publish key obtained from your PubNub account. Required when publishing messages.		String
secretKey (security)	The secret key used for message signing.		String
secure (security)	Use SSL for secure transmission.	true	boolean
subscribeKey (security)	The subscribe key obtained from your PubNub account. Required when subscribing to channels or listening for presence events		String

254.3. MESSAGE HEADERS WHEN SUBSCRIBING

Name	Description
CamelPubNubTimeToken	The Timestamp for the event.
CamelPubNubChannel	The channel for which the message belongs.

254.4. MESSAGE BODY

The message body can contain any JSON serializable data, including: Objects, Arrays, Ints and Strings. Message data should not contain special Java V4 classes or functions as these will not serialize. String content can include any single-byte or multi-byte UTF-8

Object serialization when sending is done automatically. Just pass the full object as the message payload. PubNub will takes care of object serialization.

When receiving the message body utilize objects provided by the PubNub API.

254.5. EXAMPLES

254.5.1. Publishing events

Default operation when producing. The following snippet publish the event generated by PojoBean to the channel iot.

```
from("timer:mytimer")
    // generate some data as POJO.
    .bean(PojoBean.class)
    .to("pubnub:iot?publishKey=mypublishKey");
```

254.5.2. Fire events aka BLOCKS Event Handlers

See https://www.pubnub.com/blocks-catalog/ for all kind of serverless functions that can be invoked. Example of geolocation lookup

```
from("timer:geotimer")
    .process(exchange -> exchange.getIn().setBody(new Foo("bar", "TEXT")))
    .to("pubnub:eon-maps-geolocation-input?
operation=fire&publishKey=mypubkey&subscribeKey=mysubkey");

from("pubnub:eon-map-geolocation-output?subscribeKey=mysubkey)
    // geolocation output will be logged here
    .log("${body}");
```

254.5.3. Subscribing to events

The following snippet listens for events on the iot channel. If you can add the option withPresens, you will also receive channel Join, Leave asf events.

```
from("pubnub:iot?subscribeKey=mySubscribeKey")
    .log("${body}")
    .to("mock:result");
```

254.5.4. Performing operations

herenow : Obtain information about the current state of a channel including a list of unique user-ids currently subscribed to the channel and the total occupancy count of the channel

```
from("direct:control")
    .to("pubnub:myChannel?
publishKey=mypublishKey&subscribeKey=mySubscribeKey&operation=herenow")
    .to("mock:result");
```

wherenow : Obtain information about the current list of channels to which a uuid is subscribed

```
from("direct:control")
    .to("pubnub:myChannel?
publishKey=mypublishKey&subscribeKey=mySubscribeKey&operation=wherenow&uuid=spyonme")
    .to("mock:result");
```

setstate : Used to set key/value pairs specific to a subscriber uuid.

```
from("direct:control")
    .bean(StateGenerator.class)
    .to("pubnub:myChannel?
publishKey=mypublishKey&subscribeKey=mySubscribeKey&operation=setstate&uuid=myuuid");
```

gethistory : Fetches historical messages of a channel.

```
from("direct:control")
    .to("pubnub:myChannel?
publishKey=mypublishKey&subscribeKey=mySubscribeKey&operation=gethistory");
```

There is a couple of examples in test directory that shows some of the PubNub features. They require a PubNub account, from where you can obtain a publish- and subscribe key.

The example PubNubSensorExample already contains a subscribe key provided by PubNub, so this is ready to run without a account. The example illustrates the PubNub component subscribing to a infinite stream of sensor data.

254.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- RSS

CHAPTER 255. QUARTZ COMPONENT (DEPRECATED)

Available as of Camel version 1.0

The **quartz:** component provides a scheduled delivery of messages using the Quartz Scheduler 1.x. Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).

TIP

If you are using Quartz 2.x then from Camel 2.12 onwards there is a Quartz2 component you should use

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-quartz</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

255.1. URI FORMAT

```
quartz://timerName?options
quartz://groupName/timerName?options
quartz://groupName/timerName?cron=expression
quartz://timerName?cron=expression
```

The component uses either a **CronTrigger** or a **SimpleTrigger**. If no cron expression is provided, the component uses a simple trigger. If no **groupName** is provided, the quartz component uses the **Camel** group name.

You can append query options to the URI in the following format, **?option=value&option=value&…**

255.2. OPTIONS

The Quartz component supports 8 options which are listed below.

Name	Description	Default	Type
factory (advanced)	To use the custom SchedulerFactory which is used to create the Scheduler.		SchedulerFactory
scheduler (advanced)	To use the custom configured Quartz scheduler, instead of creating a new Scheduler.		Scheduler
properties (consumer)	Properties to configure the Quartz scheduler.		Properties

Name	Description	Default	Type
propertiesFile (consumer)	File name of the properties to load from the classpath		String
startDelayedSeconds (scheduler)	Seconds to wait before starting the quartz scheduler.		int
autoStartScheduler (consumer)	Whether or not the scheduler should be auto started. This options is default true	true	boolean
enableJmx (consumer)	Whether to enable Quartz JMX which allows to manage the Quartz scheduler from JMX. This options is default true	true	boolean
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Quartz endpoint is configured using URI syntax:

```
quartz:groupName/timerName
```

with the following path and query parameters:

255.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
groupName	The quartz group name to use. The combination of group name and timer name should be unique.	Camel	String
timerName	**Required** The quartz timer name to use. The combination of group name and timer name should be unique.		String

255.2.2. Query Parameters (13 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
cron (consumer)	Specifies a cron expression to define when to trigger.		String
deleteJob (consumer)	If set to true, then the trigger automatically delete when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.	true	boolean
fireNow (consumer)	Whether to fire the scheduler asap when its started using the simple trigger (this option does not support cron)	false	boolean
pauseJob (consumer)	If set to true, then the trigger automatically pauses when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.	false	boolean
startDelayedSeconds (consumer)	Seconds to wait before starting the quartz scheduler.		int
stateful (consumer)	Uses a Quartz StatefulJob instead of the default job.	false	boolean
usingFixedCamelContextName (consumer)	If it is true, JobDataMap uses the CamelContext name directly to reference the CamelContext, if it is false, JobDataMap uses use the CamelContext management name which could be changed during the deploy time.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
jobParameters (advanced)	To configure additional options on the job.		Map
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
triggerParameters (advanced)	To configure additional options on the trigger.		Map

When using a StatefulJob, the JobDataMap is re-persisted after every execution of the job, thus preserving state for the next execution.

INFO: **Running in OSGi and having multiple bundles with quartz routes**If you run in OSGi such as Apache ServiceMix, or Apache Karaf, and have multiple bundles with Camel routes that start from Quartz endpoints, then make sure if you assign an **id** to the <camelContext> that this id is unique, as this is required by the **QuartzScheduler** in the OSGi container. If you do not set any **id** on <camelContext> then a unique id is auto assigned, and there is no problem.

255.3. CONFIGURING QUARTZ.PROPERTIES FILE

By default Quartz will look for a **quartz.properties** file in the **org/quartz** directory of the classpath. If you are using WAR deployments this means just drop the quartz.properties in **WEB-INF/classes/org/quartz**.

However the Camel Quartz component also allows you to configure properties:

Parameter	Default	Type	Description
properties	**null**	**Properties**	**Camel 2.4**: You can configure a **java.util.Properties** instance.
propertiesFile	**null**	**String**	**Camel 2.4**: File name of the properties to load from the classpath

To do this you can configure this in Spring XML as follows

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
   <property name="propertiesFile" value="com/mycompany/myquartz.properties"/>
</bean>
```

255.4. ENABLING QUARTZ SCHEDULER IN JMX

You need to configure the quartz scheduler properties to enable JMX.
That is typically setting the option **"org.quartz.scheduler.jmx.export"** to a **true** value in the configuration file.

From Camel 2.13 onwards Camel will automatic set this option to true, unless explicit disabled.

255.5. STARTING THE QUARTZ SCHEDULER

This is an example:

```
<bean id="quartz" class="org.apache.camel.component.quartz.QuartzComponent">
   <property name="startDelayedSeconds" value="5"/>
</bean>
```

255.6. CLUSTERING

Available as of Camel 2.4

If you use Quartz in clustered mode, e.g. the **JobStore** is clustered. Then from Camel 2.4 onwards the Quartz component will **not** pause/remove triggers when a node is being stopped/shutdown. This allows the trigger to keep running on the other nodes in the cluster.

Note: When running in clustered node no checking is done to ensure unique job name/group for endpoints.

255.7. MESSAGE HEADERS

Camel adds the getters from the Quartz Execution Context as header values. The following headers are added:
calendar, **fireTime**, **jobDetail**, **jobInstance**, **jobRuntTime**, **mergedJobDataMap**, **nextFireTime**, **previousFireTime**, **refireCount**, **result**, **scheduledFireTime**, **scheduler**, **trigger**, **triggerName**, **triggerGroup**.

The **fireTime** header contains the **java.util.Date** of when the exchange was fired.

255.8. USING CRON TRIGGERS

Quartz supports Cron-like expressions for specifying timers in a handy format. You can use these expressions in the **cron** URI parameter; though to preserve valid URI encoding we allow + to be used instead of spaces. Quartz provides a little tutorial on how to use cron expressions.

For example, the following will fire a message every five minutes starting at 12pm (noon) to 6pm on weekdays:

```
from("quartz://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-
FRI").to("activemq:Totally.Rocks");
```

which is equivalent to using the cron expression

```
0 0/5 12-18 ? * MON-FRI
```

The following table shows the URI character encodings we use to preserve valid URI syntax:

URI Character	Cron character
+	*Space*

255.9. SPECIFYING TIME ZONE

Available as of Camel 2.8.1
The Quartz Scheduler allows you to configure time zone per trigger. For example to use a timezone of your country, then you can do as follows:

```
quartz://groupName/timerName?cron=0+0/5+12-18+?+*+MON-
FRI&trigger.timeZone=Europe/Stockholm
```

The timeZone value is the values accepted by **java.util.TimeZone**.

In Camel 2.8.0 or older versions you would have to provide your custom **String** to **java.util.TimeZone** Type Converter to be able configure this from the endpoint uri.
From Camel 2.8.1 onwards we have included such a Type Converter in the camel-core.

255.10. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Quartz2
- Timer

CHAPTER 256. QUARTZ2 COMPONENT

Available as of Camel version 2.12

The **quartz2:** component provides a scheduled delivery of messages using the Quartz Scheduler 2.x. Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-quartz2</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

NOTE: Quartz 2.x API is not compatible with Quartz 1.x. If you need to remain on old Quartz 1.x, please use the old Quartz component instead.

256.1. URI FORMAT

```
quartz2://timerName?options
quartz2://groupName/timerName?options
quartz2://groupName/timerName?cron=expression
quartz2://timerName?cron=expression
```

The component uses either a **CronTrigger** or a **SimpleTrigger**. If no cron expression is provided, the component uses a simple trigger. If no **groupName** is provided, the quartz component uses the **Camel** group name.

You can append query options to the URI in the following format, **?option=value&option=value&…**

256.2. OPTIONS

The Quartz2 component supports 11 options which are listed below.

Name	Description	Default	Type
autoStartScheduler (scheduler)	Whether or not the scheduler should be auto started. This options is default true	true	boolean
startDelayedSeconds (scheduler)	Seconds to wait before starting the quartz scheduler.		int
prefixJobNameWith EndpointId (consumer)	Whether to prefix the quartz job with the endpoint id. This option is default false.	false	boolean

Name	Description	Default	Type
enableJmx (consumer)	Whether to enable Quartz JMX which allows to manage the Quartz scheduler from JMX. This options is default true	true	boolean
properties (consumer)	Properties to configure the Quartz scheduler.		Properties
propertiesFile (consumer)	File name of the properties to load from the classpath		String
prefixInstanceName (consumer)	Whether to prefix the Quartz Scheduler instance name with the CamelContext name. This is enabled by default, to let each CamelContext use its own Quartz scheduler instance by default. You can set this option to false to reuse Quartz scheduler instances between multiple CamelContext's.	true	boolean
interruptJobsOnShutdown (scheduler)	Whether to interrupt jobs on shutdown which forces the scheduler to shutdown quicker and attempt to interrupt any running jobs. If this is enabled then any running jobs can fail due to being interrupted.	false	boolean
schedulerFactory (advanced)	To use the custom SchedulerFactory which is used to create the Scheduler.		SchedulerFactory
scheduler (advanced)	To use the custom configured Quartz scheduler, instead of creating a new Scheduler.		Scheduler
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Quartz2 endpoint is configured using URI syntax:

```
quartz2:groupName/triggerName
```

with the following path and query parameters:

256.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
groupName	The quartz group name to use. The combination of group name and timer name should be unique.	Camel	String

Name	Description	Default	Type
triggerName	**Required** The quartz timer name to use. The combination of group name and timer name should be unique.		String

256.2.2. Query Parameters (19 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
cron (consumer)	Specifies a cron expression to define when to trigger.		String
deleteJob (consumer)	If set to true, then the trigger automatically delete when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.	true	boolean
durableJob (consumer)	Whether or not the job should remain stored after it is orphaned (no triggers point to it).	false	boolean
pauseJob (consumer)	If set to true, then the trigger automatically pauses when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.	false	boolean
recoverableJob (consumer)	Instructs the scheduler whether or not the job should be re-executed if a 'recovery' or 'fail-over' situation is encountered.	false	boolean
stateful (consumer)	Uses a Quartz PersistJobDataAfterExecution and DisallowConcurrentExecution instead of the default job.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
customCalendar (advanced)	Specifies a custom calendar to avoid specific range of date		Calendar
jobParameters (advanced)	To configure additional options on the job.		Map
prefixJobNameWithEndpointId (advanced)	Whether the job name should be prefixed with endpoint id	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
triggerParameters (advanced)	To configure additional options on the trigger.		Map
usingFixedCamelContextName (advanced)	If it is true, JobDataMap uses the CamelContext name directly to reference the CamelContext, if it is false, JobDataMap uses use the CamelContext management name which could be changed during the deploy time.	false	boolean
autoStartScheduler (scheduler)	Whether or not the scheduler should be auto started.	true	boolean
fireNow (scheduler)	If it is true will fire the trigger when the route is start when using SimpleTrigger.	false	boolean
startDelayedSeconds (scheduler)	Seconds to wait before starting the quartz scheduler.		int
triggerStartDelay (scheduler)	In case of scheduler has already started, we want the trigger start slightly after current time to ensure endpoint is fully started before the job kicks in.	500	long

For example, the following routing rule will fire two timer events to the **mock:results** endpoint:

```
from("quartz2://myGroup/myTimerName?
trigger.repeatInterval=2&trigger.repeatCount=1").routeId("myRoute")
    .to("mock:result");
```

When using **stateful=true**, the JobDataMap is re-persisted after every execution of the job, thus preserving state for the next execution.

INFO: **Running in OSGi and having multiple bundles with quartz routes**If you run in OSGi such as Apache ServiceMix, or Apache Karaf, and have multiple bundles with Camel routes that start from Quartz2 endpoints, then make sure if you assign an **id** to the <camelContext> that this id is unique, as this is required by the **QuartzScheduler** in the OSGi container. If you do not set any **id** on <camelContext> then a unique id is auto assigned, and there is no problem.

256.3. CONFIGURING QUARTZ.PROPERTIES FILE

By default Quartz will look for a **quartz.properties** file in the **org/quartz** directory of the classpath. If you are using WAR deployments this means just drop the quartz.properties in **WEB-INF/classes/org/quartz**.

However the Camel Quartz2 component also allows you to configure properties:

Parameter	Default	Type	Description
properties	**null**	**Properties**	You can configure a **java.util.Properties** instance.
propertiesFile	**null**	**String**	File name of the properties to load from the classpath

To do this you can configure this in Spring XML as follows

```
<bean id="quartz2" class="org.apache.camel.component.quartz2.QuartzComponent">
    <property name="propertiesFile" value="com/mycompany/myquartz.properties"/>
</bean>
```

256.4. ENABLING QUARTZ SCHEDULER IN JMX

You need to configure the quartz scheduler properties to enable JMX.
That is typically setting the option **"org.quartz.scheduler.jmx.export"** to a **true** value in the configuration file.

From Camel 2.13 onwards Camel will automatic set this option to true, unless explicit disabled.

256.5. STARTING THE QUARTZ SCHEDULER

The Quartz2 component offers an option to let the Quartz scheduler be started delayed, or not auto started at all.

This is an example:

```
<bean id="quartz2" class="org.apache.camel.component.quartz2.QuartzComponent">
   <property name="startDelayedSeconds" value="5"/>
</bean>
```

256.6. CLUSTERING

If you use Quartz in clustered mode, e.g. the **JobStore** is clustered. Then the Quartz2 component will **not** pause/remove triggers when a node is being stopped/shutdown. This allows the trigger to keep running on the other nodes in the cluster.

Note: When running in clustered node no checking is done to ensure unique job name/group for endpoints.

256.7. MESSAGE HEADERS

Camel adds the getters from the Quartz Execution Context as header values. The following headers are added:
calendar, **fireTime**, **jobDetail**, **jobInstance**, **jobRuntTime**, **mergedJobDataMap**, **nextFireTime**, **previousFireTime**, **refireCount**, **result**, **scheduledFireTime**, **scheduler**, **trigger**, **triggerName**, **triggerGroup**.

The **fireTime** header contains the **java.util.Date** of when the exchange was fired.

256.8. USING CRON TRIGGERS

Quartz supports Cron-like expressions for specifying timers in a handy format. You can use these expressions in the **cron** URI parameter; though to preserve valid URI encoding we allow + to be used instead of spaces.

For example, the following will fire a message every five minutes starting at 12pm (noon) to 6pm on weekdays:

```
from("quartz2://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-FRI")
   .to("activemq:Totally.Rocks");
```

which is equivalent to using the cron expression

```
0 0/5 12-18 ? * MON-FRI
```

The following table shows the URI character encodings we use to preserve valid URI syntax:

URI Character	Cron character
+	*Space*

256.9. SPECIFYING TIME ZONE

The Quartz Scheduler allows you to configure time zone per trigger. For example to use a timezone of your country, then you can do as follows:

```
quartz2://groupName/timerName?cron=0+0/5+12-18+?+*+MON-
FRI&trigger.timeZone=Europe/Stockholm
```

The timeZone value is the values accepted by **java.util.TimeZone**.

256.10. USING QUARTZSCHEDULEDPOLLCONSUMERSCHEDULER

The Quartz2 component provides a Polling Consumer scheduler which allows to use cron based scheduling for Polling Consumer such as the File and FTP consumers.

For example to use a cron based expression to poll for files every 2nd second, then a Camel route can be define simply as:

```
from("file:inbox?scheduler=quartz2&scheduler.cron=0/2+*+*+*+*+?")
  .to("bean:process");
```

Notice we define the **scheduler=quartz2** to instruct Camel to use the Quartz2 based scheduler. Then we use **scheduler.xxx** options to configure the scheduler. The Quartz2 scheduler requires the cron option to be set.

The following options is supported:

Parameter	Default	Type	Description
quartzScheduler	**null**	**org.quartz.Scheduler**	To use a custom Quartz scheduler. If none configure then the shared scheduler from the Quartz2 component is used.
cron	**null**	**String**	**Mandatory**: To define the cron expression for triggering the polls.
triggerId	**null**	**String**	To specify the trigger id. If none provided then an UUID is generated and used.
triggerGroup	**QuartzScheduledPollConsumerScheduler**	**String**	To specify the trigger group.
timeZone	**Default**	**TimeZone**	The time zone to use for the CRON trigger.

Important: Remember configuring these options from the endpoint URIs must be prefixed with **scheduler.**. For example to configure the trigger id and group:

```
    from("file:inbox?scheduler=quartz2&scheduler.cron=0/2+*+*+*+*+?
&scheduler.triggerId=myId&scheduler.triggerGroup=myGroup")
        .to("bean:process");
```

There is also a CRON scheduler in Spring, so you can use the following as well:

```
    from("file:inbox?scheduler=spring&scheduler.cron=0/2+*+*+*+*+?")
        .to("bean:process");
```

CHAPTER 257. RABBITMQ COMPONENT

Available as of Camel version 2.12

The **rabbitmq:** component allows you produce and consume messages from RabbitMQ instances. Using the RabbitMQ AMQP client, this component offers a pure RabbitMQ approach over the generic AMQP component.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-rabbitmq</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

257.1. URI FORMAT

The old syntax is **deprecated**:

```
rabbitmq://hostname[:port]/exchangeName?[options]
```

Instead the hostname and port is configured on the component level, or can be provided as uri query parameters instead.

The new syntax is:

```
rabbitmq:exchangeName?[options]
```

Where **hostname** is the hostname of the running rabbitmq instance or cluster. Port is optional and if not specified then defaults to the RabbitMQ client default (5672). The exchange name determines which exchange produced messages will sent to. In the case of consumers, the exchange name determines which exchange the queue will bind to.

257.2. OPTIONS

The RabbitMQ component supports 49 options which are listed below.

Name	Description	Default	Type
hostname (common)	The hostname of the running RabbitMQ instance or cluster.		String
portNumber (common)	Port number for the host with the running rabbitmq instance or cluster.	5672	int
username (security)	Username in case of authenticated access	guest	String

Name	Description	Default	Type
password (security)	Password for authenticated access	guest	String
vhost (common)	The vhost for the channel	/	String
addresses (common)	If this option is set, camel-rabbitmq will try to create connection based on the setting of option addresses. The addresses value is a string which looks like server1:12345, server2:12345		String
connectionFactory (common)	To use a custom RabbitMQ connection factory. When this option is set, all connection options (connectionTimeout, requestedChannelMax...) set on URI are not used		ConnectionFactory
threadPoolSize (consumer)	The consumer uses a Thread Pool Executor with a fixed number of threads. This setting allows you to set that number of threads.	10	int
autoDetectConnection Factory (advanced)	Whether to auto-detect looking up RabbitMQ connection factory from the registry. When enabled and a single instance of the connection factory is found then it will be used. An explicit connection factory can be configured on the component or endpoint level which takes precedence.	true	boolean
connectionTimeout (advanced)	Connection timeout	60000	int
requestedChannelMax (advanced)	Connection requested channel max (max number of channels offered)	0	int
requestedFrameMax (advanced)	Connection requested frame max (max size of frame offered)	0	int
requestedHeartbeat (advanced)	Connection requested heartbeat (heart-beat in seconds offered)	60	int
automaticRecovery Enabled (advanced)	Enables connection automatic recovery (uses connection implementation that performs automatic recovery when connection shutdown is not initiated by the application)		Boolean
networkRecoveryInterval (advanced)	Network recovery interval in milliseconds (interval used when recovering from network failure)	5000	Integer

Name	Description	Default	Type
topologyRecoveryEnabled (advanced)	Enables connection topology recovery (should topology recovery be performed)		Boolean
prefetchEnabled (consumer)	Enables the quality of service on the RabbitMQConsumer side. You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time	false	boolean
prefetchSize (consumer)	The maximum amount of content (measured in octets) that the server will deliver, 0 if unlimited. You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time		int
prefetchCount (consumer)	The maximum number of messages that the server will deliver, 0 if unlimited. You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time		int
prefetchGlobal (consumer)	If the settings should be applied to the entire channel rather than each consumer You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time	false	boolean
channelPoolMaxSize (producer)	Get maximum number of opened channel in pool	10	int
channelPoolMaxWait (producer)	Set the maximum number of milliseconds to wait for a channel from the pool	1000	long
requestTimeout (advanced)	Set timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds)	20000	long
requestTimeoutChecker Interval (advanced)	Set requestTimeoutCheckerInterval for inOut exchange	1000	long
transferException (advanced)	When true and an inOut Exchange failed on the consumer side send the caused Exception back in the response	false	boolean
publisher Acknowledgements (producer)	When true, the message will be published with publisher acknowledgements turned on	false	boolean

Name	Description	Default	Type
publisherAcknowledgementsTimeout (producer)	The amount of time in milliseconds to wait for a basic.ack response from RabbitMQ server		long
guaranteedDeliveries (producer)	When true, an exception will be thrown when the message cannot be delivered (basic.return) and the message is marked as mandatory. PublisherAcknowledgement will also be activated in this case. See also publisher acknowledgements - When will messages be confirmed.	false	boolean
mandatory (producer)	This flag tells the server how to react if the message cannot be routed to a queue. If this flag is set, the server will return an unroutable message with a Return method. If this flag is zero, the server silently drops the message. If the header is present rabbitmq.MANDATORY it will override this option.	false	boolean
immediate (producer)	This flag tells the server how to react if the message cannot be routed to a queue consumer immediately. If this flag is set, the server will return an undeliverable message with a Return method. If this flag is zero, the server will queue the message, but with no guarantee that it will ever be consumed. If the header is present rabbitmq.IMMEDIATE it will override this option.	false	boolean
args (advanced)	Specify arguments for configuring the different RabbitMQ concepts, a different prefix is required for each: Exchange: arg.exchange. Queue: arg.queue. Binding: arg.binding. For example to declare a queue with message ttl argument: http://localhost:5672/exchange/queueargs=arg.queue.x-message-ttl=60000		Map
clientProperties (advanced)	Connection client properties (client info used in negotiating with the server)		Map
sslProtocol (security)	Enables SSL on connection, accepted value are true, TLS and 'SSLv3		String
trustManager (security)	Configure SSL trust manager, SSL should be enabled for this option to be effective		TrustManager
autoAck (consumer)	If messages should be auto acknowledged	true	boolean

Name	Description	Default	Type
autoDelete (common)	If it is true, the exchange will be deleted when it is no longer in use	true	boolean
durable (common)	If we are declaring a durable exchange (the exchange will survive a server restart)	true	boolean
exclusive (common)	Exclusive queues may only be accessed by the current connection, and are deleted when that connection closes.	false	boolean
passive (common)	Passive queues depend on the queue already to be available at RabbitMQ.	false	boolean
skipQueueDeclare (common)	If true the producer will not declare and bind a queue. This can be used for directing messages via an existing routing key.	false	boolean
skipQueueBind (common)	If true the queue will not be bound to the exchange after declaring it	false	boolean
skipExchangeDeclare (common)	This can be used if we need to declare the queue but not the exchange	false	boolean
declare (common)	If the option is true, camel declare the exchange and queue name and bind them together. If the option is false, camel won't declare the exchange and queue name on the server.	true	boolean
deadLetterExchange (common)	The name of the dead letter exchange		String
deadLetterQueue (common)	The name of the dead letter queue		String
deadLetterRoutingKey (common)	The routing key for the dead letter exchange		String
deadLetterExchangeType (common)	The type of the dead letter exchange	direct	String
allowNullHeaders (producer)	Allow pass null values to header	false	boolean

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The RabbitMQ endpoint is configured using URI syntax:

```
rabbitmq:exchangeName
```

with the following path and query parameters:

257.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
exchangeName	**Required** The exchange name determines which exchange produced messages will sent to. In the case of consumers, the exchange name determines which exchange the queue will bind to.		String

257.2.2. Query Parameters (61 parameters):

Name	Description	Default	Type
addresses (common)	If this option is set, camel-rabbitmq will try to create connection based on the setting of option addresses. The addresses value is a string which looks like server1:12345, server2:12345		Address[]
autoDelete (common)	If it is true, the exchange will be deleted when it is no longer in use	true	boolean
connectionFactory (common)	To use a custom RabbitMQ connection factory. When this option is set, all connection options (connectionTimeout, requestedChannelMax...) set on URI are not used		ConnectionFactory
deadLetterExchange (common)	The name of the dead letter exchange		String
deadLetterExchangeType (common)	The type of the dead letter exchange	direct	String

Name	Description	Default	Type
deadLetterQueue (common)	The name of the dead letter queue		String
deadLetterRoutingKey (common)	The routing key for the dead letter exchange		String
declare (common)	If the option is true, camel declare the exchange and queue name and bind them together. If the option is false, camel won't declare the exchange and queue name on the server.	true	boolean
durable (common)	If we are declaring a durable exchange (the exchange will survive a server restart)	true	boolean
exchangeType (common)	The exchange type such as direct or topic.	direct	String
exclusive (common)	Exclusive queues may only be accessed by the current connection, and are deleted when that connection closes.	false	boolean
hostname (common)	The hostname of the running rabbitmq instance or cluster.		String
passive (common)	Passive queues depend on the queue already to be available at RabbitMQ.	false	boolean
portNumber (common)	Port number for the host with the running rabbitmq instance or cluster. Default value is 5672.		int
queue (common)	The queue to receive messages from		String
routingKey (common)	The routing key to use when binding a consumer queue to the exchange. For producer routing keys, you set the header rabbitmq.ROUTING_KEY.		String
skipExchangeDeclare (common)	This can be used if we need to declare the queue but not the exchange	false	boolean
skipQueueBind (common)	If true the queue will not be bound to the exchange after declaring it	false	boolean
skipQueueDeclare (common)	If true the producer will not declare and bind a queue. This can be used for directing messages via an existing routing key.	false	boolean

Name	Description	Default	Type
vhost (common)	The vhost for the channel	/	String
autoAck (consumer)	If messages should be auto acknowledged	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
concurrentConsumers (consumer)	Number of concurrent consumers when consuming from broker. (eg similar as to the same option for the JMS component).	1	int
prefetchCount (consumer)	The maximum number of messages that the server will deliver, 0 if unlimited. You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time		int
prefetchEnabled (consumer)	Enables the quality of service on the RabbitMQConsumer side. You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time	false	boolean
prefetchGlobal (consumer)	If the settings should be applied to the entire channel rather than each consumer You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time	false	boolean
prefetchSize (consumer)	The maximum amount of content (measured in octets) that the server will deliver, 0 if unlimited. You need to specify the option of prefetchSize, prefetchCount, prefetchGlobal at the same time		int
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
threadPoolSize (consumer)	The consumer uses a Thread Pool Executor with a fixed number of threads. This setting allows you to set that number of threads.	10	int
allowNullHeaders (producer)	Allow pass null values to header	false	boolean
bridgeEndpoint (producer)	If the bridgeEndpoint is true, the producer will ignore the message header of rabbitmq.EXCHANGE_NAME and rabbitmq.ROUTING_KEY	false	boolean
channelPoolMaxSize (producer)	Get maximum number of opened channel in pool	10	int
channelPoolMaxWait (producer)	Set the maximum number of milliseconds to wait for a channel from the pool	1000	long
guaranteedDeliveries (producer)	When true, an exception will be thrown when the message cannot be delivered (basic.return) and the message is marked as mandatory. PublisherAcknowledgement will also be activated in this case. See also publisher acknowledgements - When will messages be confirmed.	false	boolean
immediate (producer)	This flag tells the server how to react if the message cannot be routed to a queue consumer immediately. If this flag is set, the server will return an undeliverable message with a Return method. If this flag is zero, the server will queue the message, but with no guarantee that it will ever be consumed. If the header is present rabbitmq.IMMEDIATE it will override this option.	false	boolean
mandatory (producer)	This flag tells the server how to react if the message cannot be routed to a queue. If this flag is set, the server will return an unroutable message with a Return method. If this flag is zero, the server silently drops the message. If the header is present rabbitmq.MANDATORY it will override this option.	false	boolean
publisherAcknowledgements (producer)	When true, the message will be published with publisher acknowledgements turned on	false	boolean
publisherAcknowledgementsTimeout (producer)	The amount of time in milliseconds to wait for a basic.ack response from RabbitMQ server		long

Name	Description	Default	Type
args (advanced)	Specify arguments for configuring the different RabbitMQ concepts, a different prefix is required for each: Exchange: arg.exchange. Queue: arg.queue. Binding: arg.binding. For example to declare a queue with message ttl argument: http://localhost:5672/exchange/queueargs=arg.queue.x-message-ttl=60000		Map
automaticRecoveryEnabled (advanced)	Enables connection automatic recovery (uses connection implementation that performs automatic recovery when connection shutdown is not initiated by the application)		Boolean
bindingArgs (advanced)	**Deprecated** Key/value args for configuring the queue binding parameters when declare=true		Map
clientProperties (advanced)	Connection client properties (client info used in negotiating with the server)		Map
connectionTimeout (advanced)	Connection timeout	60000	int
exchangeArgs (advanced)	**Deprecated** Key/value args for configuring the exchange parameters when declare=true		Map
exchangeArgsConfigurer (advanced)	**Deprecated** Set the configurer for setting the exchange args in Channel.exchangeDeclare		ArgsConfigurer
networkRecoveryInterval (advanced)	Network recovery interval in milliseconds (interval used when recovering from network failure)	5000	Integer
queueArgs (advanced)	**Deprecated** Key/value args for configuring the queue parameters when declare=true		Map
queueArgsConfigurer (advanced)	**Deprecated** Set the configurer for setting the queue args in Channel.queueDeclare		ArgsConfigurer
requestedChannelMax (advanced)	Connection requested channel max (max number of channels offered)	0	int
requestedFrameMax (advanced)	Connection requested frame max (max size of frame offered)	0	int
requestedHeartbeat (advanced)	Connection requested heartbeat (heart-beat in seconds offered)	60	int

Name	Description	Default	Type
requestTimeout (advanced)	Set timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds)	20000	long
requestTimeoutChecker Interval (advanced)	Set requestTimeoutCheckerInterval for inOut exchange	1000	long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
topologyRecoveryEnabled (advanced)	Enables connection topology recovery (should topology recovery be performed)		Boolean
transferException (advanced)	When true and an inOut Exchange failed on the consumer side send the caused Exception back in the response	false	boolean
password (security)	Password for authenticated access	guest	String
sslProtocol (security)	Enables SSL on connection, accepted value are true, TLS and 'SSLv3		String
trustManager (security)	Configure SSL trust manager, SSL should be enabled for this option to be effective		TrustManager
username (security)	Username in case of authenticated access	guest	String

See http://www.rabbitmq.com/releases/rabbitmq-java-client/current-javadoc/com/rabbitmq/client/ConnectionFactory.html and the AMQP specification for more information on connection options.

257.3. USING CONNECTION FACTORY

To connect to RabbitMQ you can setup a **ConnectionFactory** (same as with JMS) with the login details such as:

```
<bean id="rabbitConnectionFactory" class="com.rabbitmq.client.ConnectionFactory">
  <property name="host" value="localhost"/>
  <property name="port" value="5672"/>
  <property name="username" value="camel"/>
  <property name="password" value="bugsbunny"/>
</bean>
```

> And then refer to the connection factory in the endpoint uri as shown below:
>
> ```
> <camelContext>
>   <route>
>     <from uri="direct:rabbitMQEx2"/>
>     <to uri="rabbitmq:ex2?connectionFactory=#rabbitConnectionFactory"/>
>   </route>
> </camelContext>
> ```

From Camel 2.21 onwards the **ConnectionFactory** is auto-detected by default, so you can just do

```
<camelContext>
  <route>
    <from uri="direct:rabbitMQEx2"/>
    <to uri="rabbitmq:ex2"/>
  </route>
</camelContext>
```

257.4. MESSAGE HEADERS

The following headers are set on exchanges when consuming messages.

Property	Value
rabbitmq.ROUTING_KEY	The routing key that was used to receive the message, or the routing key that will be used when producing a message
rabbitmq.EXCHANGE_NAME	The exchange the message was received from
rabbitmq.DELIVERY_TAG	The rabbitmq delivery tag of the received message
rabbitmq.REDELIVERY_TAG	Whether the message is a redelivered

Property	Value
rabbitmq.REQUEUE	**Camel 2.14.2:** This is used by the consumer to control rejection of the message. When the consumer is complete processing the exchange, and if the exchange failed, then the consumer is going to reject the message from the RabbitMQ broker. The value of this header controls this behavior. If the value is false (by default) then the message is discarded/dead-lettered. If the value is true, then the message is re-queued.

The following headers are used by the producer. If these are set on the camel exchange then they will be set on the RabbitMQ message.

Property	Value
rabbitmq.ROUTING_KEY	The routing key that will be used when sending the message
rabbitmq.EXCHANGE_NAME	The exchange the message was received from
rabbitmq.EXCHANGE_OVERRIDE_NAME	**Camel 2.21:** Used for force sending the message to this exchange instead of the endpoint configured name on the producer
rabbitmq.CONTENT_TYPE	The contentType to set on the RabbitMQ message
rabbitmq.PRIORITY	The priority header to set on the RabbitMQ message

Property	Value
rabbitmq.CORRELATIONID	The correlationId to set on the RabbitMQ message
rabbitmq.MESSAGE_ID	The message id to set on the RabbitMQ message
rabbitmq.DELIVERY_MODE	If the message should be persistent or not
rabbitmq.USERID	The userId to set on the RabbitMQ message
rabbitmq.CLUSTERID	The clusterId to set on the RabbitMQ message
rabbitmq.REPLY_TO	The replyTo to set on the RabbitMQ message
rabbitmq.CONTENT_ENCODING	The contentEncoding to set on the RabbitMQ message
rabbitmq.TYPE	The type to set on the RabbitMQ message
rabbitmq.EXPIRATION	The expiration to set on the RabbitMQ message

Property	Value
rabbitmq.TIMESTAMP	The timestamp to set on the RabbitMQ message
rabbitmq.APP_ID	The appId to set on the RabbitMQ message

Headers are set by the consumer once the message is received. The producer will also set the headers for downstream processors once the exchange has taken place. Any headers set prior to production that the producer sets will be overriden.

257.5. MESSAGE BODY

The component will use the camel exchange in body as the rabbit mq message body. The camel exchange in object must be convertible to a byte array. Otherwise the producer will throw an exception of unsupported body type.

257.6. SAMPLES

To receive messages from a queue that is bound to an exchange A with the routing key B,

```
from("rabbitmq:A?routingKey=B")
```

To receive messages from a queue with a single thread with auto acknowledge disabled.

```
from("rabbitmq:A?routingKey=B&threadPoolSize=1&autoAck=false")
```

To send messages to an exchange called C

```
to("rabbitmq:C")
```

Declaring a headers exchange and queue

```
from("rabbitmq:ex?exchangeType=headers&queue=q&bindingArgs=#bindArgs")
```

and place corresponding **Map<String, Object>** with the id of "bindArgs" in the Registry.

For example declaring a method in spring

```
@Bean(name="bindArgs")
public Map<String, Object> bindArgsBuilder() {
    return Collections.singletonMap("foo", "bar");
}
```

257.6.1. Issue when routing between exchanges (in Camel 2.20.x or older)

If you for example want to route messages from one Rabbit exchange to another as shown in the example below with foo → bar:

```
from("rabbitmq:foo")
  .to("rabbitmq:bar")
```

Then beware that Camel will route the message to itself, eg foo → foo. So why is that? This is because the consumer that receives the message (eg from) provides the message header **rabbitmq.EXCHANGE_NAME** with the name of the exchange, eg **foo**. And when the Camel producer is sending the message to **bar** then the header **rabbitmq.EXCHANGE_NAME** will override this and instead send the message to **foo**.

To avoid this you need to either:

- Remove the header:

```
from("rabbitmq:foo")
  .removeHeader("rabbitmq.EXCHANGE_NAME")
  .to("rabbitmq:bar")
```

- Or turn on **bridgeEndpoint** mode on the producer:

```
from("rabbitmq:foo")
  .to("rabbitmq:bar?bridgeEndpoint=true")
```

From Camel 2.21 onwards this has been improved so you can easily route between exchanges. The header **rabbitmq.EXCHANGE_NAME** is not longer used by the producer to override the destination exchange. Instead a new header **rabbitmq.EXCHANGE_OVERRIDE_NAME** can be used to send to a different exchange. For example to send to cheese exchange you can do

```
from("rabbitmq:foo")
  .setHeader("rabbitmq.EXCHANGE_OVERRIDE_NAME", constant("cheese"))
  .to("rabbitmq:bar")
```

CHAPTER 258. REACTIVE STREAMS COMPONENT

Available as of Camel version 2.19

The **reactive-streams:** component allows you to exchange messages with reactive stream processing libraries compatible with the reactive streams standard.

The component supports backpressure and has been tested using the reactive streams technology compatibility kit (TCK).

The Camel module provides a **reactive-streams** component that allows users to define incoming and outgoing streams within Camel routes, and a direct client API that allows using Camel endpoints directly into any external reactive framework.

Camel uses an internal implementation of the reactive streams *Publisher* and *Subscriber*, so it's not tied to any specific framework. The following reactive frameworks have been used in the integration tests: Reactor Core 3, RxJava 2.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-reactive-streams</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

258.1. URI FORMAT

```
reactive-streams://stream?[options]
```

Where **stream** is a logical stream name used to bind Camel routes to the external stream processing systems.

258.2. OPTIONS

The Reactive Streams component supports 4 options which are listed below.

Name	Description	Default	Type
internalEngineConfiguration (advanced)	Configures the internal engine for Reactive Streams.		ReactiveStreamsEngineConfiguration
backpressureStrategy (producer)	The backpressure strategy to use when pushing events to a slow subscriber.	BUFFER	ReactiveStreamsBackpressureStrategy

Name	Description	Default	Type
serviceType (advanced)	Set the type of the underlying reactive streams implementation to use. The implementation is looked up from the registry or using a ServiceLoader, the default implementation is DefaultCamelReactiveStreamsService		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Reactive Streams endpoint is configured using URI syntax:

```
reactive-streams:stream
```

with the following path and query parameters:

258.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
stream	Name of the stream channel used by the endpoint to exchange messages.		String

258.2.2. Query Parameters (10 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
concurrentConsumers (consumer)	Number of threads used to process exchanges in the Camel route.	1	int

Name	Description	Default	Type
exchangesRefillLowWatermark (consumer)	Set the low watermark of requested exchanges to the active subscription as percentage of the maxInflightExchanges. When the number of pending items from the upstream source is lower than the watermark, new items can be requested to the subscription. If set to 0, the subscriber will request items in batches of maxInflightExchanges, only after all items of the previous batch have been processed. If set to 1, the subscriber can request a new item each time an exchange is processed (chatty). Any intermediate value can be used.	0.25	double
forwardOnComplete (consumer)	Determines if onComplete events should be pushed to the Camel route.	false	boolean
forwardOnError (consumer)	Determines if onError events should be pushed to the Camel route. Exceptions will be set as message body.	false	boolean
maxInflightExchanges (consumer)	Maximum number of exchanges concurrently being processed by Camel. This parameter controls backpressure on the stream. Setting a non-positive value will disable backpressure.	128	Integer
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
backpressureStrategy (producer)	The backpressure strategy to use when pushing events to a slow subscriber.		ReactiveStreamsBackpressureStrategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

258.3. USAGE

The library is aimed to support all the communication modes needed by an application to interact with Camel data:

- **Get** data from Camel routes (In-Only from Camel)

- **Send** data to Camel routes (In-Only towards Camel)
- **Request** a transformation to a Camel route (In-Out towards Camel)
- **Process** data flowing from a Camel route using a reactive processing step (In-Out from Camel)

258.4. GETTING DATA FROM CAMEL

In order to subscribe to data flowing from a Camel route, exchanges should be redirected to a named stream, like in the following snippet:

```
from("timer:clock")
.setBody().header(Exchange.TIMER_COUNTER)
.to("reactive-streams:numbers");
```

Routes can also be written using the XML DSL.

In the example, an unbounded stream of numbers is associated to the name **numbers**. The stream can be accessed using the **CamelReactiveStreams** utility class.

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

// Getting a stream of exchanges
Publisher<Exchange> exchanges = camel.fromStream("numbers");

// Getting a stream of Integers (using Camel standard conversion system)
Publisher<Integer> numbers = camel.fromStream("numbers", Integer.class);
```

The stream can be used easily with any reactive streams compatible library. Here is an example of how to use it with RxJava 2 (although any reactive framework can be used to process events).

```
Flowable.fromPublisher(integers)
    .doOnNext(System.out::println)
    .subscribe();
```

The example prints all numbers generated by Camel into **System.out**.

258.4.1. Getting data from Camel using the direct API

For short Camel routes and for users that prefer defining the whole processing flow using functional constructs of the reactive framework (without using the Camel DSL at all), streams can also be defined using Camel URIs.

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

// Get a stream from all the files in a directory
Publisher<String> files = camel.from("file:folder", String.class);

// Use the stream in RxJava2
Flowable.fromPublisher(files)
    .doOnNext(System.out::println)
    .subscribe();
```

258.5. SENDING DATA TO CAMEL

When an external library needs to push events into a Camel route, the Reactive Streams endpoint must be set as consumer.

```
from("reactive-streams:elements")
.to("log:INFO");
```

A handle to the **elements** stream can be obtained from the **CamelReactiveStreams** utility class.

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

Subscriber<String> elements = camel.streamSubscriber("elements", String.class);
```

The subscriber can be used to push events to the Camel route that consumes from the **elements** stream.

Here is an example of how to use it with RxJava 2 (although any reactive framework can be used to publish events).

```
Flowable.interval(1, TimeUnit.SECONDS)
    .map(i -> "Item " + i)
    .subscribe(elements);
```

String items are generated every second by RxJava in the example and they are pushed into the Camel route defined above.

258.5.1. Sending data to Camel using the direct API

Also in this case, the direct API can be used to obtain a Camel subscriber from an endpoint URI.

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

// Send two strings to the "seda:queue" endpoint
Flowable.just("hello", "world")
    .subscribe(camel.subscriber("seda:queue", String.class));
```

258.6. REQUEST A TRANSFORMATION TO CAMEL

Routes defined in some Camel DSL can be used within a reactive stream framework to perform a specific transformation (the same mechanism can be also used to eg. just send data to a *http* endpoint and continue).

The following snippet shows how RxJava functional code can request the task of loading and marshalling files to Camel.

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

// Process files starting from their names
Flowable.just(new File("file1.txt"), new File("file2.txt"))
    .flatMap(file -> camel.toStream("readAndMarshal", String.class))
```

```
    // Camel output will be converted to String
    // other steps
    .subscribe();
```

In order this to work, a route like the following should be defined in the Camel context:

```
from("reactive-streams:readAndMarshal")
.marshal() // ... other details
```

258.6.1. Request a transformation to Camel using the direct API

An alternative approach consists in using the URI endpoints directly in the reactive flow:

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

// Process files starting from their names
Flowable.just(new File("file1.txt"), new File("file2.txt"))
    .flatMap(file -> camel.to("direct:process", String.class))
    // Camel output will be converted to String
    // other steps
    .subscribe();
```

When using the *to()* method instead of the *toStream*, there is no need to define the route using "reactive-streams:" endpoints (although they are used under the hood).

In this case, the Camel transformation can be just:

```
from("direct:process")
.marshal() // ... other details
```

258.7. PROCESS CAMEL DATA INTO THE REACTIVE FRAMEWORK

While a reactive streams *Publisher* allows exchanging data in a unidirectional way, Camel routes often use a in-out exchange pattern (eg. to define REST endpoints and, in general, where a reply is needed for each request).

In these circumstances, users can add a reactive processing step to the flow, to enhance a Camel route or to define the entire transformation using the reactive framework.

For example, given the following route:

```
from("timer:clock")
.setBody().header(Exchange.TIMER_COUNTER)
.to("direct:reactive")
.log("Continue with Camel route... n=${body}");
```

A reactive processing step can be associated to the "direct:reactive" endpoint:

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

camel.process("direct:reactive", Integer.class, items ->
    Flowable.fromPublisher(items) // RxJava2
        .map(n -> -n)); // make every number negative
```

Data flowing in the Camel route will be processed by the external reactive framework then continue the processing flow inside Camel.

This mechanism can also be used to define a In-Out exchange in a completely reactive way.

```
CamelReactiveStreamsService camel = CamelReactiveStreams.get(context);

// requires a rest-capable Camel component
camel.process("rest:get:orders", exchange ->
                Flowable.fromPublisher(exchange)
                        .flatMap(ex -> allOrders())); // retrieve orders asynchronously
```

See Camel examples (**camel-example-reactive-streams**) for details.

258.8. ADVANCED TOPICS

258.8.1. Controlling Backpressure (producer side)

When routing Camel exchanges to an external subscriber, backpressure is handled by an internal buffer that caches exchanges before delivering them. If the subscriber is slower than the exchange rate, the buffer may become too big. In many circumstances this must be avoided.

Considering the following route:

```
from("jms:queue")
.to("reactive-streams:flow");
```

If the JMS queue contains a high number of messages and the Subscriber associated with the **flow** stream is too slow, messages are dequeued from JMS and appended to the buffer, possibly causing a "out of memory" error. To avoid such problems, a **ThrottlingInflightRoutePolicy** can be set in the route.

```
ThrottlingInflightRoutePolicy policy = new ThrottlingInflightRoutePolicy();
policy.setMaxInflightExchanges(10);

from("jms:queue")
.routePolicy(policy)
.to("reactive-streams:flow");
```

The policy limits the maximum number of active exchanges (and so the maximum size of the buffer), keeping it lower than the threshold (**10** in the example). When more than **10** messages are in flight, the route is suspended, waiting for the subscriber to process them.

With this mechanism, the subscriber controls the route suspension/resume automatically, through backpressure. When multiple subscribers are consuming items from the same stream, the slowest one controls the route status automatically.

In other circumstances, eg. when using a **http** consumer, the route suspension makes the http service unavailable, so using the default configuration (no policy, unbounded buffer) should be preferable. Users should try to avoid memory issues by limiting the number of requests to the http service (eg. scaling out).

In contexts where a certain amount of data loss is acceptable, setting a backpressure strategy other than **BUFFER** can be a solution for dealing with fast sources.

```
from("direct:thermostat")
.to("reactive-streams:flow?backpressureStrategy=LATEST");
```

When the **LATEST** backpressure strategy is used, only the last exchange received from the route is kept by the publisher, while older data is discarded (other options are available).

258.8.2. Controlling Backpressure (consumer side)

When Camel consumes items from a reactive-streams publisher, the maximum number of inflight exchanges can be set as endpoint option.

The subscriber associated with the consumer interacts with the publisher to keep the number of messages in the route lower than the threshold.

An example of backpressure-aware route:

```
from("reactive-streams:numbers?maxInflightExchanges=10")
.to("direct:endpoint");
```

The number of items that Camel requests to the source publisher (through the reactive streams backpressure mechanism) is always lower than **10**. Messages are processed by a single thread in the Camel side.

The number of concurrent consumers (threads) can also be set as endpoint option (**concurrentConsumers**). When using 1 consumer (the default), the order of items in the source stream is maintained. When this value is increased, items will be processed concurrently by multiple threads (so not preserving the order).

258.9. CAMEL REACTIVE STREAMS STARTER

A starter module is available to spring-boot users. When using the starter, the **CamelReactiveStreamsService** can be directly injected into Spring components.

To use the starter, add the following to your spring boot pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-reactive-streams-starter</artifactId>
  <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>
```

258.10. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 259. REACTOR COMPONENT

Available as of Camel version 2.20

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-reactor</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 260. REF COMPONENT

Available as of Camel version 1.2

The **ref:** component is used for lookup of existing endpoints bound in the Registry.

260.1. URI FORMAT

```
ref:someName[?options]
```

Where **someName** is the name of an endpoint in the Registry (usually, but not always, the Spring registry). If you are using the Spring registry, **someName** would be the bean ID of an endpoint in the Spring registry.

260.2. REF OPTIONS

The Ref component has no options.

The Ref endpoint is configured using URI syntax:

```
ref:name
```

with the following path and query parameters:

260.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of endpoint to lookup in the registry.		String

260.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

260.3. RUNTIME LOOKUP

This component can be used when you need dynamic discovery of endpoints in the Registry where you can compute the URI at runtime. Then you can look up the endpoint using the following code:

```
// lookup the endpoint
String myEndpointRef = "bigspenderOrder";
Endpoint endpoint = context.getEndpoint("ref:" + myEndpointRef);

Producer producer = endpoint.createProducer();
Exchange exchange = producer.createExchange();
exchange.getIn().setBody(payloadToSend);
// send the exchange
producer.process(exchange);
```

And you could have a list of endpoints defined in the Registry such as:

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
    <endpoint id="normalOrder" uri="activemq:order.slow"/>
    <endpoint id="bigspenderOrder" uri="activemq:order.high"/>
</camelContext>
```

260.4. SAMPLE

In the sample below we use the **ref:** in the URI to reference the endpoint with the spring ID, **endpoint2**:

You could, of course, have used the **ref** attribute instead:

```
<to ref="endpoint2"/>
```

Which is the more common way to write it.

CHAPTER 261. REST COMPONENT

Available as of Camel version 2.14

The rest component allows to define REST endpoints (consumer) using the Rest DSL and plugin to other Camel components as the REST transport.

From Camel 2.18 onwards the rest component can also be used as a client (producer) to call REST services.

261.1. URI FORMAT

```
rest://method:path[:uriTemplate]?[options]
```

261.2. URI OPTIONS

The REST component supports 4 options which are listed below.

Name	Description	Default	Type
componentName (common)	The Camel Rest component to use for the REST transport, such as restlet, spark-rest. If no component has been explicit configured, then Camel will lookup if there is a Camel component that integrates with the Rest DSL, or if a org.apache.camel.spi.RestConsumerFactory (consumer) or org.apache.camel.spi.RestProducerFactory (producer) is registered in the registry. If either one is found, then that is being used.		String
apiDoc (producer)	The swagger api doc resource to use. The resource is loaded from classpath by default and must be in JSon format.		String
host (producer)	Host and port of HTTP service to use (override host in swagger schema)		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The REST endpoint is configured using URI syntax:

```
rest:method:path:uriTemplate
```

with the following path and query parameters:

261.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
method	**Required** HTTP method to use.		String
path	**Required** The base path		String
uriTemplate	The uri template		String

261.2.2. Query Parameters (15 parameters):

Name	Description	Default	Type
componentName (common)	The Camel Rest component to use for the REST transport, such as restlet, spark-rest. If no component has been explicit configured, then Camel will lookup if there is a Camel component that integrates with the Rest DSL, or if a org.apache.camel.spi.RestConsumerFactory is registered in the registry. If either one is found, then that is being used.		String
consumes (common)	Media type such as: 'text/xml', or 'application/json' this REST service accepts. By default we accept all kinds of types.		String
inType (common)	To declare the incoming POJO binding type as a FQN class name		String
outType (common)	To declare the outgoing POJO binding type as a FQN class name		String
produces (common)	Media type such as: 'text/xml', or 'application/json' this REST service returns.		String
routeId (common)	Name of the route this REST services creates		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean

Name	Description	Default	Type
description (consumer)	Human description to document this REST service		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
apiDoc (producer)	The swagger api doc resource to use. The resource is loaded from classpath by default and must be in JSon format.		String
bindingMode (producer)	Configures the binding mode for the producer. If set to anything other than 'off' the producer will try to convert the body of the incoming message from inType to the json or xml, and the response from json or xml to outType.		RestBindingMode
host (producer)	Host and port of HTTP service to use (override host in swagger schema)		String
queryParameters (producer)	Query parameters for the HTTP service to call		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

261.3. SUPPORTED REST COMPONENTS

The following components support rest consumer (Rest DSL):

- camel-coap
- camel-netty-http
- camel-netty4-http
- camel-jetty
- camel-restlet
- camel-servlet
- camel-spark-rest

- camel-undertow

The following components support rest producer:

- camel-http
- camel-http4
- camel-netty4-http
- camel-jetty
- camel-restlet
- camel-undertow

261.4. PATH AND URITEMPLATE SYNTAX

The path and uriTemplate option is defined using a REST syntax where you define the REST context path using support for parameters.

TIP:If no uriTemplate is configured then path option works the same way. It does not matter if you configure only path or if you configure both options. Though configuring both a path and uriTemplate is a more common practice with REST.

The following is a Camel route using a a path only

```
from("rest:get:hello")
  .transform().constant("Bye World");
```

And the following route uses a parameter which is mapped to a Camel header with the key "me".

```
from("rest:get:hello/{me}")
  .transform().simple("Bye ${header.me}");
```

The following examples have configured a base path as "hello" and then have two REST services configured using uriTemplates.

```
from("rest:get:hello:/{me}")
  .transform().simple("Hi ${header.me}");

from("rest:get:hello:/french/{me}")
  .transform().simple("Bonjour ${header.me}");
```

261.5. REST PRODUCER EXAMPLES

You can use the rest component to call REST services like any other Camel component.

For example to call a REST service on using **hello/{me}** you can do

```
from("direct:start")
  .to("rest:get:hello/{me}");
```

And then the dynamic value **{me}** is mapped to Camel message with the same name. So to call this REST service you can send an empty message body and a header as shown:

```
template.sendBodyAndHeader("direct:start", null, "me", "Donald Duck");
```

The Rest producer needs to know the hostname and port of the REST service, which you can configure using the host option as shown:

```
from("direct:start")
  .to("rest:get:hello/{me}?host=myserver:8080/foo");
```

Instead of using the host option, you can configure the host on the **restConfiguration** as shown:

```
restConfiguration().host("myserver:8080/foo");

from("direct:start")
  .to("rest:get:hello/{me}");
```

You can use the **producerComponent** to select which Camel component to use as the HTTP client, for example to use http4 you can do:

```
restConfiguration().host("myserver:8080/foo").producerComponent("http4");

from("direct:start")
  .to("rest:get:hello/{me}");
```

261.6. REST PRODUCER BINDING

The REST producer supports binding using JSon or XML like the rest-dsl does.

For example to use jetty with json binding mode turned on you can configure this in the rest configuration:

```
restConfiguration().component("jetty").host("localhost").port(8080).bindingMode(RestBindingMode.json
);

from("direct:start")
  .to("rest:post:user");
```

Then when calling the REST service using rest producer it will automatic bind any POJOs to json before calling the REST service:

```
  UserPojo user = new UserPojo();
  user.setId(123);
  user.setName("Donald Duck");

  template.sendBody("direct:start", user);
```

In the example above we send a POJO instance **UserPojo** as the message body. And because we have turned on JSon binding in the rest configuration, then the POJO will be marshalled from POJO to JSon before calling the REST service.

However if you want to also perform binding for the response message (eg what the REST service send back as response) you would need to configure the **outType** option to specify what is the classname of the POJO to unmarshal from JSon to POJO.

For example if the REST service returns a JSon payload that binds to **com.foo.MyResponsePojo** you can configure this as shown:

```
restConfiguration().component("jetty").host("localhost").port(8080).bindingMode(RestBindingMode.json
);

  from("direct:start")
    .to("rest:post:user?outType=com.foo.MyResponsePojo");
```

IMPORTANT

You must configure **outType** option if you want POJO binding to happen for the response messages received from calling the REST service.

261.7. MORE EXAMPLES

See Rest DSL which offers more examples and how you can use the Rest DSL to define those in a nicer RESTful way.

There is a **camel-example-servlet-rest-tomcat** example in the Apache Camel distribution, that demonstrates how to use the Rest DSL with SERVLET as transport that can be deployed on Apache Tomcat, or similar web containers.

261.8. SEE ALSO

- Rest DSL
- SERVLET

CHAPTER 262. REST SWAGGER COMPONENT

Available as of Camel version 2.19

The **rest-swagger** configures rest producers from Swagger (Open API) specification document and delegates to a component implementing the *RestProducerFactory* interface. Currently known working components are:

- http
- http4
- netty4-http
- restlet
- jetty
- undertow

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-rest-swagger</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

262.1. URI FORMAT

```
rest-swagger:[specificationPath#]operationId
```

Where **operationId** is the ID of the operation in the Swagger specification, and **specificationPath** is the path to the specification. If the **specificationPath** is not specified it defaults to **swagger.json**. The lookup mechanism uses Camels **ResourceHelper** to load the resource, which means that you can use CLASSPATH resources (**classpath:my-specification.json**), files (**file:/some/path.json**), the web (**http://api.example.com/swagger.json**) or reference a bean (**ref:nameOfBean**) or use a method of a bean (**bean:nameOfBean.methodName**) to get the specification resource, failing that Swagger's own resource loading support.

This component does not act as a HTTP client, it delegates that to another component mentioned above. The lookup mechanism searches for a single component that implements the *RestProducerFactory* interface and uses that. If the CLASSPATH contains more than one, then the property **componentName** should be set to indicate which component to delegate to.

Most of the configuration is taken from the Swagger specification but the option exists to override those by specifying them on the component or on the endpoint. Typically you would just need to override the **host** or **basePath** if those differ from the specification.

NOTE

The **host** parameter should contain the absolute URI containing scheme, hostname and port number, for instance: **https://api.example.com**

With **componentName** you specify what component is used to perform the requests, this named component needs to be present in the Camel context and implement the required *RestProducerFactory* interface – as do the components listed at the top.

If you do not specify the *componentName* at either component or endpoint level, CLASSPATH is searched for a suitable delegate. There should be only one component present on the CLASSPATH that implements the *RestProducerFactory* interface for this to work.

This component's endpoint URI is lenient which means that in addition to message headers you can specify REST operation's parameters as endpoint parameters, these will be constant for all subsequent invocations so it makes sense to use this feature only for parameters that are indeed constant for all invocations – for example API version in path such as **/api/7.3/users/{id}**.

262.2. OPTIONS

The REST Swagger component supports 7 options which are listed below.

Name	Description	Default	Type
basePath (producer)	API basePath, for example /v2. Default is unset, if set overrides the value present in Swagger specification.		String
componentName (producer)	Name of the Camel component that will perform the requests. The compnent must be present in Camel registry and it must implement RestProducerFactory service provider interface. If not set CLASSPATH is searched for single component that implements RestProducerFactory SPI. Can be overriden in endpoint configuration.		String
consumes (producer)	What payload type this component capable of consuming. Could be one type, like application/json or multiple types as application/json, application/xml; q=0.5 according to the RFC7231. This equates to the value of Accept HTTP header. If set overrides any value found in the Swagger specification. Can be overriden in endpoint configuration		String
host (producer)	Scheme hostname and port to direct the HTTP requests to in the form of https://hostname:port. Can be configured at the endpoint, component or in the correspoding REST configuration in the Camel Context. If you give this component a name (e.g. petstore) that REST configuration is consulted first, rest-swagger next, and global configuration last. If set overrides any value found in the Swagger specification, RestConfiguration. Can be overriden in endpoint configuration.		String

Name	Description	Default	Type
produces (producer)	What payload type this component is producing. For example application/json according to the RFC7231. This equates to the value of Content-Type HTTP header. If set overrides any value present in the Swagger specification. Can be overriden in endpoint configuration.		String
specificationUri (producer)	Path to the Swagger specification file. The scheme, host base path are taken from this specification, but these can be overriden with properties on the component or endpoint level. If not given the component tries to load swagger.json resource. Note that the host defined on the component and endpoint of this Component should contain the scheme, hostname and optionally the port in the URI syntax (i.e. https://api.example.com:8080). Can be overriden in endpoint configuration.	swagger.json	URI
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The REST Swagger endpoint is configured using URI syntax:

```
rest-swagger:specificationUri#operationId
```

with the following path and query parameters:

262.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
specificationUri	Path to the Swagger specification file. The scheme, host base path are taken from this specification, but these can be overriden with properties on the component or endpoint level. If not given the component tries to load swagger.json resource. Note that the host defined on the component and endpoint of this Component should contain the scheme, hostname and optionally the port in the URI syntax (i.e. https://api.example.com:8080). Overrides component configuration.	swagger.json	URI

Name	Description	Default	Type
operationId	**Required** ID of the operation from the Swagger specification.		String

262.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
basePath (producer)	API basePath, for example /v2. Default is unset, if set overrides the value present in Swagger specification and in the component configuration.		String
componentName (producer)	Name of the Camel component that will perform the requests. The compnent must be present in Camel registry and it must implement RestProducerFactory service provider interface. If not set CLASSPATH is searched for single component that implements RestProducerFactory SPI. Overrides component configuration.		String
consumes (producer)	What payload type this component capable of consuming. Could be one type, like application/json or multiple types as application/json, application/xml; q=0.5 according to the RFC7231. This equates to the value of Accept HTTP header. If set overrides any value found in the Swagger specification and. in the component configuration		String
host (producer)	Scheme hostname and port to direct the HTTP requests to in the form of https://hostname:port. Can be configured at the endpoint, component or in the correspoding REST configuration in the Camel Context. If you give this component a name (e.g. petstore) that REST configuration is consulted first, rest-swagger next, and global configuration last. If set overrides any value found in the Swagger specification, RestConfiguration. Overrides all other configuration.		String
produces (producer)	What payload type this component is producing. For example application/json according to the RFC7231. This equates to the value of Content-Type HTTP header. If set overrides any value present in the Swagger specification. Overrides all other configuration.		String

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

262.3. EXAMPLE: PETSTORE

Checkout the example in the **camel-example-rest-swagger** project in the **examples** directory.

For example if you wanted to use the *PetStore* provided REST API simply reference the specification URI and desired operation id from the Swagger specification or download the specification and store it as **swagger.json** (in the root) of CLASSPATH that way it will be automaticaly used. Let's use the Undertow component to perform all the requests and Camels excelent support for Spring Boot.

Here are our dependencies defined in Maven POM file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-undertow-starter</artifactId>
</dependency>

<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-rest-swagger-starter</artifactId>
</dependency>
```

Start by defining the *Undertow* component and the *RestSwaggerComponent*:

```
@Bean
public Component petstore(CamelContext camelContext, UndertowComponent undertow) {
    RestSwaggerComponent petstore = new RestSwaggerComponent(camelContext);
    petstore.setSpecificationUri("http://petstore.swagger.io/v2/swagger.json");
    petstore.setDelegate(undertow);

    return petstore;
}
```

NOTE

Support in Camel for Spring Boot will auto create the **UndertowComponent** Spring bean, and you can configure it using **application.properties** (or **application.yml**) using prefix **camel.component.undertow.**. We are defining the **petstore** component here in order to have a named component in the Camel context that we can use to interact with the PetStore REST API, if this is the only **rest-swagger** component used we might configure it in the same manner (using **application.properties**).

Now in our application we can simply use the **ProducerTemplate** to invoke PetStore REST methods:

```
@Autowired
ProducerTemplate template;

String getPetJsonById(int petId) {
    return template.requestBodyAndHeaders("petstore:getPetById", null, "petId", petId);
}
```

CHAPTER 263. RESTLET COMPONENT

Available as of Camel version 2.0

The **Restlet** component provides Restlet based endpoints for consuming and producing RESTful resources.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-restlet</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

263.1. URI FORMAT

```
restlet:restletUrl[?options]
```

Format of restletUrl:

```
protocol://hostname[:port][/resourcePattern]
```

Restlet promotes decoupling of protocol and application concerns. The reference implementation of Restlet Engine supports a number of protocols. However, we have tested the HTTP protocol only. The default port is port 80. We do not automatically switch default port based on the protocol yet.

You can append query options to the URI in the following format, **?option=value&option=value&...**

INFO: It seems Restlet is case sensitive in understanding headers. For example to use content-type, use Content-Type, and for location use Location and so on.

WARNING

We have received a report about drop in performance in camel-restlet in Camel 2.14.0 and 2.14.1. We have reported this to the Restlet team in issue 996. To remedy the issue then from Camel 2.14.2 onwards you can set synchronous=true as option on the endpoint uris, Or set it on the RestletComponent as a global option so all endpoints inherit this option.

263.2. OPTIONS

The Restlet component supports 22 options which are listed below.

Name	Description	Default	Type
controllerDaemon (consumer)	Indicates if the controller thread should be a daemon (not blocking JVM exit).		Boolean
controllerSleepTimeMs (consumer)	Time for the controller thread to sleep between each control.		Integer
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
inboundBufferSize (consumer)	The size of the buffer when reading messages.		Integer
maxConnectionsPerHost (common)	Maximum number of concurrent connections per host (IP address).		Integer
maxThreads (consumer)	Maximum threads that will service requests.		Integer
lowThreads (consumer)	Number of worker threads determining when the connector is considered overloaded.		Integer
maxTotalConnections (common)	Maximum number of concurrent connections in total.		Integer
minThreads (consumer)	Minimum threads waiting to service requests.		Integer
outboundBufferSize (consumer)	The size of the buffer when writing messages.		Integer
persistingConnections (consumer)	Indicates if connections should be kept alive after a call.		Boolean
pipeliningConnections (consumer)	Indicates if pipelining connections are supported.		Boolean
threadMaxIdleTimeMs (consumer)	Time for an idle thread to wait for an operation before being collected.		Integer

Name	Description	Default	Type
useForwardedForHeader (consumer)	Lookup the X-Forwarded-For header supported by popular proxies and caches and uses it to populate the Request.getClientAddresses() method result. This information is only safe for intermediary components within your local network. Other addresses could easily be changed by setting a fake header and should not be trusted for serious security checks.		Boolean
reuseAddress (consumer)	Enable/disable the SO_REUSEADDR socket option. See java.io.ServerSocketreuseAddress property for additional details.		Boolean
maxQueued (consumer)	Maximum number of calls that can be queued if there aren't any worker thread available to service them. If the value is '0', then no queue is used and calls are rejected if no worker thread is immediately available. If the value is '-1', then an unbounded queue is used and calls are never rejected.		Integer
disableStreamCache (consumer)	Determines whether or not the raw input stream from Restlet is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Restlet input stream to support reading it multiple times to ensure Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultRestletBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times.	false	boolean
port (consumer)	To configure the port number for the restlet consumer routes. This allows to configure this once to reuse the same port for these consumers.		int
synchronous (producer)	Whether to use synchronous Restlet Client for the producer. Setting this option to true can yield faster performance as it seems the Restlet synchronous Client works better.		Boolean
enabledConverters (advanced)	A list of converters to enable as full class name or simple class name. All the converters automatically registered are enabled if empty or null		List

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Restlet endpoint is configured using URI syntax:

```
restlet:protocol:host:port/uriPattern
```

with the following path and query parameters:

263.2.1. Path Parameters (4 parameters):

Name	Description	Default	Type
protocol	**Required** The protocol to use which is http or https		String
host	**Required** The hostname of the restlet service		String
port	**Required** The port number of the restlet service	80	int
uriPattern	The resource pattern such as /customer/id		String

263.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
restletMethod (common)	On a producer endpoint, specifies the request method to use. On a consumer endpoint, specifies that the endpoint consumes only restletMethod requests.	GET	Method

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
restletMethods (consumer)	Specify one or more methods separated by commas (e.g. restletMethods=post,put) to be serviced by a restlet consumer endpoint. If both restletMethod and restletMethods options are specified, the restletMethod setting is ignored. The possible methods are: ALL,CONNECT,DELETE,GET,HEAD,OPTIONS,PATCH,POST,PUT,TRACE		String
disableStreamCache (consumer)	Determines whether or not the raw input stream from Restlet is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Restlet input stream to support reading it multiple times to ensure Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultRestletBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
restletUriPatterns (consumer)	**Deprecated** Specify one ore more URI templates to be serviced by a restlet consumer endpoint, using the notation to reference a List in the Camel Registry. If a URI pattern has been defined in the endpoint URI, both the URI pattern defined in the endpoint and the restletUriPatterns option will be honored.		List

Name	Description	Default	Type
connectTimeout (producer)	The Client will give up connection if the connection is timeout, 0 for unlimited wait.	30000	int
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
socketTimeout (producer)	The Client socket receive timeout, 0 for unlimited wait.	30000	int
throwExceptionOnFailure (producer)	Whether to throw exception on a producer failure. If this option is false then the http status code is set as a message header which can be checked if it has an error value.	true	boolean
autoCloseStream (producer)	Whether to auto close the stream representation as response from calling a REST service using the restlet producer. If the response is streaming and the option streamRepresentation is enabled then you may want to auto close the InputStream from the streaming response to ensure the input stream is closed when the Camel Exchange is done being routed. However if you need to read the stream outside a Camel route, you may need to not auto close the stream.	false	boolean
streamRepresentation (producer)	Whether to support stream representation as response from calling a REST service using the restlet producer. If the response is streaming then this option can be enabled to use an java.io.InputStream as the message body on the Camel Message body. If using this option you may want to enable the autoCloseStream option as well to ensure the input stream is closed when the Camel Exchange is done being routed. However if you need to read the stream outside a Camel route, you may need to not auto close the stream.	false	boolean
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
restletBinding (advanced)	To use a custom RestletBinding to bind between Restlet and Camel message.		RestletBinding
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
restletRealm (security)	To configure the security realms of restlet as a map.		Map
sslContextParameters (security)	To configure security using SSLContextParameters.		SSLContextParameters

263.3. MESSAGE HEADERS

Name	Type	Description
Content-Type	**String**	Specifies the content type, which can be set on the OUT message by the application/processor. The value is the **content-type** of the response message. If this header is not set, the content type is based on the object type of the OUT message body. In Camel 2.3 onward, if the Content-Type header is specified in the Camel IN message, the value of the header determine the content type for the Restlet request message. Otherwise, it is defaulted to "application/x-www-form-urlencoded'. Prior to release 2.3, it is not possible to change the request content type default.
CamelAcceptContentType	**String**	**Since Camel 2.9.3, 2.10.0:** The HTTP Accept request header.
CamelHttpMethod	**String**	The HTTP request method. This is set in the IN message header.
CamelHttpQuery	**String**	The query string of the request URI. It is set on the IN message by **DefaultRestletBinding** when the restlet component receives a request.
CamelHttpResponseCode	**String** or **Integer**	The response code can be set on the OUT message by the application/processor. The value is the response code of the response message. If this header is not set, the response code is set by the restlet runtime engine.
CamelHttpUri	**String**	The HTTP request URI. This is set in the IN message header.
CamelRestletLogin	**String**	Login name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by Camel.

Name	Type	Description
CamelRestletPassword	**String**	Password name for basic authentication. It is set on the IN message by the application and gets filtered before the restlet request header by Camel.
CamelRestletRequest	**Request**	Camel 2.8: The **org.restlet.Request** object which holds all request details.
CamelRestletResponse	**Response**	Camel 2.8: The **org.restlet.Response** object. You can use this to create responses using the API from Restlet. See examples below.
org.restlet.*		Attributes of a Restlet message that get propagated to Camel IN headers.
cache-control	**String** or **List<CacheDirective>**	Camel 2.11: User can set the cache-control with the String value or the List of CacheDirective of Restlet from the camel message header.

263.4. MESSAGE BODY

Camel will store the restlet response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so that headers are preserved during routing.

263.5. SAMPLES

263.5.1. Restlet Endpoint with Authentication

The following route starts a **restlet** consumer endpoint that listens for **POST** requests on http://localhost:8080. The processor creates a response that echoes the request body and the value of the **id** header.

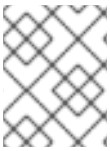
The **restletRealm** setting in the URI query is used to look up a Realm Map in the registry. If this option is specified, the restlet consumer uses the information to authenticate user logins. Only *authenticated* requests can access the resources. In this sample, we create a Spring application context that serves as a registry. The bean ID of the Realm Map should match the *restletRealmRef*.

The following sample starts a **direct** endpoint that sends requests to the server on http://localhost:8080 (that is, our restlet consumer endpoint).

That is all we need. We are ready to send a request and try out the restlet component:

The sample client sends a request to the **direct:start-auth** endpoint with the following headers:

- **CamelRestletLogin** (used internally by Camel)
- **CamelRestletPassword** (used internally by Camel)
- **id** (application header)



NOTE

org.apache.camel.restlet.auth.login and **org.apache.camel.restlet.auth.password** will not be propagated as Restlet header.

The sample client gets a response like the following:

```
received [<order foo='1'/>] as an order id = 89531
```

263.5.2. Single restlet endpoint to service multiple methods and URI templates (deprecated)

This functionality is **deprecated** so do NOT use!

It is possible to create a single route to service multiple HTTP methods using the **restletMethods** option. This snippet also shows how to retrieve the request method from the header:

In addition to servicing multiple methods, the next snippet shows how to create an endpoint that supports multiple URI templates using the **restletUriPatterns** option. The request URI is available in the header of the IN message as well. If a URI pattern has been defined in the endpoint URI (which is not the case in this sample), both the URI pattern defined in the endpoint and the **restletUriPatterns** option will be honored.

The **restletUriPatterns=#uriTemplates** option references the **List<String>** bean defined in the Spring XML configuration.

```
<util:list id="uriTemplates">
   <value>/users/{username}</value>
   <value>/atom/collection/{id}/component/{cid}</value>
</util:list>
```

263.5.3. Using Restlet API to populate response

Available as of Camel 2.8

You may want to use the **org.restlet.Response** API to populate the response. This gives you full access to the Restlet API and fine grained control of the response. See the route snippet below where we generate the response from an inlined Camel Processor:

Generating response using Restlet Response API

263.5.4. Configuring max threads on component

To configure the max threads options you must do this on the component, such as:

```
<bean id="restlet" class="org.apache.camel.component.restlet.RestletComponent">
  <property name="maxThreads" value="100"/>
</bean>
```

263.5.5. Using the Restlet servlet within a webapp

Available as of Camel 2.8
There are three possible ways to configure a Restlet application within a servlet container and using the subclassed SpringServerServlet enables configuration within Camel by injecting the Restlet Component.

Use of the Restlet servlet within a servlet container enables routes to be configured with relative paths in URIs (removing the restrictions of hard-coded absolute URIs) and for the hosting servlet container to handle incoming requests (rather than have to spawn a separate server process on a new port).

To configure, add the following to your camel-context.xml;

```
<camelContext>
  <route id="RS_RestletDemo">
    <from uri="restlet:/demo/{id}" />
    <transform>
      <simple>Request type : ${header.CamelHttpMethod} and ID : ${header.id}</simple>
    </transform>
  </route>
</camelContext>

<bean id="RestletComponent" class="org.restlet.Component" />

<bean id="RestletComponentService"
class="org.apache.camel.component.restlet.RestletComponent">
  <constructor-arg index="0">
    <ref bean="RestletComponent" />
  </constructor-arg>
</bean>
```

And add this to your web.xml;

```
<!-- Restlet Servlet -->
<servlet>
  <servlet-name>RestletServlet</servlet-name>
  <servlet-class>org.restlet.ext.spring.SpringServerServlet</servlet-class>
  <init-param>
    <param-name>org.restlet.component</param-name>
    <param-value>RestletComponent</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>RestletServlet</servlet-name>
  <url-pattern>/rs/*</url-pattern>
</servlet-mapping>
```

You will then be able to access the deployed route at **http://localhost:8080/mywebapp/rs/demo/1234** where;

localhost:8080 is the server and port of your servlet container
mywebapp is the name of your deployed webapp
Your browser will then show the following content;

```
"Request type : GET and ID : 1234"
```

You will need to add dependency on the Spring extension to restlet which you can do in your Maven pom.xml file:

```
<dependency>
  <groupId>org.restlet.jee</groupId>
  <artifactId>org.restlet.ext.spring</artifactId>
  <version>${restlet-version}</version>
</dependency>
```

And you would need to add dependency on the restlet maven repository as well:

```
<repository>
   <id>maven-restlet</id>
   <name>Public online Restlet repository</name>
   <url>http://maven.restlet.org</url>
</repository>
```

CHAPTER 264. RIBBON COMPONENT

Available as of Camel version 2.18

The ribbon component provides use of Netflix Ribbon for client side load balancing.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ribbon</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

This component helps applying load balancing feature at the client side when using ServiceCall EIP.

264.1. CONFIGURATION

- **Programmatic**

```
RibbonConfiguration configuration = new RibbonConfiguration();
configuration.addProperties("ServerListRefreshInterval", "250");

RibbonLoadBalancer loadBalancer = new RibbonLoadBalancer(configuration);

from("direct:start")
    .serviceCall()
        .name("myService")
        .loadBalancer(loadBalancer)
        .consulServiceDiscovery()
        .end()
    .to("mock:result");
```

- **Spring Boot**

  application.properties

```
camel.cloud.ribbon.properties[ServerListRefreshInterval] = 250
```

  routes

```
from("direct:start")
    .serviceCall()
        .name("myService")
        .ribbonLoadBalancer()
        .consulServiceDiscovery()
        .end()
    .to("mock:result");
```

- **XML**

```
<route>
```

```
    <from uri="direct:start"/>
    <serviceCall name="myService">
      <!-- enable ribbon load balancer -->
      <ribbonLoadBalancer>
        <properties key="ServerListRefreshInterval" value="250"/>
      </ribbonLoadBalancer>
    </serviceCall>
</route>
```

264.2. SEE ALSO

- ServiceCall EIP

CHAPTER 265. RMI COMPONENT

Available as of Camel version 1.0

The **rmi:** component binds Exchanges to the RMI protocol (JRMP).

Since this binding is just using RMI, normal RMI rules still apply regarding what methods can be invoked. This component supports only Exchanges that carry a method invocation from an interface that extends the Remote interface. All parameters in the method should be either Serializable or **Remote** objects.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-rmi</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

265.1. URI FORMAT

```
rmi://rmi-regisitry-host:rmi-registry-port/registry-path[?options]
```

For example:

```
rmi://localhost:1099/path/to/service
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

265.2. OPTIONS

The RMI component has no options.

The RMI endpoint is configured using URI syntax:

```
rmi:hostname:port/name
```

with the following path and query parameters:

265.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
hostname	Hostname of RMI server	localhost	String
name	**Required** Name to use when binding to RMI server		String
port	Port number of RMI server	1099	int

265.2.2. Query Parameters (6 parameters):

Name	Description	Default	Type
method (common)	You can set the name of the method to invoke.		String
remoteInterfaces (common)	To specific the remote interfaces.		List
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

265.3. USING

To call out to an existing RMI service registered in an RMI registry, create a route similar to the following:

```
from("pojo:foo").to("rmi://localhost:1099/foo");
```

To bind an existing camel processor or service in an RMI registry, define an RMI endpoint as follows:

```
RmiEndpoint endpoint= (RmiEndpoint) endpoint("rmi://localhost:1099/bar");
endpoint.setRemoteInterfaces(ISay.class);
from(endpoint).to("pojo:bar");
```

Note that when binding an RMI consumer endpoint, you must specify the **Remote** interfaces exposed.

In XML DSL you can do as follows from **Camel 2.7** onwards:

```
<camel:route>
  <from uri="rmi://localhost:37541/helloServiceBean?
```

```
remoteInterfaces=org.apache.camel.example.osgi.HelloService"/>
        <to uri="bean:helloServiceBean"/>
    </camel:route>
```

265.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 266. ROUTEBOX COMPONENT (DEPRECATED)

Available as of Camel version 2.6

Routebox subject for change

The **routebox** component enables the creation of specialized endpoints that offer encapsulation and a strategy based indirection service to a collection of camel routes hosted in an automatically created or user injected camel context.

Routebox endpoints are camel endpoints that may be invoked directly on camel routes. The routebox endpoint performs the following key functions

- encapsulation - acts as a blackbox, hosting a collection of camel routes stored in an inner camel context. The inner context is fully under the control of the routebox component and is **JVM bound**.
- strategy based indirection - direct payloads sent to the routebox endpoint along a camel route to specific inner routes based on a user defined internal routing strategy or a dispatch map.
- exchange propagation - forward exchanges modified by the routebox endpoint to the next segment of the camel route.

The routebox component supports both consumer and producer endpoints.

Producer endpoints are of two flavors

- Producers that send or dispatch incoming requests to a external routebox consumer endpoint
- Producers that directly invoke routes in an internal embedded camel context thereby not sending requests to an external consumer.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-routebox</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

266.1. THE NEED FOR A CAMEL ROUTEBOX ENDPOINT

The routebox component is designed to ease integration in complex environments needing

- a large collection of routes and
- involving a wide set of endpoint technologies needing integration in different ways

In such environments, it is often necessary to craft an integration solution by creating a sense of layering among camel routes effectively organizing them into

- Coarse grained or higher level routes - aggregated collection of inner or lower level routes exposed as Routebox endpoints that represent an integration focus area. For example

Focus Area	Coarse grained Route Examples
Department Focus	HR routes, Sales routes etc
Supply chain & B2B Focus	Shipping routes, Fulfillment routes, 3rd party services etc
Technology Focus	Database routes, JMS routes, Scheduled batch routes etc

- Fine grained routes - routes that execute a singular and specific business and/or integration pattern.

Requests sent to Routebox endpoints on coarse grained routes can then delegate requests to inner fine grained routes to achieve a specific integration objective, collect the final inner result, and continue to progress to the next step along the coarse-grained route.

266.2. URI FORMAT

```
routebox:routeboxname[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

266.3. OPTIONS

The RouteBox component has no options.

The RouteBox endpoint is configured using URI syntax:

```
routebox:routeboxName
```

with the following path and query parameters:

266.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
routeboxName	**Required** Logical name for the routebox (eg like a queue name)		String

266.3.2. Query Parameters (17 parameters):

Name	Description	Default	Type
dispatchMap (common)	A string representing a key in the Camel Registry matching an object value of the type HashMap. The HashMap key should contain strings that can be matched against the value set for the exchange header ROUTE_DISPATCH_KEY. The HashMap value should contain inner route consumer URI's to which requests should be directed.		Map
dispatchStrategy (common)	To use a custom RouteboxDispatchStrategy which allows to use custom dispatching instead of the default.		RouteboxDispatchStrategy
forkContext (common)	Whether to fork and create a new inner CamelContext instead of reusing the same CamelContext.	true	boolean
innerProtocol (common)	The Protocol used internally by the Routebox component. Can be Direct or SEDA. The Routebox component currently offers protocols that are JVM bound.	direct	String
queueSize (common)	Create a fixed size queue to receive requests.		int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollInterval (consumer)	The timeout used when polling from seda. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	long
threads (consumer)	Number of threads to be used by the routebox to receive requests.	20	int
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
connectionTimeout (producer)	Timeout in millis used by the producer when sending a message.	20000	long
sendToConsumer (producer)	Dictates whether a Producer endpoint sends a request to an external routebox consumer. If the setting is false, the Producer creates an embedded inner context and processes requests internally.	true	boolean
innerContext (advanced)	A string representing a key in the Camel Registry matching an object value of the type org.apache.camel.CamelContext. If a CamelContext is not provided by the user a CamelContext is automatically created for deployment of inner routes.		CamelContext
innerProducerTemplate (advanced)	The ProducerTemplate to use by the internal embeded CamelContext		ProducerTemplate
innerRegistry (advanced)	To use a custom registry for the internal embedded CamelContext.		Registry
routeBuilders (advanced)	A string representing a key in the Camel Registry matching an object value of the type List. If the user does not supply an innerContext pre-primed with inner routes, the routeBuilders option must be provided as a non-empty list of RouteBuilders containing inner routes		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

266.4. SENDING/RECEIVING MESSAGES TO/FROM THE ROUTEBOX

Before sending requests it is necessary to properly configure the routebox by loading the required URI parameters into the Registry as shown below. In the case of Spring, if the necessary beans are declared correctly, the registry is automatically populated by Camel.

266.4.1. Step 1: Loading inner route details into the Registry

```
@Override
protected JndiRegistry createRegistry() throws Exception {
    JndiRegistry registry = new JndiRegistry(createJndiContext());

    // Wire the routeDefinitions & dispatchStrategy to the outer camelContext where the routebox is
```

```
declared
    List<RouteBuilder> routes = new ArrayList<RouteBuilder>();
    routes.add(new SimpleRouteBuilder());
    registry.bind("registry", createInnerRegistry());
    registry.bind("routes", routes);

    // Wire a dispatch map to registry
    HashMap<String, String> map = new HashMap<String, String>();
    map.put("addToCatalog", "seda:addToCatalog");
    map.put("findBook", "seda:findBook");
    registry.bind("map", map);

    // Alternatively wiring a dispatch strategy to the registry
    registry.bind("strategy", new SimpleRouteDispatchStrategy());

    return registry;
}

private JndiRegistry createInnerRegistry() throws Exception {
    JndiRegistry innerRegistry = new JndiRegistry(createJndiContext());
    BookCatalog catalogBean = new BookCatalog();
    innerRegistry.bind("library", catalogBean);

    return innerRegistry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

266.4.2. Step 2: Optionaly using a Dispatch Strategy instead of a Dispatch Map

Using a dispatch Strategy involves implementing the interface *org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy* as shown in the example below.

```
public class SimpleRouteDispatchStrategy implements RouteboxDispatchStrategy {

    /* (non-Javadoc)
     * @see
org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy#selectDestinationUri(java.u
il.List, org.apache.camel.Exchange)
     */
    public URI selectDestinationUri(List<URI> activeDestinations,
            Exchange exchange) {
        URI dispatchDestination = null;

        String operation = exchange.getIn().getHeader("ROUTE_DISPATCH_KEY", String.class);
        for (URI destination : activeDestinations) {
            if (destination.toASCIIString().equalsIgnoreCase("seda:" + operation)) {
                dispatchDestination = destination;
                break;
            }
        }
```

```
        return dispatchDestination;
    }
}
```

266.4.3. Step 2: Launching a routebox consumer

When creating a route consumer, note that the # entries in the routeboxUri are matched to the created inner registry, routebuilder list and dispatchStrategy/dispatchMap in the CamelContext Registry. Note that all routebuilders and associated routes are launched in the routebox created inner context

```
private String routeboxUri = "routebox:multipleRoutes?
innerRegistry=#registry&routeBuilders=#routes&dispatchMap=#map";

public void testRouteboxRequests() throws Exception {
    CamelContext context = createCamelContext();
    template = new DefaultProducerTemplate(context);
    template.start();

    context.addRoutes(new RouteBuilder() {
        public void configure() {
            from(routeboxUri)
                .to("log:Routes operation performed?showAll=true");
        }
    });
    context.start();

    // Now use the ProducerTemplate to send the request to the routebox
    template.requestBodyAndHeader(routeboxUri, book, "ROUTE_DISPATCH_KEY",
"addToCatalog");
}
```

266.4.4. Step 3: Using a routebox producer

When sending requests to the routebox, it is not necessary for producers do not need to know the inner route endpoint URI and they can simply invoke the Routebox URI endpoint with a dispatch strategy or dispatchMap as shown below

It is necessary to set a special exchange Header called **ROUTE_DISPATCH_KEY** (optional for Dispatch Strategy) with a key that matches a key in the dispatch map so that the request can be sent to the correct inner route

```
from("direct:sendToStrategyBasedRoutebox")
    .to("routebox:multipleRoutes?
innerRegistry=#registry&routeBuilders=#routes&dispatchStrategy=#strategy")
    .to("log:Routes operation performed?showAll=true");

from ("direct:sendToMapBasedRoutebox")
    .setHeader("ROUTE_DISPATCH_KEY", constant("addToCatalog"))
    .to("routebox:multipleRoutes?
innerRegistry=#registry&routeBuilders=#routes&dispatchMap=#map")
    .to("log:Routes operation performed?showAll=true");
```

CHAPTER 267. RSS COMPONENT

Available as of Camel version 2.0

The **rss:** component is used for polling RSS feeds. Camel will default poll the feed every 60th seconds.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-rss</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Note: The component currently only supports polling (consuming) feeds.

NOTE

Camel-rss internally uses a patched version of ROME hosted on ServiceMix to solve some OSGi class loading issues.

267.1. URI FORMAT

```
rss:rssUri
```

Where **rssUri** is the URI to the RSS feed to poll.

You can append query options to the URI in the following format, **?option=value&option=value&...**

267.2. OPTIONS

The RSS component has no options.

The RSS endpoint is configured using URI syntax:

```
rss:feedUri
```

with the following path and query parameters:

267.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
feedUri	**Required** The URI to the feed to poll.		String

267.2.2. Query Parameters (27 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
feedHeader (consumer)	Sets whether to add the feed object as a header	true	boolean
filter (consumer)	Sets whether to use filtering or not of the entries.	true	boolean
lastUpdate (consumer)	Sets the timestamp to be used for filtering entries from the atom feeds. This options is only in conjunction with the splitEntries.		Date
password (consumer)	Sets the password to be used for basic authentication when polling from a HTTP feed		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
sortEntries (consumer)	Sets whether to sort entries by published date. Only works when splitEntries = true.	false	boolean
splitEntries (consumer)	Sets whether or not entries should be sent individually or whether the entire feed should be sent as a single message	true	boolean
throttleEntries (consumer)	Sets whether all entries identified in a single feed poll should be delivered immediately. If true, only one entry is processed per consumer.delay. Only applicable when splitEntries = true.	true	boolean
username (consumer)	Sets the username to be used for basic authentication when polling from a HTTP feed		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel

Name	Description	Default	Type
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

267.3. EXCHANGE DATA TYPES

Camel initializes the In body on the Exchange with a ROME **SyndFeed**. Depending on the value of the **splitEntries** flag, Camel returns either a **SyndFeed** with one **SyndEntry** or a **java.util.List** of **SyndEntrys**.

Option	Value	Behavior
splitEntries	**true**	A single entry from the current feed is set in the exchange.
splitEntries	**false**	The entire list of entries from the current feed is set in the exchange.

267.4. MESSAGE HEADERS

Header	Description
CamelRssFeed	The entire **SyncFeed** object.

267.5. RSS DATAFORMAT

The RSS component ships with an RSS dataformat that can be used to convert between String (as XML) and ROME RSS model objects.

- marshal = from ROME **SyndFeed** to XML **String**
- unmarshal = from XML **String** to ROME **SyndFeed**

A route using the RSS dataformat will look like this: **from("rss:file:src/test/data/rss20.xml?splitEntries=false&consumer.delay=1000").marshal().rss().to("mock:marshal");**

The purpose of this feature is to make it possible to use Camel's built-in expressions for manipulating RSS messages. As shown below, an XPath expression can be used to filter the RSS message. In the following example, on ly entries with Camel in the title will get through the filter.

```
`from("rss:file:src/test/data/rss20.xml?
splitEntries=true&consumer.delay=100").marshal().rss().filter().xpath("//item/title[contains(.,'Camel')]").to
("mock:result");`
```

TIP

Query parameters If the URL for the RSS feed uses query parameters, this component will resolve them. For example if the feed uses **alt=rss**, then the following example will be resolved: **from("rss:http://someserver.com/feeds/posts/default?alt=rss&splitEntries=false&consumer.delay=1000").to("bean:rss");**

267.6. FILTERING ENTRIES

You can filter out entries using XPath, as shown in the data format section above. You can also exploit Camel's Bean Integration to implement your own conditions. For instance, a filter equivalent to the XPath example above would be:

from("rss:file:src/test/data/rss20.xml?splitEntries=true&consumer.delay=100").filter().method("myFilterBean", "titleContainsCamel").to("mock:result");

The custom bean for this would be:

```
public static class FilterBean {
      public boolean titleContainsCamel(@Body SyndFeed feed) {
         SyndEntry firstEntry = (SyndEntry) feed.getEntries().get(0);
return firstEntry.getTitle().contains("Camel");
```

267.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Atom

CHAPTER 268. RSS DATAFORMAT

Available as of Camel version 2.1

The RSS component ships with an RSS dataformat that can be used to convert between String (as XML) and ROME RSS model objects.

- marshal = from ROME **SyndFeed** to XML **String**
- unmarshal = from XML **String** to ROME **SyndFeed**

A route using this would look something like this:

The purpose of this feature is to make it possible to use Camel's lovely built-in expressions for manipulating RSS messages. As shown below, an XPath expression can be used to filter the RSS message:

TIP

Query parameters If the URL for the RSS feed uses query parameters, this component will understand them as well, for example if the feed uses **alt=rss**, then you can for example do **from("rss:http://someserver.com/feeds/posts/default?alt=rss&splitEntries=false&consumer.delay=1000").to("bean:rss");**

268.1. OPTIONS

The RSS dataformat supports 1 options which are listed below.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

CHAPTER 269. SALESFORCE COMPONENT

Available as of Camel version 2.12

This component supports producer and consumer endpoints to communicate with Salesforce using Java DTOs.
There is a companion maven plugin Camel Salesforce Plugin that generates these DTOs (see further below).

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-salesforce</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

NOTE

Developers wishing to contribute to the component are instructed to look at the README.md file on instructions on how to get started and setup your environment for running integration tests.

269.1. AUTHENTICATING TO SALESFORCE

The component supports three OAuth authentication flows:

- OAuth 2.0 Username-Password Flow
- OAuth 2.0 Refresh Token Flow
- OAuth 2.0 JWT Bearer Token Flow

For each of the flow different set of properties needs to be set:

Table 269.1. Properties to set for each authentication flow

Property	Where to find it on Salesforce	Flow
clientId	Connected App, Consumer Key	All flows
clientSecret	Connected App, Consumer Secret	Username-Password, Refresh Token
userName	Salesforce user username	Username-Password, JWT Bearer Token
password	Salesforce user password	Username-Password
refreshToken	From OAuth flow callback	Refresh Token

Property	Where to find it on Salesforce	Flow
keystore	Connected App, Digital Certificate	JWT Bearer Token

The component auto determines what flow you're trying to configure, to be remove ambiguity set the **authenticationType** property.

NOTE

Using Username-Password Flow in production is not encouraged.

NOTE

The certificate used in JWT Bearer Token Flow can be a selfsigned certificate. The KeyStore holding the certificate and the private key must contain only single certificate-private key entry.

269.2. URI FORMAT

When used as a consumer, receiving streaming events, the URI scheme is:

```
salesforce:topic?options
```

When used as a producer, invoking the Salesforce RSET APIs, the URI scheme is:

```
salesforce:operationName?options
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

269.3. PASSING IN SALESFORCE HEADERS AND FETCHING SALESFORCE RESPONSE HEADERS

With Camel 2.21 there is support to pass Salesforce headers via inbound message headers, header names that start with **Sforce** or **x-sfdc** on the Camel message will be passed on in the request, and response headers that start with **Sforce** will be present in the outboud message headers.

For example to fetch API limits you can specify:

```
// in your Camel route set the header before Salesforce endpoint
//...
  .setHeader("Sforce-Limit-Info", constant("api-usage"))
  .to("salesforce:getGlobalObjects")
  .to(myProcessor);

// myProcessor will receive `Sforce-Limit-Info` header on the outbound
// message
class MyProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        Message in = exchange.getIn();
```

```
        String apiLimits = in.getHeader("Sforce-Limit-Info", String.class);
    }
}
```

269.4. SUPPORTED SALESFORCE APIS

The component supports the following Salesforce APIs

Producer endpoints can use the following APIs. Most of the APIs process one record at a time, the Query API can retrieve multiple Records.

269.4.1. Rest API

You can use the following for **operationName**:

- getVersions - Gets supported Salesforce REST API versions
- getResources - Gets available Salesforce REST Resource endpoints
- getGlobalObjects - Gets metadata for all available SObject types
- getBasicInfo - Gets basic metadata for a specific SObject type
- getDescription - Gets comprehensive metadata for a specific SObject type
- getSObject - Gets an SObject using its Salesforce Id
- createSObject - Creates an SObject
- updateSObject - Updates an SObject using Id
- deleteSObject - Deletes an SObject using Id
- getSObjectWithId - Gets an SObject using an external (user defined) id field
- upsertSObject - Updates or inserts an SObject using an external id
- deleteSObjectWithId - Deletes an SObject using an external id
- query - Runs a Salesforce SOQL query
- queryMore - Retrieves more results (in case of large number of results) using result link returned from the 'query' API
- queryAll - Runs a SOQL query. It returns the results that are deleted because of a merge or delete. Also returns the information about archived Task and Event records.
- search - Runs a Salesforce SOSL query
- limits - fetching organization API usage limits
- recent - fetching recent items
- approval - submit a record or records (batch) for approval process
- approvals - fetch a list of all approval processes

- composite - submit up to 25 possibly related REST requests and receive individual responses
- composite-tree - create up to 200 records with parent-child relationships (up to 5 levels) in one go
- composite-batch - submit a composition of requests in batch
- getBlobField - Retrieves the specified blob field from an individual record.
- apexCall - Executes a user defined APEX REST API call.

For example, the following producer endpoint uses the upsertSObject API, with the sObjectIdName parameter specifying 'Name' as the external id field. The request message body should be an SObject DTO generated using the maven plugin. The response message will either be **null** if an existing record was updated, or **CreateSObjectResult** with an id of the new record, or a list of errors while creating the new object.

```
...to("salesforce:upsertSObject?sObjectIdName=Name")...
```

269.4.2. Rest Bulk API

Producer endpoints can use the following APIs. All Job data formats, i.e. xml, csv, zip/xml, and zip/csv are supported.
The request and response have to be marshalled/unmarshalled by the route. Usually the request will be some stream source like a CSV file,
and the response may also be saved to a file to be correlated with the request.

You can use the following for **operationName**:

- createJob - Creates a Salesforce Bulk Job
- getJob - Gets a Job using its Salesforce Id
- closeJob - Closes a Job
- abortJob - Aborts a Job
- createBatch - Submits a Batch within a Bulk Job
- getBatch - Gets a Batch using Id
- getAllBatches - Gets all Batches for a Bulk Job Id
- getRequest - Gets Request data (XML/CSV) for a Batch
- getResults - Gets the results of the Batch when its complete
- createBatchQuery - Creates a Batch from an SOQL query
- getQueryResultIds - Gets a list of Result Ids for a Batch Query
- getQueryResult - Gets results for a Result Id
- getRecentReports - Gets up to 200 of the reports you most recently viewed by sending a GET request to the Report List resource.

- getReportDescription - Retrieves the report, report type, and related metadata for a report, either in a tabular or summary or matrix format.
- executeSyncReport - Runs a report synchronously with or without changing filters and returns the latest summary data.
- executeAsyncReport - Runs an instance of a report asynchronously with or without filters and returns the summary data with or without details.
- getReportInstances - Returns a list of instances for a report that you requested to be run asynchronously. Each item in the list is treated as a separate instance of the report.
- getReportResults: Contains the results of running a report.

For example, the following producer endpoint uses the createBatch API to create a Job Batch. The in message must contain a body that can be converted into an **InputStream** (usually UTF-8 CSV or XML content from a file, etc.) and header fields 'jobId' for the Job and 'contentType' for the Job content type, which can be XML, CSV, ZIP_XML or ZIP_CSV. The put message body will contain **BatchInfo** on success, or throw a **SalesforceException** on error.

```
...to("salesforce:createBatchJob")..
```

269.4.3. Rest Streaming API

Consumer endpoints can use the following sytax for streaming endpoints to receive Salesforce notifications on create/update.

To create and subscribe to a topic

```
from("salesforce:CamelTestTopic?
notifyForFields=ALL&notifyForOperations=ALL&sObjectName=Merchandise__c&updateTopic=true&sO
bjectQuery=SELECT Id, Name FROM Merchandise__c")...
```

To subscribe to an existing topic

```
from("salesforce:CamelTestTopic&sObjectName=Merchandise__c")...
```

269.4.4. Platform events

To emit a platform event use **createSObject** operation. And set the message body can be JSON string or InputStream with key-value data – in that case **sObjectName** needs to be set to the API name of the event, or a class that extends from AbstractDTOBase with the appropriate class name for the event.

For example using a DTO:

```
class Order_Event__e extends AbstractDTOBase {
  @JsonProperty("OrderNumber")
  private String orderNumber;
  // ... other properties and getters/setters
}

from("timer:tick")
    .process(exchange -> {
        final Message in = exchange.getIn();
```

```
        String orderNumber = "ORD" + String.valueOf(in.getHeader(Exchange.TIMER_COUNTER));
        Order_Event__e event = new Order_Event__e();
        event.setOrderNumber(orderNumber);
        in.setBody(event);
    })
    .to("salesforce:createSObject");
```

Or using JSON event data:

```
from("timer:tick")
    .process(exchange -> {
        final Message in = exchange.getIn();
        String orderNumber = "ORD" + String.valueOf(in.getHeader(Exchange.TIMER_COUNTER));
        in.setBody("{\"OrderNumber\":\"" + orderNumber + "\"}");
    })
    .to("salesforce:createSObject?sObjectName=Order_Event__e");
```

To receive platform events use the consumer endpoint with the API name of the platform event prefixed with **event/** (or **/event/**), e.g.: **salesforce:events/Order_Event__e**. Processor consuming from that endpoint will receive either **org.apache.camel.component.salesforce.api.dto.PlatformEvent** object or **org.cometd.bayeux.Message** in the body depending on the **rawPayload** being **false** or **true** respectively.

For example, in the simplest form to consume one event:

```
PlatformEvent event = consumer.receiveBody("salesforce:event/Order_Event__e",
PlatformEvent.class);
```

269.5. EXAMPLES

269.5.1. Uploading a document to a ContentWorkspace

Create the ContentVersion in Java, using a Processor instance:

```
public class ContentProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        Message message = exchange.getIn();

        ContentVersion cv = new ContentVersion();
        ContentWorkspace cw = getWorkspace(exchange);
        cv.setFirstPublishLocationId(cw.getId());
        cv.setTitle("test document");
        cv.setPathOnClient("test_doc.html");
        byte[] document = message.getBody(byte[].class);
        ObjectMapper mapper = new ObjectMapper();
        String enc = mapper.convertValue(document, String.class);
        cv.setVersionDataUrl(enc);
        message.setBody(cv);
    }

    protected ContentWorkspace getWorkSpace(Exchange exchange) {
        // Look up the content workspace somehow, maybe use enrich() to add it to a
        // header that can be extracted here
```

```
        ....
    }
}
```

Give the output from the processor to the Salesforce component:

```
from("file:///home/camel/library")
    .to(new ContentProcessor())     // convert bytes from the file into a ContentVersion SObject
                                    // for the salesforce component
    .to("salesforce:createSObject");
```

269.6. USING SALESFORCE LIMITS API

With **salesforce:limits** operation you can fetch of API limits from Salesforce and then act upon that data received. The result of **salesforce:limits** operation is mapped to **org.apache.camel.component.salesforce.api.dto.Limits** class and can be used in a custom processors or expressions.

For instance, consider that you need to limit the API usage of Salesforce so that 10% of daily API requests is left for other routes. The body of output message contains an instance of **org.apache.camel.component.salesforce.api.dto.Limits** object that can be used in conjunction with Content Based Router and Content Based Router and Spring Expression Language (SpEL) to choose when to perform queries.

Notice how multiplying **1.0** with the integer value held in **body.dailyApiRequests.remaining** makes the expression evaluate as with floating point arithmetic, without it - it would end up making integral division which would result with either **0** (some API limits consumed) or **1** (no API limits consumed).

```
from("direct:querySalesforce")
    .to("salesforce:limits")
    .choice()
    .when(spel("#{1.0 * body.dailyApiRequests.remaining / body.dailyApiRequests.max < 0.1}"))
        .to("salesforce:query?...")
    .otherwise()
        .setBody(constant("Used up Salesforce API limits, leaving 10% for critical routes"))
    .endChoice()
```

269.7. WORKING WITH APPROVALS

All the properties are named exactly the same as in the Salesforce REST API prefixed with **approval.**. You can set approval properties by setting **approval.PropertyName** of the Endpoint these will be used as template – meaning that any property not present in either body or header will be taken from the Endpoint configuration. Or you can set the approval template on the Endpoint by assigning **approval** property to a reference onto a bean in the Registry.

You can also provide header values using the same **approval.PropertyName** in the incoming message headers.

And finally body can contain one **AprovalRequest** or an **Iterable** of **ApprovalRequest** objects to process as a batch.

The important thing to remember is the priority of the values specified in these three mechanisms:

1. value in body takes precedence before any other

2. value in message header takes precedence before template value

3. value in template is set if no other value in header or body was given

For example to send one record for approval using values in headers use:

Given a route:

```
from("direct:example1")//
        .setHeader("approval.ContextId", simple("${body['contextId']}"))
        .setHeader("approval.NextApproverIds", simple("${body['nextApproverIds']}"))
        .to("salesforce:approval?"//
           + "approval.actionType=Submit"//
           + "&approval.comments=this is a test"//
           + "&approval.processDefinitionNameOrId=Test_Account_Process"//
           + "&approval.skipEntryCriteria=true");
```

You could send a record for approval using:

```
final Map<String, String> body = new HashMap<>();
body.put("contextId", accountIds.iterator().next());
body.put("nextApproverIds", userId);

final ApprovalResult result = template.requestBody("direct:example1", body, ApprovalResult.class);
```

269.8. USING SALESFORCE RECENT ITEMS API

To fetch the recent items use **salesforce:recent** operation. This operation returns an **java.util.List** of **org.apache.camel.component.salesforce.api.dto.RecentItem** objects (**List<RecentItem>**) that in turn contain the **Id**, **Name** and **Attributes** (with **type** and **url** properties). You can limit the number of returned items by specifying **limit** parameter set to maximum number of records to return. For example:

```
from("direct:fetchRecentItems")
    to("salesforce:recent")
        .split().body()
            .log("${body.name} at ${body.attributes.url}");
```

269.9. WORKING WITH APPROVALS

All the properties are named exactly the same as in the Salesforce REST API prefixed with **approval.**. You can set approval properties by setting **approval.PropertyName** of the Endpoint these will be used as template – meaning that any property not present in either body or header will be taken from the Endpoint configuration. Or you can set the approval template on the Endpoint by assigning **approval** property to a reference onto a bean in the Registry.

You can also provide header values using the same **approval.PropertyName** in the incoming message headers.

And finally body can contain one **AprovalRequest** or an **Iterable** of **ApprovalRequest** objects to process as a batch.

The important thing to remember is the priority of the values specified in these three mechanisms:

1. value in body takes precedence before any other

2. value in message header takes precedence before template value

3. value in template is set if no other value in header or body was given

For example to send one record for approval using values in headers use:

Given a route:

```
from("direct:example1")//
        .setHeader("approval.ContextId", simple("${body['contextId']}"))
        .setHeader("approval.NextApproverIds", simple("${body['nextApproverIds']}"))
        .to("salesforce:approval?"//
           + "approvalActionType=Submit"//
           + "&approvalComments=this is a test"//
           + "&approvalProcessDefinitionNameOrId=Test_Account_Process"//
           + "&approvalSkipEntryCriteria=true");
```

You could send a record for approval using:

```
final Map<String, String> body = new HashMap<>();
body.put("contextId", accountIds.iterator().next());
body.put("nextApproverIds", userId);

final ApprovalResult result = template.requestBody("direct:example1", body, ApprovalResult.class);
```

269.10. USING SALESFORCE COMPOSITE API TO SUBMIT SOBJECT TREE

To create up to 200 records including parent-child relationships use **salesforce:composite-tree** operation. This requires an instance of **org.apache.camel.component.salesforce.api.dto.composite.SObjectTree** in the input message and returns the same tree of objects in the output message. The **org.apache.camel.component.salesforce.api.dto.AbstractSObjectBase** instances within the tree get updated with the identifier values (**Id** property) or their corresponding **org.apache.camel.component.salesforce.api.dto.composite.SObjectNode** is populated with **errors** on failure.

Note that for some records operation can succeed and for some it can fail – so you need to manually check for errors.

Easiest way to use this functionality is to use the DTOs generated by the **camel-salesforce-maven-plugin**, but you also have the option of customizing the references that identify the each object in the tree, for instance primary keys from your database.

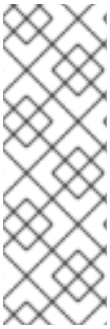
Lets look at an example:

```
Account account = ...
Contact president = ...
Contact marketing = ...

Account anotherAccount = ...
Contact sales = ...
Asset someAsset = ...
```

```
// build the tree
SObjectTree request = new SObjectTree();
request.addObject(account).addChildren(president, marketing);
request.addObject(anotherAccount).addChild(sales).addChild(someAsset);

final SObjectTree response = template.requestBody("salesforce:composite-tree", tree,
SObjectTree.class);
final Map<Boolean, List<SObjectNode>> result = response.allNodes()
                                        .collect(Collectors.groupingBy(SObjectNode::hasErrors));

final List<SObjectNode> withErrors = result.get(true);
final List<SObjectNode> succeeded = result.get(false);

final String firstId = succeeded.get(0).getId();
```

269.11. USING SALESFORCE COMPOSITE API TO SUBMIT MULTIPLE REQUESTS IN A BATCH

The Composite API batch operation (**composite-batch**) allows you to accumulate multiple requests in a batch and then submit them in one go, saving the round trip cost of multiple individual requests. Each response is then received in a list of responses with the order preserved, so that the n-th requests response is in the n-th place of the response.



NOTE

The results can vary from API to API so the result of the request is given as a **java.lang.Object**. In most cases the result will be a **java.util.Map** with string keys and values or other **java.util.Map** as value. Requests made in JSON format hold some type information (i.e. it is known what values are strings and what values are numbers), so in general those will be more type friendly. Note that the responses will vary between XML and JSON, this is due to the responses from Salesforce API being different. So be careful if you switch between formats without changing the response handling code.

Lets look at an example:

```
final String acountId = ...
final SObjectBatch batch = new SObjectBatch("38.0");

final Account updates = new Account();
updates.setName("NewName");
batch.addUpdate("Account", accountId, updates);

final Account newAccount = new Account();
newAccount.setName("Account created from Composite batch API");
batch.addCreate(newAccount);

batch.addGet("Account", accountId, "Name", "BillingPostalCode");

batch.addDelete("Account", accountId);

final SObjectBatchResponse response = template.requestBody("salesforce:composite-batch?
format=JSON", batch, SObjectBatchResponse.class);

boolean hasErrors = response.hasErrors(); // if any of the requests has resulted in either 4xx or 5xx
```

```
HTTP status
final List<SObjectBatchResult> results = response.getResults(); // results of three operations sent in batch

final SObjectBatchResult updateResult = results.get(0); // update result
final int updateStatus = updateResult.getStatusCode(); // probably 204
final Object updateResultData = updateResult.getResult(); // probably null

final SObjectBatchResult createResult = results.get(1); // create result
@SuppressWarnings("unchecked")
final Map<String, Object> createData = (Map<String, Object>) createResult.getResult();
final String newAccountId = createData.get("id"); // id of the new account, this is for JSON, for XML it would be createData.get("Result").get("id")

final SObjectBatchResult retrieveResult = results.get(2); // retrieve result
@SuppressWarnings("unchecked")
final Map<String, Object> retrieveData = (Map<String, Object>) retrieveResult.getResult();
final String accountName = retrieveData.get("Name"); // Name of the retrieved account, this is for JSON, for XML it would be createData.get("Account").get("Name")
final String accountBillingPostalCode = retrieveData.get("BillingPostalCode"); // Name of the retrieved account, this is for JSON, for XML it would be createData.get("Account").get("BillingPostalCode")

final SObjectBatchResult deleteResult = results.get(3); // delete result
final int updateStatus = deleteResult.getStatusCode(); // probably 204
final Object updateResultData = deleteResult.getResult(); // probably null
```

269.12. USING SALESFORCE COMPOSITE API TO SUBMIT MULTIPLE CHAINED REQUESTS

The **composite** operation allows submitting up to 25 requests that can be chained together, for instance identifier generated in previous request can be used in subsequent request. Individual requests and responses are linked with the provided *reference*.

NOTE

Composite API supports only JSON payloads.

NOTE

As with the batch API the results can vary from API to API so the result of the request is given as a **java.lang.Object**. In most cases the result will be a **java.util.Map** with string keys and values or other **java.util.Map** as value. Requests made in JSON format hold some type information (i.e. it is known what values are strings and what values are numbers), so in general those will be more type friendly.

Lets look at an example:

```
SObjectComposite composite = new SObjectComposite("38.0", true);

// first insert operation via an external id
final Account updateAccount = new TestAccount();
updateAccount.setName("Salesforce");
updateAccount.setBillingStreet("Landmark @ 1 Market Street");
```

```
updateAccount.setBillingCity("San Francisco");
updateAccount.setBillingState("California");
updateAccount.setIndustry(Account_IndustryEnum.TECHNOLOGY);
composite.addUpdate("Account", "001xx000003DIpcAAG", updateAccount, "UpdatedAccount");

final Contact newContact = new TestContact();
newContact.setLastName("John Doe");
newContact.setPhone("1234567890");
composite.addCreate(newContact, "NewContact");

final AccountContactJunction__c junction = new AccountContactJunction__c();
junction.setAccount__c("001xx000003DIpcAAG");
junction.setContactId__c("@{NewContact.id}");
composite.addCreate(junction, "JunctionRecord");

final SObjectCompositeResponse response = template.requestBody("salesforce:composite?
format=JSON", composite, SObjectCompositeResponse.class);
final List<SObjectCompositeResult> results = response.getCompositeResponse();

final SObjectCompositeResult accountUpdateResult = results.stream().filter(r ->
"UpdatedAccount".equals(r.getReferenceId())).findFirst().get()
final int statusCode = accountUpdateResult.getHttpStatusCode(); // should be 200
final Map<String, ?> accountUpdateBody = accountUpdateResult.getBody();

final SObjectCompositeResult contactCreationResult = results.stream().filter(r ->
"JunctionRecord".equals(r.getReferenceId())).findFirst().get()
```

269.13. CAMEL SALESFORCE MAVEN PLUGIN

This Maven plugin generates DTOs for the Camel Salesforce.

269.14. OPTIONS

The Salesforce component supports 29 options which are listed below.

Name	Description	Default	Type
authenticationType (security)	Explicit authentication method to be used, one of USERNAME_PASSWORD, REFRESH_TOKEN or JWT. Salesforce component can auto-determine the authentication method to use from the properties set, set this property to eliminate any ambiguity.		AuthenticationType
loginConfig (security)	All authentication configuration in one nested bean, all properties set there can be set directly on the component as well		SalesforceLoginConfig
instanceUrl (security)	URL of the Salesforce instance used after authantication, by default received from Salesforce on successful authentication		String

Name	Description	Default	Type
loginUrl (security)	**Required** URL of the Salesforce instance used for authentication, by default set to https://login.salesforce.com	https://login.salesforce.com	String
clientId (security)	**Required** OAuth Consumer Key of the connected app configured in the Salesforce instance setup. Typically a connected app needs to be configured but one can be provided by installing a package.		String
clientSecret (security)	OAuth Consumer Secret of the connected app configured in the Salesforce instance setup.		String
keystore (security)	KeyStore parameters to use in OAuth JWT flow. The KeyStore should contain only one entry with private key and certificate. Salesforce does not verify the certificate chain, so this can easily be a selfsigned certificate. Make sure that you upload the certificate to the corresponding connected app.		KeyStoreParameters
refreshToken (security)	Refresh token already obtained in the refresh token OAuth flow. One needs to setup a web application and configure a callback URL to receive the refresh token, or configure using the builtin callback at https://login.salesforce.com/services/oauth2/success or https://test.salesforce.com/services/oauth2/success and then retrive the refresh_token from the URL at the end of the flow. Note that in development organizations Salesforce allows hosting the callback web application at localhost.		String
userName (security)	Username used in OAuth flow to gain access to access token. It's easy to get started with password OAuth flow, but in general one should avoid it as it is deemed less secure than other flows.		String
password (security)	Password used in OAuth flow to gain access to access token. It's easy to get started with password OAuth flow, but in general one should avoid it as it is deemed less secure than other flows. Make sure that you append security token to the end of the password if using one.		String

Name	Description	Default	Type
lazyLogin (security)	If set to true prevents the component from authenticating to Salesforce with the start of the component. You would generaly set this to the (default) false and authenticate early and be immediately aware of any authentication issues.	false	boolean
config (common)	Global endpoint configuration - use to set values that are common to all endpoints		SalesforceEndpoint Config
httpClientProperties (common)	Used to set any properties that can be configured on the underlying HTTP client. Have a look at properties of SalesforceHttpClient and the Jetty HttpClient for all available options.		Map
longPollingTransport Properties (common)	Used to set any properties that can be configured on the LongPollingTransport used by the BayeuxClient (CometD) used by the streaming api		Map
sslContextParameters (security)	SSL parameters to use, see SSLContextParameters class for all available options.		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters	false	boolean
httpProxyHost (proxy)	Hostname of the HTTP proxy server to use.		String
httpProxyPort (proxy)	Port number of the HTTP proxy server to use.		Integer
httpProxyUsername (security)	Username to use to authenticate against the HTTP proxy server.		String
httpProxyPassword (security)	Password to use to authenticate against the HTTP proxy server.		String
isHttpProxySocks4 (proxy)	If set to true the configures the HTTP proxy to use as a SOCKS4 proxy.	false	boolean
isHttpProxySecure (security)	If set to false disables the use of TLS when accessing the HTTP proxy.	true	boolean

Name	Description	Default	Type
httpProxyIncluded Addresses (proxy)	A list of addresses for which HTTP proxy server should be used.		Set
httpProxyExcluded Addresses (proxy)	A list of addresses for which HTTP proxy server should not be used.		Set
httpProxyAuthUri (security)	Used in authentication against the HTTP proxy server, needs to match the URI of the proxy server in order for the httpProxyUsername and httpProxyPassword to be used for authentication.		String
httpProxyRealm (security)	Realm of the proxy server, used in preemptive Basic/Digest authentication methods against the HTTP proxy server.		String
httpProxyUseDigest Auth (security)	If set to true Digest authentication will be used when authenticating to the HTTP proxy,otherwise Basic authorization method will be used	false	boolean
packages (common)	In what packages are the generated DTO classes. Typically the classes would be generated using camel-salesforce-maven-plugin. Set it if using the generated DTOs to gain the benefit of using short SObject names in parameters/header values.		String[]
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Salesforce endpoint is configured using URI syntax:

```
salesforce:operationName:topicName
```

with the following path and query parameters:

269.14.1. Path Parameters (2 parameters):

Name	Description	Default	Type
operationName	The operation to use		OperationName

Name	Description	Default	Type
topicName	The name of the topic to use		String

269.14.2. Query Parameters (44 parameters):

Name	Description	Default	Type
apexMethod (common)	APEX method name		String
apexQueryParams (common)	Query params for APEX method		Map
apexUrl (common)	APEX method URL		String
apiVersion (common)	Salesforce API version, defaults to SalesforceEndpointConfig.DEFAULT_VERSION		String
backoffIncrement (common)	Backoff interval increment for Streaming connection restart attempts for failures beyond CometD auto-reconnect.		long
batchId (common)	Bulk API Batch ID		String
contentType (common)	Bulk API content type, one of XML, CSV, ZIP_XML, ZIP_CSV		ContentType
defaultReplayId (common)	Default replayId setting if no value is found in link initialReplayIdMap		Long
format (common)	Payload format to use for Salesforce API calls, either JSON or XML, defaults to JSON		PayloadFormat
httpClient (common)	Custom Jetty Http Client to use to connect to Salesforce.		SalesforceHttpClient
includeDetails (common)	Include details in Salesforce1 Analytics report, defaults to false.		Boolean
initialReplayIdMap (common)	Replay IDs to start from per channel name.		Map
instanceId (common)	Salesforce1 Analytics report execution instance ID		String

Name	Description	Default	Type
jobId (common)	Bulk API Job ID		String
limit (common)	Limit on number of returned records. Applicable to some of the API, check the Salesforce documentation.		Integer
maxBackoff (common)	Maximum backoff interval for Streaming connection restart attempts for failures beyond CometD auto-reconnect.		long
notFoundBehaviour (common)	Sets the behaviour of 404 not found status received from Salesforce API. Should the body be set to NULL link NotFoundBehaviourNULL or should a exception be signaled on the exchange link NotFoundBehaviourEXCEPTION - the default.		NotFoundBehaviour
notifyForFields (common)	Notify for fields, options are ALL, REFERENCED, SELECT, WHERE		NotifyForFieldsEnum
notifyForOperationCreate (common)	Notify for create operation, defaults to false (API version = 29.0)		Boolean
notifyForOperationDelete (common)	Notify for delete operation, defaults to false (API version = 29.0)		Boolean
notifyForOperations (common)	Notify for operations, options are ALL, CREATE, EXTENDED, UPDATE (API version 29.0)		NotifyForOperations Enum
notifyForOperationUndelete (common)	Notify for un-delete operation, defaults to false (API version = 29.0)		Boolean
notifyForOperationUpdate (common)	Notify for update operation, defaults to false (API version = 29.0)		Boolean
objectMapper (common)	Custom Jackson ObjectMapper to use when serializing/deserializing Salesforce objects.		ObjectMapper
rawPayload (common)	Use raw payload String for request and response (either JSON or XML depending on format), instead of DTOs, false by default	false	boolean
reportId (common)	Salesforce1 Analytics report Id		String

Name	Description	Default	Type
reportMetadata (common)	Salesforce1 Analytics report metadata for filtering		ReportMetadata
resultId (common)	Bulk API Result ID		String
serializeNulls (common)	Should the NULL values of given DTO be serialized with empty (NULL) values. This affects only JSON data format.	false	boolean
sObjectBlobFieldName (common)	SObject blob field name		String
sObjectClass (common)	Fully qualified SObject class name, usually generated using camel-salesforce-maven-plugin		String
sObjectFields (common)	SObject fields to retrieve		String
sObjectId (common)	SObject ID if required by API		String
sObjectIdName (common)	SObject external ID field name		String
sObjectIdValue (common)	SObject external ID field value		String
sObjectName (common)	SObject name if required or supported by API		String
sObjectQuery (common)	Salesforce SOQL query string		String
sObjectSearch (common)	Salesforce SOSL search string		String
updateTopic (common)	Whether to update an existing Push Topic when using the Streaming API, defaults to false	false	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
replayId (consumer)	The replayId value to use when subscribing		Long
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

For obvious security reasons it is recommended that the clientId, clientSecret, userName and password fields be not set in the pom.xml. The plugin should be configured for the rest of the properties, and can be executed using the following command:

```
mvn camel-salesforce:generate -DcamelSalesforce.clientId=<clientid> -
DcamelSalesforce.clientSecret=<clientsecret> \
    -DcamelSalesforce.userName=<username> -DcamelSalesforce.password=<password>
```

The generated DTOs use Jackson and XStream annotations. All Salesforce field types are supported. Date and time fields are mapped to Joda DateTime, and picklist fields are mapped to generated Java Enumerations.

269.15. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 270. SAP COMPONENT

The SAP component is a package consisting of a suite of ten different SAP components. There are remote function call (RFC) components that support the sRFC, tRFC, and qRFC protocols; and there are IDoc components that facilitate communication using messages in IDoc format. The component uses the SAP Java Connector (SAP JCo) library to facilitate bidirectional communication with SAP and the SAP IDoc library to facilitate the transmission of documents in the Intermediate Document (IDoc) format.

270.1. OVERVIEW

Dependencies

Maven users need to add the following dependency to their **pom.xml** file to use this component:

```
<dependency>
  <groupId>org.fusesource</groupId>
  <artifactId>camel-sap</artifactId>
  <version>x.x.x</version>
<dependency>
```

Additional platform restrictions for the SAP component

Because the SAP component depends on the third-party JCo 3.0 and IDoc 3.0 libraries, it can only be installed on the platforms that these libraries support. For more details about the platform restrictions, see Red Hat JBoss Fuse Supported Configurations .

SAP JCo and SAP IDoc libraries

A prerequisite for using the SAP component is that the SAP Java Connector (SAP JCo) libraries and the SAP IDoc library are installed into the **lib/** directory of the Java runtime. You must make sure that you download the appropriate set of SAP libraries for your target operating system from the SAP Service Marketplace.

The names of the library files vary depending on the target operating system, as shown in Table 270.1, “Required SAP Libraries”.

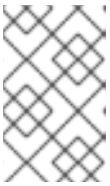
Table 270.1. Required SAP Libraries

SAP Component	Linux and UNIX	Windows
SAP JCo 3	**sapjco3.jar** **libsapjco3.so**	**sapjco3.jar** **sapjco3.dll**
SAP IDoc	**sapidoc3.jar**	**sapidoc3.jar**

Deploying in a Fuse OSGi Container

You can install the SAP JCo libraries and the SAP IDoc library into the JBoss Fuse OSGi container as follows:

1. Download the SAP JCo libraries and the SAP IDoc library from the SAP Service Marketplace (http://service.sap.com/public/connectors), making sure to choose the appropriate version of the libraries for your operating system.



NOTE

You require version 3.0.11 or greater of the JCo library and version 3.0.10 or greater of the IDoc library. You must have an **SAP Service Marketplace Account** in order to download and use these libraries.

2. Copy the **sapjco3.jar**, **libsapjco3.so** (or **sapjco3.dll** on Windows), and **sapidoc3.jar** library files into the **lib/** directory of your Fuse installation.

3. Open both the configuration properties file, **etc/config.properties**, and the custom properties file, **etc/custom.properties**, in a text editor. In the **etc/config.properties** file, look for the **org.osgi.framework.system.packages.extra** property and copy the complete property setting (this setting extends over multiple lines, with a backslash character, \, used to indicate line continuation). Now paste this setting into the **etc/custom.properties** file.
You can now add the extra packages required to support the SAP libraries. In the **etc/custom.properties** file, add the required packages to the **org.osgi.framework.system.packages.extra** setting as shown:

```
org.osgi.framework.system.packages.extra = \
... , \
com.sap.conn.idoc, \
com.sap.conn.idoc.jco, \
com.sap.conn.jco, \
com.sap.conn.jco.ext, \
com.sap.conn.jco.monitor, \
com.sap.conn.jco.rt, \
com.sap.conn.jco.server
```

Don't forget to include a comma and a backslash, **,** \, at the end of each line preceding the new entries, so that the list is properly continued.

4. You need to restart the container for these changes to take effect.

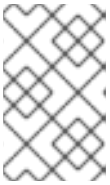
5. You need to install the **camel-sap** feature in the container. In the Karaf console, enter the following command:

```
JBossFuse:karaf@root> features:install camel-sap
```

Deploying in a JBoss EAP container

To deploy the SAP component in a JBoss EAP container, perform the following steps:

1. Download the SAP JCo libraries and the SAP IDoc library from the SAP Service Marketplace (http://service.sap.com/public/connectors), making sure to choose the appropriate version of the libraries for your operating system.



NOTE

You require version 3.0.11 or greater of the JCo library and version 3.0.10 or greater of the IDoc library. You must have an **SAP Service Marketplace Account** in order to download and use these libraries.

2. Copy the JCo library files and the IDoc library file into the appropriate subdirectory of your JBoss EAP installation. For example, if your host platform is 64-bit Linux (**linux-x86_64**), install the library files as follows:

```
cp sapjco3.jar sapidoc3.jar
$JBOSS_HOME/modules/system/layers/fuse/com/sap/conn/jco/main/
mkdir -p $JBOSS_HOME/modules/system/layers/fuse/com/sap/conn/jco/main/lib/linux-
x86_64
cp libsapjco3.so
$JBOSS_HOME/modules/system/layers/fuse/com/sap/conn/jco/main/lib/linux-x86_64/
```

IMPORTANT

For installing native libraries (such as **libsapjco3.so**) into the JBoss EAP installation, there is a standardized convention for naming the library subdirectory, which must be followed. In the case of 64-bit Linux, the subdirectory is **linux-x86_64**. For other platforms, see https://docs.jboss.org/author/display/MODULES/Native+Libraries.

3. Create a new file called **$JBOSS_HOME/modules/system/layers/fuse/org/wildfly/camel/extras/main/module.xml** and **add** the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.1" name="org.wildfly.camel.extras">

 <dependencies>
  <module name="org.fusesource.camel.component.sap" export="true" services="export" />
 </dependencies>

</module>
```

URI format

There are two different kinds of endpoint provided by the SAP component: the Remote Function Call (RFC) endpoints, and the Intermediate Document (IDoc) endpoints.

The URI formats for the RFC endpoints are as follows:

```
sap-srfc-destination:destinationName:rfcName
sap-trfc-destination:destinationName:rfcName
sap-qrfc-destination:destinationName:queueName:rfcName
sap-srfc-server:serverName:rfcName[?options]
sap-trfc-server:serverName:rfcName[?options]
```

The URI formats for the IDoc endpoints are as follows:

```
sap-idoc-
destination:destinationName:idocType[:idocTypeExtension[:systemRelease[:applicationRelease]]]
sap-idoclist-
destination:destinationName:idocType[:idocTypeExtension[:systemRelease[:applicationRelease]]]
sap-qidoc-
destination:destinationName:queueName:idocType[:idocTypeExtension[:systemRelease[:applicationR
elease]]]
sap-qidoclist-
destination:destinationName:queueName:idocType[:idocTypeExtension[:systemRelease[:applicationR
elease]]]
sap-idoclist-server:serverName:idocType[:idocTypeExtension[:systemRelease[:applicationRelease]]]
[?options]
```

The URI formats prefixed by sap-*endpointKind*-destination are used to define destination endpoints (in other words, Camel producer endpoints) and *destinationName* is the name of a specific outbound connection to an SAP instance. Outbound connections are named and configured at the component level, as described in Section 270.2.2, "Destination Configuration".

The URI formats prefixed by sap-*endpointKind*-server are used to define server endpoints (in other words, Camel consumer endpoints) and *serverName* is the name of a specific inbound connection from an SAP instance. Inbound connections are named and configured at the component level, as described in the Section 270.2.3, "Server Configuration".

The other components of an RFC endpoint URI are as follows:

rfcName

(Required) In a destination endpoint URI, is the name of the RFC invoked by the endpoint in the connected SAP instance. In a server endpoint URI, is the name of the RFC handled by the endpoint when invoked from the connected SAP instance.

queueName

Specifies the queue this endpoint sends an SAP request to.

The other components of an IDoc endpoint URI are as follows:

idocType

(Required) Specifies the Basic IDoc Type of an IDoc produced by this endpoint.

idocTypeExtension

Specifies the IDoc Type Extension, if any, of an IDoc produced by this endpoint.

systemRelease

Specifies the associated SAP Basis Release, if any, of an IDoc produced by this endpoint.

applicationRelease

Specifes the associated Application Release, if any, of an IDoc produced by this endpoint.

queueName

Specifies the queue this endpoint sends an SAP request to.

Options for RFC destination endpoints

The RFC destination endpoints (**sap-srfc-destination**, **sap-trfc-destination**, and **sap-qrfc-destination**) support the following URI options:

Name	Default	Description
stateful	**false**	If **true**, specifies that this endpoint initiates an SAP stateful session
transacted	**false**	If **true**, specifies that this endpoint initiates an SAP transaction

Options for RFC server endpoints

The SAP RFC server endpoints (**sap-srfc-server** and **sap-trfc-server**) support the following URI options:

Name	Default	Description
stateful	**false**	If **true**, specifies that this endpoint initiates an SAP stateful session.
propagateExceptions	**false**	**(sap-trfc-server endpoint only)** If **true**, specifies that this endpoint propagates exceptions back to the caller in SAP, instead of the exchange's exception handler

Options for the IDoc List Server endpoint

The SAP IDoc List Server endpoint (**sap-idoclist-server**) supports the following URI options:

Name	Default	Description
stateful	**false**	If **true**, specifies that this endpoint initiates an SAP stateful session.
propagateExceptions	**false**	If **true**, specifies that this endpoint propagates exceptions back to the caller in SAP, instead of the exchange's exception handler

Summary of the RFC and IDoc endpoints

The SAP component package provides the following RFC and IDoc endpoints:

sap-srfc-destination

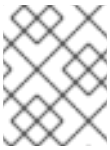
JBoss Fuse SAP Synchronous Remote Function Call Destination Camel component. This endpoint should be used in cases where Camel routes require synchronous delivery of requests to and responses from an SAP system.

NOTE

The sRFC protocol used by this component delivers requests and responses to and from an SAP system with **best effort**. In case of a communication error while sending a request, the completion status of a remote function call in the receiving SAP system remains **in doubt**.

sap-trfc-destination

JBoss Fuse SAP Transactional Remote Function Call Destination Camel component. This endpoint should be used in cases where requests must be delivered to the receiving SAP system **at most once**. To accomplish this, the component generates a transaction ID, **tid**, which accompanies every request sent through the component in a route's exchange. The receiving SAP system records the **tid** accompanying a request before delivering the request; if the SAP system receives the request again with the same **tid** it will not deliver the request. Thus if a route encounters a communication error when sending a request through an endpoint of this component, it can retry sending the request within the same exchange knowing it will be delivered and executed only once.



NOTE

The tRFC protocol used by this component is asynchronous and does not return a response. Thus the endpoints of this component do not return a response message.

NOTE

This component does not guarantee the order of a series of requests through its endpoints, and the delivery and execution order of these requests may differ on the receiving SAP system due to communication errors and resends of a request. For guaranteed delivery order, please see the JBoss Fuse SAP Queued Remote Function Call Destination Camel component.

sap-qrfc-destination

JBoss Fuse SAP Queued Remote Function Call Destination Camel component. This component extends the capabilities of the JBoss Fuse Transactional Remote Function Call Destination camel component by adding **in order** delivery guarantees to the delivery of requests through its endpoints. This endpoint should be used in cases where a series of requests depend on each other and must be delivered to the receiving SAP system **at most once** and **in order**. The component accomplishes the **at most once** delivery guarantees using the same mechanisms as the JBoss Fuse SAP Transactional Remote Function Call Destination Camel component. The ordering guarantee is accomplished by serializing the requests in the order they are received by the SAP system to an *inbound queue*. Inbound queues are processed by the *QIN scheduler* within SAP. When the inbound queue is **activated**, the QIN Scheduler will execute the queue requests in order.

NOTE

The qRFC protocol used by this component is asynchronous and does not return a response. Thus the endpoints of this component do not return a response message.

sap-srfc-server

JBoss Fuse SAP Synchronous Remote Function Call Server Camel component. This component and its endpoints should be used in cases where a Camel route is required to synchronously handle requests from and responses to an SAP system.

sap-trfc-server

JBoss Fuse SAP Transactional Remote Function Call Server Camel component. This endpoint should be used in cases where the sending SAP system requires **at most once** delivery of its requests to a Camel route. To accomplish this, the sending SAP system generates a transaction ID, **tid**, which accompanies every request it sends to the component's endpoints. The sending SAP system will first check with the component whether a given **tid** has been received by it before sending a series of requests associated with the **tid**. The component will check the list of received **tid**s it maintains, record the sent **tid** if it is not in that list, and then respond to the sending SAP system, indicating whether or not the **tid** had already been recorded. The sending SAP system will only then send the series of requests, if the **tid** has not been previously recorded. This enables a sending SAP system to reliably send a series of requests once to a camel route.

sap-idoc-destination

JBoss Fuse SAP IDoc Destination Camel component. This endpoint should be used in cases where a Camel route is required to send a list of Intermediate Documents (IDocs) to an SAP system.

sap-idoclist-destination

JBoss Fuse SAP IDoc List Destination Camel component. This endpoint should be used in cases where a Camel route is required to send a list of Intermediate documents (IDocs) list to an SAP system.

sap-qidoc-destination

JBoss Fuse SAP Queued IDoc Destination Camel component. This component and its endpoints should be used in cases where a Camel route is required to send a list of Intermediate documents (IDocs) to an SAP system in order.

sap-qidoclist-destination

JBoss Fuse SAP Queued IDoc List Destination Camel component. This component and its endpoints should be used in cases where a camel route is required to send a list of Intermediate documents (IDocs) list to an SAP system in order.

sap-idoclist-server

JBoss Fuse SAP IDoc List Server Camel component. This endpoint should be used in cases where a sending SAP system requires delivery of Intermediate Document lists to a Camel route. This component uses the tRFC protocol to communicate with SAP as described in the **sap-trfc-server-standalone** quick start.

SAP RFC destination endpoint

An RFC destination endpoint supports outbound communication to SAP, which enable these endpoints to make RFC calls out to ABAP function modules in SAP. An RFC destination endpoint is configured to make an RFC call to a specific ABAP function over a specific connection to an SAP instance. An RFC destination is a logical designation for an outbound connection and has a unique name. An RFC destination is specified by a set of connection parameters called *destination data*.

An RFC destination endpoint will extract an RFC request from the input message of the IN-OUT exchanges it receives and dispatch that request in a function call to SAP. The response from the function call will be returned in the output message of the exchange. Since SAP RFC destination endpoints only support outbound communication, an RFC destination endpoint only supports the creation of producers.

SAP RFC server endpoint

An RFC server endpoint supports inbound communication from SAP, which enables ABAP applications in SAP to make RFC calls into server endpoints. An ABAP application interacts with an RFC server endpoint as if it were a remote function module. An RFC server endpoint is configured to receive an RFC call to a specific RFC function over a specific connection from an SAP instance. An RFC server is a logical designation for an inbound connection and has a unique name. An RFC server is specified by a set of connection parameters called *server data*.

An RFC server endpoint will handle an incoming RFC request and dispatch it as the input message of an IN-OUT exchange. The output message of the exchange will be returned as the response of the RFC call. Since SAP RFC server endpoints only support inbound communication, an RFC server endpoint only supports the creation of consumers.

SAP IDoc and IDoc list destination endpoints

An IDoc destination endpoint supports outbound communication to SAP, which can then perform further processing on the IDoc message. An IDoc document represents a business transaction, which can easily be exchanged with non-SAP systems. An IDoc destination is specified by a set of connection parameters called *destination data*.

An IDoc list destination endpoint is similar to an IDoc destination endpoint, except that the messages it handles consist of a **list** of IDoc documents.

SAP IDoc list server endpoint

An IDoc list server endpoint supports inbound communication from SAP, enabling a Camel route to receive a list of IDoc documents from an SAP system. An IDoc list server is specified by a set of connection parameters called *server data*.

Meta-data repositories

A meta-data repository is used to store the following kinds of meta-data:

Interface descriptions of function modules

This meta-data is used by the JCo and ABAP runtimes to check RFC calls to ensure the type-safe transfer of data between communication partners before dispatching those calls. A repository is populated with repository data. Repository data is a map of named function templates. A function template contains the meta-data describing all the parameters and their typing information passed to and from a function module and has the unique name of the function module it describes.

IDoc type descriptions

This meta-data is used by the IDoc runtime to ensure that the IDoc documents are correctly formatted before being sent to a communication partner. A basic IDoc type consists of a name, a list of permitted segments, and a description of the hierarchical relationship between the segments. Some additional constraints can be imposed on the segments: a segment can be mandatory or optional; and it is possible to specify a minimum/maximum range for each segment (defining the number of allowed repetitions of that segment).

SAP destination and server endpoints thus require access to a repository, in order to send and receive RFC calls and in order to send and receive IDoc documents. For RFC calls, the meta-data for all function modules invoked and handled by the endpoints must reside within the repository; and for IDoc endpoints, the meta-data for all IDoc types and IDoc type extensions handled by the endpoints must reside within the repository. The location of the repository used by a destination and server endpoint is specified in the destination data and the server data of their respective connections.

In the case of an SAP destination endpoint, the repository it uses typically resides in an SAP system and

it defaults to the SAP system it is connected to. This default requires no explicit configuration in the destination data. Furthermore, the meta-data for the remote function call that a destination endpoint makes will already exist in a repository for any existing function module that it calls. The meta-data for calls made by destination endpoints thus require no configuration in the SAP component.

On the other hand, the meta-data for function calls handled by server endpoints do not typically reside in the repository of an SAP system and must instead be provided by a repository residing in the SAP component. The SAP component maintains a map of named meta-data repositories. The name of a repository corresponds to the name of the server to which it provides meta-data.

270.2. CONFIGURATION

The SAP component maintains three maps to store destination data, server data and repository data. The *destination data store* and the *server data store* are configured on a special configuration object, **SapConnectionConfiguration**, which automatically gets injected into the SAP component (in the context of Blueprint XML configuration or Spring XML configuration files). The *repository data store* must be configured directly on the relevant SAP component.

When deploying in EAP, **SapConnectionConfiguration** should be deployed separately. Only one instance should exist per EAP instance. If multiple instances are created, an exception is thrown: **java.lang.IllegalStateException: DestinationDataProvider already registered**.

270.2.1. Configuration Overview

Overview

The SAP component maintains three maps to store destination data, server data and repository data. The component's property, **destinationDataStore**, stores destination data keyed by destination name, the property, **serverDataStore**, stores server data keyed by server name and the property, **repositoryDataStore**, stores repository data keyed by repository name. These configurations must be passed to the component during its initialization.

Example

The following example shows how to configure a sample destination data store and a sample server data store in a Blueprint XML file. The **sap-configuration** bean (of type **SapConnectionConfiguration**) will automatically be injected into any SAP component that is used in this XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
   ...
   <!-- Configures the Inbound and Outbound SAP Connections -->
   <bean id="sap-configuration"
      class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
      <property name="destinationDataStore">
         <map>
            <entry key="quickstartDest" value-ref="quickstartDestinationData" />
         </map>
      </property>
      <property name="serverDataStore">
         <map>
            <entry key="quickstartServer" value-ref="quickstartServerData" />
         </map>
      </property>
```

```
    </bean>

    <!-- Configures an Outbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment *** -->
    <bean id="quickstartDestinationData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl">
        <property name="ashost" value="example.com" />
        <property name="sysnr" value="00" />
        <property name="client" value="000" />
        <property name="user" value="username" />
        <property name="passwd" value="passowrd" />
        <property name="lang" value="en" />
    </bean>

    <!-- Configures an Inbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment ** -->
    <bean id="quickstartServerData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.ServerDataImpl">
        <property name="gwhost" value="example.com" />
        <property name="gwserv" value="3300" />
        <!-- The following property values should not be changed -->
        <property name="progid" value="QUICKSTART" />
        <property name="repositoryDestination" value="quickstartDest" />
        <property name="connectionCount" value="2" />
    </bean>
</blueprint>
```

270.2.2. Destination Configuration

Overview

The configurations for destinations are maintained in the **destinationDataStore** property of the SAP component. Each entry in this map configures a distinct outbound connection to an SAP instance. The key for each entry is the name of the outbound connection and is used in the *destinationName* component of a destination endpoint URI as described in the URI format section.

The value for each entry is a destination data configuration object - **org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl** - that specifies the configuration of an outbound SAP connection.

Sample destination configuration

The following Blueprint XML code shows how to configure a sample destination with the name, **quickstartDest**.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
    ...
    <!-- Create interceptor to support tRFC processing -->
    <bean id="currentProcessorDefinitionInterceptor"
          class="org.fusesource.camel.component.sap.CurrentProcessorDefinitionInterceptStrategy" />

    <!-- Configures the Inbound and Outbound SAP Connections -->
    <bean id="sap-configuration"
```

```
        class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
        <property name="destinationDataStore">
            <map>
                <entry key="quickstartDest" value-ref="quickstartDestinationData" />
            </map>
        </property>
    </bean>

    <!-- Configures an Outbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment *** -->
    <bean id="quickstartDestinationData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl">
        <property name="ashost" value="example.com" />
        <property name="sysnr" value="00" />
        <property name="client" value="000" />
        <property name="user" value="username" />
        <property name="passwd" value="password" />
        <property name="lang" value="en" />
    </bean>

</blueprint>
```

For example, after configuring the destination as shown in the preceding Blueprint XML file, you could invoke the **BAPI_FLCUST_GETLIST** remote function call on the **quickstartDest** destination using the following URI:

```
sap-srfc-destination:quickstartDest:BAPI_FLCUST_GETLIST
```

Interceptor for tRFC and qRFC destinations

The preceding sample destination configuration shows the instantiation of a **CurrentProcessorDefinitionInterceptStrategy** object. This object installs an interceptor in the Camel runtime, which enables the Camel SAP component to keep track of its position within a Camel route while it is handling RFC transactions. For more details, see the section called “Transactional RFC destination endpoints”.

IMPORTANT

This interceptor is critically important for transactional RFC destination endpoints (such as **sap-trfc-destination** and **sap-qrfc-destination**) and must be installed in the Camel runtime in order for outbound transactional RFC communication to be properly managed. The Destination RFC Transaction Handlers will issue warnings into the Camel log if the strategy is not found at runtime and in this situation the Camel runtime will need to be re-provisioned and restarted to properly manage outbound transactional RFC communication.

Logon and authentication options

The following table lists the **logon and authentication** options for configuring a destination in the SAP destination data store:

Name	Default Value	Description

client		SAP client, mandatory logon parameter
user		Logon user, logon parameter for password based authentication
aliasUser		Logon user alias, can be used instead of logon user
userId		User identity which is used for logon to the ABAP AS. Used by the JCo runtime, if the destination configuration uses SSO/assertion ticket, certificate, current user ,or SNC environment for authentication. The user ID is mandatory, if neither user nor user alias is set. This ID will never be sent to the SAP backend, it will be used by the JCo runtime locally.
passwd		Logon password, logon parameter for password-based authentication
lang		Logon language, if not defined, the default user language is used
mysapsso2		Use the specified SAP Cookie Version 2 as logon ticket for SSO based authentication
x509cert		Use the specified X509 certificate for certificate based authentication
lcheck		Postpone the authentication until the first call - 1 (enable). Used in special cases only .
useSapGui		Use a visible, hidden, or do not use SAP GUI
codePage		Additional logon parameter to define the codepage that will used to convert the logon parameters. Used in special cases only

getsso2		Order a SSO ticket after logon, the obtained ticket is available in the destination attributes
denyInitialPassword		If set to **1**, using initial passwords will lead to an exception (default is **0**).

Connection options

The following table lists the **connection** options for configuring a destination in the SAP destination data store:

Name	Default Value	Description
saprouter		SAP Router string for connection to systems behind a SAP Router. SAP Router string contains the chain of SAP Routers and its port numbers and has the form: **(/H/<host>[/S/<port>])+**
sysnr		System number of the SAP ABAP application server, mandatory for a direct connection
ashost		SAP ABAP application server, mandatory for a direct connection
mshost		SAP message server, mandatory property for a load balancing connection
msserv		SAP message server port, optional property for a load balancing connection. In order to resolve the service names sapmsXXX a lookup in **etc/services** is performed by the network layer of the operating system. If using port numbers instead of symbolic service names, no look-ups are performed and no additional entries are needed.

gwhost		Allows specifying a concrete gateway, which should be used for establishing the connection to an application server. If not specified the gateway on the application server is used
gwserv		Should be set, when using gwhost. Allows specifying the port used on that gateway. If not specified the port of the gateway on the application server is used. In order to resolve the service names sapgwXXX a lookup in etc/services is performed by the network layer of the operating system. If using port numbers instead of symbolic service names, no lookups are performed and no additional entries are needed.
r3name		System ID of the SAP system, mandatory property for a load balancing connection.
group		Group of SAP application servers, mandatory property for a load balancing connection

Connection pool options

The following table lists the **connection pool** options for configuring a destination in the SAP destination data store:

Name	Default Value	Description
peakLimit	**0**	Maximum number of active outbound connections that can be created for a destination simultaneously. A value of **0** allows an unlimited number of active connections, otherwise if the value is less than the value of **jpoolCapacity**, it will be automatically increased to this value. Default setting is the value of **poolCapacity**, or in case of **poolCapacity** not being specified as well, the default is **0** (unlimited).

poolCapacity	**1**	Maximum number of idle outbound connections kept open by the destination. A value of **0** has the effect that there is no connection pooling (default is **1**).
expirationTime		Time in milliseconds after which a free connection held internally by the destination can be closed
expirationPeriod		Period in milliseconds after which the destination checks the released connections for expiration.
maxGetTime		Maximum time in milliseconds to wait for a connection, if the maximum allowed number of connections has already been allocated by the application.

Secure network connection options

The following table lists the **secure network** options for configuring a destination in the SAP destination data store:

Name	Default Value	Description
sncMode		Secure network connection (SNC) mode, **0** (off) or **1** (on)
sncPartnername		SNC partner, for example: **p:CN=R3, O=XYZ-INC, C=EN**
sncQop		SNC level of security: **1** to **9**
sncMyname		Own SNC name. Overrides environment settings
sncLibrary		Path to library that provides SNC service

Repository options

The following table lists the **repository** options for configuring a destination in the SAP destination data store:

Name	Default Value	Description

repositoryDest		Specifies which destination should be used as repository.
repositoryUser		If a repository destination is not set, and this property is set, it will be used as user for repository calls. This enables you to use a different user for repository look-ups.
repositoryPasswd		The password for a repository user. Mandatory, if a repository user should be used.
repositorySnc		**(Optional)** If SNC is used for this destination, it is possible to turn it off for repository connections, if this property is set to **0**. Default setting is the value of **jco.client.snc**_mode. For special cases only.

repositoryRoundtripOptimization		Enable the **RFC_METADATA_GET** API, which provides repository data in one single round trip. **1** Activates use of **RFC_METADATA_GET** in ABAP System, **0** Deactivates **RFC_METADATA_GET** in ABAP System. If the property is not set, the destination initially does a remote call to check whether **RFC_METADATA_GET** is available. If it is available, the destination will use it. **Note:** If the repository is already initialized (for example because it is used by some other destination) this property does not have any effect. Generally, this property is related to the ABAP System, and should have the same value on all destinations pointing to the same ABAP System. See note 1456826 for backend prerequisites.

Trace configuration options

The following table lists the **trace configuration** options for configuring a destination in the SAP destination data store:

Name	Default Value	Description
trace		Enable/disable RFC trace (**0** or **1**)
cpicTrace		Enable/disable CPIC trace **[0..3]**

270.2.3. Server Configuration

Overview

The configurations for servers are maintained in the **serverDataStore** property of the SAP component. Each entry in this map configures a distinct inbound connection from an SAP instance. The key for each entry is the name of the outbound connection and is used in the **serverName** component of a server endpoint URI as described in the URI format section.

The value for each entry is a *server data configuration object*, **org.fusesource.camel.component.sap.model.rfc.impl.ServerDataImpl**, that defines the configuration of an inbound SAP connection.

Sample server configuration

The following Blueprint XML code shows how to create a sample server configuration with the name, **quickstartServer**.

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
    ...
    <!-- Configures the Inbound and Outbound SAP Connections -->
    <bean id="sap-configuration"
        class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
        <property name="destinationDataStore">
            <map>
                <entry key="quickstartDest" value-ref="quickstartDestinationData" />
            </map>
        </property>
        <property name="serverDataStore">
            <map>
                <entry key="quickstartServer" value-ref="quickstartServerData" />
            </map>
        </property>
    </bean>

    <!-- Configures an Outbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment *** -->
    <bean id="quickstartDestinationData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl">
        <property name="ashost" value="example.com" />
        <property name="sysnr" value="00" />
        <property name="client" value="000" />
        <property name="user" value="username" />
        <property name="passwd" value="passowrd" />
        <property name="lang" value="en" />
    </bean>

    <!-- Configures an Inbound SAP Connection -->
    <!-- *** Please enter the connection property values for your environment ** -->
    <bean id="quickstartServerData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.ServerDataImpl">
        <property name="gwhost" value="example.com" />
        <property name="gwserv" value="3300" />
        <!-- The following property values should not be changed -->
        <property name="progid" value="QUICKSTART" />
        <property name="repositoryDestination" value="quickstartDest" />
        <property name="connectionCount" value="2" />
    </bean>
</blueprint>
```

Notice how this example also configures a destination connection, **quickstartDest**, which the server uses to retrieve meta-data from a remote SAP instance. This destination is configured in the server data through the **repositoryDestination** option. If you do not configure this option, you would need to create

a local meta-data repository instead (see Section 270.2.4, "Repository Configuration").

For example, after configuring the destination as shown in the preceding Blueprint XML file, you could handle the **BAPI_FLCUST_GETLIST** remote function call from an invoking client, using the following URI:

```
sap-srfc-server:quickstartServer:BAPI_FLCUST_GETLIST
```

Required options

The required options for the server data configuration object are, as follows:

Name	Default Value	Description
gwhost		Gateway host on which the server connection should be registered.
gwserv		Gateway service, which is the port on which a registration can be done. In order to resolve the service names **sapgwXXX**, a look-up in **etc/services** is performed by the network layer of the operating system. If using port numbers instead of symbolic service names, no look-ups are performed and no additional entries are needed.
progid		The program ID with which the registration is done. Serves as identifier on the gateway and in the destination in the ABAP system.
repositoryDestination		Specifies a destination name that the server can use in order to retrieve meta-data from a meta-data repository hosted in a remote SAP server.
connectionCount		The number of connections that should be registered at the gateway.

Secure network connection options

The secure network connection options for the server data configuration object are, as follows:

Name	Default Value	Description

sncMode		Secure network connection (SNC) mode, **0** (off) or **1** (on)
sncQop		SNC level of security, **1** to **9**
sncMyname		SNC name of your server. Overrides the default SNC name. Typically something like **p:CN=JCoServer, O=ACompany, C=EN**.
sncLib		Path to library which provides SNC service. If this property is not provided, the value of the **jco.middleware.snc_lib** property is used instead

Other options

The other options for the server data configuration object are, as follows:

Name	Default Value	Description
saprouter		SAP router string to use for a system protected by a firewall, which can therefore only be reached through a SAProuter, when registering the server at the gateway of that ABAP System. A typical router string is **/H/firewall.hostname/H/**
maxStartupDelay		The maximum time (in seconds) between two start-up attempts in case of failures. The waiting time is doubled from initially 1 second after each start-up failure until either the maximum value is reached or the server could be started successfully.
trace		Enable/disable RFC trace (**0** or **1**)
workerThreadCount		The maximum number of threads used by the server connection. If not set, the value for the **connectionCount** is used as the **workerThreadCount**. The maximum number of threads can not exceed 99.

workerThreadMinCount		The minimum number of threads used by server connection. If not set, the value for **connectionCount** is used as the **workerThreadMinCount**.

270.2.4. Repository Configuration

Overview

The configurations for repositories are maintained in the **repositoryDataStore** property of the SAP Component. Each entry in this map configures a distinct repository. The key for each entry is the name of the repository and this key also corresponds to the name of server to which this repository is attached.

The value of each entry is a repository data configuration object, **org.fusesource.camel.component.sap.model.rfc.impl.RepositoryDataImpl**, that defines the contents of a meta-data repository. A repository data object is a map of function template configuration objects, **org.fuesource.camel.component.sap.model.rfc.impl.FunctionTemplateImpl**. Each entry in this map specifies the interface of a function module and the key for each entry is the name of the function module specified.

Repository data example

The following code shows a simple example of configuring a meta-data repository:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ... >
    ...
    <!-- Configures the sap-srfc-server component -->
    <bean id="sap-configuration"
        class="org.fusesource.camel.component.sap.SapConnectionConfiguration">
        <property name="repositoryDataStore">
            <map>
                <entry key="nplServer" value-ref="nplRepositoryData" />
            </map>
        </property>
    </bean>

    <!-- Configures a Meta-Data Repository -->
    <bean id="nplRepositoryData"
        class="org.fusesource.camel.component.sap.model.rfc.impl.RepositoryDataImpl">
        <property name="functionTemplates">
            <map>
                <entry key="BOOK_FLIGHT" value-ref="bookFlightFunctionTemplate" />
            </map>
        </property>
    </bean>
    ...
</blueprint>
```

Function template properties

The interface of a function module consists of four parameter lists by which data is transferred back and forth to the function module in an RFC call. Each parameter list consists of one or more fields, each of which is a named parameter transferred in an RFC call. The following parameter lists and exception list are supported:

- The *import parameter list* contains parameter values that are sent to a function module in an RFC call;
- The *export parameter list* contains parameter values that are returned by a function module in an RFC call;
- The *changing parameter list* contains parameter values that are sent to and returned by a function module in an RFC call;
- The *table parameter list* contains internal table values that are sent to and returned by a function module in an RFC call.
- The interface of a function module also consists of an *exception list* of ABAP exceptions that may be raised when the module is invoked in an RFC call.

A function template describes the name and type of parameters in each parameter list of a function interface and the ABAP exceptions thrown by the function. A function template object maintains five property lists of meta-data objects, as described in the following table.

Property	Description
importParameterList	A list of list field meta-data objects, **org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMeataDataImpl**. Specifies the parameters that are sent in an RFC call to a function module.
changingParameterList	A list of list field meta-data objects, **org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMeataDataImpl**. Specifies the parameters that sent and returned in an RFC call to and from a function module.
exportParameterList	A list of list field meta-data objects, **org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMeataDataImpl**. Specifies the parameters that are returned in an RFC call from a function module.
tableParameterList	A list of list field meta-data objects, **org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMeataDataImpl**. Specifies the table parameters that are sent and returned in an RFC call to and from a function module.

exceptionList	A list of ABAP exception meta-data objects, **org.fusesource.camel.component.sap.model.rfc.impl.AbapExceptionImpl**. Specifies the ABAP exceptions potentially raised in an RFC call of function module.

Function template example

The following example shows an outline of how to configure a function template:

```
<bean id="bookFlightFunctionTemplate"
    class="org.fusesource.camel.component.sap.model.rfc.impl.FunctionTemplateImpl">
    <property name="importParameterList">
        <list>
            ...
        </list>
    </property>
    <property name="changingParameterList">
        <list>
            ...
        </list>
    </property>
    <property name="exportParameterList">
        <list>
            ...
        </list>
    </property>
    <property name="tableParameterList">
        <list>
            ...
        </list>
    </property>
    <property name="exceptionList">
        <list>
            ...
        </list>
    </property>
</bean>
```

List field meta-data properties

A list field meta-data object, **org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMeataDataImpl**, specifies the name and type of a field in a parameter list. For an elementary parameter field (**CHAR**, **DATE**, **BCD**, **TIME**, **BYTE**, **NUM**, **FLOAT**, **INT**, **INT1**, **INT2**, **DECF16**, **DECF34**, **STRING**, **XSTRING**), the following table lists the configuration properties that may be set on a list field meta-data object:

Name	Default Value	Description
name	-	The name of the parameter field.

type	-	The parameter type of the field.
byteLength	-	The field length in bytes for a non-Unicode layout. This value depends on the parameter type. See Section 270.5, "Message Body for RFC".
unicodeByteLength	-	The field length in bytes for a Unicode layout. This value depends on the parameter type. See Section 270.5, "Message Body for RFC".
decimals	**0**	The number of decimals in field value; only required for parameter types BCD and FLOAT. See Section 270.5, "Message Body for RFC".
optional	**false**	If **true**, the field is optional and need not be set in a RFC call

Note that all elementary parameter fields require that the **name**, **type**, **byteLength** and **unicodeByteLength** properties be specified in the field meta-data object. In addition, the **BCD**, **FLOAT**, **DECF16** and **DECF34** fields require the decimal property to be specified in the field meta-data object.

For a complex parameter field of type **TABLE** or **STRUCTURE**, the following table lists the configuration properties that may be set on a list field meta-data object:

Name	Default Value	Description
name	-	The name of the parameter field
type	-	The parameter type of the field
recordMetaData	-	The meta-data for the structure or table. A record meta-data object, **org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl**, is passed to specify the fields in the structure or table rows.
optional	**false**	If **true**, the field is optional and need not be set in a RFC call

Note that all complex parameter fields require that the **name**, **type** and **recordMetaData** properties be specified in the field meta-data object. The value of the **recordMetaData** property is a record field

meta-data object, **org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl**, which specifies the structure of a nested structure or the structure of a table row.

Elementary list field meta-data example

The following meta-data configuration specifies an optional, 24-digit packed BCD number parameter with two decimal places named **TICKET_PRICE**:

```
<bean    class="org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMetaDataImpl">
   <property name="name" value="TICKET_PRICE" />
   <property name="type" value="BCD" />
   <property name="byteLength" value="12" />
   <property name="unicodeByteLength" value="24" />
   <property name="decimals" value="2" />
   <property name="optional" value="true" />
</bean>
```

Complex list field meta-data example

The following meta-data configuration specifies a required **TABLE** parameter named **CONNINFO** with a row structure specified by the **connectionInfo** record meta-data object:

```
<bean    class="org.fusesource.camel.component.sap.model.rfc.impl.ListFieldMetaDataImpl">
   <property name="name" value="CONNINFO" />
   <property name="type" value="TABLE" />
   <property name="recordMetaData" ref="connectionInfo" />
</bean>
```

Record meta-data properties

A record meta-data object, **org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl**, specifies the name and contents of a nested **STRUCTURE** or the row of a **TABLE** parameter. A record meta-data object maintains a list of record field meta data objects, **org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl**, which specify the parameters that reside in the nested structure or table row.

The following table lists configuration properties that may be set on a record meta-data object:

Name	Default Value	Description
name	-	The name of the record.
recordFieldMetaData	-	The list of record field meta-data objects, **org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl**. Specifies the fields contained within the structure.

NOTE

All properties of the record meta-data object are required.

Record meta-data example

The following example shows how to configure a record meta-data object:

```
<bean    id="connectionInfo"
        class="org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl">
    <property name="name" value="CONNECTION_INFO" />
    <property name="recordFieldMetaData">
        <list>
            ...
        </list>
    </property>
</bean>
```

Record field meta-data properties

A record field meta-data object, **org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl**, specifies the name and type of a parameter field withing a structure.

A record field meta-data object is similar to a parameter field meta-data object, except that the offsets of the individual field locations within the nested structure or table row must be additionally specified. The non-Unicode and Unicode offsets of an individual field must be calculated and specified from the sum of non-Unicode and Unicode byte lengths of the preceding fields in the structure or row. Note that failure to properly specify the offsets of fields in nested structures and table rows will cause the field storage of parameters in the underlying JCo and ABAP runtimes to overlap and prevent the proper transfer of values in RFC calls.

For an elementary parameter field (**CHAR**, **DATE**, **BCD**, **TIME**, **BYTE**, **NUM**, **FLOAT**, **INT**, **INT1**, **INT2**, **DECF16**, **DECF34**, **STRING**, **XSTRING**), the following table lists the configuration properties that may be set on a record field meta-data object:

Name	Default Value	Description
name	-	The name of the parameter field
type	-	The parameter type of the field
byteLength	-	The field length in bytes for a non-Unicode layout. This value depends on the parameter type. See Section 270.5, “Message Body for RFC”.
unicodeByteLength	-	The field length in bytes for a Unicode layout. This value depends on the parameter type. See Section 270.5, “Message Body for RFC”.

byteOffset	-	The field offset in bytes for non-Unicode layout. This offset is the byte location of the field within the enclosing structure.
unicodeByteOffset	-	The field offset in bytes for Unicode layout. This offset is the byte location of the field within the enclosing structure.
decimals	**0**	The number of decimals in field value; only required for parameter types **BCD** and **FLOAT**. See Section 270.5, "Message Body for RFC".

For a complex parameter field of type **TABLE** or **STRUCTURE**, the following table lists the configuration properties that may be set on a record field meta-data object:

Name	Default Value	Description
name	-	The name of the parameter field
type	-	The parameter type of the field
byteOffset	-	The field offset in bytes for non-Unicode layout. This offset is the byte location of the field within the enclosing structure.
unicodeByteOffset	-	The field offset in bytes for Unicode layout. This offset is the byte location of the field within the enclosing structure.
recordMetaData	-	The meta-data for the structure or table. A record meta-data object, **org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl**, is passed to specify the fields in the structure or table rows.

Elementary record field meta-data example

The following meta-data configuration specifies a **DATE** field parameter named **ARRDATE** located 85 bytes into the enclosing structure in the case of a non-Unicode layout and located 170 bytes into the enclosing structure in the case of a Unicode layout:

```
<bean    class="org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl">
   <property name="name" value="ARRDATE" />
   <property name="type" value="DATE" />
   <property name="byteLength" value="8" />
   <property name="unicodeByteLength" value="16" />
   <property name="byteOffset" value="85" />
   <property name="unicodeByteOffset" value="170" />
</bean>
```

Complex record field meta-data example

The following meta-data configuration specifies a **STRUCTURE** field parameter named **FLTINFO** with a structure specified by the **flightInfo** record meta-data object. The parameter is located at the beginning of the enclosing structure in both the case of a non-Unicode and Unicode layout.

```
<bean    class="org.fusesource.camel.component.sap.model.rfc.impl.FieldMetaDataImpl">
   <property name="name" value="FLTINFO" />
   <property name="type" value="STRUCTURE" />
   <property name="byteOffset" value="0" />
   <property name="unicodeByteOffset" value="0" />
   <property name="recordMetaData" ref="flightInfo" />
</bean>
```

270.3. MESSAGE HEADERS

The SAP component supports the following message headers:

Header	Description

CamelSap.scheme	The URI scheme of the last endpoint to process the message. Use one of the following values: **sap-srfc-destination** **sap-trfc-destination** **sap-qrfc-destination** **sap-srfc-server** **sap-trfc-server** **sap-idoc-destination** **sap-idoclist-destination** **sap-qidoc-destination** **sap-qidoclist-destination** **sap-idoclist-server**
CamelSap.destinationName	The destination name of the last destination endpoint to process the message.
CamelSap.serverName	The server name of the last server endpoint to process the message.
CamelSap.queueName	The queue name of the last queuing endpoint to process the message.
CamelSap.rfcName	The RFC name of the last RFC endpoint to process the message.
CamelSap.idocType	The IDoc type of the last IDoc endpoint to process the message.
CamelSap.idocTypeExtension	The IDoc type extension, if any, of the last IDoc endpoint to process the message.
CamelSap.systemRelease	The system release, if any, of the last IDoc endpoint to process the message.
CamelSap.applicationRelease	The application release, if any, of the last IDoc endpoint to process the message.

270.4. EXCHANGE PROPERTIES

The SAP component adds the following exchange properties:

Property	Description

CamelSap.destinationPropertiesMap	A map containing the properties of each SAP destination encountered by the exchange. The map is keyed by destination name and each entry is a **java.util.Properties** object containing the configuration properties of that destination.
CamelSap.serverPropertiesMap	A map containing the properties of each SAP server encountered by the exchange. The map is keyed by server name and each entry is a **java.util.Properties** object containing the configuration properties of that server.

270.5. MESSAGE BODY FOR RFC

Request and response objects

An SAP endpoint expects to receive a message with a message body containing an SAP request object and will return a message with a message body containing an SAP response object. SAP requests and responses are fixed map data structures containing named fields with each field having a predefined data type.

Note that the named fields in an SAP request and response are specific to an SAP endpoint, with each endpoint defining the parameters in the SAP request and response it will accept. An SAP endpoint provides factory methods to create the request and response objects that are specific to it.

```
public class SAPEndpoint ... {
    ...
    public Structure getRequest() throws Exception;

    public Structure getResponse() throws Exception;
    ...
}
```

Structure objects

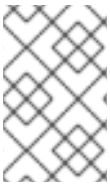
Both SAP request and response objects are represented in Java as a structure object which supports the **org.fusesource.camel.component.sap.model.rfc.Structure** interface. This interface extends both the **java.util.Map** and **org.eclipse.emf.ecore.EObject** interfaces.

```
public interface Structure extends org.eclipse.emf.ecore.EObject,
                                   java.util.Map<String, Object> {

    <T> T get(Object key, Class<T> type);

}
```

The field values in a structure object are accessed through the field's getter methods in the map interface. In addition, the structure interface provides a type-restricted method to retrieve field values.

Structure objects are implemented in the component runtime using the Eclipse Modeling Framework (EMF) and support that framework's **EObject** interface. Instances of a structure object have attached meta-data which define and restrict the structure and contents of the map of fields it provides. This

meta-data can be accessed and introspected using the standard methods provided by EMF. Please refer to the EMF documentation for further details.



NOTE

Attempts to get a parameter not defined on a structure object will return null. Attempts to set a parameter not defined on a structure will throw an exception as well as attempts to set the value of a parameter with an incorrect type.

As discussed in the following sections, structure objects can contain fields that contain values of the complex field types, **STRUCTURE** and **TABLE**. Note that it is unnecessary to create instances of these types and add them to the structure. Instances of these field values are created on demand if necessary when accessed in the enclosing structure.

Field types

The fields that reside within the structure object of an SAP request or response may be either *elementary* or *complex*. An elementary field contains a single scalar value, whereas a complex field will contain one or more fields of either a elementary or complex type.

Elementary field types

An elementary field may be either a character, numeric, hexadecimal or string field type. The following table summarizes the types of elementary fields that may reside in a structure object:

Field Type	Corresponding Java Type	Byte Length	Unicode Byte Length	Number Decimals Digits	Description
CHAR	**java.lang.String**	1 to 65535	1 to 65535	-	ABAP Type 'C': Fixed sized character string
DATE	**java.util.Date**	8	16	-	ABAP Type 'D': Date (format: YYYYMMDD)
BCD	**java.math.BigDecimal**	1 to 16	1 to 16	0 to 14	ABAP Type 'P': Packed BCD number. A BCD number contains two digits per byte.
TIME	**java.util.Date**	6	12	-	ABAP Type 'T': Time (format: HHMMSS)
BYTE	**byte[]**	1 to 65535	1 to 65535	-	ABAP Type 'X':Fixed sized byte array

NUM	**java.lang.String**	1 to 65535	1 to 65535	-	ABAP Type 'N': Fixed sized numeric character string
FLOAT	**java.lang.Double**	8	8	0 to 15	ABAP Type 'F': Floating point number
INT	**java.lang.Integer**	4	4	-	ABAP Type 'I': 4-byte Integer
INT2	**java.lang.Integer**	2	2	-	ABAP Type 'S': 2-byte Integer
INT1	**java.lang.Integer**	1	1	-	ABAP Type 'B': 1-byte Integer
DECF16	**java.match.BigDecimal**	8	8	16	ABAP Type 'decfloat16': 8-byte Decimal Floating Point Number
DECF34	**java.math.BigDecimal**	16	16	34	ABAP Type 'decfloat34': 16-byte Decimal Floating Point Number
STRING	**java.lang.String**	8	8	-	ABAP Type 'G': Variable length character string
XSTRING	**byte[]**	8	8	-	ABAP Type 'Y': Variable length byte array

Character field types

A character field contains a fixed sized character string that may use either a non-Unicode or Unicode character encoding in the underlying JCo and ABAP runtimes. Non-Unicode character strings encode one character per byte. Unicode characters strings are encoded in two bytes using UTF-16 encoding. Character field values are represented in Java as **java.lang.String** objects and the underlying JCo runtime is responsible for the conversion to their ABAP representation.

A character field declares its field length in its associated **byteLength** and **unicodeByteLength** properties, which determine the length of the field's character string in each encoding system.

CHAR

A **CHAR** character field is a text field containing alphanumeric characters and corresponds to the ABAP type C.

NUM

A **NUM** character field is a numeric text field containing numeric characters only and corresponds to the ABAP type N.

DATE

A **DATE** character field is an 8 character date field with the year, month and day formatted as **YYYYMMDD** and corresponds to the ABAP type D.

TIME

A **TIME** character field is a 6 character time field with the hours, minutes and seconds formatted as **HHMMSS** and corresponds to the ABAP type T.

Numeric field types

A numeric field contains a number. The following numeric field types are supported:

INT

An **INT** numeric field is an integer field stored as a 4-byte integer value in the underlying JCo and ABAP runtimes and corresponds to the ABAP type I. An **INT** field value is represented in Java as a **java.lang.Integer** object.

INT2

An **INT2** numeric field is an integer field stored as a 2-byte integer value in the underlying JCo and ABAP runtimes and corresponds to the ABAP type S. An **INT2** field value is represented in Java as a **java.lang.Integer** object.

INT1

An **INT1** field is an integer field stored as a 1-byte integer value in the underlying JCo and ABAP runtimes value and corresponds to the ABAP type B. An **INT1** field value is represented in Java as a **java.lang.Integer** object.

FLOAT

A **FLOAT** field is a binary floating point number field stored as an 8-byte double value in the underlying JCo and ABAP runtimes and corresponds to the ABAP type F. A **FLOAT** field declares the number of decimal digits that the field's value contains in its associated decimal property. In the case of a **FLOAT** field, this decimal property can have a value between 1 and 15 digits. A **FLOAT** field value is represented in Java as a **java.lang.Double** object.

BCD

A **BCD** field is a binary coded decimal field stored as a 1 to 16 byte packed number in the underlying JCo and ABAP runtimes and corresponds to the ABAP type P. A packed number stores two decimal digits per byte. A **BCD** field declares its field length in its associated **byteLength** and **unicodeByteLength** properties. In the case of a **BCD** field, these properties can have a value between 1 and 16 bytes and both properties will have the same value. A **BCD** field declares the number of decimal digits that the field's value contains in its associated decimal property. In the case of a **BCD** field, this decimal property can have a value between 1 and 14 digits. A **BCD** field value is represented in Java as a **java.math.BigDecimal**.

DECF16

A **DECF16** field is a decimal floating point stored as an 8-byte IEEE 754 decimal64 floating point value in the underlying JCo and ABAP runtimes and corresponds to the ABAP type **decfloat16**. The value of a **DECF16** field has 16 decimal digits. The value of a **DECF16** field is represented in Java as **java.math.BigDecimal**.

DECF34

A **DECF34** field is a decimal floating point stored as a 16-byte IEEE 754 decimal128 floating point value in the underlying JCo and ABAP runtimes and corresponds to the ABAP type **decfloat34**. The value of a **DECF34** field has 34 decimal digits. The value of a **DECF34** field is represented in Java as **java.math.BigDecimal**.

Hexadecimal field types

A hexadecimal field contains raw binary data. The following hexadecimal field types are supported:

BYTE

A **BYTE** field is a fixed sized byte string stored as a byte array in the underlying JCo and ABAP runtimes and corresponds to the ABAP type X. A **BYTE** field declares its field length in its associated **byteLength** and **unicodeByteLength** properties. In the case of a **BYTE** field, these properties can have a value between 1 and 65535 bytes and both properties will have the same value. The value of a **BYTE** field is represented in Java as a **byte[]** object.

String field types

A string field references a variable length string value. The length of that string value is not fixed until runtime. The storage for the string value is dynamically created in the underlying JCo and ABAP runtimes. The storage for the string field itself is fixed and contains only a string header.

STRING

A **STRING** field refers to a character string and is stored in the underlying JCo and ABAP runtimes as an 8-byte value. It corresponds to the ABAP type G. The value of the **STRING** field is represented in Java as a **java.lang.String** object.

XSTRING

An **XSTRING** field refers to a byte string and is stored in the underlying JCo and ABAP runtimes as an 8-byte value. It corresponds to the ABAP type Y. The value of the **STRING** field is represented in Java as a **byte[]** object.

Complex field types

A complex field may be either a structure or table field type. The following table summarizes these complex field types.

Field Type	Corresponding Java Type	Byte Length	Unicode Byte Length	Number Decimals Digits	Description
STRUCTURE	**org.fusesource.camel.component.sap.model.rfc.Structure**	Total of individual field byte lengths	Total of individual field Unicode byte lengths	-	ABAP Type 'u' & 'v': Heterogeneous Structure

TABLE	**org.fusesource.camel.component.sap.model.rfc.Table**	Byte length of row structure	Unicode byte length of row structure	-	ABAP Type 'h': Table

Structure field types

A **STRUCTURE** field contains a structure object and is stored in the underlying JCo and ABAP runtimes as an ABAP structure record. It corresponds to either an ABAP type **u** or **v**. The value of a **STRUCTURE** field is represented in Java as a structure object with the interface **org.fusesource.camel.component.sap.model.rfc.Structure**.

Table field types

A **TABLE** field contains a table object and is stored in the underlying JCo and ABAP runtimes as an ABAP internal table. It corresponds to the ABAP type **h**. The value of the field is represented in Java by a table object with the interface **org.fusesource.camel.component.sap.model.rfc.Table**.

Table objects

A table object is a homogeneous list data structure containing rows of structure objects with the same structure. This interface extends both the **java.util.List** and **org.eclipse.emf.ecore.EObject** interfaces.

```
public interface Table<S extends Structure>
    extends org.eclipse.emf.ecore.EObject,
    java.util.List<S> {

    /**
     * Creates and adds table row at end of row list
     */
    S add();

    /**
     * Creates and adds table row at index in row list
     */
    S add(int index);

}
```

The list of rows in a table object are accessed and managed using the standard methods defined in the list interface. In addition the table interface provides two factory methods for creating and adding structure objects to the row list.

Table objects are implemented in the component runtime using the Eclipse Modeling Framework (EMF) and support that framework's EObject interface. Instances of a table object have attached meta-data which define and restrict the structure and contents of the rows it provides. This meta-data can be accessed and introspected using the standard methods provided by EMF. Please refer to the EMF documentation for further details.

NOTE

Attempts to add or set a row structure value of the wrong type will throw an exception.

270.6. MESSAGE BODY FOR IDOC

IDoc message type

When using one of the IDoc Camel SAP endpoints, the type of the message body depends on which particular endpoint you are using.

For a **sap-idoc-destination** endpoint or a **sap-qidoc-destination** endpoint, the message body is of **Document** type:

```
org.fusesource.camel.component.sap.model.idoc.Document
```

For a **sap-idoclist-destination** endpoint, a **sap-qidoclist-destination** endpoint, or a **sap-idoclist-server** endpoint, the message body is of **DocumentList** type:

```
org.fusesource.camel.component.sap.model.idoc.DocumentList
```

The IDoc document model

For the Camel SAP component, an IDoc document is modelled using the Eclipse Modelling Framework (EMF), which provides a wrapper API around the underlying SAP IDoc API. The most important types in this model are:

```
org.fusesource.camel.component.sap.model.idoc.Document
org.fusesource.camel.component.sap.model.idoc.Segment
```

The **Document** type represents an IDoc document instance. In outline, the **Document** interface exposes the following methods:

```
// Java
package org.fusesource.camel.component.sap.model.idoc;
...
public interface Document extends EObject {
    // Access the field values from the IDoc control record
    String getArchiveKey();
    void setArchiveKey(String value);
    String getClient();
    void setClient(String value);
    ...

    // Access the IDoc document contents
    Segment getRootSegment();
}
```

The following kinds of method are exposed by the **Document** interface:

Methods for accessing the control record

Most of the methods are for accessing or modifying field values of the IDoc control record. These methods are of the form *AttributeName*, *AttributeName*, where *AttributeName* is the name of a field value (see Table 270.2, "IDoc Document Attributes").

Method for accessing the document contents

The **getRootSegment** method provides access to the document contents (IDoc data records), returning the contents as a **Segment** object. Each **Segment** object can contain an arbitrary number of child segments, and the segments can be nested to an arbitrary degree.
Note, however, that the precise layout of the segment hierarchy is defined by the particular *IDoc type* of the document. When creating (or reading) a segment hierarchy, therefore, you must be sure to follow the exact structure as defined by the IDoc type.

The **Segment** type is used to access the data records of the IDoc document, where the segments are laid out in accordance with the structure defined by the document's IDoc type. In outline, the **Segment** interface exposes the following methods:

```
// Java
package org.fusesource.camel.component.sap.model.idoc;
...
public interface Segment extends EObject, java.util.Map<String, Object> {
   // Returns the value of the '<em><b>Parent</b></em>' reference.
   Segment getParent();

   // Return a immutable list of all child segments
   <S extends Segment> EList<S> getChildren();

   // Returns a list of child segments of the specified segment type.
   <S extends Segment> SegmentList<S> getChildren(String segmentType);

   EList<String> getTypes();

   Document getDocument();

   String getDescription();

   String getType();

   String getDefinition();

   int getHierarchyLevel();

   String getIdocType();

   String getIdocTypeExtension();

   String getSystemRelease();

   String getApplicationRelease();

   int getNumFields();

   long getMaxOccurrence();

   long getMinOccurrence();

   boolean isMandatory();

   boolean isQualified();

   int getRecordLength();
```

```
    <T> T get(Object key, Class<T> type);
}
```

The **getChildren(String segmentType)** method is particularly useful for adding new (nested) children to a segment. It returns an object of type, **SegmentList**, which is defined as follows:

```
// Java
package org.fusesource.camel.component.sap.model.idoc;
...
public interface SegmentList<S extends Segment> extends EObject, EList<S> {
    S add();

    S add(int index);
}
```

Hence, to create a data record of **E1SCU_CRE** type, you could use Java code like the following:

```
Segment rootSegment = document.getRootSegment();

Segment E1SCU_CRE_Segment = rootSegment.getChildren("E1SCU_CRE").add();
```

How an IDoc is related to a Document object

According to the SAP documentation, an IDoc document consists of the following main parts:

Control record

The control record (which contains the meta-data for the IDoc document) is represented by the attributes on the **Document** object – see Table 270.2, "IDoc Document Attributes" for details.

Data records

The data records are represented by the **Segment** objects, which are constructed as a nested hierarchy of segments. You can access the root segment through the **Document.getRootSegment** method.

Status records

In the Camel SAP component, the status records are **not** represented by the document model. But you do have access to the latest status value through the **status** attribute on the control record.

Example of creating a Document instance

For example, Example 270.1, "Creating an IDoc Document in Java" shows how to create an IDoc document with the IDoc type, **FLCUSTOMER_CREATEFROMDATA01**, using the IDoc model API in Java.

Example 270.1. Creating an IDoc Document in Java

```
// Java
import org.fusesource.camel.component.sap.model.idoc.Document;
import org.fusesource.camel.component.sap.model.idoc.Segment;
import org.fusesource.camel.component.sap.util.IDocUtil;

import org.fusesource.camel.component.sap.model.idoc.Document;
import org.fusesource.camel.component.sap.model.idoc.DocumentList;
```

```
import org.fusesource.camel.component.sap.model.idoc.IdocFactory;
import org.fusesource.camel.component.sap.model.idoc.IdocPackage;
import org.fusesource.camel.component.sap.model.idoc.Segment;
import org.fusesource.camel.component.sap.model.idoc.SegmentChildren;
...
//
// Create a new IDoc instance using the modelling classes
//

// Get the SAP Endpoint bean from the Camel context.
// In this example, it's a 'sap-idoc-destination' endpoint.
SapTransactionalIDocDestinationEndpoint endpoint =
    exchange.getContext().getEndpoint(
        "bean:SapEndpointBeanID",
        SapTransactionalIDocDestinationEndpoint.class
    );

// The endpoint automatically populates some required control record attributes
Document document = endpoint.createDocument()

// Initialize additional control record attributes
document.setMessageType("FLCUSTOMER_CREATEFROMDATA");
document.setRecipientPartnerNumber("QUICKCLNT");
document.setRecipientPartnerType("LS");
document.setSenderPartnerNumber("QUICKSTART");
document.setSenderPartnerType("LS");

Segment rootSegment = document.getRootSegment();

Segment E1SCU_CRE_Segment = rootSegment.getChildren("E1SCU_CRE").add();

Segment E1BPSCUNEW_Segment =
E1SCU_CRE_Segment.getChildren("E1BPSCUNEW").add();
E1BPSCUNEW_Segment.put("CUSTNAME", "Fred Flintstone");
E1BPSCUNEW_Segment.put("FORM", "Mr.");
E1BPSCUNEW_Segment.put("STREET", "123 Rubble Lane");
E1BPSCUNEW_Segment.put("POSTCODE", "01234");
E1BPSCUNEW_Segment.put("CITY", "Bedrock");
E1BPSCUNEW_Segment.put("COUNTR", "US");
E1BPSCUNEW_Segment.put("PHONE", "800-555-1212");
E1BPSCUNEW_Segment.put("EMAIL", "fred@bedrock.com");
E1BPSCUNEW_Segment.put("CUSTTYPE", "P");
E1BPSCUNEW_Segment.put("DISCOUNT", "005");
E1BPSCUNEW_Segment.put("LANGU", "E");
```

Document attributes

Table 270.2, "IDoc Document Attributes" shows the control record attributes that you can set on the **Document** object.

Table 270.2. IDoc Document Attributes

Attribute	Length	SAP Field	Description
archiveKey	70	**ARCKEY**	EDI archive key
client	3	**MANDT**	Client
creationDate	8	**CREDAT**	Date IDoc was created
creationTime	6	**CRETIM**	Time IDoc was created
direction	1	**DIRECT**	Direction
eDIMessage	14	**REFMES**	Reference to message
eDIMessageGroup	14	**REFGRP**	Reference to message group
eDIMessageType	6	**STDMES**	EDI message type
eDIStandardFlag	1	**STD**	EDI standard
eDIStandardVersion	6	**STDVRS**	Version of EDI standard
eDITransmissionFile	14	**REFINT**	Reference to interchange file
iDocCompoundType	8	**DOCTYP**	IDoc type
iDocNumber	16	**DOCNUM**	IDoc number
iDocSAPRelease	4	**DOCREL**	SAP Release of IDoc
iDocType	30	**IDOCTP**	Name of basic IDoc type
iDocTypeExtension	30	**CIMTYP**	Name of extension type
messageCode	3	**MESCOD**	Logical message code
messageFunction	3	**MESFCT**	Logical message function
messageType	30	**MESTYP**	Logical message type
outputMode	1	**OUTMOD**	Output mode
recipientAddress	10	**RCVSAD**	Receiver address (SADR)

Attribute	Length	SAP Field	Description
recipientLogicalAddress	70	**RCVLAD**	Logical address of receiver
recipientPartnerFunction	2	**RCVPFC**	Partner function of receiver
recipientPartnerNumber	10	**RCVPRN**	Partner number of receiver
recipientPartnerType	2	**RCVPRT**	Partner type of receiver
recipientPort	10	**RCVPOR**	Receiver port (SAP System, EDI subsystem)
senderAddress		**SNDSAD**	Sender address (SADR)
senderLogicalAddress	70	**SNDLAD**	Logical address of sender
senderPartnerFunction	2	**SNDPFC**	Partner function of sender
senderPartnerNumber	10	**SNDPRN**	Partner number of sender
senderPartnerType	2	**SNDPRT**	Partner type of sender
senderPort	10	**SNDPOR**	Sender port (SAP System, EDI subsystem)
serialization	20	**SERIAL**	EDI/ALE: Serialization field
status	2	**STATUS**	Status of IDoc
testFlag	1	**TEST**	Test flag

Setting document attributes in Java

When setting the control record attributes in Java (from Table 270.2, "IDoc Document Attributes"), the usual convention for Java bean properties is followed. That is, a **name** attribute can be accessed through the **getName** and **setName** methods, for getting and setting the attribute value. For example, the **iDocType**, **iDocTypeExtension**, and **messageType** attributes can be set as follows on a **Document** object:

```
// Java
document.setIDocType("FLCUSTOMER_CREATEFROMDATA01");
```

```
document.setIDocTypeExtension("");
document.setMessageType("FLCUSTOMER_CREATEFROMDATA");
```

Setting document attributes in XML

When setting the control record attributes in XML, the attributes must be set on the **idoc:Document** element. For example, the **iDocType**, **iDocTypeExtension**, and **messageType** attributes can be set as follows:

```
<?xml version="1.0" encoding="ASCII"?>
<idoc:Document ...
        iDocType="FLCUSTOMER_CREATEFROMDATA01"
        iDocTypeExtension=""
        messageType="FLCUSTOMER_CREATEFROMDATA" ... >
  ...
</idoc:Document>
```

270.7. TRANSACTION SUPPORT

BAPI transaction model

The SAP Component supports the BAPI transaction model for outbound communication with SAP. A destination endpoint with a URL containing the transacted option set to **true** will, if necessary, initiate a stateful session on the outbound connection of the endpoint and register a Camel Synchronization object with the exchange. This synchronization object will call the BAPI service method **BAPI_TRANSACTION_COMMIT** and end the stateful session when the processing of the message exchange is complete. If the processing of the message exchange fails, the synchronization object will call the BAPI server method **BAPI_TRANSACTION_ROLLBACK** and end the stateful session.

RFC transaction model

The tRFC protocol accomplishes an AT-MOST-ONCE delivery and processing guarantee by identifying each transactional request with a unique transaction identifier (TID). A TID accompanies each request sent in the protocol. A sending application using the tRFC protocol must identify each instance of a request with a unique TID when sending the request. An application may send a request with a given TID multiple times, but the protocol ensures that the request is delivered and processed in the receiving system at most once. An application may choose to resend a request with a given TID when encountering a communication or system error when sending the request, and is thus in doubt as to whether that request was delivered and processed in the receiving system. By resending a request when encountering an communication error, a client application using the tRFC protocol can thus ensure EXACTLY-ONCE delivery and processing guarantees for its request.

Which transaction model to use?

A BAPI transaction is an application level transaction, in the sense that it imposes ACID guarantees on the persistent data changes performed by a BAPI method or RFC function in the SAP database. An RFC transaction is a communication transaction, in the sense that it imposes delivery guarantees (AT-MOST-ONCE, EXACTLY-ONCE, EXACTLY-ONCE-IN-ORDER) on requests to a BAPI method and/or RFC function.

Transactional RFC destination endpoints

The following destination endpoints support RFC transactions:

- **sap-trfc-destination**
- **sap-qrfc-destination**

A single Camel route can include multiple transactional RFC destination endpoints, sending messages to multiple RFC destinations and even sending messages to the same RFC destination multiple times. This implies that the Camel SAP component potentially needs to keep track of **many** transaction IDs (TIDs) for each **Exchange** object passing along a route. Now if the route processing fails and must be retried, the situation gets quite complicated. The RFC transaction semantics demand that each RFC destination along the route must be invoked using the **same** TID that was used the first time around (and where the TIDs for each of the destinations are distinct from each other). In other words, the Camel SAP component must keep track of which TID was used at which point along the route, and remember this information, so that the TIDs can be replayed in the correct order.

By default, Camel does not provide a mechanism that enables an **Exchange** to know where it is in a route. To provide such a mechanism, it is necessary to install the **CurrentProcessorDefinitionInterceptStrategy** interceptor into the Camel runtime. This interceptor must be installed into the Camel runtime, in order for the Camel SAP component to keep track of the TIDs in a route. For details of how to configure the interceptor, see the section called “Interceptor for tRFC and qRFC destinations”.

Transactional RFC server endpoints

The following server endpoints support RFC transactions:

- **sap-trfc-server**

When a Camel exchange processing a transactional request encounters a processing error, Camel handles the processing error through its standard error handling mechanisms. If the Camel route processing the exchange is configured to propagate the error back to the caller, the SAP server endpoint that initiated the exchange takes note of the failure and the sending SAP system is notified of the error. The sending SAP system can then respond by sending another transaction request with the same TID to process the request again.

270.8. XML SERIALIZATION FOR RFC

Overview

SAP request and response objects support an XML serialization format which enable these objects to be serialized to and from an XML document.

XML namespace

Each RFC in a repository defines a specific XML name space for the elements which compose the serialized forms of its Request and Response objects. The form of this namespace URL is as follows:

```
http://sap.fusesource.org/rfc/<Repository Name>/<RFC Name>
```

RFC namespace URLs have a common **http://sap.fusesource.org/rfc** prefix followed by the name of the repository in which the RFC’s metadata is defined. The final component in the URL is the name of the RFC itself.

Request and response XML documents

An SAP request object will be serialized into an XML document with the root element of that document named Request and scoped by the namespace of the request's RFC.

```
<?xml version="1.0" encoding="ASCII"?>
<BOOK_FLIGHT:Request
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
   ...
</BOOK_FLIGHT:Request>
```

An SAP response object will be serialized into an XML document with the root element of that document named Response and scoped by the namespace of the response's RFC.

```
<?xml version="1.0" encoding="ASCII"?>
<BOOK_FLIGHT:Response
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
   ...
</BOOK_FLIGHT:Response>
```

Structure fields

Structure fields in parameter lists or nested structures are serialized as elements. The element name of the serialized structure corresponds to the field name of the structure within the enclosing parameter list, structure or table row entry it resides.

```
<BOOK_FLIGHT:FLTINFO
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
   ...
</BOOK_FLIGHT:FLTINFO>
```

Note that the type name of the structure element in the RFC namespace will correspond to the name of the record meta data object which defines the structure, as in the following example:

```
<xs:schema
    targetNamespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
   ...
   <xs:complexType name="FLTINFO_STRUCTURE”>
   ...
   </xs:complexType>
   ...
</xs:schema>
```

This distinction will be important when specifying a JAXB bean to marshal and unmarshal the structure as will be seen in Section 270.12, “Example 3: Handling Requests from SAP” .

Table fields

Table fields in parameter lists or nested structures are serialized as elements. The element name of the serialized structure will correspond to the field name of the table within the enclosing parameter list, structure, or table row entry it resides. The table element will contain a series of row elements to hold the serialized values of the table's row entries.

```
<BOOK_FLIGHT:CONNINFO
```

```
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT">
  <row ... > ... </row>
  ...
  <row ... > ... </row>
</BOOK_FLIGHT:CONNINFO>
```

Note that the type name of the table element in the RFC namespace will correspond to the name of the record meta data object which defines the row structure of the table suffixed by **_TABLE**. The type name of the table row element in the RFC name corresponds to the name of the record meta data object which defines the row structure of the table, as in the following example:

```
<xs:schema
    targetNamespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
  <xs:complextType name="CONNECTION_INFO_STRUCTURE_TABLE”>
     <xs:sequence>
        <xs:element
           name="row”
           minOccures="0”
           maxOccurs="unbounded”
           type="CONNECTION_INFO_STRUCTURE”/>
        ...
        <xs:sequence>
     </xs:sequence>
  </xs:complexType>

  <xs:complextType name="CONNECTION_INFO_STRUCTURE”>
        ...
  </xs:complexType>
  ...
</xs:schema>
```

This distinction will be important when specifying a JAXB bean to marshal and unmarshal the structure as will be seen in Section 270.12, “Example 3: Handling Requests from SAP” .

Elementary fields

Elementary fields in parameter lists or nested structures are serialized as attributes on the element of the enclosing parameter list or structure. The attribute name of the serialized field corresponds to the field name of the field within the enclosing parameter list, structure, or table row entry it resides, as in the following example:

```
<?xml version="1.0" encoding="ASCII"?>
<BOOK_FLIGHT:Request
    xmlns:BOOK_FLIGHT="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT"
    CUSTNAME="James Legrand"
  PASSFORM="Mr"
  PASSNAME="Travelin Joe"
  PASSBIRTH="1990-03-17T00:00:00.000-0500"
  FLIGHTDATE="2014-03-19T00:00:00.000-0400"
  TRAVELAGENCYNUMBER="00000110"
  DESTINATION_FROM="SFO"
  DESTINATION_TO="FRA"/>
```

Date and time formats

Date and Time fields are serialized into attribute values using the following format:

```
yyyy-MM-dd'T'HH:mm:ss.SSSZ
```

Date fields will be serialized with only the year, month, day and timezone components set:

```
DEPDATE="2014-03-19T00:00:00.000-0400"
```

Time fields will be serialized with only the hour, minute, second, millisecond and timezone components set:

```
DEPTIME="1970-01-01T16:00:00.000-0500"
```

270.9. XML SERIALIZATION FOR IDOC

Overview

An IDoc message body can be serialized into an XML string format, with the help of a built-in type converter.

XML namespace

Each serialized IDoc is associated with an XML namespace, which has the following general format:

```
http://sap.fusesource.org/idoc/repositoryName/idocType/idocTypeExtension/systemRelease/applicationRelease
```

Both the *repositoryName* (name of the remote SAP meta-data repository) and the *idocType* (IDoc document type) are mandatory, but the other components of the namespace can be left blank. For example, you could have an XML namespace like the following:

```
http://sap.fusesource.org/idoc/MY_REPO/FLCUSTOMER_CREATEFROMDATA01///
```

Built-in type converter

The Camel SAP component has a built-in type converter, which is capable of converting a **Document** object or a **DocumentList** object to and from a **String** type.

For example, to serialize a **Document** object to an XML string, you can simply add the following line to a route in XML DSL:

```
<convertBodyTo type="java.lang.String"/>
```

You can also use this approach to a serialized XML message into a **Document** object. For example, given that the current message body is a serialized XML string, you can convert it back into a **Document** object by adding the following line to a route in XML DSL:

```
<convertBodyTo type="org.fusesource.camel.component.sap.model.idoc.Document"/>
```

Sample IDoc message body in XML format

When you convert an IDoc message to a **String**, it is serialized into an XML document, where the root element is either **idoc:Document** (for a single document) or **idoc:DocumentList** (for a list of documents). Example 270.2, "IDoc Message Body in XML" shows a single IDoc document that has been serialized to an **idoc:Document** element.

Example 270.2. IDoc Message Body in XML

```
<?xml version="1.0" encoding="ASCII"?>
<idoc:Document
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:FLCUSTOMER_CREATEFROMDATA01---
="http://sap.fusesource.org/idoc/XXX/FLCUSTOMER_CREATEFROMDATA01///"
    xmlns:idoc="http://sap.fusesource.org/idoc"
    creationDate="2015-01-28T12:39:13.980-0500"
    creationTime="2015-01-28T12:39:13.980-0500"
    iDocType="FLCUSTOMER_CREATEFROMDATA01"
    iDocTypeExtension=""
    messageType="FLCUSTOMER_CREATEFROMDATA"
    recipientPartnerNumber="QUICKCLNT"
    recipientPartnerType="LS"
    senderPartnerNumber="QUICKSTART"
    senderPartnerType="LS">
  <rootSegment xsi:type="FLCUSTOMER_CREATEFROMDATA01---:ROOT" document="/">
    <segmentChildren parent="//@rootSegment">
      <E1SCU_CRE parent="//@rootSegment" document="/">
        <segmentChildren parent="//@rootSegment/@segmentChildren/@E1SCU_CRE.0">
          <E1BPSCUNEW parent="//@rootSegment/@segmentChildren/@E1SCU_CRE.0"
              document="/"
              CUSTNAME="Fred Flintstone" FORM="Mr."
              STREET="123 Rubble Lane"
              POSTCODE="01234"
              CITY="Bedrock"
              COUNTR="US"
              PHONE="800-555-1212"
              EMAIL="fred@bedrock.com"
              CUSTTYPE="P"
              DISCOUNT="005"
              LANGU="E"/>
        </segmentChildren>
      </E1SCU_CRE>
    </segmentChildren>
  </rootSegment>
</idoc:Document>
```

270.10. EXAMPLE 1: READING DATA FROM SAP

Overview

This example demonstrates a route which reads **FlightCustomer** business object data from SAP. The route invokes the **FlightCustomer** BAPI method, **BAPI_FLCUST_GETLIST**, using an SAP synchronous RFC destination endpoint to retrieve the data.

Java DSL for route

The Java DSL for the example route is as follows:

```
from("direct:getFlightCustomerInfo")
    .to("bean:createFlightCustomerGetListRequest")
    .to("sap-srfc-destination:nplDest:BAPI_FLCUST_GETLIST")
    .to("bean:returnFlightCustomerInfo");
```

XML DSL for route

And the Spring DSL for the same route is as follows:

```
<route>
    <from uri="direct:getFlightCustomerInfo"/>
    <to uri="bean:createFlightCustomerGetListRequest"/>
    <to uri="sap-srfc-destination:nplDest:BAPI_FLCUST_GETLIST"/>
    <to uri="bean:returnFlightCustomerInfo"/>
</route>
```

createFlightCustomerGetListRequest bean

The **createFlightCustomerGetListRequest** bean is responsible for building an SAP request object in its exchange method that is used in the RFC call of the subsequent SAP endpoint . The following code snippet demonstrates the sequence of operations to build the request object:

```
public void create(Exchange exchange) throws Exception {

    // Get SAP Endpoint to be called from context.
    SapSynchronousRfcDestinationEndpoint endpoint =
        exchange.getContext().getEndpoint("sap-srfc-destination:nplDest:BAPI_FLCUST_GETLIST",
                                          SapSynchronousRfcDestinationEndpoint.class);

    // Retrieve bean from message containing Flight Customer name to
    // look up.
    BookFlightRequest bookFlightRequest =
        exchange.getIn().getBody(BookFlightRequest.class);

    // Create SAP Request object from target endpoint.
    Structure request = endpoint.getRequest();

    // Add Customer Name to request if set
    if (bookFlightRequest.getCustomerName() != null &&
        bookFlightRequest.getCustomerName().length() > 0) {
            request.put("CUSTOMER_NAME",
                        bookFlightRequest.getCustomerName());
        }
    } else {
        throw new Exception("No Customer Name");
    }

    // Put request object into body of exchange message.
    exchange.getIn().setBody(request);
}
```

returnFlightCustomerInfo bean

The **returnFlightCustomerInfo** bean is responsible for extracting data from the SAP response object in its exchange method that it receives from the previous SAP endpoint . The following code snippet demonstrates the sequence of operations to extract the data from the response object:

```
public void createFlightCustomerInfo(Exchange exchange) throws Exception {

    // Retrieve SAP response object from body of exchange message.
    Structure flightCustomerGetListResponse =
        exchange.getIn().getBody(Structure.class);

    if (flightCustomerGetListResponse == null) {
        throw new Exception("No Flight Customer Get List Response");
    }

    // Check BAPI return parameter for errors
    @SuppressWarnings("unchecked")
    Table<Structure> bapiReturn =
        flightCustomerGetListResponse.get("RETURN", Table.class);
    Structure bapiReturnEntry = bapiReturn.get(0);
    if (bapiReturnEntry.get("TYPE", String.class) != "S") {
        String message = bapiReturnEntry.get("MESSAGE", String.class);
        throw new Exception("BAPI call failed: " + message);
    }

    // Get customer list table from response object.
    @SuppressWarnings("unchecked")
    Table<? extends Structure> customerList =
        flightCustomerGetListResponse.get("CUSTOMER_LIST", Table.class);

    if (customerList == null || customerList.size() == 0) {
        throw new Exception("No Customer Info.");
    }

    // Get Flight Customer data from first row of table.
    Structure customer = customerList.get(0);

    // Create bean to hold Flight Customer data.
    FlightCustomerInfo flightCustomerInfo = new FlightCustomerInfo();

    // Get customer id from Flight Customer data and add to bean.
    String customerId = customer.get("CUSTOMERID", String.class);
    if (customerId != null) {
        flightCustomerInfo.setCustomerNumber(customerId);
    }

    ...

    // Put bean into body of exchange message.
    exchange.getIn().setHeader("flightCustomerInfo", flightCustomerInfo);

}
```

270.11. EXAMPLE 2: WRITING DATA TO SAP

Overview

This example demonstrates a route which creates a **FlightTrip** business object instance in SAP. The route invokes the **FlightTrip** BAPI method, **BAPI_FLTRIP_CREATE**, using a destination endpoint to create the object.

Java DSL for route

The Java DSL for the example route is as follows:

```
from("direct:createFlightTrip")
    .to("bean:createFlightTripRequest")
    .to("sap-srfc-destination:nplDest:BAPI_FLTRIP_CREATE?transacted=true")
    .to("bean:returnFlightTripResponse");
```

XML DSL for route

And the Spring DSL for the same route is as follows:

```
<route>
    <from uri="direct:createFlightTrip"/>
    <to uri="bean:createFlightTripRequest"/>
    <to uri="sap-srfc-destination:nplDest:BAPI_FLTRIP_CREATE?transacted=true"/>
    <to uri="bean:returnFlightTripResponse"/>
</route>
```

Transaction support

Note that the URL for the SAP endpoint has the **transacted** option set to **true**. As discussed in Section 270.7, "Transaction Support", when this option is enabled the endpoint ensures that an SAP transaction session has been initiated before invoking the RFC call. Because this endpoint's RFC creates new data in SAP, this options is necessary to make the route's changes permanent in SAP.

Populating request parameters

The **createFlightTripRequest** and **returnFlightTripResponse** beans are responsible for populating request parameters into the SAP request and extracting response parameters from the SAP response respectively following the same sequence of operations as demonstrated in the previous example.

270.12. EXAMPLE 3: HANDLING REQUESTS FROM SAP

Overview

This example demonstrates a route which handles a request from SAP to the **BOOK_FLIGHT** RFC, which is implemented by the route. In addition, it demonstrates the component's XML serialization support, using JAXB to unmarshal and marshal SAP request objects and response objects to custom beans.

This route creates a **FlightTrip** business object on behalf of a travel agent, **FlightCustomer**. The route first unmarshals the SAP request object received by the SAP server endpoint into a custom JAXB bean. This custom bean is then multicasted in the exchange to three sub-routes, which gather the travel agent, flight connection and passenger information required to create the flight trip. The final sub-route

creates the flight trip object in SAP as demonstrated in the previous example. The final sub-route also creates and returns a custom JAXB bean which is marshaled into an SAP response object and returned by the server endpoint.

Java DSL for route

The Java DSL for the example route is as follows:

```
DataFormat jaxb = new JaxbDataFormat("org.fusesource.sap.example.jaxb");

from("sap-srfc-server:nplserver:BOOK_FLIGHT")
    .unmarshal(jaxb)
    .multicast()
    .to("direct:getFlightConnectionInfo",
        "direct:getFlightCustomerInfo",
        "direct:getPassengerInfo")
    .end()
    .to("direct:createFlightTrip")
    .marshal(jaxb);
```

XML DSL for route

And the XML DSL for the same route is as follows:

```
<route>
    <from uri="sap-srfc-server:nplserver:BOOK_FLIGHT"/>
    <unmarshal>
        <jaxb contextPath="org.fusesource.sap.example.jaxb"/>
    </unmarshal>
    <multicast>
        <to uri="direct:getFlightConnectionInfo"/>
        <to uri="direct:getFlightCustomerInfo"/>
        <to uri="direct:getPassengerInfo"/>
    </multicast>
    <to uri="direct:createFlightTrip"/>
    <marshal>
        <jaxb contextPath="org.fusesource.sap.example.jaxb"/>
    </marshal>
</route>
```

BookFlightRequest bean

The following listing illustrates a JAXB bean which unmarshals from the serialized form of an SAP **BOOK_FLIGHT** request object:

```
@XmlRootElement(name="Request",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class BookFlightRequest {

    @XmlAttribute(name="CUSTNAME")
    private String customerName;
```

```
    @XmlAttribute(name="FLIGHTDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date flightDate;

    @XmlAttribute(name="TRAVELAGENCYNUMBER")
    private String travelAgencyNumber;

    @XmlAttribute(name="DESTINATION_FROM")
    private String startAirportCode;

    @XmlAttribute(name="DESTINATION_TO")
    private String endAirportCode;

    @XmlAttribute(name="PASSFORM")
    private String passengerFormOfAddress;

    @XmlAttribute(name="PASSNAME")
    private String passengerName;

    @XmlAttribute(name="PASSBIRTH")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date passengerDateOfBirth;

    @XmlAttribute(name="CLASS")
    private String flightClass;

    ...
}
```

BookFlightResponse bean

The following listing illustrates a JAXB bean which marshals to the serialized form of an SAP **BOOK_FLIGHT** response object:

```
@XmlRootElement(name="Response",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class BookFlightResponse {

    @XmlAttribute(name="TRIPNUMBER")
    private String tripNumber;

    @XmlAttribute(name="TICKET_PRICE")
    private BigDecimal ticketPrice;

    @XmlAttribute(name="TICKET_TAX")
    private BigDecimal ticketTax;

    @XmlAttribute(name="CURRENCY")
    private String currency;

    @XmlAttribute(name="PASSFORM")
    private String passengerFormOfAddress;

    @XmlAttribute(name="PASSNAME")
```

```
    private String passengerName;

    @XmlAttribute(name="PASSBIRTH")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date passengerDateOfBirth;

    @XmlElement(name="FLTINFO")
    private FlightInfo flightInfo;

    @XmlElement(name="CONNINFO")
    private ConnectionInfoTable connectionInfo;

    ...
}
```

NOTE

The complex parameter fields of the response object are serialized as child elements of the response.

FlightInfo bean

The following listing illustrates a JAXB bean which marshals to the serialized form of the complex structure parameter **FLTINFO**:

```
@XmlRootElement(name="FLTINFO",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class FlightInfo {

    @XmlAttribute(name="FLIGHTTIME")
    private String flightTime;

    @XmlAttribute(name="CITYFROM")
    private String cityFrom;

    @XmlAttribute(name="DEPDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date departureDate;

    @XmlAttribute(name="DEPTIME")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date departureTime;

    @XmlAttribute(name="CITYTO")
    private String cityTo;

    @XmlAttribute(name="ARRDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date arrivalDate;

    @XmlAttribute(name="ARRTIME")
    @XmlJavaTypeAdapter(DateAdapter.class)
    private Date arrivalTime;
```

```
    ...
}
```

ConnectionInfoTable bean

The following listing illustrates a JAXB bean which marshals to the serialized form of the complex table parameter, **CONNINFO**. Note that the name of the root element type of the JAXB bean corresponds to the name of the row structure type suffixed with **_TABLE** and the bean contains a list of row elements.

```
@XmlRootElement(name="CONNINFO_TABLE",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class ConnectionInfoTable {

    @XmlElement(name="row")
    List<ConnectionInfo> rows;

    ...
}
```

ConnectionInfo bean

The following listing illustrates a JAXB bean, which marshals to the serialized form of the above tables row elements:

```
@XmlRootElement(name="CONNINFO",
namespace="http://sap.fusesource.org/rfc/nplServer/BOOK_FLIGHT")
@XmlAccessorType(XmlAccessType.FIELD)
public class ConnectionInfo {

    @XmlAttribute(name="CONNID")
    String connectionId;

    @XmlAttribute(name="AIRLINE")
    String airline;

    @XmlAttribute(name="PLANETYPE")
    String planeType;

    @XmlAttribute(name="CITYFROM")
    String cityFrom;

    @XmlAttribute(name="DEPDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date departureDate;

    @XmlAttribute(name="DEPTIME")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date departureTime;

    @XmlAttribute(name="CITYTO")
    String cityTo;
```

```
    @XmlAttribute(name="ARRDATE")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date arrivalDate;

    @XmlAttribute(name="ARRTIME")
    @XmlJavaTypeAdapter(DateAdapter.class)
    Date arrivalTime;

    ...
}
```

CHAPTER 271. SAP NETWEAVER COMPONENT

Available as of Camel version 2.12

The **sap-netweaver** integrates with the SAP NetWeaver Gateway using HTTP transports.

This camel component supports only producer endpoints.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sap-netweaver</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

271.1. URI FORMAT

The URI scheme for a sap netweaver gateway component is as follows

```
sap-netweaver:https://host:8080/path?username=foo&password=secret
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

271.2. PREREQUISITES

You would need to have an account to the SAP NetWeaver system to be able to leverage this component. SAP provides a demo setup where you can requires for an account.

This component uses the basic authentication scheme for logging into SAP NetWeaver.

271.3. SAPNETWEAVER OPTIONS

The SAP NetWeaver component has no options.

The SAP NetWeaver endpoint is configured using URI syntax:

```
sap-netweaver:url
```

with the following path and query parameters:

271.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
url	**Required** Url to the SAP net-weaver gateway server.		String

271.3.2. Query Parameters (6 parameters):

Name	Description	Default	Type
flatternMap (producer)	If the JSON Map contains only a single entry, then flattern by storing that single entry value as the message body.	true	boolean
json (producer)	Whether to return data in JSON format. If this option is false, then XML is returned in Atom format.	true	boolean
jsonAsMap (producer)	To transform the JSON from a String to a Map in the message body.	true	boolean
password (producer)	**Required** Password for account.		String
username (producer)	**Required** Username for account.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

271.4. MESSAGE HEADERS

The following headers can be used by the producer.

Name	Type	Description
CamelNetWeaverCommand	**String**	**Mandatory**: The command to execute inMS ADO.Net Data Service format.

271.5. EXAMPLES

This example is using the flight demo example from SAP, which is available online over the internet here.

In the route below we request the SAP NetWeaver demo server using the following url

> https://*sapes1.sapdevcenter.com/sap/opu/odata/IWBEP/RMTSAMPLEFLIGHT_2/*

And we want to execute the following command

> FlightCollection(AirLineID='AA',FlightConnectionID='0017',FlightDate=datetime'2012-08-29T00%3A00%3A00')

To get flight details for the given flight. The command syntax is in MS ADO.Net Data Service format.

We have the following Camel route

```
from("direct:start")
    .setHeader(NetWeaverConstants.COMMAND, constant(command))
    .toF("sap-netweaver:%s?username=%s&password=%s", url, username, password)
    .to("log:response")
    .to("velocity:flight-info.vm")
```

Where url, username, password and command is defined as:

```
    private String username = "P1909969254";
    private String password = "TODO";
    private String url =
"https://sapes1.sapdevcenter.com/sap/opu/odata/IWBEP/RMTSAMPLEFLIGHT_2/";
    private String command =
"FlightCollection(AirLineID='AA',FlightConnectionID='0017',FlightDate=datetime'2012-08-
29T00%3A00%3A00')";
```

The password is invalid. You would need to create an account at SAP first to run the demo.

The velocity template is used for formatting the response to a basic HTML page

```
<html>
  <body>
  Flight information:

  <p/>
  <br/>Airline ID: $body["AirLineID"]
  <br/>Aircraft Type: $body["AirCraftType"]
  <br/>Departure city: $body["FlightDetails"]["DepartureCity"]
  <br/>Departure airport: $body["FlightDetails"]["DepartureAirPort"]
  <br/>Destination city: $body["FlightDetails"]["DestinationCity"]
  <br/>Destination airport: $body["FlightDetails"]["DestinationAirPort"]

  </body>
</html>
```

When running the application you get sampel output:

```
Flight information:
Airline ID: AA
Aircraft Type: 747-400
Departure city: new york
Departure airport: JFK
Destination city: SAN FRANCISCO
Destination airport: SFO
```

271.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint

- Getting Started
- HTTP

CHAPTER 272. SCHEDULER COMPONENT

Available as of Camel version 2.15

The **scheduler:** component is used to generate message exchanges when a scheduler fires. This component is similar to the Timer component, but it offers more functionality in terms of scheduling. Also this component uses JDK **ScheduledExecutorService**. Where as the timer uses a JDK **Timer**.

You can only consume events from this endpoint.

272.1. URI FORMAT

```
scheduler:name[?options]
```

Where **name** is the name of the scheduler, which is created and shared across endpoints. So if you use the same name for all your timer endpoints, only one scheduler thread pool and thread will be used - but you can configure the thread pool to allow more concurrent threads.

You can append query options to the URI in the following format, **?option=value&option=value&...**

Note: The IN body of the generated exchange is **null**. So **exchange.getIn().getBody()** returns **null**.

272.2. OPTIONS

The Scheduler component supports 2 options which are listed below.

Name	Description	Default	Type
concurrentTasks (scheduler)	Number of threads used by the scheduling thread pool. Is by default using a single thread	1	int
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Scheduler endpoint is configured using URI syntax:

```
scheduler:name
```

with the following path and query parameters:

272.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** The name of the scheduler		String

272.2.2. Query Parameters (20 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel. In other words the error occurred while the polling was gathering information, for instance access to a file network failed so Camel cannot access it to scan for files. The default implementation will log the caused exception at WARN level and ignore it.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
concurrentTasks (scheduler)	Number of threads used by the scheduling thread pool. Is by default using a single thread	1	int
delay (scheduler)	Milliseconds before the next poll. The default value is 500. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. The default value is 1000. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool. This option allows you to share a thread pool among multiple consumers.		ScheduledExecutor Service

Name	Description	Default	Type
scheduler (scheduler)	Allow to plugin a custom org.apache.camel.spi.ScheduledPollConsumerScheduler to use as the scheduler for firing when the polling consumer runs. The default implementation uses the ScheduledExecutorService and there is a Quartz2, and Spring based which supports CRON expressions. Notice: If using a custom scheduler then the options for initialDelay, useFixedDelay, timeUnit, and scheduledExecutorService may not be in use. Use the text quartz2 to refer to use the Quartz2 scheduler; and use the text spring to use the Spring based; and use the text myScheduler to refer to a custom scheduler by its id in the Registry. See Quartz2 page for an example.	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

272.3. MORE INFORMATION

This component is a scheduler Polling Consumer where you can find more information about the options above, and examples at the Polling Consumer page.

272.4. EXCHANGE PROPERTIES

When the timer is fired, it adds the following information as properties to the **Exchange**:

Name	Type	Description
Exchange.TIMER_NAME	**String**	The value of the **name** option.

Name	Type	Description
Exchange.TIMER_FIRED_TIME	**Date**	The time when the consumer fired.

272.5. SAMPLE

To set up a route that generates an event every 60 seconds:

```
from("scheduler://foo?period=60s").to("bean:myBean?method=someMethodName");
```

The above route will generate an event and then invoke the **someMethodName** method on the bean called **myBean** in the Registry such as JNDI or Spring.

And the route in Spring DSL:

```
<route>
  <from uri="scheduler://foo?period=60s"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

272.6. FORCING THE SCHEDULER TO TRIGGER IMMEDIATELY WHEN COMPLETED

To let the scheduler trigger as soon as the previous task is complete, you can set the option **greedy=true**. But beware then the scheduler will keep firing all the time. So use this with caution.

272.7. FORCING THE SCHEDULER TO BE IDLE

There can be use cases where you want the scheduler to trigger and be greedy. But sometimes you want "tell the scheduler" that there was no task to poll, so the scheduler can change into idle mode using the backoff options. To do this you would need to set a property on the exchange with the key **Exchange.SCHEDULER_POLLED_MESSAGES** to a boolean value of false. This will cause the consumer to indicate that there was no messages polled.

The consumer will otherwise as by default return 1 message polled to the scheduler, every time the consumer has completed processing the exchange.

272.8. SEE ALSO

- Timer
- Quartz

CHAPTER 273. SCHEMATRON COMPONENT

Available as of Camel version 2.15

Schematron is an XML-based language for validating XML instance documents. It is used to make assertions about data in an XML document and it is also used to express operational and business rules. Schematron is an ISO Standard. The schematron component uses the leading implementation of ISO schematron. It is an XSLT based implementation. The schematron rules is run through four XSLT pipelines, which generates a final XSLT which will be used as the basis for running the assertion against the XML document. The component is written in a way that Schematron rules are loaded at the start of the endpoint (only once) this is to minimise the overhead of instantiating a Java Templates object representing the rules.

273.1. URI FORMAT

```
schematron://path?[options]
```

273.2. URI OPTIONS

The Schematron component has no options.

The Schematron endpoint is configured using URI syntax:

```
schematron:path
```

with the following path and query parameters:

273.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
path	**Required** The path to the schematron rules file. Can either be in class path or location in the file system.		String

273.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
abort (producer)	Flag to abort the route and throw a schematron validation exception.	false	boolean
rules (producer)	To use the given schematron rules instead of loading from the path		Templates
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
uriResolver (advanced)	Set the URIResolver to be used for resolving schematron includes in the rules file.		URIResolver

273.3. HEADERS

Name	Description	Type	In/Out
CamelSchematronValidationStatus	The schematron validation status: SUCCESS / FAILED	String	IN
CamelSchematronValidationReport	The schematrion report body in XML format. See an example below	String	IN

273.4. URI AND PATH SYNTAX

The following example shows how to invoke the schematron processor in Java DSL. The schematron rules file is sourced from the class path:

```
from("direct:start").to("schematron://sch/schematron.sch").to("mock:result")
```

The following example shows how to invoke the schematron processor in XML DSL. The schematrion rules file is sourced from the file system:

```
<route>
  <from uri="direct:start" />
  <to uri="schematron:///usr/local/sch/schematron.sch" />
  <log message="Schematron validation status: ${in.header.CamelSchematronValidationStatus}" />
```

```
  <choice>
    <when>
      <simple>${in.header.CamelSchematronValidationStatus} == 'SUCCESS'</simple>
      <to uri="mock:success" />
    </when>
    <otherwise>
      <log message="Failed schematron validation" />
      <setBody>
        <header>CamelSchematronValidationReport</header>
      </setBody>
      <to uri="mock:failure" />
    </otherwise>
  </choice>
</route>
```

TIP

Where to store schematron rules? Schematron rules can change with business requirement, as such it is recommended to store these rules somewhere in file system. When the schematron component endpoint is started, the rules are compiled into XSLT as a Java Templates Object. This is done only once to minimise the overhead of instantiating Java Templates object, which can be an expensive operation for large set of rules and given that the process goes through four pipelines of XSLT transformations. So if you happen to store the rules in the file system, in the event of an update, all you need is to restart the route or the component. No harm in storing these rules in the class path though, but you will have to build and deploy the component to pick up the changes.

273.5. SCHEMATRON RULES AND REPORT SAMPLES

Here is an example of schematron rules

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron">
  <title>Check Sections 12/07</title>
  <pattern id="section-check">
    <rule context="section">
      <assert test="title">This section has no title</assert>
      <assert test="para">This section has no paragraphs</assert>
    </rule>
  </pattern>
</schema>
```

Here is an example of schematron report:

```
<?xml version="1.0" encoding="UTF-8"?>
<svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl"
 xmlns:iso="http://purl.oclc.org/dsdl/schematron"
 xmlns:saxon="http://saxon.sf.net/"
 xmlns:schold="http://www.ascc.net/xml/schematron"
 xmlns:xhtml="http://www.w3.org/1999/xhtml"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" schemaVersion="" title="">

  <svrl:active-pattern document="" />
  <svrl:fired-rule context="chapter" />
```

```
    <svrl:failed-assert test="title" location="/doc[1]/chapter[1]">
      <svrl:text>A chapter should have a title</svrl:text>
    </svrl:failed-assert>
    <svrl:fired-rule context="chapter" />
    <svrl:failed-assert test="title" location="/doc[1]/chapter[2]">
      <svrl:text>A chapter should have a title</svrl:text>
    </svrl:failed-assert>
    <svrl:fired-rule context="chapter" />
  </svrl:schematron-output>
```

TIP

Useful Links and resources* Introduction to Schematron by Mulleberry technologies. An excellent document in PDF to get you started on Schematron. * Schematron official site. This contains links to other resources

CHAPTER 274. SCP COMPONENT

Available as of Camel version 2.10

The **camel-jsch** component supports the SCP protocol using the Client API of the Jsch project. Jsch is already used in camel by the FTP component for the **sftp:** protocol.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jsch</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

274.1. URI FORMAT

```
scp://host[:port]/destination[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

The file name can be specified either in the <path> part of the URI or as a "CamelFileName" header on the message (**Exchange.FILE_NAME** if used in code).

274.2. OPTIONS

The SCP component supports 2 options which are listed below.

Name	Description	Default	Type
verboseLogging (producer)	JSCH is verbose logging out of the box. Therefore we turn the logging down to DEBUG logging by default. But setting this option to true turns on the verbose logging again.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The SCP endpoint is configured using URI syntax:

```
scp:host:port/directoryName
```

with the following path and query parameters:

274.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	**Required** Hostname of the FTP server		String
port	Port of the FTP server		int
directoryName	The starting directory		String

274.2.2. Query Parameters (20 parameters):

Name	Description	Default	Type
disconnect (common)	Whether or not to disconnect from remote FTP server right after use. Disconnect will only disconnect the current connection to the FTP server. If you have a consumer which you want to stop, then you need to stop the consumer/route instead.	false	boolean
chmod (producer)	Allows you to set chmod on the stored file. For example chmod=664.	664	String
fileName (producer)	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the CamelFileName header. (Note: The header itself can also be an Expression). The expression options support both String and Expression types. If the expression is a String type, it is always evaluated using the File Language. If the expression is an Expression type, the specified Expression type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: mydata-$date:now:yyyyMMdd.txt. The producers support the CamelOverruleFileName header which takes precedence over any existing CamelFileName header; the CamelOverruleFileName is a header that is used only once, and makes it easier as this avoids to temporary store CamelFileName and have to restore it afterwards.		String

Name	Description	Default	Type
flatten (producer)	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to true on the producer enforces that any file name in CamelFileName header will be stripped for any leading paths.	false	boolean
strictHostKeyChecking (producer)	Sets whether to use strict host key checking. Possible values are: no, yes	no	String
allowNullBody (producer)	Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a GenericFileWriteException of 'Cannot write null body to file.' will be thrown. If the fileExist option is set to 'Override', then the file will be truncated, and if set to append the file will remain unchanged.	false	boolean
disconnectOnBatchComplete (producer)	Whether or not to disconnect from remote FTP server right after a Batch upload is complete. disconnectOnBatchComplete will only disconnect the current connection to the FTP server.	false	boolean
connectTimeout (advanced)	Sets the connect timeout for waiting for a connection to be established Used by both FTPClient and JSCH	10000	int
soTimeout (advanced)	Sets the so timeout Used only by FTPClient	300000	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
timeout (advanced)	Sets the data timeout for waiting for reply Used only by FTPClient	30000	int
knownHostsFile (security)	Sets the known_hosts file, so that the jsch endpoint can do host key verification. You can prefix with classpath: to load the file from classpath instead of file system.		String
password (security)	Password to use for login		String

Name	Description	Default	Type
preferredAuthentications (security)	Set a comma separated list of authentications that will be used in order of preference. Possible authentication methods are defined by JCraft JSCH. Some examples include: gssapi-with-mic,publickey,keyboard-interactive,password If not specified the JSCH and/or system defaults will be used.		String
privateKeyBytes (security)	Set the private key bytes to that the endpoint can do private key verification. This must be used only if privateKeyFile wasn't set. Otherwise the file will have the priority.		byte[]
privateKeyFile (security)	Set the private key file to that the endpoint can do private key verification. You can prefix with classpath: to load the file from classpath instead of file system.		String
privateKeyFilePassphrase (security)	Set the private key file passphrase to that the endpoint can do private key verification.		String
username (security)	Username to use for login		String
useUserKnownHostsFile (security)	If knownHostFile has not been explicit configured, then use the host file from System.getProperty(user.home) /.ssh/known_hosts	true	boolean
ciphers (security)	Set a comma separated list of ciphers that will be used in order of preference. Possible cipher names are defined by JCraft JSCH. Some examples include: aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,blowfish-cbc,aes192-cbc,aes256-cbc. If not specified the default list from JSCH will be used.		String

274.3. LIMITATIONS

Currently camel-jsch only supports a Producer (i.e. copy files to another host).

274.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 275. CAMEL SCR (DEPRECATED)

Available as of Camel 2.15

SCR stands for Service Component Runtime and is an implementation of OSGi Declarative Services specification. SCR enables any plain old Java object to expose and use OSGi services with no boilerplate code.

OSGi framework knows your object by looking at SCR descriptor files in its bundle which are typically generated from Java annotations by a plugin such as **org.apache.felix:maven-scr-plugin**.

Running Camel in an SCR bundle is a great alternative for Spring DM and Blueprint based solutions having significantly fewer lines of code between you and the OSGi framework. Using SCR your bundle can remain completely in Java world; there is no need to edit XML or properties files. This offers you full control over everything and means your IDE of choice knows exactly what is going on in your project.

275.1. CAMEL SCR SUPPORT

Camel-scr bundle is not included in Apache Camel versions prior 2.15.0, but the artifact itself can be used with any Camel version since 2.12.0.

org.apache.camel/camel-scr bundle provides a base class, **AbstractCamelRunner**, which manages a Camel context for you and a helper class, **ScrHelper**, for using your SCR properties in unit tests. Camel-scr feature for Apache Karaf defines all features and bundles required for running Camel in SCR bundles.

AbstractCamelRunner class ties CamelContext's lifecycle to Service Component's lifecycle and handles configuration with help of Camel's PropertiesComponent. All you have to do to make a Service Component out of your java class is to extend it from **AbstractCamelRunner** and add the following **org.apache.felix.scr.annotations** on class level:

Add required annotations

```
@Component
@References({
    @Reference(name = "camelComponent",referenceInterface = ComponentResolver.class,
        cardinality = ReferenceCardinality.MANDATORY_MULTIPLE, policy =
ReferencePolicy.DYNAMIC,
        policyOption = ReferencePolicyOption.GREEDY, bind = "gotCamelComponent", unbind =
"lostCamelComponent")
})
```

Then implement **getRouteBuilders()** method which returns the Camel routes you want to run:

Implement getRouteBuilders()

```
    @Override
    protected List<RoutesBuilder> getRouteBuilders() {
        List<RoutesBuilder> routesBuilders = new ArrayList<>();
        routesBuilders.add(new YourRouteBuilderHere(registry));
        routesBuilders.add(new AnotherRouteBuilderHere(registry));
        return routesBuilders;
    }
```

And finally provide the default configuration with:

Default configuration in annotations

```
@Properties({
  @Property(name = "camelContextId", value = "my-test"),
  @Property(name = "active", value = "true"),
  @Property(name = "...", value = "..."),
  ...
})
```

That's all. And if you used **camel-archetype-scr** to generate a project all this is already taken care of.

Below is an example of a complete Service Component class, generated by **camel-archetype-scr:**

CamelScrExample.java

```
// This file was generated from org.apache.camel.archetypes/camel-archetype-scr/2.15-SNAPSHOT
package example;

import java.util.ArrayList;
import java.util.List;

import org.apache.camel.scr.AbstractCamelRunner;
import example.internal.CamelScrExampleRoute;
import org.apache.camel.RoutesBuilder;
import org.apache.camel.spi.ComponentResolver;
import org.apache.felix.scr.annotations.*;

@Component(label = CamelScrExample.COMPONENT_LABEL, description =
CamelScrExample.COMPONENT_DESCRIPTION, immediate = true, metatype = true)
@Properties({
   @Property(name = "camelContextId", value = "camel-scr-example"),
   @Property(name = "camelRouteId", value = "foo/timer-log"),
   @Property(name = "active", value = "true"),
   @Property(name = "from", value = "timer:foo?period=5000"),
   @Property(name = "to", value = "log:foo?showHeaders=true"),
   @Property(name = "messageOk", value = "Success: {{from}} -> {{to}}"),
   @Property(name = "messageError", value = "Failure: {{from}} -> {{to}}"),
   @Property(name = "maximumRedeliveries", value = "0"),
   @Property(name = "redeliveryDelay", value = "5000"),
   @Property(name = "backOffMultiplier", value = "2"),
   @Property(name = "maximumRedeliveryDelay", value = "60000")
})
@References({
   @Reference(name = "camelComponent",referenceInterface = ComponentResolver.class,
      cardinality = ReferenceCardinality.MANDATORY_MULTIPLE, policy =
ReferencePolicy.DYNAMIC,
      policyOption = ReferencePolicyOption.GREEDY, bind = "gotCamelComponent", unbind =
"lostCamelComponent")
})
public class CamelScrExample extends AbstractCamelRunner {

   public static final String COMPONENT_LABEL = "example.CamelScrExample";
   public static final String COMPONENT_DESCRIPTION = "This is the description for camel-scr-
example.";
```

```
    @Override
    protected List<RoutesBuilder> getRouteBuilders() {
        List<RoutesBuilder> routesBuilders = new ArrayList<>();
        routesBuilders.add(new CamelScrExampleRoute(registry));
        return routesBuilders;
    }
}
```

CamelContextId and **active** properties control the CamelContext's name (defaults to "camel-runner-default") and whether it will be started or not (defaults to "false"), respectively. In addition to these you can add and use as many properties as you like. Camel's PropertiesComponent handles recursive properties and prefixing with fallback without problem.

AbstractCamelRunner will make these properties available to your RouteBuilders with help of Camel's PropertiesComponent and it will also inject these values into your Service Component's and RouteBuilder's fields when their names match. The fields can be declared with any visibility level, and many types are supported (String, int, boolean, URL, ...).

Below is an example of a RouteBuilder class generated by **camel-archetype-scr**:

CamelScrExampleRoute.java

```
// This file was generated from org.apache.camel.archetypes/camel-archetype-scr/2.15-SNAPSHOT
package example.internal;

import org.apache.camel.LoggingLevel;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.SimpleRegistry;
import org.apache.commons.lang.Validate;

public class CamelScrExampleRoute extends RouteBuilder {

    SimpleRegistry registry;

    // Configured fields
    private String camelRouteId;
    private Integer maximumRedeliveries;
    private Long redeliveryDelay;
    private Double backOffMultiplier;
    private Long maximumRedeliveryDelay;

    public CamelScrExampleRoute(final SimpleRegistry registry) {
        this.registry = registry;
    }

    @Override
    public void configure() throws Exception {
        checkProperties();

        // Add a bean to Camel context registry
        registry.put("test", "bean");
```

```
        errorHandler(defaultErrorHandler()
            .retryAttemptedLogLevel(LoggingLevel.WARN)
            .maximumRedeliveries(maximumRedeliveries)
            .redeliveryDelay(redeliveryDelay)
            .backOffMultiplier(backOffMultiplier)
            .maximumRedeliveryDelay(maximumRedeliveryDelay));

        from("{{from}}")
            .startupOrder(2)
            .routeId(camelRouteId)
            .onCompletion()
                .to("direct:processCompletion")
            .end()
            .removeHeaders("CamelHttp*")
            .to("{{to}}");


        from("direct:processCompletion")
            .startupOrder(1)
            .routeId(camelRouteId + ".completion")
            .choice()
                .when(simple("${exception} == null"))
                    .log("{{messageOk}}")
                .otherwise()
                    .log(LoggingLevel.ERROR, "{{messageError}}")
            .end();
        }
    }

    public void checkProperties() {
        Validate.notNull(camelRouteId, "camelRouteId property is not set");
        Validate.notNull(maximumRedeliveries, "maximumRedeliveries property is not set");
        Validate.notNull(redeliveryDelay, "redeliveryDelay property is not set");
        Validate.notNull(backOffMultiplier, "backOffMultiplier property is not set");
        Validate.notNull(maximumRedeliveryDelay, "maximumRedeliveryDelay property is not set");
    }
}
```

Let's take a look at **CamelScrExampleRoute** in more detail.

```
    // Configured fields
    private String camelRouteId;
    private Integer maximumRedeliveries;
    private Long redeliveryDelay;
    private Double backOffMultiplier;
    private Long maximumRedeliveryDelay;
```

The values of these fields are set with values from properties by matching their names.

```
// Add a bean to Camel context registry
registry.put("test", "bean");
```

If you need to add some beans to CamelContext's registry for your routes, you can do it like this.

```
public void checkProperties() {
    Validate.notNull(camelRouteId, "camelRouteId property is not set");
    Validate.notNull(maximumRedeliveries, "maximumRedeliveries property is not set");
    Validate.notNull(redeliveryDelay, "redeliveryDelay property is not set");
    Validate.notNull(backOffMultiplier, "backOffMultiplier property is not set");
    Validate.notNull(maximumRedeliveryDelay, "maximumRedeliveryDelay property is not set");
}
```

It is a good idea to check that required parameters are set and they have meaningful values before allowing the routes to start.

```
from("{{from}}")
    .startupOrder(2)
    .routeId(camelRouteId)
    .onCompletion()
        .to("direct:processCompletion")
    .end()
    .removeHeaders("CamelHttp*")
    .to("{{to}}");


from("direct:processCompletion")
    .startupOrder(1)
    .routeId(camelRouteId + ".completion")
    .choice()
        .when(simple("${exception} == null"))
            .log("{{messageOk}}")
        .otherwise()
            .log(LoggingLevel.ERROR, "{{messageError}}")
    .end();
```

Note that pretty much everything in the route is configured with properties. This essentially makes your RouteBuilder a template. SCR allows you to create more instances of your routes just by providing alternative configurations. More on this in section *Using Camel SCR bundle as a template* .

275.2. ABSTRACTCAMELRUNNER'S LIFECYCLE IN SCR

1. When component's configuration policy and mandatory references are satisfied SCR calls **activate()**. This creates and sets up a CamelContext through the following call chain: **activate() → prepare() → createCamelContext() → setupPropertiesComponent() → configure() → setupCamelContext()**. Finally, the context is scheduled to start after a delay defined in **AbstractCamelRunner.START_DELAY** with **runWithDelay()**.

2. When Camel components (**ComponentResolver** services, to be exact) are registered in OSGi, SCR calls **gotCamelComponent**`()` which reschedules/delays the CamelContext start further

by the same **AbstractCamelRunner.START_DELAY**. This in effect makes CamelContext wait until all Camel components are loaded or there is a sufficient gap between them. The same logic will tell a failed-to-start CamelContext to try again whenever we add more Camel components.

3. When Camel components are unregistered SCR calls **lostCamelComponent`()`**. This call does nothing.

4. When one of the requirements that caused the call to **activate() is lost SCR will call deactivate()**. This will shutdown the CamelContext.

In (non-OSGi) unit tests you should use **prepare() → run() → stop()** instead of **activate() → deactivate()** for more fine-grained control. Also, this allows us to avoid possible SCR specific operations in tests.

275.3. USING CAMEL-ARCHETYPE-SCR

The easiest way to create an Camel SCR bundle project is to use **camel-archetype-scr** and Maven.

You can generate a project with the following steps:

Generating a project

```
$ mvn archetype:generate -Dfilter=org.apache.camel.archetypes:camel-archetype-scr

Choose archetype:
1: local -> org.apache.camel.archetypes:camel-archetype-scr (Creates a new Camel SCR bundle project for Karaf)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): : 1
Define value for property 'groupId': : example
[INFO] Using property: groupId = example
Define value for property 'artifactId': : camel-scr-example
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': example: :
[INFO] Using property: archetypeArtifactId = camel-archetype-scr
[INFO] Using property: archetypeGroupId = org.apache.camel.archetypes
[INFO] Using property: archetypeVersion = 2.15-SNAPSHOT
Define value for property 'className': : CamelScrExample
Confirm properties configuration:
groupId: example
artifactId: camel-scr-example
version: 1.0-SNAPSHOT
package: example
archetypeArtifactId: camel-archetype-scr
archetypeGroupId: org.apache.camel.archetypes
archetypeVersion: 2.15-SNAPSHOT
className: CamelScrExample
Y: :
```

Done!

Now run:

```
mvn install
```

and the bundle is ready to be deployed.

275.4. UNIT TESTING CAMEL ROUTES

Service Component is a POJO and has no special requirements for (non-OSGi) unit testing. There are however some techniques that are specific to Camel SCR or just make testing easier.

Below is an example unit test, generated by **camel-archetype-scr**:

```
// This file was generated from org.apache.camel.archetypes/camel-archetype-scr/2.15-SNAPSHOT
package example;

import java.util.List;

import org.apache.camel.scr.internal.ScrHelper;
import org.apache.camel.builder.AdviceWithRouteBuilder;
import org.apache.camel.component.mock.MockComponent;
import org.apache.camel.component.mock.MockEndpoint;
import org.apache.camel.model.ModelCamelContext;
import org.apache.camel.model.RouteDefinition;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TestName;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

@RunWith(JUnit4.class)
public class CamelScrExampleTest {

    Logger log = LoggerFactory.getLogger(getClass());

    @Rule
    public TestName testName = new TestName();

    CamelScrExample integration;
    ModelCamelContext context;

    @Before
    public void setUp() throws Exception {
        log.info("********************************************************************************");
        log.info("Test: " + testName.getMethodName());
        log.info("********************************************************************************");

        // Set property prefix for unit testing
        System.setProperty(CamelScrExample.PROPERTY_PREFIX, "unit");

        // Prepare the integration
        integration = new CamelScrExample();
        integration.prepare(null, ScrHelper.getScrProperties(integration.getClass().getName()));
        context = integration.getContext();

        // Disable JMX for test
        context.disableJMX();
```

```
        // Fake a component for test
        context.addComponent("amq", new MockComponent());
    }

    @After
    public void tearDown() throws Exception {
        integration.stop();
    }

    @Test
    public void testRoutes() throws Exception {
        // Adjust routes
        List<RouteDefinition> routes = context.getRouteDefinitions();

        routes.get(0).adviceWith(context, new AdviceWithRouteBuilder() {
            @Override
            public void configure() throws Exception {
                // Replace "from" endpoint with direct:start
                replaceFromWith("direct:start");
                // Mock and skip result endpoint
                mockEndpoints("log:*");
            }
        });

        MockEndpoint resultEndpoint = context.getEndpoint("mock:log:foo", MockEndpoint.class);
        // resultEndpoint.expectedMessageCount(1); // If you want to just check the number of messages
        resultEndpoint.expectedBodiesReceived("hello"); // If you want to check the contents

        // Start the integration
        integration.run();

        // Send the test message
        context.createProducerTemplate().sendBody("direct:start", "hello");

        resultEndpoint.assertIsSatisfied();
    }
}
```

Now, let's take a look at the interesting bits one by one.

Using property prefixing

```
        // Set property prefix for unit testing
        System.setProperty(CamelScrExample.PROPERTY_PREFIX, "unit");
```

This allows you to override parts of the configuration by prefixing properties with "unit.". For example, **unit.from** overrides **from** for the unit test.

Prefixes can be used to handle the differences between the runtime environments where your routes might run. Moving the unchanged bundle through development, testing and production environments is a typical use case.

Getting test configuration from annotations

```
integration.prepare(null, ScrHelper.getScrProperties(integration.getClass().getName()));
```

Here we configure the Service Component in test with the same properties that would be used in OSGi environment.

Mocking components for test

```
// Fake a component for test
context.addComponent("amq", new MockComponent());
```

Components that are not available in test can be mocked like this to allow the route to start.

Adjusting routes for test

```
// Adjust routes
List<RouteDefinition> routes = context.getRouteDefinitions();

routes.get(0).adviceWith(context, new AdviceWithRouteBuilder() {
    @Override
    public void configure() throws Exception {
        // Replace "from" endpoint with direct:start
        replaceFromWith("direct:start");
        // Mock and skip result endpoint
        mockEndpoints("log:*");
    }
});
```

Camel's AdviceWith feature allows routes to be modified for test.

Starting the routes

```
// Start the integration
integration.run();
```

Here we start the Service Component and along with it the routes.

Sending a test message

```
// Send the test message
context.createProducerTemplate().sendBody("direct:start", "hello");
```

Here we send a message to a route in test.

275.5. RUNNING THE BUNDLE IN APACHE KARAF

Once the bundle has been built with **mvn install** it's ready to be deployed. To deploy the bundle on Apache Karaf perform the following steps on Karaf command line:

Deploying the bundle in Apache Karaf

```
# Add Camel feature repository
karaf@root> features:chooseurl camel 2.15-SNAPSHOT

# Install camel-scr feature
karaf@root> features:install camel-scr

# Install commons-lang, used in the example route to validate parameters
karaf@root> osgi:install mvn:commons-lang/commons-lang/2.6

# Install and start your bundle
karaf@root> osgi:install -s mvn:example/camel-scr-example/1.0-SNAPSHOT

# See how it's running
karaf@root> log:tail -n 10

Press ctrl-c to stop watching the log.
```

275.5.1. Overriding the default configuration

By default, Service Component's configuration PID equals the fully qualified name of its class. You can change the example bundle's properties with Karaf's **config:*** commands:

Override a property

```
# Override 'messageOk' property
karaf@root> config:propset -p example.CamelScrExample messageOk "This is better logging"
```

Or you can change the configuration by editing property files in Karaf's **etc** folder.

275.5.2. Using Camel SCR bundle as a template

Let's say you have a Camel SCR bundle that implements an integration pattern that you use frequently, say, **from → to**, with success/failure logging and redelivery which also happens to be the pattern our example route implements. You probably don't want to create a separate bundle for every instance. No worries, SCR has you covered.

Create a configuration PID for your Service Component, but add a tail with a dash and SCR will use that configuration to create a new instance of your component.

Creating a new Service Component instance

```
# Create a PID with a tail
karaf@root> config:edit example.CamelScrExample-anotherone

# Override some properties
karaf@root> config:propset camelContextId my-other-context
karaf@root> config:propset to "file://removeme?fileName=removemetoo.txt"
```

```
# Save the PID
karaf@root> config:update
```

This will start a new CamelContext with your overridden properties. How convenient.

275.6. NOTES

When designing a Service Component to be a template you typically don't want it to start without a "tailed" configuration i.e. with the default configuration.

To prevent your Service Component from starting with the default configuration add **policy = ConfigurationPolicy.REQUIRE `to the class level `@Component** annotation.

CHAPTER 276. XML SECURITY DATAFORMAT

Available as of Camel version 2.0

The XMLSecurity Data Format facilitates encryption and decryption of XML payloads at the Document, Element, and Element Content levels (including simultaneous multi-node encryption/decryption using XPath). To sign messages using the XML Signature specification, please see the Camel XML Security component.

The encryption capability is based on formats supported using the Apache XML Security (Santuario) project. Symmetric encryption/decryption is currently supported using Triple-DES and AES (128, 192, and 256) encryption formats. Additional formats can be easily added later as needed. This capability allows Camel users to encrypt/decrypt payloads while being dispatched or received along a route.

Available as of Camel 2.9
The XMLSecurity Data Format supports asymmetric key encryption. In this encryption model a symmetric key is generated and used to perform XML content encryption or decryption. This "content encryption key" is then itself encrypted using an asymmetric encryption algorithm that leverages the recipient's public key as the "key encryption key". Use of an asymmetric key encryption algorithm ensures that only the holder of the recipient's private key can access the generated symmetric encryption key. Thus, only the private key holder can decode the message. The XMLSecurity Data Format handles all of the logic required to encrypt and decrypt the message content and encryption key(s) using asymmetric key encryption.

The XMLSecurity Data Format also has improved support for namespaces when processing the XPath queries that select content for encryption. A namespace definition mapping can be included as part of the data format configuration. This enables true namespace matching, even if the prefix values in the XPath query and the target xml document are not equivalent strings.

276.1. XMLSECURITY OPTIONS

The XML Security dataformat supports 12 options which are listed below.

Name	Default	Java Type	Description
xmlCipherAlgorithm	**TRIPLEDES**	**String**	The cipher algorithm to be used for encryption/decryption of the XML message content. The available choices are: XMLCipher.TRIPLEDES XMLCipher.AES_128 XMLCipher.AES_128_GCM XMLCipher.AES_192 XMLCipher.AES_192_GCM XMLCipher.AES_256 XMLCipher.AES_256_GCM XMLCipher.SEED_128 XMLCipher.CAMELLIA_128 XMLCipher.CAMELLIA_192 XMLCipher.CAMELLIA_256 The default value is MLCipher.TRIPLEDES
passPhrase		**String**	A String used as passPhrase to encrypt/decrypt content. The passPhrase has to be provided. If no passPhrase is specified, a default passPhrase is used. The passPhrase needs to be put together in conjunction with the appropriate encryption algorithm. For example using TRIPLEDES the passPhase can be a Only another 24 Byte key

Name	Default	Java Type	Description
secureTag		String	The XPath reference to the XML Element selected for encryption/decryption. If no tag is specified, the entire payload is encrypted/decrypted.
secureTagContents	false	Boolean	A boolean value to specify whether the XML Element is to be encrypted or the contents of the XML Element false = Element Level true = Element Content Level
keyCipherAlgorithm	RSA_OAEP	String	The cipher algorithm to be used for encryption/decryption of the asymmetric key. The available choices are: XMLCipher.RSA_v1dot5 XMLCipher.RSA_OAEP XMLCipher.RSA_OAEP_11 The default value is XMLCipher.RSA_OAEP
recipientKeyAlias		String	The key alias to be used when retrieving the recipient's public or private key from a KeyStore when performing asymmetric key encryption or decryption.
keyOrTrustStoreParametersId		String	Refers to a KeyStore instance to lookup in the registry, which is used for configuration options for creating and loading a KeyStore instance that represents the sender's trustStore or recipient's keyStore.
keyPassword		String	The password to be used for retrieving the private key from the KeyStore. This key is used for asymmetric decryption.
digestAlgorithm	SHA1	String	The digest algorithm to use with the RSA OAEP algorithm. The available choices are: XMLCipher.SHA1 XMLCipher.SHA256 XMLCipher.SHA512 The default value is XMLCipher.SHA1
mgfAlgorithm	MGF1_SHA1	String	The MGF Algorithm to use with the RSA OAEP algorithm. The available choices are: EncryptionConstants.MGF1_SHA1 EncryptionConstants.MGF1_SHA256 EncryptionConstants.MGF1_SHA512 The default value is EncryptionConstants.MGF1_SHA1
addKeyValueForEncryptedKey	true	Boolean	Whether to add the public key used to encrypt the session key as a KeyValue in the EncryptedKey structure or not.
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

276.1.1. Key Cipher Algorithm

As of Camel 2.12.0, the default Key Cipher Algorithm is now XMLCipher.RSA_OAEP instead of XMLCipher.RSA_v1dot5. Usage of XMLCipher.RSA_v1dot5 is discouraged due to various attacks. Requests that use RSA v1.5 as the key cipher algorithm will be rejected unless it has been explicitly configured as the key cipher algorithm.

276.2. MARSHAL

In order to encrypt the payload, the **marshal** processor needs to be applied on the route followed by the **secureXML()** tag.

276.3. UNMARSHAL

In order to decrypt the payload, the **unmarshal** processor needs to be applied on the route followed by the **secureXML()** tag.

276.4. EXAMPLES

Given below are several examples of how marshalling could be performed at the Document, Element, and Content levels.

276.4.1. Full Payload encryption/decryption

```
from("direct:start")
    .marshal().secureXML()
    .unmarshal().secureXML()
    .to("direct:end");
```

276.4.2. Partial Payload Content Only encryption/decryption

```
String tagXPATH = "//cheesesites/italy/cheese";
boolean secureTagContent = true;
...
from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent)
    .unmarshal().secureXML(tagXPATH, secureTagContent)
    .to("direct:end");
```

276.4.3. Partial Multi Node Payload Content Only encryption/decryption

```
String tagXPATH = "//cheesesites/*/cheese";
boolean secureTagContent = true;
...
from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent)
    .unmarshal().secureXML(tagXPATH, secureTagContent)
    .to("direct:end");
```

276.4.4. Partial Payload Content Only encryption/decryption with choice of passPhrase(password)

```
String tagXPATH = "//cheesesites/italy/cheese";
boolean secureTagContent = true;
...
String passPhrase = "Just another 24 Byte key";
from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent, passPhrase)
    .unmarshal().secureXML(tagXPATH, secureTagContent, passPhrase)
    .to("direct:end");
```

276.4.5. Partial Payload Content Only encryption/decryption with passPhrase(password) and Algorithm

```
import org.apache.xml.security.encryption.XMLCipher;
....
String tagXPATH = "//cheesesites/italy/cheese";
boolean secureTagContent = true;
String passPhrase = "Just another 24 Byte key";
String algorithm= XMLCipher.TRIPLEDES;
from("direct:start")
    .marshal().secureXML(tagXPATH, secureTagContent, passPhrase, algorithm)
    .unmarshal().secureXML(tagXPATH, secureTagContent, passPhrase, algorithm)
    .to("direct:end");
```

276.4.6. Partial Payload Content with Namespace support

Java DSL

```
final Map<String, String> namespaces = new HashMap<String, String>();
namespaces.put("cust", "http://cheese.xmlsecurity.camel.apache.org/");

final KeyStoreParameters tsParameters = new KeyStoreParameters();
tsParameters.setPassword("password");
tsParameters.setResource("sender.ts");

context.addRoutes(new RouteBuilder() {
    public void configure() {
        from("direct:start")
          .marshal().secureXML("//cust:cheesesites/italy", namespaces, true, "recipient",
                               testCypherAlgorithm, XMLCipher.RSA_v1dot5, tsParameters)
          .to("mock:encrypted");
    }
}
```

Spring XML

A namespace prefix that is defined as part of the **camelContext** definition can be re-used in context within the data format **secureTag** attribute of the **secureXML** element.

```
<camelContext id="springXmlSecurityDataFormatTestCamelContext"
              xmlns="http://camel.apache.org/schema/spring"
              xmlns:cheese="http://cheese.xmlsecurity.camel.apache.org/">
    <route>
        <from uri="direct://start"/>
```

```
        <marshal>
          <secureXML secureTag="//cheese:cheesesites/italy"
                secureTagContents="true"/>
        </marshal>
        ...
```

276.4.7. Asymmetric Key Encryption

Spring XML Sender

```
<!-- trust store configuration -->
<camel:keyStoreParameters id="trustStoreParams" resource="./sender.ts" password="password"/>

<camelContext id="springXmlSecurityDataFormatTestCamelContext"
          xmlns="http://camel.apache.org/schema/spring"
          xmlns:cheese="http://cheese.xmlsecurity.camel.apache.org/">
  <route>
    <from uri="direct://start"/>
      <marshal>
        <secureXML secureTag="//cheese:cheesesites/italy"
              secureTagContents="true"
              xmlCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
              keyCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
              recipientKeyAlias="recipient"
              keyOrTrustStoreParametersId="trustStoreParams"/>
      </marshal>
      ...
```

Spring XML Recipient

```
<!-- key store configuration -->
<camel:keyStoreParameters id="keyStoreParams" resource="./recipient.ks" password="password" />

<camelContext id="springXmlSecurityDataFormatTestCamelContext"
          xmlns="http://camel.apache.org/schema/spring"
          xmlns:cheese="http://cheese.xmlsecurity.camel.apache.org/">
  <route>
    <from uri="direct://encrypted"/>
      <unmarshal>
        <secureXML secureTag="//cheese:cheesesites/italy"
              secureTagContents="true"
              xmlCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
              keyCipherAlgorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
              recipientKeyAlias="recipient"
              keyOrTrustStoreParametersId="keyStoreParams"
              keyPassword="privateKeyPassword" />
      </unmarshal>
      ...
```

276.5. DEPENDENCIES

This data format is provided within the **camel-xmlsecurity** component.

CHAPTER 277. SEDA COMPONENT

Available as of Camel version 1.1

The **seda:** component provides asynchronous SEDA behavior, so that messages are exchanged on a BlockingQueue and consumers are invoked in a separate thread from the producer.

Note that queues are only visible within a *single* CamelContext. If you want to communicate across **CamelContext** instances (for example, communicating between Web applications), see the VM component.

This component does not implement any kind of persistence or recovery, if the VM terminates while messages are yet to be processed. If you need persistence, reliability or distributed SEDA, try using either JMS or ActiveMQ.

TIP:*Synchronous* The Direct component provides synchronous invocation of any consumers when a producer sends a message exchange.

277.1. URI FORMAT

```
seda:someName[?options]
```

Where **someName** can be any string that uniquely identifies the endpoint within the current CamelContext.

You can append query options to the URI in the following format: **?option=value&option=value&…**

277.2. OPTIONS

The SEDA component supports 4 options which are listed below.

Name	Description	Default	Type
queueSize (advanced)	Sets the default maximum capacity of the SEDA queue (i.e., the number of messages it can hold).		int
concurrentConsumers (consumer)	Sets the default number of concurrent threads processing exchanges.	1	int
defaultQueueFactory (advanced)	Sets the default queue factory.		Exchange>
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The SEDA endpoint is configured using URI syntax:

```
seda:name
```

with the following path and query parameters:

277.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of queue		String

277.2.2. Query Parameters (16 parameters):

Name	Description	Default	Type
size (common)	The maximum capacity of the SEDA queue (i.e., the number of messages it can hold).	2147483647	int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
concurrentConsumers (consumer)	Number of concurrent threads processing exchanges.	1	int
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
limitConcurrentConsumers (consumer)	Whether to limit the number of concurrentConsumers to the maximum of 500. By default, an exception will be thrown if an endpoint is configured with a greater number. You can disable that check by turning this option off.	true	boolean

Name	Description	Default	Type
multipleConsumers (consumer)	Specifies whether multiple consumers are allowed. If enabled, you can use SEDA for Publish-Subscribe messaging. That is, you can send a message to the SEDA queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.	false	boolean
pollTimeout (consumer)	The timeout used when polling. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
purgeWhenStopping (consumer)	Whether to purge the task queue when stopping the consumer/route. This allows to stop faster, as any pending messages on the queue is discarded.	false	boolean
blockWhenFull (producer)	Whether a thread that sends messages to a full SEDA queue will block until the queue's capacity is no longer exhausted. By default, an exception will be thrown stating that the queue is full. By enabling this option, the calling thread will instead block and wait until the message can be accepted.	false	boolean
discardIfNoConsumers (producer)	Whether the producer should discard the message (do not add the message to the queue), when sending to a queue with no active consumers. Only one of the options discardIfNoConsumers and failIfNoConsumers can be enabled at the same time.	false	boolean
failIfNoConsumers (producer)	Whether the producer should fail by throwing an exception, when sending to a queue with no active consumers. Only one of the options discardIfNoConsumers and failIfNoConsumers can be enabled at the same time.	false	boolean
timeout (producer)	Timeout (in milliseconds) before a SEDA producer will stop waiting for an asynchronous task to complete. You can disable timeout by using 0 or a negative value.	30000	long
waitForTaskToComplete (producer)	Option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: Always, Never or IfReplyExpected. The first two values are self-explanatory. The last value, IfReplyExpected, will only wait if the message is Request Reply based. The default option is IfReplyExpected.	IfReplyExpected	WaitForTaskToComplete

Name	Description	Default	Type
queue (advanced)	Define the queue instance which will be used by the endpoint. This option is only for rare use-cases where you want to use a custom queue instance.		BlockingQueue
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

277.3. CHOOSING BLOCKINGQUEUE IMPLEMENTATION

Available as of Camel 2.12

By default, the SEDA component always intantiates LinkedBlockingQueue, but you can use different implementation, you can reference your own BlockingQueue implementation, in this case the size option is not used

```
<bean id="arrayQueue" class="java.util.ArrayBlockingQueue">
  <constructor-arg index="0" value="10" ><!-- size -->
  <constructor-arg index="1" value="true" ><!-- fairness -->
</bean>

<!-- ... and later -->
<from>seda:array?queue=#arrayQueue</from>
```

Or you can reference a BlockingQueueFactory implementation, 3 implementations are provided LinkedBlockingQueueFactory, ArrayBlockingQueueFactory and PriorityBlockingQueueFactory:

```
<bean id="priorityQueueFactory"
class="org.apache.camel.component.seda.PriorityBlockingQueueFactory">
  <property name="comparator">
    <bean class="org.apache.camel.demo.MyExchangeComparator" />
  </property>
</bean>

<!-- ... and later -->
<from>seda:priority?queueFactory=#priorityQueueFactory&size=100</from>
```

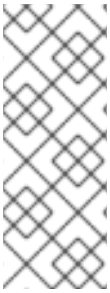
277.4. USE OF REQUEST REPLY

The SEDA component supports using Request Reply, where the caller will wait for the Async route to complete. For instance:

```
from("mina:tcp://0.0.0.0:9876?textline=true&sync=true").to("seda:input");

from("seda:input").to("bean:processInput").to("bean:createResponse");
```

In the route above, we have a TCP listener on port 9876 that accepts incoming requests. The request is routed to the **seda:input** queue. As it is a Request Reply message, we wait for the response. When the

consumer on the **seda:input** queue is complete, it copies the response to the original message response.



NOTE

until 2.2: Works only with 2 endpoints Using Request Reply over SEDA or VM only works with 2 endpoints. You **cannot** chain endpoints by sending to A → B → C etc. Only between A → B. The reason is the implementation logic is fairly simple. To support 3+ endpoints makes the logic much more complex to handle ordering and notification between the waiting threads properly. This has been improved in **Camel 2.3** onwards, which allows you to chain as many endpoints as you like.

277.5. CONCURRENT CONSUMERS

By default, the SEDA endpoint uses a single consumer thread, but you can configure it to use concurrent consumer threads. So instead of thread pools you can use:

```
from("seda:stageName?concurrentConsumers=5").process(...)
```

As for the difference between the two, note a *thread pool* can increase/shrink dynamically at runtime depending on load, whereas the number of concurrent consumers is always fixed.

277.6. THREAD POOLS

Be aware that adding a thread pool to a SEDA endpoint by doing something like:

```
from("seda:stageName").thread(5).process(...)
```

Can wind up with two **BlockQueues**: one from the SEDA endpoint, and one from the workqueue of the thread pool, which may not be what you want. Instead, you might wish to configure a Direct endpoint with a thread pool, which can process messages both synchronously and asynchronously. For example:

```
from("direct:stageName").thread(5).process(...)
```

You can also directly configure number of threads that process messages on a SEDA endpoint using the **concurrentConsumers** option.

277.7. SAMPLE

In the route below we use the SEDA queue to send the request to this async queue to be able to send a fire-and-forget message for further processing in another thread, and return a constant reply in this thread to the original caller.

Here we send a Hello World message and expects the reply to be OK.

The "Hello World" message will be consumed from the SEDA queue from another thread for further processing. Since this is from a unit test, it will be sent to a **mock** endpoint where we can do assertions in the unit test.

277.8. USING MULTIPLECONSUMERS

Available as of Camel 2.2

In this example we have defined two consumers and registered them as spring beans.

Since we have specified **multipleConsumers=true** on the seda foo endpoint we can have those two consumers receive their own copy of the message as a kind of pub-sub style messaging.

As the beans are part of an unit test they simply send the message to a mock endpoint, but notice how we can use @Consume to consume from the seda queue.

277.9. EXTRACTING QUEUE INFORMATION.

If needed, information such as queue size, etc. can be obtained without using JMX in this fashion:

```
SedaEndpoint seda = context.getEndpoint("seda:xxxx");
int size = seda.getExchanges().size();
```

277.10. SEE ALSO

- VM
- Disruptor
- Direct
- Async

CHAPTER 278. JAVA OBJECT SERIALIZATION DATAFORMAT

Available as of Camel version 2.12

Serialization is a Data Format which uses the standard Java Serialization mechanism to unmarshal a binary payload into Java objects or to marshal Java objects into a binary blob.
For example the following uses Java serialization to unmarshal a binary file then send it as an ObjectMessage to ActiveMQ

```
from("file://foo/bar").
  unmarshal().serialization().
  to("activemq:Some.Queue");
```

278.1. OPTIONS

The Java Object Serialization dataformat supports 1 options which are listed below.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

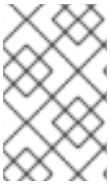
278.2. DEPENDENCIES

This data format is provided in **camel-core** so no additional dependencies is needed.

CHAPTER 279. SERVICENOW COMPONENT

Available as of Camel version 2.18

The ServiceNow component provides access to ServiceNow platform through their REST API.



NOTE

From Camel 2.18.1 the component supports multiple version of ServiceNow platform with default to Helsinki. Supported version are Table 279.1, “API Mapping” and Table 279.2, “API Mapping”

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-servicenow</artifactId>
    <version>${camel-version}</version>
</dependency>
```

279.1. URI FORMAT

```
servicenow://instanceName?[options]
```

279.2. OPTIONS

The ServiceNow component supports 14 options which are listed below.

Name	Description	Default	Type
instanceName (advanced)	The ServiceNow instance name		String
configuration (advanced)	The ServiceNow default configuration		ServiceNowConfiguration
apiUrl (producer)	The ServiceNow REST API url		String
userName (security)	ServiceNow user account name		String
password (security)	ServiceNow account password		String
oauthClientId (security)	OAuth2 ClientID		String

Name	Description	Default	Type
oauthClientSecret (security)	OAuth2 ClientSecret		String
oauthTokenUrl (security)	OAuth token Url		String
proxyHost (advanced)	The proxy host name		String
proxyPort (advanced)	The proxy port number		Integer
proxyUserName (security)	Username for proxy authentication		String
proxyPassword (security)	Password for proxy authentication		String
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The ServiceNow endpoint is configured using URI syntax:

```
servicenow:instanceName
```

with the following path and query parameters:

279.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
instanceName	**Required** The ServiceNow instance name		String

279.2.2. Query Parameters (44 parameters):

Name	Description	Default	Type
display (producer)	Set this parameter to true to return only scorecards where the indicator Display field is selected. Set this parameter to all to return scorecards with any Display field value. This parameter is true by default.	true	String
displayValue (producer)	Return the display value (true), actual value (false), or both (all) for reference fields (default: false)	false	String
excludeReferenceLink (producer)	True to exclude Table API links for reference fields (default: false)		Boolean
favorites (producer)	Set this parameter to true to return only scorecards that are favorites of the querying user.		Boolean
includeAggregates (producer)	Set this parameter to true to always return all available aggregates for an indicator, including when an aggregate has already been applied. If a value is not specified, this parameter defaults to false and returns no aggregates.		Boolean
includeAvailableAggregates (producer)	Set this parameter to true to return all available aggregates for an indicator when no aggregate has been applied. If a value is not specified, this parameter defaults to false and returns no aggregates.		Boolean
includeAvailableBreakdowns (producer)	Set this parameter to true to return all available breakdowns for an indicator. If a value is not specified, this parameter defaults to false and returns no breakdowns.		Boolean
includeScoreNotes (producer)	Set this parameter to true to return all notes associated with the score. The note element contains the note text as well as the author and timestamp when the note was added.		Boolean
includeScores (producer)	Set this parameter to true to return all scores for a scorecard. If a value is not specified, this parameter defaults to false and returns only the most recent score value.		Boolean
inputDisplayValue (producer)	True to set raw value of input fields (default: false)		Boolean
key (producer)	Set this parameter to true to return only scorecards for key indicators.		Boolean

Name	Description	Default	Type
models (producer)	Defines both request and response models		String
perPage (producer)	Enter the maximum number of scorecards each query can return. By default this value is 10, and the maximum is 100.	10	Integer
release (producer)	The ServiceNow release to target, default to Helsinki See https://docs.servicenow.com	HELSINKI	ServiceNowRelease
requestModels (producer)	Defines the request model		String
resource (producer)	The default resource, can be overridden by header CamelServiceNowResource		String
responseModels (producer)	Defines the response model		String
sortBy (producer)	Specify the value to use when sorting results. By default, queries sort records by value.		String
sortDir (producer)	Specify the sort direction, ascending or descending. By default, queries sort records in descending order. Use sysparm_sortdir=asc to sort in ascending order.		String
suppressAutoSysField (producer)	True to suppress auto generation of system fields (default: false)		Boolean
suppressPaginationHeader (producer)	Set this value to true to remove the Link header from the response. The Link header allows you to request additional pages of data when the number of records matching your query exceeds the query limit		Boolean
table (producer)	The default table, can be overridden by header CamelServiceNowTable		String
target (producer)	Set this parameter to true to return only scorecards that have a target.		Boolean
topLevelOnly (producer)	Gets only those categories whose parent is a catalog.		Boolean
apiVersion (advanced)	The ServiceNow REST API version, default latest		String

Name	Description	Default	Type
dateFormat (advanced)	The date format used for Json serialization/deserialization	yyyy-MM-dd	String
dateTimeFormat (advanced)	The date-time format used for Json serialization/deserialization	yyyy-MM-dd HH:mm:ss	String
httpClientPolicy (advanced)	To configure http-client		HTTPClientPolicy
mapper (advanced)	Sets Jackson's ObjectMapper to use for request/reply		ObjectMapper
proxyAuthorizationPolicy (advanced)	To configure proxy authentication		ProxyAuthorization Policy
retrieveTargetRecordOn Import (advanced)	Set this parameter to true to retrieve the target record when using import set api. The import set result is then replaced by the target record	false	Boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
timeFormat (advanced)	The time format used for Json serialization/deserialization	HH:mm:ss	String
proxyHost (proxy)	The proxy host name		String
proxyPort (proxy)	The proxy port number		Integer
apiUrl (security)	The ServiceNow REST API url		String
oauthClientId (security)	OAuth2 ClientID		String
oauthClientSecret (security)	OAuth2 ClientSecret		String
oauthTokenUrl (security)	OAuth token Url		String

Name	Description	Default	Type
password (security)	**Required** ServiceNow account password, MUST be provided		String
proxyPassword (security)	Password for proxy authentication		String
proxyUserName (security)	Username for proxy authentication		String
sslContextParameters (security)	To configure security using SSLContextParameters. See http://camel.apache.org/camel-configuration-utilities.html		SSLContextParameters
userName (security)	**Required** ServiceNow user account name, MUST be provided		String

279.3. HEADERS

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowResource	String	-	-	The resource to access
CamelServiceNowAction	String	-	-	The action to perform
CamelServiceNowActionSubject	-	-	String	The subject to which the action should be applied
CamelServiceNowModel	Class	-	-	The data model

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowRequestModel	Class	-	-	The request data model
CamelServiceNowResponseModel	Class	-	-	The response data model
CamelServiceNowOffsetNext	-	-	-	-
CamelServiceNowOffsetPrev	-	-	-	-
CamelServiceNowOffsetFirst	-	-	-	-
CamelServiceNowOffsetLast	-	-	-	-
CamelServiceNowContentType	-	-	-	-
CamelServiceNowContentEncoding	-	-	-	-

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowContentMeta	-	-	-	-
CamelServiceNowSysId	String	sys_id	-	-
CamelServiceNowUserSysId	String	user_sysid	-	-
CamelServiceNowUserId	String	user_id	-	-
CamelServiceNowCartItemId	String	cart_item_id	-	-
CamelServiceNowFileName	String	file_name	-	-
CamelServiceNowTable	String	table_name	-	-
CamelServiceNowTableSysId	String	table_sys_id	-	-

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowEncryptionContext	String	encryption_context	-	-
CamelServiceNowCategory	String	sysparm_category	-	-
CamelServiceNowType	String	sysparm_type	-	-
CamelServiceNowCatalog	String	sysparm_catalog	-	-
CamelServiceNowQuery	String	sysparm_query	-	-
CamelServiceNowDisplayValue	String	sysparm_display_value	displayValue	-
CamelServiceNowInputDisplayValue	Boolean	sysparm_input_display_value	inputDisplayValue	-

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowExcludeReferenceLink	Boolean	sysparm_exclude_reference_link	excludeReferenceLink	-
CamelServiceNowFields	String	sysparm_fields	-	-
CamelServiceNowLimit	Integer	sysparm_limit	-	-
CamelServiceNowText	String	sysparm_text	-	-
CamelServiceNowOffset	Integer	sysparm_offset	-	-
CamelServiceNowView	String	sysparm_view	-	-
CamelServiceNowSuppressAutoSysField	Boolean	sysparm_suppress_auto_sys_field	suppressAutoSysField	-
CamelServiceNowSuppressPaginationHeader	Boolean	sysparm_suppress_pagination_header	suppressPaginationHeader	-

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowMinFields	String	sysparm_min_fields	-	-
CamelServiceNowMaxFields	String	sysparm_max_fields	-	-
CamelServiceNowSumFields	String	sysparm_sum_fields	-	-
CamelServiceNowAvgFields	String	sysparm_avg_fields	-	-
CamelServiceNowCount	Boolean	sysparm_count	-	-
CamelServiceGroupBy	String	sysparm_group_by	-	-
CamelServiceOrderBy	String	sysparm_order_by	-	-
CamelServiceHaving	String	sysparm_having	-	-
CamelServiceNowUUID	String	sysparm_uuid	-	-

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowBreakdown	String	sysparm_breakdown	-	-
CamelServiceNowIncludeScores	Boolean	sysparm_include_scores	includeScores	-
CamelServiceNowIncludeScoreNotes	Boolean	sysparm_include_score_notes	includeScoreNotes	-
CamelServiceNowIncludeAggregates	Boolean	sysparm_include_aggregates	includeAggregates	-
CamelServiceNowIncludeAvailableBreakdowns	Boolean	sysparm_include_available_breakdowns	includeAvailableBreakdowns	-
CamelServiceNowIncludeAvailableAggregates	Boolean	sysparm_include_available_aggregates	includeAvailableAggregates	-
CamelServiceNowFavorites	Boolean	sysparm_favorites	favorites	-

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowKey	Boolean	sysparm_key	key	-
CamelServiceNowTarget	Boolean	sysparm_target	target	-
CamelServiceNowDisplay	String	sysparm_display	display	-
CamelServiceNowPerPage	Integer	sysparm_per_page	perPage	-
CamelServiceNowSortBy	String	sysparm_sortby	sortBy	-
CamelServiceNowSortDir	String	sysparm_sortdit	sortDir	-
CamelServiceNowContains	String	sysparm_contains	-	-
CamelServiceNowTags	String	sysparm_tags	-	-
CamelServiceNowPage	String	sysparm_page	-	-

Name	Type	ServiceNow API Parameter	Endpoint option	Description
CamelServiceNowElementsFilter	String	sysparm_elements_filter	-	-
CamelServiceNowBreakdownRelation	String	sysparm_breakdown_relation	-	-
CamelServiceNowDataSource	String	sysparm_data_source	-	-
CamelServiceNowTopLevelOnly	Boolean	sysparm_top_level_only	topLevelOnly	-
CamelServiceNowApiVersion	String	-	-	The REST API version
CamelServiceNowResponseMeta	Map	-	-	Meta data provided along with a response

Table 279.1. API Mapping

Camel ServiceNowResource	Camel ServiceNowAction	Method	API URI
TABLE	RETRIEVE	GET	/api/now/v1/table/{table_name}/{sys_id}
	CREATE	POST	/api/now/v1/table/{table_name}
	MODIFY	PUT	/api/now/v1/table/{table_name}/{sys_id}
	DELETE	DELETE	/api/now/v1/table/{table_name}/{sys_id}
	UPDATE	PATCH	/api/now/v1/table/{table_name}/{sys_id}
AGGREGATE	RETRIEVE	GET	/api/now/v1/stats/{table_name}
IMPORT	RETRIEVE	GET	/api/now/import/{table_name}/{sys_id}
	CREATE	POST	/api/now/import/{table_name}

NOTE

Fuji REST API Documentation

Table 279.2. API Mapping

Camel ServiceNowResource	Camel ServiceNowAction	Camel ServiceNowActionSubject	Method	API URI
TABLE	RETRIEVE		GET	/api/now/v1/table/{table_name}/{sys_id}
	CREATE		POST	/api/now/v1/table/{table_name}

Camel ServiceNowResource	Camel ServiceNowAction	Camel ServiceNowActionSubject	Method	API URI
	MODIFY		PUT	/api/now/v1/table/{table_name}/{sys_id}
	DELETE		DELETE	/api/now/v1/table/{table_name}/{sys_id}
	UPDATE		PATCH	/api/now/v1/table/{table_name}/{sys_id}
AGGREGATE	RETRIEVE		GET	/api/now/v1/stats/{table_name}
IMPORT	RETRIEVE		GET	/api/now/import/{table_name}/{sys_id}
	CREATE		POST	/api/now/import/{table_name}
ATTACHMENT	RETRIEVE		GET	/api/now/api/now/attachment/{sys_id}
	CONTENT		GET	/api/now/attachment/{sys_id}/file
	UPLOAD		POST	/api/now/api/now/attachment/file
	DELETE		DELETE	/api/now/attachment/{sys_id}
SCORECARDS	RETRIEVE	PERFORMANCE_ANALYTICS	GET	/api/now/pa/scorecards
MISC	RETRIEVE	USER_ROLE_INHERITANCE	GET	/api/global/user_role_inheritance

CamelServiceNowResource	CamelServiceNowAction	CamelServiceNowActionSubject	Method	API URI
	CREATE	IDENTIFY_RECONCILE	POST	/api/now/identifyreconcile
SERVICE_CATALOG	RETRIEVE		GET	/sn_sc/servicecatalog/catalogs/{sys_id}
	RETRIEVE	CATEGORIES	GET	/sn_sc/servicecatalog/catalogs/{sys_id}/categories
SERVICE_CATALOG_ITEMS	RETRIEVE		GET	/sn_sc/servicecatalog/items/{sys_id}
	RETRIEVE	SUBMIT_GUIDE	POST	/sn_sc/servicecatalog/items/{sys_id}/submit_guide
	RETRIEVE	CHECKOUT_GUIDE	POST	/sn_sc/servicecatalog/items/{sys_id}/checkout_guide
	CREATE	SUBJECT_CART	POST	/sn_sc/servicecatalog/items/{sys_id}/add_to_cart
	CREATE	SUBJECT_PRODUCER	POST	/sn_sc/servicecatalog/items/{sys_id}/submit_producer
SERVICE_CATALOG_CARTS	RETRIEVE		GET	/sn_sc/servicecatalog/cart
	RETRIEVE	DELIVERY_ADDRESS	GET	/sn_sc/servicecatalog/cart/delivery_address/{user_id}
	RETRIEVE	CHECKOUT	POST	/sn_sc/servicecatalog/cart/checkout

Camel ServiceNowResource	Camel ServiceNowAction	Camel ServiceNowActionSubject	Method	API URI
	UPDATE		POST	/sn_sc/servicecatalog/cart/{cart_item_id}
	UPDATE	CHECKOUT	POST	/sn_sc/servicecatalog/cart/submit_order
	DELETE		DELETE	/sn_sc/servicecatalog/cart/{sys_id}/empty
SERVICE_CATALOG_CATEGORIES	RETRIEVE		GET	/sn_sc/servicecatalog/categories/{sys_id}

NOTE

Helsinki REST API Documentation

279.4. USAGE EXAMPLES:

Retrieve 10 Incidents

```
context.addRoutes(new RouteBuilder() {
    public void configure() {
        from("direct:servicenow")
            .to("servicenow:{{env:SERVICENOW_INSTANCE}}"
                + "?userName={{env:SERVICENOW_USERNAME}}"
                + "&password={{env:SERVICENOW_PASSWORD}}"
                + "&oauthClientId={{env:SERVICENOW_OAUTH2_CLIENT_ID}}"
                + "&oauthClientSecret={{env:SERVICENOW_OAUTH2_CLIENT_SECRET}}"
            .to("mock:servicenow");
    }
});

FluentProducerTemplate.on(context)
    .withHeader(ServiceNowConstants.RESOURCE, "table")
    .withHeader(ServiceNowConstants.ACTION, ServiceNowConstants.ACTION_RETRIEVE)
    .withHeader(ServiceNowConstants.SYSPARM_LIMIT.getId(), "10")
    .withHeader(ServiceNowConstants.TABLE, "incident")
    .withHeader(ServiceNowConstants.MODEL, Incident.class)
    .to("direct:servicenow")
    .send();
```

CHAPTER 280. SERVLET COMPONENT

Available as of Camel version 2.0

The **servlet:** component provides HTTP based endpoints for consuming HTTP requests that arrive at a HTTP endpoint that is bound to a published Servlet.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-servlet</artifactId>
  <version>x.x.x</version>
  <\!-\- use the same version as your Camel core version \-->
</dependency>
```

INFO: **Stream**. Servlet is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**. If you find a situation where the message body appears to be empty or you need to access the data multiple times (eg: doing multicasting, or redelivery error handling) you should use Stream caching or convert the message body to a **String** which is safe to be read multiple times.

280.1. URI FORMAT

```
servlet://relative_path[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

280.2. OPTIONS

The Servlet component supports 8 options which are listed below.

Name	Description	Default	Type
servletName (consumer)	Default name of servlet to use. The default name is CamelServlet.		String
httpRegistry (consumer)	To use a custom org.apache.camel.component.servlet.HttpRegistry.		HttpRegistry
attachmentMultipart Binding (consumer)	Whether to automatic bind multipart/form-data as attachments on the Camel Exchange. The options attachmentMultipartBinding=true and disableStreamCache=false cannot work together. Remove disableStreamCache to use AttachmentMultipartBinding. This is turn off by default as this may require servlet specific configuration to enable this when using Servlet's.	false	boolean

Name	Description	Default	Type
httpBinding (advanced)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
httpConfiguration (advanced)	To use the shared HttpConfiguration as base configuration.		HttpConfiguration
allowJavaSerializedObject (advanced)	Whether to allow java serialization when a request uses context-type=application/x-java-serialized-object. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Servlet endpoint is configured using URI syntax:

```
servlet:contextPath
```

with the following path and query parameters:

280.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
contextPath	**Required** The context-path to use		String

280.2.2. Query Parameters (21 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
disableStreamCache (common)	Determines whether or not the raw input stream from Servlet is cached or not (Camel will read the stream into a in memory/overflow to file, Stream caching) cache. By default Camel will cache the Servlet input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support reading the stream multiple times. If you use Servlet to bridge/proxy an endpoint then consider enabling this option to improve performance, in case you do not need to read the message payload multiple times. The http/http4 producer will by default cache the response body stream. If setting this option to true, then the producers will not cache the response body stream but use the response stream as-is as the message body.	false	boolean
headerFilterStrategy (common)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
httpBinding (common)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		HttpBinding
async (consumer)	Configure the consumer to work in async mode	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
chunked (consumer)	If this option is false the Servlet will disable the HTTP streaming and set the content-length header on the response	true	boolean
httpMethodRestrict (consumer)	Used to only allow consuming if the HttpMethod matches, such as GET/POST/PUT etc. Multiple methods can be specified separated by comma.		String

Name	Description	Default	Type
matchOnUriPrefix (consumer)	Whether or not the consumer should try to find a target consumer by matching the URI prefix if no exact match is found.	false	boolean
responseBufferSize (consumer)	To use a custom buffer size on the javax.servlet.ServletResponse.		Integer
servletName (consumer)	Name of the servlet to use	Camel Servlet	String
transferException (consumer)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
attachmentMultipartBinding (consumer)	Whether to automatic bind multipart/form-data as attachments on the Camel Exchange. The options attachmentMultipartBinding=true and disableStreamCache=false cannot work together. Remove disableStreamCache to use AttachmentMultipartBinding. This is turn off by default as this may require servlet specific configuration to enable this when using Servlet's.	false	boolean
eagerCheckContentAvailable (consumer)	Whether to eager check whether the HTTP requests has content if the content-length header is 0 or not present. This can be turned on in case HTTP clients do not send streamed data.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
optionsEnabled (consumer)	Specifies whether to enable HTTP OPTIONS for this Servlet consumer. By default OPTIONS is turned off.	false	boolean

Name	Description	Default	Type
traceEnabled (consumer)	Specifies whether to enable HTTP TRACE for this Servlet consumer. By default TRACE is turned off.	false	boolean
mapHttpMessageBody (advanced)	If this option is true then IN exchange Body of the exchange will be mapped to HTTP body. Setting this to false will avoid the HTTP mapping.	true	boolean
mapHttpMessageFormUrlEncodedBody (advanced)	If this option is true then IN exchange Form Encoded body of the exchange will be mapped to HTTP. Setting this to false will avoid the HTTP Form Encoded body mapping.	true	boolean
mapHttpMessageHeaders (advanced)	If this option is true then IN exchange Headers of the exchange will be mapped to HTTP headers. Setting this to false will avoid the HTTP Headers mapping.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

280.3. MESSAGE HEADERS

Camel will apply the same Message Headers as the HTTP component.

Camel will also populate **all request.parameter** and **request.headers**. For example, if a client request has the URL, **http://myserver/myserver?orderid=123**, the exchange will contain a header named **orderid** with the value 123.

280.4. USAGE

You can consume only from endpoints generated by the Servlet component. Therefore, it should be used only as input into your Camel routes. To issue HTTP requests against other HTTP endpoints, use the HTTP Component

280.5. PUTTING CAMEL JARS IN THE APP SERVER BOOT CLASSPATH

If you put the Camel JARs such as **camel-core**, **camel-servlet**, etc. in the boot classpath of your application server (eg usually in its lib directory), then mind that the servlet mapping list is now shared between multiple deployed Camel application in the app server.

Mind that putting Camel JARs in the boot classpath of the application server is generally not best practice!

So in those situations you **must** define a custom and unique servlet name in each of your Camel application, eg in the **web.xml** define:

```
<servlet>
  <servlet-name>MyServlet</servlet-name>
```

```
  <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

And in your Camel endpoints then include the servlet name as well

```
<route>
  <from uri="servlet://foo?servletName=MyServlet"/>
  ...
</route>
```

From **Camel 2.11** onwards Camel will detect this duplicate and fail to start the application. You can control to ignore this duplicate by setting the servlet init-parameter ignoreDuplicateServletName to true as follows:

```
  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <display-name>Camel Http Transport Servlet</display-name>
    <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servlet-class>
    <init-param>
      <param-name>ignoreDuplicateServletName</param-name>
      <param-value>true</param-value>
    </init-param>
  </servlet>
```

But its **strongly advised** to use unique servlet-name for each Camel application to avoid this duplication clash, as well any unforeseen side-effects.

280.6. SAMPLE

INFO: From Camel 2.7 onwards it's easier to use Servlet in Spring web applications. See Servlet Tomcat Example for details.

In this sample, we define a route that exposes a HTTP service at **http://localhost:8080/camel/services/hello**.
First, you need to publish the CamelHttpTransportServlet through the normal Web Container, or OSGi Service.
Use the **Web.xml** file to publish the CamelHttpTransportServlet as follows:

Then you can define your route as follows:

NOTE

Specify the relative path for camel-servlet endpoint Since we are binding the Http transport with a published servlet, and we don't know the servlet's application context path, the **camel-servlet** endpoint uses the relative path to specify the endpoint's URL. A client can access the **camel-servlet** endpoint through the servlet publish address: **("http://localhost:8080/camel/services") + RELATIVE_PATH("/hello")**.

280.6.1. Sample when using Spring 3.x

See Servlet Tomcat Example

280.6.2. Sample when using Spring 2.x

When using the Servlet component in a Camel/Spring application it's often required to load the Spring ApplicationContext *after* the Servlet component has started. This can be accomplished by using Spring's **ContextLoaderServlet** instead of **ContextLoaderListener**. In that case you'll need to start **ContextLoaderServlet** after CamelHttpTransportServlet like this:

```
<web-app>
  <servlet>
      <servlet-name>CamelServlet</servlet-name>
      <servlet-class>
         org.apache.camel.component.servlet.CamelHttpTransportServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet>
      <servlet-name>SpringApplicationContext</servlet-name>
      <servlet-class>
         org.springframework.web.context.ContextLoaderServlet
      </servlet-class>
      <load-on-startup>2</load-on-startup>
  </servlet>
<web-app>
```

280.6.3. Sample when using OSGi

From **Camel 2.6.0**, you can publish the CamelHttpTransportServlet as an OSGi service with help of SpringDM like this.

Then use this service in your camel route like this:

For versions prior to Camel 2.6 you can use an **Activator** to publish the CamelHttpTransportServlet on the OSGi platform

280.6.4. Usage with Spring-Boot

From **Camel 2.19.0** onwards, the *camel-servlet-starter* library binds automatically all the rest endpoints under the "/camel/*" context path. The following table summarizes the additional configuration properties available in the camel-servlet-starter library. The automatic mapping of the Camel servlet can also be disabled.

Spring-Boot Property	Default	Description
camel.component.servlet.mapping.enabled	**true**	Enables the automatic mapping of the servlet component into the Spring web context
camel.component.servlet.mapping.context-path	**/camel/***	Context path used by the servlet component for automatic mapping

Spring-Boot Property	Default	Description
camel.component.servlet.mapping.servlet-name	**CamelServlet**	The name of the Camel servlet

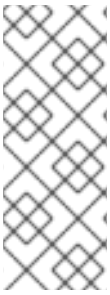
280.7. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Servlet Tomcat Example
- Servlet Tomcat No Spring Example
- HTTP
- Jetty

280.8. SERVLETLISTENER COMPONENT

Available as of Camel 2.11

This component is used for bootstrapping Camel applications in web applications. For example beforehand people would have to find their own way of bootstrapping Camel, or rely on 3rd party frameworks such as Spring to do it.



NOTE

Sidebar This component supports Servlet 2.x onwards, which mean it works also in older web containers; which is the goal of this component. Though Servlet 2.x requires to use a web.xml file as configuration. For Servlet 3.x containers you can use annotation driven configuration to boostrap Camel using the @WebListener, and implement your own class, where you boostrap Camel. Doing this still puts the challenge how to let end users easily configure Camel, which you get for free with the old school web.xml file.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-servletlistener</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

280.8.1. Using

You would need to chose one of the following implementations of the abstract class **org.apache.camel.component.servletlistener.CamelServletContextListener**.

- **JndiCamelServletContextListener** which uses the **JndiRegistry** to leverage JNDI for its registry.
- **SimpleCamelServletContextListener** which uses the **SimpleRegistry** to leverage a **java.util.Map** as its registry.

To use this you need to configure the **org.apache.camel.component.servletlistener.CamelServletContextListener** in the **WEB-INF/web.xml** file as shown below:

280.8.2. Options

The **org.apache.camel.component.servletlistener.CamelServletContextListener** supports the following options which can be configured as context-param in the web.xml file.

Option	Type	Description
propertyPlaceholder.XXX		To configure property placeholders in Camel. You should prefix the option with "propertyPlaceholder.", for example to configure the location, use propertyPlaceholder.location as name. You can configure all the options from the Properties component.
jmx.XXX		To configure JMX. You should prefix the option with "jmx.", for example to disable JMX, use jmx.disabled as name. You can configure all the options from **org.apache.camel.spi.ManagementAgent**. As well the options mentioned on the JMX page.
name	**String**	To configure the name of the CamelContext.
messageHistory	**Boolean**	**Camel 2.12.2:** Whether to enable or disable Message History (enabled by default).
streamCache	**Boolean**	Whether to enable Stream caching.
trace	**Boolean**	Whether to enable Tracer.
delayer	**Long**	To set a delay value for Delay Interceptor.
handleFault	**Boolean**	Whether to enable handle fault.
errorHandlerRef	**String**	Refers to a context scoped Error Handler to be used.

Option	Type	Description
autoStartup	**Boolean**	Whether to start all routes when starting Camel.
useMDCLogging	**Boolean**	Whether to use MDC logging.
useBreadcrumb	**Boolean**	Whether to use breadcrumb.
managementNamePattern	**String**	To set a custom naming pattern for JMX MBeans.
threadNamePattern	**String**	To set a custom naming pattern for threads.
properties.XXX		To set custom properties on **CamelContext.getProperties**. This is seldom in use.
routebuilder.XXX		To configure routes to be used. See below for more details.
CamelContextLifecycle		Refers to a FQN classname of an implementation of **org.apache.camel.component.servletlistener.CamelContextLifecycle**. Which allows to execute custom code before and after CamelContext has been started or stopped. See below for further details.
XXX		To set any option on CamelContext.

280.8.3. Examples

See Servlet Tomcat No Spring Example .

280.8.4. Accessing the created CamelContext

Available as of Camel 2.14/2.13.3/2.12.5

The created **CamelContext** is stored on the **ServletContext** as an attribute with the key "CamelContext". You can get hold of the CamelContext if you can get hold of the **ServletContext** as shown below:

```
ServletContext sc = ...
CamelContext camel = (CamelContext) sc.getAttribute("CamelContext");
```

280.8.5. Configuring routes

You need to configure which routes to use in the web.xml file. You can do this in a number of ways, though all the parameters must be prefixed with "routeBuilder".

280.8.5.1. Using a RouteBuilder class

By default Camel will assume the param-value is a FQN classname for a Camel RouteBuilder class, as shown below:

```
<context-param>
  <param-name>routeBuilder-MyRoute</param-name>
  <param-value>org.apache.camel.component.servletlistener.MyRoute</param-value>
</context-param>
```

You can specify multiple classes in the same param-value as shown below:

```
<context-param>
  <param-name>routeBuilder-routes</param-name>
  <!-- we can define multiple values separated by comma -->
  <param-value>
    org.apache.camel.component.servletlistener.MyRoute,
    org.apache.camel.component.servletlistener.routes.BarRouteBuilder
  </param-value>
</context-param>
```

The name of the parameter does not have a meaning at runtime. It just need to be unique and start with "routeBuilder". In the example above we have "routeBuilder-routes". But you could just as well have named it "routeBuilder.foo".

280.8.5.2. Using package scanning

You can also tell Camel to use package scanning, which mean it will look in the given package for all classes of RouteBuilder types and automatic adding them as Camel routes. To do that you need to prefix the value with "packagescan:" as shown below:

```
<context-param>
  <param-name>routeBuilder-MyRoute</param-name>
  <!-- define the routes using package scanning by prefixing with packagescan: -->
  <param-value>packagescan:org.apache.camel.component.servletlistener.routes</param-value>
</context-param>
```

280.8.5.3. Using a XML file

You can also define Camel routes using XML DSL, though as we are not using Spring or Blueprint the XML file can only contain Camel route(s).

In the web.xml you refer to the XML file which can be from "classpath", "file" or a "http" url, as shown below:

```
<context-param>
```

```
    <param-name>routeBuilder-MyRoute</param-name>
    <param-value>classpath:routes/myRoutes.xml</param-value>
  </context-param>
```

And the XML file is:

routes/myRoutes.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- the xmlns="http://camel.apache.org/schema/spring" is needed -->
<routes xmlns="http://camel.apache.org/schema/spring">

  <route id="foo">
    <from uri="direct:foo"/>
    <to uri="mock:foo"/>
  </route>

  <route id="bar">
    <from uri="direct:bar"/>
    <to uri="mock:bar"/>
  </route>

</routes>
```

Notice that in the XML file the root tag is <routes> which must use the namespace "http://camel.apache.org/schema/spring". This namespace is having the spring in the name, but that is because of historical reasons, as Spring was the first and only XML DSL back in the time. At runtime no Spring JARs is needed. Maybe in Camel 3.0 the namespace can be renamed to a generic name.

280.8.5.4. Configuring propert placeholders

Here is a snippet of a web.xml configuration for setting up property placeholders to load **myproperties.properties** from the classpath

```
  <!-- setup property placeholder to load properties from classpath -->
  <!-- we do this by setting the param-name with propertyPlaceholder. as prefix and then any options
such as location, cache etc -->
  <context-param>
    <param-name>propertyPlaceholder.location</param-name>
    <param-value>classpath:myproperties.properties</param-value>
  </context-param>
  <!-- for example to disable cache on properties component, you do -->
  <context-param>
    <param-name>propertyPlaceholder.cache</param-name>
    <param-value>false</param-value>
  </context-param>
```

280.8.5.5. Configuring JMX

Here is a snippet of a web.xml configuration for configuring JMX, such as disabling JMX.

```
  <!-- configure JMX by using names that is prefixed with jmx. -->
  <!-- in this example we disable JMX -->
  <context-param>
```

```
    <param-name>jmx.disabled</param-name>
    <param-value>true</param-value>
  </context-param>
```

JNDI or Simple as Camel Registry ^^^^^^^^^^^^^^^^^^

This component uses either JNDI or Simple as the Registry.
This allows you to lookup Beans and other services in JNDI, and as well to bind and unbind your own Beans.

This is done from Java code by implementing the **org.apache.camel.component.servletlistener.CamelContextLifecycle**.

280.8.5.6. Using custom CamelContextLifecycle

In the code below we use the callbacks **beforeStart** and **afterStop** to enlist our custom bean in the Simple Registry, and as well to cleanup when we stop.

Then we need to register this class in the web.xml file as shown below, using the parameter name "CamelContextLifecycle". The value must be a FQN which refers to the class implementing the **org.apache.camel.component.servletlistener.CamelContextLifecycle** interface.

```
  <context-param>
    <param-name>CamelContextLifecycle</param-name>
    <param-value>org.apache.camel.component.servletlistener.MyLifecycle</param-value>
  </context-param>
```

As we enlisted our HelloBean Bean using the name "myBean" we can refer to this Bean in the Camel routes as shown below:

```
public class MyBeanRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("seda:foo").routeId("foo")
            .to("bean:myBean")
            .to("mock:foo");
    }
}
```

Important: If you use **org.apache.camel.component.servletlistener.JndiCamelServletContextListener** then the **CamelContextLifecycle** must use the **JndiRegistry** as well. And likewise if the servlet is **org.apache.camel.component.servletlistener.SimpleCamelServletContextListener** then the **CamelContextLifecycle** must use the **SimpleRegistry**

280.8.6. See Also

- SERVLET
- Servlet Tomcat Example
- Servlet Tomcat No Spring Example

CHAPTER 281. SFTP COMPONENT

Available as of Camel version 1.1

This component provides access to remote file systems over the FTP and SFTP protocols.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

For more information you can look at FTP component

281.1. URI OPTIONS

The options below are exclusive for the FTPS component.

The SFTP component has no options.

The SFTP endpoint is configured using URI syntax:

```
sftp:host:port/directoryName
```

with the following path and query parameters:

281.1.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	**Required** Hostname of the FTP server		String
port	Port of the FTP server		int
directoryName	The starting directory		String

281.1.2. Query Parameters (111 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
charset (common)	This option is used to specify the encoding of the file. You can use this on the consumer, to specify the encodings of the files, which allow Camel to know the charset it should load the file content in case the file content is being accessed. Likewise when writing a file, you can use this option to specify which charset to write the file as well. Do mind that when writing the file Camel may have to read the message content into memory to be able to convert the data into the configured charset, so do not use this if you have big messages.		String
disconnect (common)	Whether or not to disconnect from remote FTP server right after use. Disconnect will only disconnect the current connection to the FTP server. If you have a consumer which you want to stop, then you need to stop the consumer/route instead.	false	boolean
doneFileName (common)	Producer: If provided, then Camel will write a 2nd done file when the original file has been written. The done file will be empty. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The done file will always be written in the same folder as the original file. Consumer: If provided, Camel will only consume files if a done file exists. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders.The done file is always expected in the same folder as the original file. Only $file.name and $file.name.noext is supported as dynamic placeholders.		String

Name	Description	Default	Type
fileName (common)	Use Expression such as File Language to dynamically set the filename. For consumers, it's used as a filename filter. For producers, it's used to evaluate the filename to write. If an expression is set, it take precedence over the CamelFileName header. (Note: The header itself can also be an Expression). The expression options support both String and Expression types. If the expression is a String type, it is always evaluated using the File Language. If the expression is an Expression type, the specified Expression type is used - this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: mydata-$date:now:yyyyMMdd.txt. The producers support the CamelOverruleFileName header which takes precedence over any existing CamelFileName header; the CamelOverruleFileName is a header that is used only once, and makes it easier as this avoids to temporary store CamelFileName and have to restore it afterwards.		String
jschLoggingLevel (common)	The logging level to use for JSCH activity logging. As JSCH is verbose at by default at INFO level the threshold is WARN by default.	WARN	LoggingLevel
separator (common)	Sets the path separator to be used. UNIX = Uses unix style path separator Windows = Uses windows style path separator Auto = (is default) Use existing path separator in file name	UNIX	PathSeparator
fastExistsCheck (common)	If set this option to be true, camel-ftp will use the list file directly to check if the file exists. Since some FTP server may not support to list the file directly, if the option is false, camel-ftp will use the old way to list the directory and check if the file exists. This option also influences readLock=changed to control whether it performs a fast check to update file information or not. This can be used to speed up the process if the FTP server has a lot of files.	false	boolean

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delete (consumer)	If true, the file will be deleted after it is processed successfully.	false	boolean
moveFailed (consumer)	Sets the move failure expression based on Simple language. For example, to move files into a .error subdirectory use: .error. Note: When moving the files to the fail location Camel will handle the error and will not pick up the file again.		String
noop (consumer)	If true, the file is not moved or deleted in any way. This option is good for readonly data, or for ETL type requirements. If noop=true, Camel will set idempotent=true as well, to avoid consuming the same files over and over again.	false	boolean
preMove (consumer)	Expression (such as File Language) used to dynamically set the filename when moving it before processing. For example to move in-progress files into the order directory set this value to order.		String
preSort (consumer)	When pre-sort is enabled then the consumer will sort the file and directory names during polling, that was retrieved from the file system. You may want to do this in case you need to operate on the files in a sorted order. The pre-sort is executed before the consumer starts to filter, and accept files to process by Camel. This option is default=false meaning disabled.	false	boolean
recursive (consumer)	If a directory, will look for files in all the sub-directories as well.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean

Name	Description	Default	Type
streamDownload (consumer)	Sets the download method to use when not using a local working directory. If set to true, the remote files are streamed to the route as they are read. When set to false, the remote files are loaded into memory before being sent into the route.	false	boolean
directoryMustExist (consumer)	Similar to startingDirectoryMustExist but this applies during polling recursive sub directories.	false	boolean
download (consumer)	Whether the FTP consumer should download the file. If this option is set to false, then the message body will be null, but the consumer will still trigger a Camel Exchange that has details about the file such as file name, file size, etc. It's just that the file will not be downloaded.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
ignoreFileNotFoundOrPermissionError (consumer)	Whether to ignore when (trying to list files in directories or when downloading a file), which does not exist or due to permission error. By default when a directory or file does not exists or insufficient permission, then an exception is thrown. Setting this option to true allows to ignore that instead.	false	boolean
inProgressRepository (consumer)	A pluggable in-progress repository org.apache.camel.spi.IdempotentRepository. The in-progress repository is used to account the current in progress files being consumed. By default a memory based repository is used.		String>
localWorkDirectory (consumer)	When consuming, a local work directory can be used to store the remote file content directly in local files, to avoid loading the content into memory. This is beneficial, if you consume a very big remote file and thus can conserve memory.		String

Name	Description	Default	Type
onCompletionException Handler (consumer)	To use a custom org.apache.camel.spi.ExceptionHandler to handle any thrown exceptions that happens during the file on completion process where the consumer does either a commit or rollback. The default implementation will log any exception at WARN level and ignore.		ExceptionHandler
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
processStrategy (consumer)	A pluggable org.apache.camel.component.file.GenericFileProcessStrategy allowing you to implement your own readLock option or similar. Can also be used when special conditions must be met before a file can be consumed, such as a special ready file exists. If this option is set then the readLock option does not apply.		GenericFileProcess s Strategy<T>
startingDirectoryMustExist (consumer)	Whether the starting directory must exist. Mind that the autoCreate option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable autoCreate and enable this to ensure the starting directory must exist. Will thrown an exception if the directory doesn't exist.	false	boolean
useList (consumer)	Whether to allow using LIST command when downloading a file. Default is true. In some use cases you may want to download a specific file and are not allowed to use the LIST command, and therefore you can set this option to false. Notice when using this option, then the specific file to download does not include meta-data information such as file size, timestamp, permissions etc, because those information is only possible to retrieve when LIST command is in use.	true	boolean

Name	Description	Default	Type
fileExist (producer)	What to do if a file already exists with the same name. Override, which is the default, replaces the existing file. Append - adds content to the existing file. Fail - throws a GenericFileOperationException, indicating that there is already an existing file. Ignore - silently ignores the problem and does not override the existing file, but assumes everything is okay. Move - option requires to use the moveExisting option to be configured as well. The option eagerDeleteTargetFile can be used to control what to do if an moving the file, and there exists already an existing file, otherwise causing the move operation to fail. The Move option will move any existing files, before writing the target file. TryRename is only applicable if tempFileName option is in use. This allows to try renaming the file from the temporary name to the actual name, without doing any exists check. This check may be faster on some file systems and especially FTP servers.	Override	GenericFileExist
flatten (producer)	Flatten is used to flatten the file name path to strip any leading paths, so it's just the file name. This allows you to consume recursively into sub-directories, but when you eg write the files to another directory they will be written in a single directory. Setting this to true on the producer enforces that any file name in CamelFileName header will be stripped for any leading paths.	false	boolean
moveExisting (producer)	Expression (such as File Language) used to compute file name to use when fileExist=Move is configured. To move files into a backup subdirectory just enter backup. This option only supports the following File Language tokens: file:name, file:name.ext, file:name.noext, file:onlyname, file:onlyname.noext, file:ext, and file:parent. Notice the file:parent is not supported by the FTP component, as the FTP component can only move any existing files to a relative directory based on current dir as base.		String
tempFileName (producer)	The same as tempPrefix option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language.		String

Name	Description	Default	Type
tempPrefix (producer)	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also avoid consumers (not using exclusive read locks) reading in progress files. Is often used by FTP when uploading big files.		String
allowNullBody (producer)	Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a GenericFileWriteException of 'Cannot write null body to file.' will be thrown. If the fileExist option is set to 'Override', then the file will be truncated, and if set to append the file will remain unchanged.	false	boolean
chmod (producer)	Allows you to set chmod on the stored file. For example chmod=640.		String
disconnectOnBatchComplete (producer)	Whether or not to disconnect from remote FTP server right after a Batch upload is complete. disconnectOnBatchComplete will only disconnect the current connection to the FTP server.	false	boolean
eagerDeleteTargetFile (producer)	Whether or not to eagerly delete any existing target file. This option only applies when you use fileExists=Override and the tempFileName option as well. You can use this to disable (set it to false) deleting the target file before the temp file is written. For example you may write big files and want the target file to exists during the temp file is being written. This ensure the target file is only deleted until the very last moment, just before the temp file is being renamed to the target filename. This option is also used to control whether to delete any existing files when fileExist=Move is enabled, and an existing file exists. If this option copyAndDeleteOnRenameFails false, then an exception will be thrown if an existing file existed, if its true, then the existing file is deleted before the move operation.	true	boolean

Name	Description	Default	Type
keepLastModified (producer)	Will keep the last modified timestamp from the source file (if any). Will use the Exchange.FILE_LAST_MODIFIED header to located the timestamp. This header can contain either a java.util.Date or long with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. Note: This option only applies to the file producer. You cannot use this option with any of the ftp producers.	false	boolean
sendNoop (producer)	Whether to send a noop command as a pre-write check before uploading files to the FTP server. This is enabled by default as a validation of the connection is still valid, which allows to silently re-connect to be able to upload the file. However if this causes problems, you can turn this option off.	true	boolean
autoCreate (advanced)	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means the directory the files should be written to.	true	boolean
bufferSize (advanced)	Write buffer sized in bytes.	131072	int
bulkRequests (advanced)	Specifies how many requests may be outstanding at any one time. Increasing this value may slightly improve file transfer speed but will increase memory usage.		Integer
compression (advanced)	To use compression. Specify a level from 1 to 10. Important: You must manually add the needed JSCH zlib JAR to the classpath for compression support.		int
connectTimeout (advanced)	Sets the connect timeout for waiting for a connection to be established Used by both FTPClient and JSCH	10000	int
maximumReconnectAttempts (advanced)	Specifies the maximum reconnect attempts Camel performs when it tries to connect to the remote FTP server. Use 0 to disable this behavior.		int
proxy (advanced)	To use a custom configured com.jcraft.jsch.Proxy. This proxy is used to consume/send messages from the target SFTP host.		Proxy
reconnectDelay (advanced)	Delay in millis Camel will wait before performing a reconnect attempt.		long

Name	Description	Default	Type
serverAliveCountMax (advanced)	Allows you to set the serverAliveCountMax of the sftp session	1	int
serverAliveInterval (advanced)	Allows you to set the serverAliveInterval of the sftp session		int
soTimeout (advanced)	Sets the so timeout FTP and FTPS Only for Camel 2.4. SFTP for Camel 2.14.3/2.15.3/2.16 onwards. Is the SocketOptions.SO_TIMEOUT value in millis. Recommended option is to set this to 300000 so as not have a hanged connection. On SFTP this option is set as timeout on the JSCH Session instance.	300000	int
stepwise (advanced)	Sets whether we should stepwise change directories while traversing file structures when downloading files, or as well when uploading a file to a directory. You can disable this if you for example are in a situation where you cannot change directory on the FTP server due security reasons.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
throwExceptionOnConnectFailed (advanced)	Should an exception be thrown if connection failed (exhausted) By default exception is not thrown and a WARN is logged. You can use this to enable exception being thrown and handle the thrown exception from the org.apache.camel.spi.PollingConsumerPollStrategy rollback method.	false	boolean
timeout (advanced)	Sets the data timeout for waiting for reply Used only by FTPClient	30000	int
antExclude (filter)	Ant style filter exclusion. If both antInclude and antExclude are used, antExclude takes precedence over antInclude. Multiple exclusions may be specified in comma-delimited format.		String
antFilterCaseSensitive (filter)	Sets case sensitive flag on ant filter	true	boolean
antInclude (filter)	Ant style filter inclusion. Multiple inclusions may be specified in comma-delimited format.		String

Name	Description	Default	Type
eagerMaxMessagesPerPoll (filter)	Allows for controlling whether the limit from maxMessagesPerPoll is eager or not. If eager then the limit is during the scanning of files. Where as false would scan all files, and then perform sorting. Setting this option to false allows for sorting all files first, and then limit the poll. Mind that this requires a higher memory usage as all file details are in memory to perform the sorting.	true	boolean
exclude (filter)	Is used to exclude files, if filename matches the regex pattern (matching is case in-senstive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
filter (filter)	Pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. Will skip files if filter returns false in its accept() method.		GenericFileFilter< T>
filterDirectory (filter)	Filters the directory based on Simple language. For example to filter on current date, you can use a simple date pattern such as $date:now:yyyMMdd		String
filterFile (filter)	Filters the file based on Simple language. For example to filter on file size, you can use $file:size 5000		String
idempotent (filter)	Option to use the Idempotent Consumer EIP pattern to let Camel skip already processed files. Will by default use a memory based LRUCache that holds 1000 entries. If noop=true then idempotent will be enabled as well to avoid consuming the same files over and over again.	false	Boolean
idempotentKey (filter)	To use a custom idempotent key. By default the absolute path of the file is used. You can use the File Language, for example to use the file name and file size, you can do: idempotentKey=$file:name-$file:size		String
idempotentRepository (filter)	A pluggable repository org.apache.camel.spi.IdempotentRepository which by default use MemoryMessageIdRepository if none is specified and idempotent is true.		String>

Name	Description	Default	Type
include (filter)	Is used to include files, if filename matches the regex pattern (matching is case in-sensitive). Notice if you use symbols such as plus sign and others you would need to configure this using the RAW() syntax if configuring this as an endpoint uri. See more details at configuring endpoint uris		String
maxDepth (filter)	The maximum depth to traverse when recursively processing a directory.	2147483647	int
maxMessagesPerPoll (filter)	To define a maximum messages to gather per poll. By default no maximum is set. Can be used to set a limit of e.g. 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disabled it. Notice: If this option is in use then the File and FTP components will limit before any sorting. For example if you have 100000 files and use maxMessagesPerPoll=500, then only the first 500 files will be picked up, and then sorted. You can use the eagerMaxMessagesPerPoll option and set this to false to allow to scan all files first and then sort afterwards.		int
minDepth (filter)	The minimum depth to start processing when recursively processing a directory. Using minDepth=1 means the base directory. Using minDepth=2 means the first sub directory.		int
move (filter)	Expression (such as Simple Language) used to dynamically set the filename when moving it after processing. To move files into a .done subdirectory just enter .done.		String
exclusiveReadLockStrategy (lock)	Pluggable read-lock as a org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy implementation.		GenericFileExclusiveReadLockStrategy<T>

Name	Description	Default	Type
readLock (lock)	Used by consumer, to only poll the files if it has exclusive read-lock on the file (i.e. the file is not in-progress or being written). Camel will wait until the file lock is granted. This option provides the build in strategies: none - No read lock is in use markerFile - Camel creates a marker file (fileName.camelLock) and then holds a lock on it. This option is not available for the FTP component changed - Changed is using file length/modification timestamp to detect whether the file is currently being copied or not. Will at least use 1 sec to determine this, so this option cannot consume files as fast as the others, but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. The option readLockCheckInterval can be used to set the check frequency. fileLock - is for using java.nio.channels.FileLock. This option is not avail for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks. rename - rename is for using a try to rename the file as a test if we can get exclusive read-lock. idempotent - (only for file component) idempotent is for using a idempotentRepository as the read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-changed - (only for file component) idempotent-changed is for using a idempotentRepository and changed as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. idempotent-rename - (only for file component) idempotent-rename is for using a idempotentRepository and rename as the combined read-lock. This allows to use read locks that supports clustering if the idempotent repository implementation supports that. Notice: The various read locks is not all suited to work in clustered mode, where concurrent consumers on different nodes is competing for the same files on a shared file system. The markerFile using a close to atomic operation to create the empty marker file, but its not guaranteed to work in a cluster. The fileLock may work better but then the file system need to support distributed file locks, and so on. Using the idempotent read lock can support clustering if the idempotent repository supports clustering, such as Hazelcast Component or Infinispan.	none	String

Name	Description	Default	Type
readLockCheckInterval (lock)	Interval in millis for the read-lock, if supported by the read lock. This interval is used for sleeping between attempts to acquire the read lock. For example when using the changed read lock, you can set a higher interval period to cater for slow writes. The default of 1 sec. may be too fast if the producer is very slow writing the file. Notice: For FTP the default readLockCheckInterval is 5000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	1000	long
readLockDeleteOrphanLock Files (lock)	Whether or not read lock with marker files should upon startup delete any orphan read lock files, which may have been left on the file system, if Camel was not properly shutdown (such as a JVM crash). If turning this option to false then any orphaned lock file will cause Camel to not attempt to pickup that file, this could also be due another node is concurrently reading files from the same shared directory.	true	boolean
readLockLoggingLevel (lock)	Logging level used when a read lock could not be acquired. By default a WARN is logged. You can change this level, for example to OFF to not have any logging. This option is only applicable for readLock of types: changed, fileLock, idempotent, idempotent-changed, idempotent-rename, rename.	DEBUG	LoggingLevel
readLockMarkerFile (lock)	Whether to use marker file with the changed, rename, or exclusive read lock types. By default a marker file is used as well to guard against other processes picking up the same files. This behavior can be turned off by setting this option to false. For example if you do not want to write marker files to the file systems by the Camel application.	true	boolean
readLockMinAge (lock)	This option applied only for readLock=change. This option allows to specify a minimum age the file must be before attempting to acquire the read lock. For example use readLockMinAge=300s to require the file is at last 5 minutes old. This can speedup the changed read lock as it will only attempt to acquire files which are at least that given age.	0	long

Name	Description	Default	Type
readLockMinLength (lock)	This option applied only for readLock=changed. This option allows you to configure a minimum file length. By default Camel expects the file to contain data, and thus the default value is 1. You can set this option to zero, to allow consuming zero-length files.	1	long
readLockRemoveOnCommit (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file is succeeded and a commit happens. By default the file is not removed which ensures that any race-condition do not occur so another active node may attempt to grab the file. Instead the idempotent repository may support eviction strategies that you can configure to evict the file name entry after X minutes - this ensures no problems with race conditions.	false	boolean
readLockRemoveOnRollback (lock)	This option applied only for readLock=idempotent. This option allows to specify whether to remove the file name entry from the idempotent repository when processing the file failed and a rollback happens. If this option is false, then the file name entry is confirmed (as if the file did a commit).	true	boolean
readLockTimeout (lock)	Optional timeout in millis for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout triggered, then Camel will skip the file. At next poll Camel, will try the file again, and this time maybe the read-lock could be granted. Use a value of 0 or lower to indicate forever. Currently fileLock, changed and rename support the timeout. Notice: For FTP the default readLockTimeout value is 20000 instead of 10000. The readLockTimeout value must be higher than readLockCheckInterval, but a rule of thumb is to have a timeout that is at least 2 or more times higher than the readLockCheckInterval. This is needed to ensure that amble time is allowed for the read lock process to try to grab the lock before the timeout was hit.	10000	long
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

Name	Description	Default	Type
shuffle (sort)	To shuffle the list of files (sort in random order)	false	boolean
sortBy (sort)	Built-in sort by using the File Language. Supports nested sorts, so you can have a sort by file name and as a 2nd group sort by modified date.		String
sorter (sort)	Pluggable sorter as a java.util.Comparator class.		GenericFile<T>>
ciphers (security)	Set a comma separated list of ciphers that will be used in order of preference. Possible cipher names are defined by JCraft JSCH. Some examples include: aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,blowfish-cbc,aes192-cbc,aes256-cbc. If not specified the default list from JSCH will be used.		String
keyPair (security)	Sets a key pair of the public and private key so to that the SFTP endpoint can do public/private key verification.		KeyPair
knownHosts (security)	Sets the known_hosts from the byte array, so that the SFTP endpoint can do host key verification.		byte[]
knownHostsFile (security)	Sets the known_hosts file, so that the SFTP endpoint can do host key verification.		String
knownHostsUri (security)	Sets the known_hosts file (loaded from classpath by default), so that the SFTP endpoint can do host key verification.		String
password (security)	Password to use for login		String
preferredAuthentications (security)	Set the preferred authentications which SFTP endpoint will used. Some example include:password,publickey. If not specified the default list from JSCH will be used.		String
privateKey (security)	Set the private key as byte so that the SFTP endpoint can do private key verification.		byte[]
privateKeyFile (security)	Set the private key file so that the SFTP endpoint can do private key verification.		String
privateKeyPassphrase (security)	Set the private key file passphrase so that the SFTP endpoint can do private key verification.		String

Name	Description	Default	Type
privateKeyUri (security)	Set the private key file (loaded from classpath by default) so that the SFTP endpoint can do private key verification.		String
strictHostKeyChecking (security)	Sets whether to use strict host key checking.	no	String
username (security)	Username to use for login		String
useUserKnownHostsFile (security)	If knownHostFile has not been explicit configured then use the host file from System.getProperty(user.home)/.ssh/known_hosts	true	boolean

CHAPTER 282. SHIRO SECURITY COMPONENT

Available as of Camel 2.5

The **shiro-security** component in Camel is a security focused component, based on the Apache Shiro security project.

Apache Shiro is a powerful and flexible open-source security framework that cleanly handles authentication, authorization, enterprise session management and cryptography. The objective of the Apache Shiro project is to provide the most robust and comprehensive application security framework available while also being very easy to understand and extremely simple to use.

This camel shiro-security component allows authentication and authorization support to be applied to different segments of a camel route.

Shiro security is applied on a route using a Camel Policy. A Policy in Camel utilizes a strategy pattern for applying interceptors on Camel Processors. It offering the ability to apply cross-cutting concerns (for example. security, transactions etc) on sections/segments of a camel route.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-shiro</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

282.1. SHIRO SECURITY BASICS

To employ Shiro security on a camel route, a ShiroSecurityPolicy object must be instantiated with security configuration details (including users, passwords, roles etc). This object must then be applied to a camel route. This ShiroSecurityPolicy Object may also be registered in the Camel registry (JNDI or ApplicationContextRegistry) and then utilized on other routes in the Camel Context.

Configuration details are provided to the ShiroSecurityPolicy using an Ini file (properties file) or an Ini object. The Ini file is a standard Shiro configuration file containing user/role details as shown below

```
[users]
# user 'ringo' with password 'starr' and the 'sec-level1' role
ringo = starr, sec-level1
george = harrison, sec-level2
john = lennon, sec-level3
paul = mccartney, sec-level3

[roles]
# 'sec-level3' role has all permissions, indicated by the
# wildcard '*'
sec-level3 = *

# The 'sec-level2' role can do anything with access of permission
# readonly (*) to help
sec-level2 = zone1:*
```

```
# The 'sec-level1' role can do anything with access of permission
# readonly
sec-level1 = zone1:readonly:*
```

282.2. INSTANTIATING A SHIROSECURITYPOLICY OBJECT

A ShiroSecurityPolicy object is instantiated as follows

```
    private final String iniResourcePath = "classpath:shiro.ini";
    private final byte[] passPhrase = {
        (byte) 0x08, (byte) 0x09, (byte) 0x0A, (byte) 0x0B,
        (byte) 0x0C, (byte) 0x0D, (byte) 0x0E, (byte) 0x0F,
        (byte) 0x10, (byte) 0x11, (byte) 0x12, (byte) 0x13,
        (byte) 0x14, (byte) 0x15, (byte) 0x16, (byte) 0x17};
    List<permission> permissionsList = new ArrayList<permission>();
    Permission permission = new WildcardPermission("zone1:readwrite:*");
    permissionsList.add(permission);

    final ShiroSecurityPolicy securityPolicy =
        new ShiroSecurityPolicy(iniResourcePath, passPhrase, true, permissionsList);
```

282.3. SHIROSECURITYPOLICY OPTIONS

Name	Default Value	Type	Description
iniResourcePath or ini	**none**	Resource String or Ini Object	A mandatory Resource String for the iniResourcePath or an instance of an Ini object must be passed to the security policy. Resources can be acquired from the file system, classpath, or URLs when prefixed with "file:, classpath:, or url:" respectively. For e.g "classpath:shiro.ini"
passPhrase	**An AES 128 based key**	byte[]	A passPhrase to decrypt ShiroSecurityToken(s) sent along with Message Exchanges
alwaysReauthenticate	**true**	boolean	Setting to ensure re-authentication on every individual request. If set to false, the user is authenticated and locked such than only requests from the same user going forward are authenticated.

Name	Default Value	Type	Description
permissionsList	**none**	List<Permission>	A List of permissions required in order for an authenticated user to be authorized to perform further action i.e continue further on the route. If no Permissions list or Roles List (see below) is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required. Note that the default is that authorization is granted if any of the Permission Objects in the list are applicable.
rolesList	**none**	List<String>	**Camel 2.13:** A List of roles required in order for an authenticated user to be authorized to perform further action i.e continue further on the route. If no roles list or permissions list (see above) is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required. Note that the default is that authorization is granted if any of the roles in the list are applicable.
cipherService	**AES**	org.apache.shiro.crypto.CipherService	Shiro ships with AES & Blowfish based CipherServices. You may use one these or pass in your own Cipher implementation
base64	**false**	**boolean**	**Camel 2.12:** To use base64 encoding for the security token header, which allows transferring the header over JMS etc. This option must also be set on **ShiroSecurityTokenInjector** as well.
allPermissionsRequired	**false**	**boolean**	**Camel 2.13:** The default is that authorization is granted if any of the Permission Objects in the permissionsList parameter are applicable. Set this to true to require all of the Permissions to be met.
allRolesRequired	**false**	**boolean**	**Camel 2.13:** The default is that authorization is granted if any of the roles in the rolesList parameter are applicable. Set this to true to require all of the roles to be met.

282.4. APPLYING SHIRO AUTHENTICATION ON A CAMEL ROUTE

The ShiroSecurityPolicy, tests and permits incoming message exchanges containing a encrypted SecurityToken in the Message Header to proceed further following proper authentication. The SecurityToken object contains a Username/Password details that are used to determine where the user is a valid user.

```
protected RouteBuilder createRouteBuilder() throws Exception {
```

```
    final ShiroSecurityPolicy securityPolicy =
        new ShiroSecurityPolicy("classpath:shiro.ini", passPhrase);

    return new RouteBuilder() {
        public void configure() {
            onException(UnknownAccountException.class).
                to("mock:authenticationException");
            onException(IncorrectCredentialsException.class).
                to("mock:authenticationException");
            onException(LockedAccountException.class).
                to("mock:authenticationException");
            onException(AuthenticationException.class).
                to("mock:authenticationException");

            from("direct:secureEndpoint").
                to("log:incoming payload").
                policy(securityPolicy).
                to("mock:success");
        }
    };
}
```

282.5. APPLYING SHIRO AUTHORIZATION ON A CAMEL ROUTE

Authorization can be applied on a camel route by associating a Permissions List with the ShiroSecurityPolicy. The Permissions List specifies the permissions necessary for the user to proceed with the execution of the route segment. If the user does not have the proper permission set, the request is not authorized to continue any further.

```
protected RouteBuilder createRouteBuilder() throws Exception {
    final ShiroSecurityPolicy securityPolicy =
        new ShiroSecurityPolicy("./src/test/resources/securityconfig.ini", passPhrase);

    return new RouteBuilder() {
        public void configure() {
            onException(UnknownAccountException.class).
                to("mock:authenticationException");
            onException(IncorrectCredentialsException.class).
                to("mock:authenticationException");
            onException(LockedAccountException.class).
                to("mock:authenticationException");
            onException(AuthenticationException.class).
                to("mock:authenticationException");

            from("direct:secureEndpoint").
                to("log:incoming payload").
                policy(securityPolicy).
                to("mock:success");
        }
    };
}
```

282.6. CREATING A SHIROSECURITYTOKEN AND INJECTING IT INTO A MESSAGE EXCHANGE

A ShiroSecurityToken object may be created and injected into a Message Exchange using a Shiro Processor called ShiroSecurityTokenInjector. An example of injecting a ShiroSecurityToken using a ShiroSecurityTokenInjector in the client is shown below

```
ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");
ShiroSecurityTokenInjector shiroSecurityTokenInjector =
    new ShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

from("direct:client").
    process(shiroSecurityTokenInjector).
    to("direct:secureEndpoint");
```

282.7. SENDING MESSAGES TO ROUTES SECURED BY A SHIROSECURITYPOLICY

Messages and Message Exchanges sent along the camel route where the security policy is applied need to be accompanied by a SecurityToken in the Exchange Header. The SecurityToken is an encrypted object that holds a Username and Password. The SecurityToken is encrypted using AES 128 bit security by default and can be changed to any cipher of your choice.

Given below is an example of how a request may be sent using a ProducerTemplate in Camel along with a SecurityToken

```
@Test
public void testSuccessfulShiroAuthenticationWithNoAuthorization() throws Exception {
    //Incorrect password
    ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "stirr");

    // TestShiroSecurityTokenInjector extends ShiroSecurityTokenInjector
    TestShiroSecurityTokenInjector shiroSecurityTokenInjector =
        new TestShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

    successEndpoint.expectedMessageCount(1);
    failureEndpoint.expectedMessageCount(0);

    template.send("direct:secureEndpoint", shiroSecurityTokenInjector);

    successEndpoint.assertIsSatisfied();
    failureEndpoint.assertIsSatisfied();
}
```

282.8. SENDING MESSAGES TO ROUTES SECURED BY A SHIROSECURITYPOLICY (MUCH EASIER FROM CAMEL 2.12 ONWARDS)

From **Camel 2.12** onwards its even easier as you can provide the subject in two different ways.

282.8.1. Using ShiroSecurityToken

You can send a message to a Camel route with a header of key **ShiroSecurityConstants.SHIRO_SECURITY_TOKEN** of the type **org.apache.camel.component.shiro.security.ShiroSecurityToken** that contains the username and password. For example:

```
        ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");

        template.sendBodyAndHeader("direct:secureEndpoint", "Beatle Mania",
    ShiroSecurityConstants.SHIRO_SECURITY_TOKEN, shiroSecurityToken);
```

You can also provide the username and password in two different headers as shown below:

```
        Map<String, Object> headers = new HashMap<String, Object>();
        headers.put(ShiroSecurityConstants.SHIRO_SECURITY_USERNAME, "ringo");
        headers.put(ShiroSecurityConstants.SHIRO_SECURITY_PASSWORD, "starr");
        template.sendBodyAndHeaders("direct:secureEndpoint", "Beatle Mania", headers);
```

When you use the username and password headers, then the ShiroSecurityPolicy in the Camel route will automatic transform those into a single header with key ShiroSecurityConstants.SHIRO_SECURITY_TOKEN with the token. Then token is either a **ShiroSecurityToken** instance, or a base64 representation as a String (the latter is when you have set base64=true).

CHAPTER 283. SIMPLE LANGUAGE

Available as of Camel version 1.1

The Simple Expression Language was a really simple language when it was created, but has since grown more powerful. It is primarily intended for being a really small and simple language for evaluating Expressions and Predicates without requiring any new dependencies or knowledge of XPath; so it is ideal for testing in camel-core. The idea was to cover 95% of the common use cases when you need a little bit of expression based script in your Camel routes.

However for much more complex use cases you are generally recommended to choose a more expressive and powerful language such as:

- SpEL
- Mvel
- Groovy
- JavaScript
- OGNL
- one of the supported Scripting Languages

The simple language uses **${body}** placeholders for complex expressions where the expression contains constant literals. The $\{ } placeholders can be omitted if the expression is only the token itself.

TIP

Alternative syntax From Camel 2.5 onwards you can also use the alternative syntax which uses **$simple{ }** as placeholders. This can be used in situations to avoid clashes when using for example Spring property placeholder together with Camel.

283.1. SIMPLE LANGUAGE CHANGES IN CAMEL 2.9 ONWARDS

The Simple language have been improved from Camel 2.9 onwards to use a better syntax parser, which can do index precise error messages, so you know exactly what is wrong and where the problem is. For example if you have made a typo in one of the operators, then previously the parser would not be able to detect this, and cause the evaluation to be true. There are a few changes in the syntax which are no longer backwards compatible. When using Simple language as a Predicate then the literal text **must** be enclosed in either single or double quotes. For example: **"${body} == 'Camel'"**. Notice how we have single quotes around the literal. The old style of using **"body"** and **"header.foo"** to refer to the message body and header is @deprecated, and it is encouraged to always use $\{ } tokens for the built-in functions. The range operator now requires the range to be in single quote as well as shown: **"${header.zip} between '30000..39999'"**.

To get the body of the in message: **"body"**, or **"in.body"** or **"${body}"**.

A complex expression must use $\{ } placeholders, such as: **"Hello ${in.header.name} how are you?"**.

You can have multiple functions in the same expression: **"Hello ${in.header.name} this is ${in.header.me} speaking"**.
However you can **not** nest functions in Camel 2.8.x or older (i.e. having another $\{ } placeholder in an existing, is not allowed).
From **Camel 2.9** onwards you can nest functions.

283.2. SIMPLE LANGUAGE OPTIONS

The Simple language supports 2 options which are listed below.

Name	Default	Java Type	Description
resultType		**String**	Sets the class name of the result type (type from output)
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

283.3. VARIABLES

Variable	Type	Description
camelId	String	**Camel 2.10:** the CamelContext name
camelContext.**OGNL**	Object	**Camel 2.11:** the CamelContext invoked using a Camel OGNL expression.
exchange	Exchange	**Camel 2.16:** the Exchange
exchange.**OGNL**	Object	**Camel 2.16:** the Exchange invoked using a Camel OGNL expression.
exchangeId	String	**Camel 2.3:** the exchange id
id	String	the input message id
body	Object	the input body
in.body	Object	the input body
body.**OGNL**	Object	**Camel 2.3:** the input body invoked using a Camel OGNL expression.
in.body.**OGNL**	Object	**Camel 2.3:** the input body invoked using a Camel OGNL expression.
bodyAs(*type*)	Type	**Camel 2.3:** Converts the body to the given type determined by its classname. The converted body can be null.

Variable	Type	Description
bodyAs(*type*).**OGNL**	Object	**Camel 2.18:** Converts the body to the given type determined by its classname and then invoke methods using a Camel OGNL expression. The converted body can be null.
mandatoryBodyAs(*type*)	Type	**Camel 2.5:** Converts the body to the given type determined by its classname, and expects the body to be not null.
mandatoryBodyAs(*type*).**OGNL**	Object	**Camel 2.18:** Converts the body to the given type determined by its classname and then invoke methods using a Camel OGNL expression.
out.body	Object	the output body
header.foo	Object	refer to the input foo header
header[foo]	Object	**Camel 2.9.2:** refer to the input foo header
headers.foo	Object	refer to the input foo header
headers[foo]	Object	**Camel 2.9.2:** refer to the input foo header
in.header.foo	Object	refer to the input foo header
in.header[foo]	Object	**Camel 2.9.2:** refer to the input foo header
in.headers.foo	Object	refer to the input foo header
in.headers[foo]	Object	**Camel 2.9.2:** refer to the input foo header
header.foo[bar]	Object	**Camel 2.3:** regard input foo header as a map and perform lookup on the map with bar as key

Variable	Type	Description
in.header.foo[bar]	Object	**Camel 2.3:** regard input foo header as a map and perform lookup on the map with bar as key
in.headers.foo[bar]	Object	**Camel 2.3:** regard input foo header as a map and perform lookup on the map with bar as key
header.foo.**OGNL**	Object	**Camel 2.3:** refer to the input foo header and invoke its value using a Camel OGNL expression.
in.header.foo.**OGNL**	Object	**Camel 2.3:** refer to the input foo header and invoke its value using a Camel OGNL expression.
in.headers.foo.**OGNL**	Object	**Camel 2.3:** refer to the input foo header and invoke its value using a Camel OGNL expression.
out.header.foo	Object	refer to the out header foo
out.header[foo]	Object	**Camel 2.9.2:** refer to the out header foo
out.headers.foo	Object	refer to the out header foo
out.headers[foo]	Object	**Camel 2.9.2:** refer to the out header foo
headerAs(*key*, *type*)	Type	**Camel 2.5:** Converts the header to the given type determined by its classname
headers	Map	**Camel 2.9:** refer to the input headers
in.headers	Map	**Camel 2.9:** refer to the input headers

Variable	Type	Description
property.foo	Object	**Deprecated:** refer to the foo property on the exchange
exchangeProperty.foo	Object	**Camel 2.15:** refer to the foo property on the exchange
property[foo]	Object	**Deprecated:** refer to the foo property on the exchange
exchangeProperty[foo]	Object	**Camel 2.15:** refer to the foo property on the exchange
property.foo.**OGNL**	Object	**Deprecated:** refer to the foo property on the exchange and invoke its value using a Camel OGNL expression.
exchangeProperty.foo.**OGNL**	Object	**Camel 2.15:** refer to the foo property on the exchange and invoke its value using a Camel OGNL expression.
sys.foo	String	refer to the system property
sysenv.foo	String	**Camel 2.3:** refer to the system environment
exception	Object	**Camel 2.4:** Refer to the exception object on the exchange, is**null** if no exception set on exchange. Will fallback and grab caught exceptions (**Exchange.EXCEPTION_CAUGHT**) if the Exchange has any.
exception.**OGNL**	Object	**Camel 2.4:** Refer to the exchange exception invoked using a Camel OGNL expression object
exception.message	String	Refer to the exception.message on the exchange, is **null** if no exception set on exchange. Will fallback and grab caught exceptions (**Exchange.EXCEPTION_CAUGHT**) if the Exchange has any.
exception.stacktrace	String	**Camel 2.6.** Refer to the exception.stacktrace on the exchange, is**null** if no exception set on exchange. Will fallback and grab caught exceptions (**Exchange.EXCEPTION_CAUGHT**) if the Exchange has any.

Variable	Type	Description
date:_command_	Date	Evaluates to a Date object. Supported commands are: **now** for current timestamp, **in.header.xxx** or **header.xxx** to use the Date object in the IN header with the key xxx. **out.header.xxx** to use the Date object in the OUT header with the key xxx. **property.xxx** to use the Date object in the exchange property with the key xxx.**file** for the last modified timestamp of the file (available with a File consumer). Command accepts offsets such as: **now-24h** or **in.header.xxx+1h** or even **now+1h30m-100**.
date:_command:pattern_	String	Date formatting using **java.text.SimpleDataFormat** patterns.
date-with-timezone:_command:timezone:pattern_	String	Date formatting using **java.text.SimpleDataFormat** timezones and patterns.
bean:_bean expression_	Object	Invoking a bean expression using the Bean language. Specifying a method name you must use dot as separator. We also support the ?method=methodname syntax that is used by the Bean component.
properties:_locations:key_	String	**Deprecated (use properties-location instead) Camel 2.3:** Lookup a property with the given key. The **locations** option is optional. See more at Using PropertyPlaceholder.
properties-location:_http://locationskey[locations:key]_	String	**Camel 2.14.1:** Lookup a property with the given key. The**locations** option is optional. See more at Using PropertyPlaceholder.
properties:key:default	String	**Camel 2.14.1**: Lookup a property with the given key. If the key does not exists or has no value, then an optional default value can be specified.
routeId	String	**Camel 2.11:** Returns the id of the current route the Exchange is being routed.

Variable	Type	Description
threadName	String	**Camel 2.3:** Returns the name of the current thread. Can be used for logging purpose.
ref:xxx	Object	**Camel 2.6:** To lookup a bean from the Registry with the given id.
type:name.field	Object	**Camel 2.11:** To refer to a type or field by its FQN name. To refer to a field you can append .FIELD_NAME. For example you can refer to the constant field from Exchange as: **org.apache.camel.Exchange.FILE_NAME**
null	null	**Camel 2.12.3:** represents a **null**
random_(value)_	Integer	*Camel 2.16.0:*returns a random Integer between 0 (included) and *value* (excluded)
random_(min, max)_	Integer	*Camel 2.16.0:*returns a random Integer between *min* (included) and *max* (excluded)
collate(group)	List	**Camel 2.17:** The collate function iterates the message body and groups the data into sub lists of specified size. This can be used with the Splitter EIP to split a message body and group/batch the splitted sub message into a group of N sub lists. This method works similar to the collate method in Groovy.
skip(number)	Iterator	**Camel 2.19:** The skip function iterates the message body and skips the first number of items. This can be used with the Splitter EIP to split a message body and skip the first N number of items.
messageHistory	String	**Camel 2.17:** The message history of the current exchange how it has been routed. This is similar to the route stack-trace message history the error handler logs in case of an unhandled exception.
messageHistory(false)	String	**Camel 2.17:** As messageHistory but without the exchange details (only includes the route strack-trace). This can be used if you do not want to log sensitive data from the message itself.

283.4. OGNL EXPRESSION SUPPORT

Available as of Camel 2.3

INFO:Camel's OGNL support is for invoking methods only. You cannot access fields. From **Camel 2.11.1** onwards we added special support for accessing the length field of Java arrays.

The Simple and Bean language now supports a Camel OGNL notation for invoking beans in a chain like fashion. Suppose the Message IN body contains a POJO which has a **getAddress()** method.

Then you can use Camel OGNL notation to access the address object:

```
simple("${body.address}")
simple("${body.address.street}")
simple("${body.address.zip}")
```

Camel understands the shorthand names for getters, but you can invoke any method or use the real name such as:

```
simple("${body.address}")
simple("${body.getAddress.getStreet}")
simple("${body.address.getZip}")
simple("${body.doSomething}")
```

You can also use the null safe operator (**?.**) to avoid NPE if for example the body does NOT have an address

```
simple("${body?.address?.street}")
```

It is also possible to index in **Map** or **List** types, so you can do:

```
simple("${body[foo].name}")
```

To assume the body is **Map** based and lookup the value with **foo** as key, and invoke the **getName** method on that value.

If the key has space, then you **must** enclose the key with quotes, for example 'foo bar':

```
simple("${body['foo bar'].name}")
```

You can access the **Map** or **List** objects directly using their key name (with or without dots) :

```
simple("${body[foo]}")
simple("${body[this.is.foo]}")
```

Suppose there was no value with the key **foo** then you can use the null safe operator to avoid the NPE as shown:

```
simple("${body[foo]?.name}")
```

You can also access **List** types, for example to get lines from the address you can do:

```
simple("${body.address.lines[0]}")
simple("${body.address.lines[1]}")
simple("${body.address.lines[2]}")
```

There is a special **last** keyword which can be used to get the last value from a list.

```
simple("${body.address.lines[last]}")
```

And to get the 2nd last you can subtract a number, so we can use **last-1** to indicate this:

```
simple("${body.address.lines[last-1]}")
```

And the 3rd last is of course:

```
simple("${body.address.lines[last-2]}")
```

And you can call the size method on the list with

```
simple("${body.address.lines.size}")
```

From **Camel 2.11.1** onwards we added support for the length field for Java arrays as well, eg:

```
String[] lines = new String[]{"foo", "bar", "cat"};
exchange.getIn().setBody(lines);

simple("There are ${body.length} lines")
```

And yes you can combine this with the operator support as shown below:

```
simple("${body.address.zip} > 1000")
```

283.5. OPERATOR SUPPORT

The parser is limited to only support a single operator.

To enable it the left value must be enclosed in $\{ }. The syntax is:

```
${leftValue} OP rightValue
```

Where the **rightValue** can be a String literal enclosed in **' '**, **null**, a constant value or another expression enclosed in $\{ }.

IMPORTANT

There **must** be spaces around the operator.

Camel will automatically type convert the rightValue type to the leftValue type, so it is able to eg. convert a string into a numeric so you can use > comparison for numeric values.

The following operators are supported:

Operator	Description
==	equals
=~	**Camel 2.16:** equals ignore case (will ignore case when comparing String values)
>	greater than
>=	greater than or equals

Operator	Description
<	less than
<=	less than or equals
!=	not equals
contains	For testing if contains in a string based value
not contains	For testing if not contains in a string based value
~~	For testing if contains by ignoring case sensitivity in a string based value
regex	For matching against a given regular expression pattern defined as a String value
not regex	For not matching against a given regular expression pattern defined as a String value
in	For matching if in a set of values, each element must be separated by comma. If you want to include an empty value, then it must be defined using double comma, eg ',,bronze,silver,gold', which is a set of four values with an empty value and then the three medals.
not in	For matching if not in a set of values, each element must be separated by comma. If you want to include an empty value, then it must be defined using double comma, eg ',,bronze,silver,gold', which is a set of four values with an empty value and then the three medals.
is	For matching if the left hand side type is an instanceof the value.
not is	For matching if the left hand side type is not an instanceof the value.
range	For matching if the left hand side is within a range of values defined as numbers: **from..to**. From **Camel 2.9** onwards the range values must be enclosed in single quotes.
not range	For matching if the left hand side is not within a range of values defined as numbers: **from..to**. From **Camel 2.9** onwards the range values must be enclosed in single quotes.

Operator	Description
starts with	**Camel 2.17.1, 2.18**: For testing if the left hand side string starts with the right hand string.
ends with	**Camel 2.17.1, 2.18**: For testing if the left hand side string ends with the right hand string.

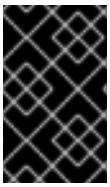
And the following unary operators can be used:

Operator	Description
++	**Camel 2.9:** To increment a number by one. The left hand side must be a function, otherwise parsed as literal.
–	**Camel 2.9:** To decrement a number by one. The left hand side must be a function, otherwise parsed as literal.
\	**Camel 2.9.3 to 2.10.x** To escape a value, eg \$, to indicate a $ sign. Special: Use \n for new line, \t for tab, and \r for carriage return. **Notice:** Escaping is **not** supported using the File Language. **Notice:** From Camel 2.11 onwards the escape character is no longer support, but replaced with the following three special escaping.
\n	**Camel 2.11:** To use newline character.
\t	**Camel 2.11:** To use tab character.
\r	**Camel 2.11:** To use carriage return character.
\}	**Camel 2.18:** To use the } character as text

And the following logical operators can be used to group expressions:

Operator	Description
and	**deprecated** use && instead. The logical and operator is used to group two expressions.
or	**deprecated** use || instead. The logical or operator is used to group two expressions.

Operator	Description
&&	**Camel 2.9:** The logical and operator is used to group two expressions.
||	**Camel 2.9:** The logical or operator is used to group two expressions.



IMPORTANT

Using and,or operators In **Camel 2.4 or older** the **and** or **or** can only be used **once** in a simple language expression. From **Camel 2.5** onwards you can use these operators multiple times.

The syntax for AND is:

```
${leftValue} OP rightValue and ${leftValue} OP rightValue
```

And the syntax for OR is:

```
${leftValue} OP rightValue or ${leftValue} OP rightValue
```

Some examples:

```
// exact equals match
simple("${in.header.foo} == 'foo'")

// ignore case when comparing, so if the header has value FOO this will match
simple("${in.header.foo} =~ 'foo'")

// here Camel will type convert '100' into the type of in.header.bar and if it is an Integer '100' will also be converter to an Integer
simple("${in.header.bar} == '100'")

simple("${in.header.bar} == 100")

// 100 will be converter to the type of in.header.bar so we can do > comparison
simple("${in.header.bar} > 100")
```

283.5.1. Comparing with different types

When you compare with different types such as String and int, then you have to take a bit care. Camel will use the type from the left hand side as 1st priority. And fallback to the right hand side type if both values couldn't be compared based on that type.
This means you can flip the values to enforce a specific type. Suppose the bar value above is a String. Then you can flip the equation:

```
simple("100 < ${in.header.bar}")
```

which then ensures the int type is used as 1st priority.

This may change in the future if the Camel team improves the binary comparison operations to prefer numeric types over String based. It's most often the String type which causes problem when comparing with numbers.

```
// testing for null
simple("${in.header.baz} == null")

// testing for not null
simple("${in.header.baz} != null")
```

And a bit more advanced example where the right value is another expression

```
simple("${in.header.date} == ${date:now:yyyyMMdd}")

simple("${in.header.type} == ${bean:orderService?method=getOrderType}")
```

And an example with contains, testing if the title contains the word Camel

```
simple("${in.header.title} contains 'Camel'")
```

And an example with regex, testing if the number header is a 4 digit value:

```
simple("${in.header.number} regex '\\d{4}'")
```

And finally an example if the header equals any of the values in the list. Each element must be separated by comma, and no space around.
This also works for numbers etc, as Camel will convert each element into the type of the left hand side.

```
simple("${in.header.type} in 'gold,silver'")
```

And for all the last 3 we also support the negate test using not:

```
simple("${in.header.type} not in 'gold,silver'")
```

And you can test if the type is a certain instance, eg for instance a String

```
simple("${in.header.type} is 'java.lang.String'")
```

We have added a shorthand for all **java.lang** types so you can write it as:

```
simple("${in.header.type} is 'String'")
```

Ranges are also supported. The range interval requires numbers and both from and end are inclusive.
For instance to test whether a value is between 100 and 199:

```
simple("${in.header.number} range 100..199")
```

Notice we use **..** in the range without spaces. It is based on the same syntax as Groovy.

From **Camel 2.9** onwards the range value must be in single quotes

```
simple("${in.header.number} range '100..199'")
```

283.5.2. Using Spring XML

As the Spring XML does not have all the power as the Java DSL with all its various builder methods, you have to resort to use some other languages for testing with simple operators. Now you can do this with the simple language. In the sample below we want to test if the header is a widget order:

```
<from uri="seda:orders">
  <filter>
    <simple>${in.header.type} == 'widget'</simple>
    <to uri="bean:orderService?method=handleWidget"/>
  </filter>
</from>
```

283.6. USING AND / OR

If you have two expressions you can combine them with the **and** or **or** operator.

TIP

Camel 2.9 onwards Use && or || from Camel 2.9 onwards.

For instance:

```
simple("${in.header.title} contains 'Camel' and ${in.header.type'} == 'gold'")
```

And of course the **or** is also supported. The sample would be:

```
simple("${in.header.title} contains 'Camel' or ${in.header.type'} == 'gold'")
```

Notice: Currently **and** or **or** can only be used **once** in a simple language expression. This might change in the future.
So you **cannot** do:

```
simple("${in.header.title} contains 'Camel' and ${in.header.type'} == 'gold' and ${in.header.number}
range 100..200")
```

283.7. SAMPLES

In the Spring XML sample below we filter based on a header value:

```
<from uri="seda:orders">
  <filter>
    <simple>${in.header.foo}</simple>
    <to uri="mock:fooOrders"/>
  </filter>
</from>
```

The Simple language can be used for the predicate test above in the Message Filter pattern, where we test if the in message has a **foo** header (a header with the key **foo** exists). If the expression evaluates to **true** then the message is routed to the **mock:fooOrders** endpoint, otherwise the message is dropped.

The same example in Java DSL:

```
from("seda:orders")
    .filter().simple("${in.header.foo}")
        .to("seda:fooOrders");
```

You can also use the simple language for simple text concatenations such as:

```
from("direct:hello")
    .transform().simple("Hello ${in.header.user} how are you?")
    .to("mock:reply");
```

Notice that we must use $\{ } placeholders in the expression now to allow Camel to parse it correctly.

And this sample uses the date command to output current date.

```
from("direct:hello")
    .transform().simple("The today is ${date:now:yyyyMMdd} and it is a great day.")
    .to("mock:reply");
```

And in the sample below we invoke the bean language to invoke a method on a bean to be included in the returned string:

```
from("direct:order")
    .transform().simple("OrderId: ${bean:orderIdGenerator}")
    .to("mock:reply");
```

Where **orderIdGenerator** is the id of the bean registered in the Registry. If using Spring then it is the Spring bean id.

If we want to declare which method to invoke on the order id generator bean we must prepend **.method name** such as below where we invoke the **generateId** method.

```
from("direct:order")
    .transform().simple("OrderId: ${bean:orderIdGenerator.generateId}")
    .to("mock:reply");
```

We can use the **?method=methodname** option that we are familiar with the Bean component itself:

```
from("direct:order")
    .transform().simple("OrderId: ${bean:orderIdGenerator?method=generateId}")
    .to("mock:reply");
```

And from Camel 2.3 onwards you can also convert the body to a given type, for example to ensure that it is a String you can do:

```
<transform>
  <simple>Hello ${bodyAs(String)} how are you?</simple>
</transform>
```

There are a few types which have a shorthand notation, so we can use **String** instead of **java.lang.String**. These are: **byte[], String, Integer, Long**. All other types must use their FQN name, e.g. **org.w3c.dom.Document**.

It is also possible to lookup a value from a header **Map** in **Camel 2.3** onwards:

```
<transform>
  <simple>The gold value is ${header.type[gold]}</simple>
</transform>
```

In the code above we lookup the header with name **type** and regard it as a **java.util.Map** and we then lookup with the key **gold** and return the value. If the header is not convertible to Map an exception is thrown. If the header with name **type** does not exist **null** is returned.

From Camel 2.9 onwards you can nest functions, such as shown below:

```
<setHeader headerName="myHeader">
  <simple>${properties:${header.someKey}}</simple>
</setHeader>
```

283.8. REFERRING TO CONSTANTS OR ENUMS

Available as of Camel 2.11

Suppose you have an enum for customers

And in a Content Based Router we can use the Simple language to refer to this enum, to check the message which enum it matches.

283.9. USING NEW LINES OR TABS IN XML DSLS

Available as of Camel 2.9.3

From Camel 2.9.3 onwards it is easier to specify new lines or tabs in XML DSLs as you can escape the value now

```
<transform>
  <simple>The following text\nis on a new line</simple>
</transform>
```

283.10. LEADING AND TRAILING WHITESPACE HANDLING

Available as of Camel 2.10.0

From Camel 2.10.0 onwards, the trim attribute of the expression can be used to control whether the leading and trailing whitespace characters are removed or preserved. The default value is true, which removes the whitespace characters.

```
<setBody>
  <simple trim="false">You get some trailing whitespace characters.     </simple>
</setBody>
```

283.11. SETTING RESULT TYPE

Available as of Camel 2.8

You can now provide a result type to the Simple expression, which means the result of the evaluation will be converted to the desired type. This is most useable to define types such as booleans, integers, etc.

For example to set a header as a boolean type you can do:

```
.setHeader("cool", simple("true", Boolean.class))
```

And in XML DSL

```
<setHeader headerName="cool">
  <!-- use resultType to indicate that the type should be a java.lang.Boolean -->
  <simple resultType="java.lang.Boolean">true</simple>
</setHeader>
```

283.12. CHANGING FUNCTION START AND END TOKENS

Available as of Camel 2.9.1

You can configure the function start and end tokens - $\{ } using the setters **changeFunctionStartToken** and **changeFunctionEndToken** on **SimpleLanguage**, using Java code. From Spring XML you can define a <bean> tag with the new changed tokens in the properties as shown below:

```
<!-- configure Simple to use custom prefix/suffix tokens -->
<bean id="simple" class="org.apache.camel.language.simple.SimpleLanguage">
  <property name="functionStartToken" value="["/>
  <property name="functionEndToken" value="]"/>
</bean>
```

In the example above we use [ ] as the changed tokens.

Notice by changing the start/end token you change those in all the Camel applications which share the same **camel-core** on their classpath.
For example in an OSGi server this may affect many applications, where as a Web Application as a WAR file it only affects the Web Application.

283.13. LOADING SCRIPT FROM EXTERNAL RESOURCE

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as **"classpath:"**, **"file:"**, or **"http:"**.
This is done using the following syntax: **"resource:scheme:location"**, eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").simple("resource:classpath:mysimple.txt")
```

283.14. SETTING SPRING BEANS TO EXCHANGE PROPERTIES

Available as of Camel 2.6

You can set a spring bean into an exchange property as shown below:

```
<bean id="myBeanId" class="my.package.MyCustomClass" />
...
<route>
  ...
  <setProperty propertyName="monitoring.message">
    <simple>ref:myBeanId</simple>
  </setProperty>
  ...
</route>
```

283.15. DEPENDENCIES

The Simple language is part of **camel-core**.

CHAPTER 284. SIP COMPONENT

Available as of Camel version 2.5

The **sip** component in Camel is a communication component, based on the Jain SIP implementation (available under the JCP license).

Session Initiation Protocol (SIP) is an IETF-defined signaling protocol, widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol (IP).The SIP protocol is an Application Layer protocol designed to be independent of the underlying transport layer; it can run on Transmission Control Protocol (TCP), User Datagram Protocol (UDP) or Stream Control Transmission Protocol (SCTP).

The Jain SIP implementation supports TCP and UDP only.

The Camel SIP component **only** supports the SIP Publish and Subscribe capability as described in the RFC3903 - Session Initiation Protocol (SIP) Extension for Event

This camel component supports both producer and consumer endpoints.

Camel SIP Producers (Event Publishers) and SIP Consumers (Event Subscribers) communicate event & state information to each other using an intermediary entity called a SIP Presence Agent (a stateful brokering entity).

For SIP based communication, a SIP Stack with a listener **must** be instantiated on both the SIP Producer and Consumer (using separate ports if using localhost). This is necessary in order to support the handshakes & acknowledgements exchanged between the SIP Stacks during communication.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sip</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

284.1. URI FORMAT

The URI scheme for a sip endpoint is as follows:

```
sip://johndoe@localhost:99999[?options]
sips://johndoe@localhost:99999/[?options]
```

This component supports producer and consumer endpoints for both TCP and UDP.

You can append query options to the URI in the following format, **?option=value&option=value&…**

284.2. OPTIONS

The SIP Component offers an extensive set of configuration options & capability to create custom stateful headers needed to propagate state via the SIP protocol.

The SIP component has no options.

The SIP endpoint is configured using URI syntax:

```
sip:uri
```

with the following path and query parameters:

284.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
uri	**Required** URI of the SIP server to connect to (the username and password can be included such as: john:secretmyserver:9999)		URI

284.2.2. Query Parameters (44 parameters):

Name	Description	Default	Type
cacheConnections (common)	Should connections be cached by the SipStack to reduce cost of connection creation. This is useful if the connection is used for long running conversations.	false	boolean
contentSubType (common)	Setting for contentSubType can be set to any valid MimeSubType.	plain	String
contentType (common)	Setting for contentType can be set to any valid MimeType.	text	String
eventHeaderName (common)	Setting for a String based event type.		String
eventId (common)	Setting for a String based event Id. Mandatory setting unless a registry based FromHeader is specified		String
fromHost (common)	Hostname of the message originator. Mandatory setting unless a registry based FromHeader is specified		String
fromPort (common)	Port of the message originator. Mandatory setting unless a registry based FromHeader is specified		int
fromUser (common)	Username of the message originator. Mandatory setting unless a registry based custom FromHeader is specified.		String

Name	Description	Default	Type
msgExpiration (common)	The amount of time a message received at an endpoint is considered valid	3600	int
receiveTimeoutMillis (common)	Setting for specifying amount of time to wait for a Response and/or Acknowledgement can be received from another SIP stack	10000	long
stackName (common)	Name of the SIP Stack instance associated with an SIP Endpoint.	NAME_NOT_SET	String
toHost (common)	Hostname of the message receiver. Mandatory setting unless a registry based ToHeader is specified		String
toPort (common)	Portname of the message receiver. Mandatory setting unless a registry based ToHeader is specified		int
toUser (common)	Username of the message receiver. Mandatory setting unless a registry based custom ToHeader is specified.		String
transport (common)	Setting for choice of transport protocol. Valid choices are tcp or udp.	tcp	String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
consumer (consumer)	This setting is used to determine whether the kind of header (FromHeader,ToHeader etc) that needs to be created for this endpoint	false	boolean
presenceAgent (consumer)	This setting is used to distinguish between a Presence Agent & a consumer. This is due to the fact that the SIP Camel component ships with a basic Presence Agent (for testing purposes only). Consumers have to set this flag to true.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
addressFactory (advanced)	To use a custom AddressFactory		AddressFactory
callIdHeader (advanced)	A custom Header object containing call details. Must implement the type javax.sip.header.CallIdHeader		CallIdHeader
contactHeader (advanced)	An optional custom Header object containing verbose contact details (email, phone number etc). Must implement the type javax.sip.header.ContactHeader		ContactHeader
contentTypeHeader (advanced)	A custom Header object containing message content details. Must implement the type javax.sip.header.ContentTypeHeader		ContentTypeHeader
eventHeader (advanced)	A custom Header object containing event details. Must implement the type javax.sip.header.EventHeader		EventHeader
expiresHeader (advanced)	A custom Header object containing message expiration details. Must implement the type javax.sip.header.ExpiresHeader		ExpiresHeader
extensionHeader (advanced)	A custom Header object containing user/application specific details. Must implement the type javax.sip.header.ExtensionHeader		ExtensionHeader
fromHeader (advanced)	A custom Header object containing message originator settings. Must implement the type javax.sip.header.FromHeader		FromHeader
headerFactory (advanced)	To use a custom HeaderFactory		HeaderFactory
listeningPoint (advanced)	To use a custom ListeningPoint implementation		ListeningPoint

Name	Description	Default	Type
maxForwardsHeader (advanced)	A custom Header object containing details on maximum proxy forwards. This header places a limit on the viaHeaders possible. Must implement the type javax.sip.header.MaxForwardsHeader		MaxForwardsHeader
maxMessageSize (advanced)	Setting for maximum allowed Message size in bytes.	1048576	int
messageFactory (advanced)	To use a custom MessageFactory		MessageFactory
sipFactory (advanced)	To use a custom SipFactory to create the SipStack to be used		SipFactory
sipStack (advanced)	To use a custom SipStack		SipStack
sipUri (advanced)	To use a custom SipURI. If none configured, then the SipUri fallback to use the options toUser toHost:toPort		SipURI
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
toHeader (advanced)	A custom Header object containing message receiver settings. Must implement the type javax.sip.header.ToHeader		ToHeader
viaHeaders (advanced)	List of custom Header objects of the type javax.sip.header.ViaHeader. Each ViaHeader containing a proxy address for request forwarding. (Note this header is automatically updated by each proxy when the request arrives at its listener)		List
implementationDebugLogFile (logging)	Name of client debug log file to use for logging		String
implementationServerLogFile (logging)	Name of server log file to use for logging		String
implementationTraceLevel (logging)	Logging level for tracing	0	String

Name	Description	Default	Type
maxForwards (proxy)	Number of maximum proxy forwards		int
useRouterForAllUris (proxy)	This setting is used when requests are sent to the Presence Agent via a proxy.	false	boolean

284.3. SENDING MESSAGES TO/FROM A SIP ENDPOINT

284.3.1. Creating a Camel SIP Publisher

In the example below, a SIP Publisher is created to send SIP Event publications to a user "agent@localhost:5152". This is the address of the SIP Presence Agent which acts as a broker between the SIP Publisher and Subscriber

- using a SIP Stack named client
- using a registry based eventHeader called evtHdrName
- using a registry based eventId called evtId
- from a SIP Stack with Listener set up as user2@localhost:3534
- The Event being published is EVENT_A
- A Mandatory Header called REQUEST_METHOD is set to Request.Publish thereby setting up the endpoint as a Event publisher"

```
producerTemplate.sendBodyAndHeader(
    "sip://agent@localhost:5152?
stackName=client&eventHeaderName=evtHdrName&eventId=evtid&fromUser=user2&fromHost=localhost&fromPort=3534",
    "EVENT_A",
    "REQUEST_METHOD",
    Request.PUBLISH);
```

284.3.2. Creating a Camel SIP Subscriber

In the example below, a SIP Subscriber is created to receive SIP Event publications sent to a user "johndoe@localhost:5154"

- using a SIP Stack named Subscriber
- registering with a Presence Agent user called agent@localhost:5152
- using a registry based eventHeader called evtHdrName. The evtHdrName contains the Event which is se to "Event_A"
- using a registry based eventId called evtId

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            // Create PresenceAgent
            from("sip://agent@localhost:5152?
stackName=PresenceAgent&presenceAgent=true&eventHeaderName=evtHdrName&eventId=evtid")
                .to("mock:neverland");

            // Create Sip Consumer(Event Subscriber)
            from("sip://johndoe@localhost:5154?
stackName=Subscriber&toUser=agent&toHost=localhost&toPort=5152&eventHeaderName=evtHdrNam
e&eventId=evtid")
                .to("log:ReceivedEvent?level=DEBUG")
                .to("mock:notification");

        }
    };
}
```

The Camel SIP component also ships with a Presence Agent that is meant to be used for Testing and Demo purposes only. An example of instantiating a Presence Agent is given above.

Note that the Presence Agent is set up as a user agent@localhost:5152 and is capable of communicating with both Publisher as well as Subscriber. It has a separate SIP stackName distinct from Publisher as well as Subscriber. While it is set up as a Camel Consumer, it does not actually send any messages along the route to the endpoint "mock:neverland".

CHAPTER 285. SIMPLE JMS BATCH COMPONENT

Available as of Camel version 2.16

SJMS Batch is a specialized component for highly performant, transactional batch consumption from a JMS queue. It can be thought of as a hybrid of a consumer-only component and an aggregator.

A common use case in Camel is to consume messages from a queue and aggregate them before sending the aggregated state to another endpoint. In order to ensure that data is not lost if the system performing the processing fails, it is typically consumed within a transaction from the queue, and once aggregated stored in a persistent **AggregationRepository**, such as the one found in the JDBC Component.

The behavior of the aggregator pattern involves fetching data from the **AggregationRepository** before an incoming message is aggregated, and writing back the result afterwards. By nature, the reads and writes take progressively longer as the number of aggregated artifacts increases. A rough example of this using arbitrary time units that demonstrates the impact of this is as follows:

Item	Read Time	Write Time	Total Time
0	0	1	1
1	1	2	4
2	2	3	9
3	3	4	16
4	4	5	25
5	5	6	36
6	6	7	49
7	7	8	64
8	8	9	81
9	9	10	100

In contrast, consumption performance using the SJMS Batch component is linear. Each message is consumed and aggregated using an **AggregationStrategy** before the next one is fetched. As all of the consumption and aggregation is performed in a single JMS transaction no external storage is required to persist the intermediate state - this avoids the read and write costs described above. In practice, this yields multiple orders of magnitude higher throughput.

Once a completion condition is met, either by size or period since first message, the aggregated **Exchange** is passed into the route. During the processing of this **Exchange**, if an exception is thrown or the system shuts down, all of the original consumed messages end up back on the queue (or are placed on the dead-letter queue depending on the broker configuration).

Unlike using a regular aggregator, there is no facility for an aggregation condition, meaning that it is not possible to batch consume into multiple groups of messages. All consumed messages are aggregated together into a single batch.

Multiple JMS consumer support is available which allows you to consume in parallel using the one route, and at the same time use facilities like JMS message groups to group related messages.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

285.1. URI FORMAT

```
sjms:[queue:]destinationName[?options]
```

Where **destinationName** is a JMS queue. By default, the **destinationName** is interpreted as a queue name.

```
sjms:FOO.BAR
```

You can include the optional **queue:** prefix, if you prefer:

```
sjms:queue:FOO.BAR
```

Topic consumption is not supported, as there is no advantage to using batch consumption within that context. Topic messages are usually non-persistent, and loss is acceptable. If consumed within a transaction that fails, a topic message will likely not be redelivered by the broker. A plain SJMS consumer endpoint can be used in conjunction with a regular non-persistence backed aggregator in this scenario.

285.2. COMPONENT OPTIONS AND CONFIGURATIONS

The Simple JMS Batch component supports 5 options which are listed below.

Name	Description	Default	Type
connectionFactory (advanced)	A ConnectionFactory is required to enable the SjmsBatchComponent.		ConnectionFactory

Name	Description	Default	Type
asyncStartListener (advanced)	Whether to startup the consumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean
recoveryInterval (advanced)	Specifies the interval between recovery attempts, i.e. when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.	5000	int
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Simple JMS Batch endpoint is configured using URI syntax:

```
sjms-batch:destinationName
```

with the following path and query parameters:

285.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
destinationName	**Required** The destination name. Only queues are supported, names may be prefixed by 'queue:'.		String

285.2.2. Query Parameters (23 parameters):

Name	Description	Default	Type
aggregationStrategy (consumer)	**Required** The aggregation strategy to use, which merges all the batched messages into a single message		AggregationStrategy
allowNullBody (consumer)	Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
completionInterval (consumer)	The completion interval in millis, which causes batches to be completed in a scheduled fixed rate every interval. The batch may be empty if the timeout triggered and there was no messages in the batch. Notice you cannot use both completion timeout and completion interval at the same time, only one can be configured.	1000	int
completionPredicate (consumer)	The completion predicate, which causes batches to be completed when the predicate evaluates as true. The predicate can also be configured using the simple language using the string syntax. You may want to set the option eagerCheckCompletion to true to let the predicate match the incoming message, as otherwise it matches the aggregated message.		String
completionSize (consumer)	The number of messages consumed at which the batch will be completed	200	int
completionTimeout (consumer)	The timeout in millis from receipt of the first first message when the batch will be completed. The batch may be empty if the timeout triggered and there was no messages in the batch. Notice you cannot use both completion timeout and completion interval at the same time, only one can be configured.	500	int
consumerCount (consumer)	The number of JMS sessions to consume from	1	int

Name	Description	Default	Type
eagerCheckCompletion (consumer)	Use eager completion checking which means that the completionPredicate will use the incoming Exchange. As opposed to without eager completion checking the completionPredicate will use the aggregated Exchange.	false	boolean
includeAllJMSXProperties (consumer)	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.	false	boolean
mapJmsMessage (consumer)	Specifies whether Camel should auto map the received JMS message to a suited payload type, such as javax.jms.TextMessage to a String etc. See section about how mapping works below for more details.	true	boolean
pollDuration (consumer)	The duration in milliseconds of each poll for messages. completionTimeOut will be used if it is shorter and a batch has started.	1000	int
sendEmptyMessageWhenIdle (consumer)	If using completion timeout or interval, then the batch may be empty if the timeout triggered and there was no messages in the batch. If this option is true and the batch is empty then an empty message is added to the batch so an empty message is routed.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
asyncStartListener (advanced)	Whether to startup the consumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean

Name	Description	Default	Type
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		JmsKeyFormatStrategy
keepAliveDelay (advanced)	The delay in millis between attempts to re-establish a valid session. If this is a positive value the SjmsBatchConsumer will attempt to create a new session if it sees an IllegalStateException during message consumption. This delay value allows you to pause between attempts to prevent spamming the logs. If this is a negative value (default is -1) then the SjmsBatchConsumer will behave as it always has before - that is it will bail out and the route will shut down if it sees an IllegalStateException.	-1	int
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
recoveryInterval (advanced)	Specifies the interval between recovery attempts, i.e. when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.	5000	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
timeoutCheckerExecutorService (advanced)	If using the completionInterval option a background thread is created to trigger the completion interval. Set this option to provide a custom thread pool to be used rather than creating a new thread for every consumer.		ScheduledExecutorService

The **completionSize** endpoint attribute is used in conjunction with **completionTimeout**, where the first condition to be met will cause the aggregated **Exchange** to be emitted down the route.

CHAPTER 286. SIMPLE JMS COMPONENT

Available as of Camel version 2.11

The Simple JMS Component, or SJMS, is a JMS client for use with Camel that uses well known best practices when it comes to JMS client creation and configuration. SJMS contains a brand new JMS client API written explicitly for Camel eliminating third party messaging implementations keeping it light and resilient. The following features is included:

- Standard Queue and Topic Support (Durable & Non-Durable)
- InOnly & InOut MEP Support
- Asynchronous Producer and Consumer Processing
- Internal JMS Transaction Support

Additional key features include:

- Plugable Connection Resource Management
- Session, Consumer, & Producer Pooling & Caching Management
- Batch Consumers and Producers
- Transacted Batch Consumers & Producers
- Support for Customizable Transaction Commit Strategies (Local JMS Transactions only)

NOTE

Why the S in SJMS

S stands for Simple and Standard and Springless. Also camel-jms was already taken.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

286.1. URI FORMAT

```
sjms:[queue:|topic:]destinationName[?options]
```

Where **destinationName** is a JMS queue or topic name. By default, the **destinationName** is interpreted as a queue name. For example, to connect to the queue, **FOO.BAR** use:

```
sjms:FOO.BAR
```

You can include the optional **queue:** prefix, if you prefer:

```
sjms:queue:FOO.BAR
```

To connect to a topic, you *must* include the **topic:** prefix. For example, to connect to the topic, **Stocks.Prices**, use:

```
sjms:topic:Stocks.Prices
```

You append query options to the URI using the following format, **?option=value&option=value&...**

286.2. COMPONENT OPTIONS AND CONFIGURATIONS

The Simple JMS component supports 15 options which are listed below.

Name	Description	Default	Type
connectionFactory (advanced)	A ConnectionFactory is required to enable the SjmsComponent. It can be set directly or set set as part of a ConnectionResource.		ConnectionFactory
connectionResource (advanced)	A ConnectionResource is an interface that allows for customization and container control of the ConnectionFactory. See Plugable Connection Resource Management for further details.		ConnectionResource
connectionCount (common)	The maximum number of connections available to endpoints started under this component	1	Integer
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides one implementation out of the box: default. The default strategy will safely marshal dots and hyphens (. and -). Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		JmsKeyFormatStrategy
transactionCommit Strategy (transaction)	To configure which kind of commit strategy to use. Camel provides two implementations out of the box, default and batch.		TransactionCommit Strategy
destinationCreation Strategy (advanced)	To use a custom DestinationCreationStrategy.		DestinationCreation Strategy
timedTaskManager (advanced)	To use a custom TimedTaskManager		TimedTaskManager

Name	Description	Default	Type
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
connectionTestOnBorrow (advanced)	When using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource then should each javax.jms.Connection be tested (calling start) before returned from the pool.	true	boolean
connectionUsername (security)	The username to use when creating javax.jms.Connection when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.		String
connectionPassword (security)	The password to use when creating javax.jms.Connection when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.		String
connectionClientId (advanced)	The client ID to use when creating javax.jms.Connection when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.		String
connectionMaxWait (advanced)	The max wait time in millis to block and wait on free connection when the pool is exhausted when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.	5000	long
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Simple JMS endpoint is configured using URI syntax:

```
sjms:destinationType:destinationName
```

with the following path and query parameters:

286.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
destinationType	The kind of destination to use	queue	String
destinationName	**Required** DestinationName is a JMS queue or topic name. By default, the destinationName is interpreted as a queue name.		String

286.2.2. Query Parameters (34 parameters):

Name	Description	Default	Type
acknowledgementMode (common)	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE	AUTO_ACKNOWLEDGE	SessionAcknowledgement Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
consumerCount (consumer)	Sets the number of consumer listeners used for this endpoint.	1	int
durableSubscriptionId (consumer)	Sets the durable subscription Id required for durable topics.		String
synchronous (consumer)	Sets whether synchronous processing should be strictly used or Camel is allowed to use asynchronous processing (if supported).	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
messageSelector (consumer)	Sets the JMS Message selector syntax.		String
namedReplyTo (producer)	Sets the reply to destination name used for InOut producer endpoints.		String
persistent (producer)	Flag used to enable/disable message persistence.	true	boolean
producerCount (producer)	Sets the number of producers used for this endpoint.	1	int
ttl (producer)	Flag used to adjust the Time To Live value of produced messages.	-1	long
allowNullBody (producer)	Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.	true	boolean
prefillPool (producer)	Whether to prefill the producer connection pool on startup, or create connections lazy when needed.	true	boolean
responseTimeOut (producer)	Sets the amount of time we should wait before timing out a InOut response.	5000	long
asyncStartListener (advanced)	Whether to startup the consumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean
asyncStopListener (advanced)	Whether to stop the consumer message listener asynchronously, when stopping a route.	false	boolean
connectionCount (advanced)	The maximum number of connections available to this endpoint		Integer

Name	Description	Default	Type
connectionFactory (advanced)	Initializes the connectionFactory for the endpoint, which takes precedence over the component's connectionFactory, if any		ConnectionFactory
connectionResource (advanced)	Initializes the connectionResource for the endpoint, which takes precedence over the component's connectionResource, if any		ConnectionResource
destinationCreationStrategy (advanced)	To use a custom DestinationCreationStrategy.		DestinationCreation Strategy
exceptionListener (advanced)	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.		ExceptionListener
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
includeAllJMSXProperties (advanced)	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.	false	boolean
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		JmsKeyFormatStrategy
mapJmsMessage (advanced)	Specifies whether Camel should auto map the received JMS message to a suited payload type, such as javax.jms.TextMessage to a String etc. See section about how mapping works below for more details.	true	boolean
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
errorHandlerLoggingLevel (logging)	Allows to configure the default errorHandler logging level for logging uncaught exceptions.	WARN	LoggingLevel

Name	Description	Default	Type
errorHandlerLogStackTrace (logging)	Allows to control whether stacktraces should be logged or not, by the default errorHandler.	true	boolean
transacted (transaction)	Specifies whether to use transacted mode	false	boolean
transactionBatchCount (transaction)	If transacted sets the number of messages to process before committing a transaction.	-1	int
transactionBatchTimeout (transaction)	Sets timeout (in millis) for batch transactions, the value should be 1000 or higher.	5000	long
transactionCommitStrategy (transaction)	Sets the commit strategy.		TransactionCommit Strategy
sharedJMSSession (transaction)	Specifies whether to share JMS session with other SJMS endpoints. Turn this off if your route is accessing to multiple JMS providers. If you need transaction against multiple JMS providers, use jms component to leverage XA transaction.	true	boolean

Below is an example of how to configure the **SjmsComponent** with its required **ConnectionFactory** provider. It will create a single connection by default and store it using the components internal pooling APIs to ensure that it is able to service Session creation requests in a thread safe manner.

```
SjmsComponent component = new SjmsComponent();
component.setConnectionFactory(new ActiveMQConnectionFactory("tcp://localhost:61616"));
getContext().addComponent("sjms", component);
```

For a SJMS component that is required to support a durable subscription, you can override the default **ConnectionFactoryResource** instance and set the **clientId** property.

```
ConnectionFactoryResource connectionResource = new ConnectionFactoryResource();
connectionResource.setConnectionFactory(new
ActiveMQConnectionFactory("tcp://localhost:61616"));
connectionResource.setClientId("myclient-id");

SjmsComponent component = new SjmsComponent();
component.setConnectionResource(connectionResource);
component.setMaxConnections(1);
```

286.3. PRODUCER USAGE

286.3.1. InOnly Producer - (Default)

The *InOnly* producer is the default behavior of the SJMS Producer Endpoint.

```
from("direct:start")
    .to("sjms:queue:bar");
```

286.3.2. InOut Producer

To enable *InOut* behavior append the **exchangePattern** attribute to the URI. By default it will use a dedicated TemporaryQueue for each consumer.

```
from("direct:start")
    .to("sjms:queue:bar?exchangePattern=InOut");
```

You can specify a **namedReplyTo** though which can provide a better monitor point.

```
from("direct:start")
    .to("sjms:queue:bar?exchangePattern=InOut&namedReplyTo=my.reply.to.queue");
```

286.4. CONSUMER USAGE

286.4.1. InOnly Consumer - (Default)

The *InOnly* xonsumer is the default Exchange behavior of the SJMS Consumer Endpoint.

```
from("sjms:queue:bar")
    .to("mock:result");
```

286.4.2. InOut Consumer

To enable *InOut* behavior append the **exchangePattern** attribute to the URI.

```
from("sjms:queue:in.out.test?exchangePattern=InOut")
    .transform(constant("Bye Camel"));
```

286.5. ADVANCED USAGE NOTES

286.5.1. Plugable Connection Resource Management

SJMS provides JMS **Connection** resource management through built-in connection pooling. This eliminates the need to depend on third party API pooling logic. However there may be times that you are required to use an external Connection resource manager such as those provided by J2EE or OSGi containers. For this SJMS provides an interface that can be used to override the internal SJMS Connection pooling capabilities. This is accomplished through the **ConnectionResource** interface.

The **ConnectionResource** provides methods for borrowing and returning Connections as needed is the contract used to provide **Connection** pools to the SJMS component. A user should use when it is necessary to integrate SJMS with an external connection pooling manager.

It is recommended though that for standard **ConnectionFactory** providers you use the **ConnectionFactoryResource** implementation that is provided with SJMS as-is or extend as it is optimized for this component.

Below is an example of using the plugable ConnectionResource with the ActiveMQ **PooledConnectionFactory**:

```
public class AMQConnectionResource implements ConnectionResource {
    private PooledConnectionFactory pcf;

    public AMQConnectionResource(String connectString, int maxConnections) {
        super();
        pcf = new PooledConnectionFactory(connectString);
        pcf.setMaxConnections(maxConnections);
        pcf.start();
    }

    public void stop() {
        pcf.stop();
    }

    @Override
    public Connection borrowConnection() throws Exception {
        Connection answer = pcf.createConnection();
        answer.start();
        return answer;
    }

    @Override
    public Connection borrowConnection(long timeout) throws Exception {
        // SNIPPED...
    }

    @Override
    public void returnConnection(Connection connection) throws Exception {
        // Do nothing since there isn't a way to return a Connection
        // to the instance of PooledConnectionFactory
        log.info("Connection returned");
    }
}
```

Then pass in the **ConnectionResource** to the **SjmsComponent**:

```
CamelContext camelContext = new DefaultCamelContext();
AMQConnectionResource pool = new AMQConnectionResource("tcp://localhost:33333", 1);
SjmsComponent component = new SjmsComponent();
component.setConnectionResource(pool);
camelContext.addComponent("sjms", component);
```

To see the full example of its usage please refer to the **ConnectionResourceIT**.

286.5.2. Batch Message Support

The SjmsProducer supports publishing a collection of messages by creating an Exchange that encapsulates a **List**. This SjmsProducer will take then iterate through the contents of the List and publish each message individually.

If when producing a batch of messages there is the need to set headers that are unique to each message you can use the SJMS **BatchMessage** class. When the SjmsProducer encounters a **BatchMessage** list it will iterate each **BatchMessage** and publish the included payload and headers.

Below is an example of using the BatchMessage class. First we create a list of **BatchMessage**:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Then publish the list:

```
template.sendBody("sjms:queue:batch.queue", messages);
```

286.5.3. Customizable Transaction Commit Strategies (Local JMS Transactions only)

SJMS provides a developer the means to create a custom and plugable transaction strategy through the use of the **TransactionCommitStrategy** interface. This allows a user to define a unique set of circumstances that the **SessionTransactionSynchronization** will use to determine when to commit the Session. An example of its use is the **BatchTransactionCommitStrategy** which is detailed further in the next section.

286.5.4. Transacted Batch Consumers & Producers

The SJMS component has been designed to support the batching of local JMS transactions on both the Producer and Consumer endpoints. How they are handled on each is very different though.

The SJMS consumer endpoint is a straightforward implementation that will process X messages before committing them with the associated Session. To enable batched transaction on the consumer first enable transactions by setting the **transacted** parameter to true and then adding the **transactionBatchCount** and setting it to any value that is greater than 0. For example the following configuration will commit the Session every 10 messages:

```
sjms:queue:transacted.batch.consumer?transacted=true&transactionBatchCount=10
```

If an exception occurs during the processing of a batch on the consumer endpoint, the Session rollback is invoked causing the messages to be redelivered to the next available consumer. The counter is also reset to 0 for the **BatchTransactionCommitStrategy** for the associated Session as well. It is the responsibility of the user to ensure they put hooks in their processors of batch messages to watch for messages with the JMSRedelivered header set to true. This is the indicator that messages were rolled back at some point and that a verification of a successful processing should occur.

A transacted batch consumer also carries with it an instance of an internal timer that waits a default amount of time (5000ms) between messages before committing the open transactions on the Session. The default value of 5000ms (minimum of 1000ms) should be adequate for most use-cases but if further tuning is necessary simply set the **transactionBatchTimeout** parameter.

```
sjms:queue:transacted.batch.consumer?
transacted=true&transactionBatchCount=10&transactionBatchTimeout=2000
```

The minimal value that will be accepted is 1000ms as the amount of context switching may cause unnecessary performance impacts without gaining benefit.

The producer endpoint is handled much differently though. With the producer after each message is delivered to its destination the Exchange is closed and there is no longer a reference to that message. To make a available all the messages available for redelivery you simply enable transactions on a Producer Endpoint that is publishing BatchMessages. The transaction will commit at the conclusion of the exchange which includes all messages in the batch list. Nothing additional need be configured. For example:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Now publish the List with transactions enabled:

```
template.sendBody("sjms:queue:batch.queue?transacted=true", messages);
```

286.6. ADDITIONAL NOTES

286.6.1. Message Header Format

The SJMS Component uses the same header format strategy that is used in the Camel JMS Component. This plugable strategy ensures that messages sent over the wire conform to the JMS Message spec.

For the **exchange.in.header** the following rules apply for the header keys:

- Keys starting with **JMS** or **JMSX** are reserved.
- **exchange.in.headers** keys must be literals and all be valid Java identifiers (do not use dots in the key name).
- Camel replaces dots & hyphens and the reverse when when consuming JMS messages:
  - is replaced by *DOT* and the reverse replacement when Camel consumes the message.
  - is replaced by *HYPHEN* and the reverse replacement when Camel consumes the message. See also the option **jmsKeyFormatStrategy**, which allows use of your own custom strategy for formatting keys.

For the **exchange.in.header**, the following rules apply for the header values:

286.6.2. Message Content

To deliver content over the wire we must ensure that the body of the message that is being delivered adheres to the JMS Message Specification. Therefore, all that are produced must either be primitives or their counter objects (such as **Integer**, **Long**, **Character**). The types, **String**, **CharSequence**, **Date**,

BigDecimal and **BigInteger** are all converted to their **toString()** representation. All other types are dropped.

286.6.3. Clustering

When using *InOut* with SJMS in a clustered environment you must either use TemporaryQueue destinations or use a unique named reply to destination per InOut producer endpoint. Message correlation is handled by the endpoint, not with message selectors at the broker. The InOut Producer Endpoint uses Java Concurrency Exchangers cached by the Message **JMSCorrelationID**. This provides a nice performance increase while reducing the overhead on the broker since all the messages are consumed from the destination in the order they are produced by the interested consumer.

Currently the only correlation strategy is to use the **JMSCorrelationId**. The *InOut* Consumer uses this strategy as well ensuring that all responses messages to the included **JMSReplyTo** destination also have the **JMSCorrelationId** copied from the request as well.

286.7. TRANSACTION SUPPORT

SJMS currently only supports the use of internal JMS Transactions. There is no support for the Camel Transaction Processor or the Java Transaction API (JTA).

286.7.1. Does Springless Mean I Can't Use Spring?

Not at all. Below is an example of the SJMS component using the Spring DSL:

```
<route
    id="inout.named.reply.to.producer.route">
    <from
        uri="direct:invoke.named.reply.to.queue" />
    <to
        uri="sjms:queue:named.reply.to.queue?
namedReplyTo=my.response.queue&amp;amp;exchangePattern=InOut" />
</route>
```

Springless refers to moving away from the dependency on the Spring JMS API. A new JMS client API is being developed from the ground up to power SJMS.

CHAPTER 287. SIMPLE JMS2 COMPONENT

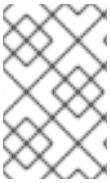
Available as of Camel version 2.19

The Simple JMS 2.0 Component, or SJMS2, is a JMS client for use with Camel that uses well known best practices when it comes to JMS client creation and configuration. SJMS2 contains a brand new JMS 2.0 client API written explicitly for Camel eliminating third party messaging implementations keeping it light and resilient. The following features is included:

- Standard Queue and Topic Support (Durable & Non-Durable)
- InOnly & InOut MEP Support
- Asynchronous Producer and Consumer Processing
- Internal JMS Transaction Support

Additional key features include:

- Plugable Connection Resource Management
- Session, Consumer, & Producer Pooling & Caching Management
- Batch Consumers and Producers
- Transacted Batch Consumers & Producers
- Support for Customizable Transaction Commit Strategies (Local JMS Transactions only)



NOTE

Why the S in SJMS

S stands for Simple and Standard and Springless. Also camel-jms was already taken.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms2</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

287.1. URI FORMAT

```
sjms2:[queue:|topic:]destinationName[?options]
```

Where **destinationName** is a JMS queue or topic name. By default, the **destinationName** is interpreted as a queue name. For example, to connect to the queue, **FOO.BAR** use:

```
sjms2:FOO.BAR
```

You can include the optional **queue:** prefix, if you prefer:

```
sjms2:queue:FOO.BAR
```

To connect to a topic, you *must* include the **topic:** prefix. For example, to connect to the topic, **Stocks.Prices**, use:

```
sjms2:topic:Stocks.Prices
```

You append query options to the URI using the following format, **?option=value&option=value&...**

287.2. COMPONENT OPTIONS AND CONFIGURATIONS

The Simple JMS2 component supports 15 options which are listed below.

Name	Description	Default	Type
connectionFactory (advanced)	A ConnectionFactory is required to enable the SjmsComponent. It can be set directly or set set as part of a ConnectionResource.		ConnectionFactory
connectionResource (advanced)	A ConnectionResource is an interface that allows for customization and container control of the ConnectionFactory. See Plugable Connection Resource Management for further details.		ConnectionResource
connectionCount (common)	The maximum number of connections available to endpoints started under this component	1	Integer
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides one implementation out of the box: default. The default strategy will safely marshal dots and hyphens (. and -). Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		JmsKeyFormatStrategy
transactionCommit Strategy (transaction)	To configure which kind of commit strategy to use. Camel provides two implementations out of the box, default and batch.		TransactionCommit Strategy
destinationCreation Strategy (advanced)	To use a custom DestinationCreationStrategy.		DestinationCreation Strategy
timedTaskManager (advanced)	To use a custom TimedTaskManager		TimedTaskManager

Name	Description	Default	Type
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
connectionTestOnBorrow (advanced)	When using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource then should each javax.jms.Connection be tested (calling start) before returned from the pool.	true	boolean
connectionUsername (security)	The username to use when creating javax.jms.Connection when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.		String
connectionPassword (security)	The password to use when creating javax.jms.Connection when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.		String
connectionClientId (advanced)	The client ID to use when creating javax.jms.Connection when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.		String
connectionMaxWait (advanced)	The max wait time in millis to block and wait on free connection when the pool is exhausted when using the default org.apache.camel.component.sjms.jms.ConnectionFactoryResource.	5000	long
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Simple JMS2 endpoint is configured using URI syntax:

```
sjms2:destinationType:destinationName
```

with the following path and query parameters:

287.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
destinationType	The kind of destination to use	queue	String
destinationName	**Required** DestinationName is a JMS queue or topic name. By default, the destinationName is interpreted as a queue name.		String

287.2.2. Query Parameters (37 parameters):

Name	Description	Default	Type
acknowledgementMode (common)	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE	AUTO_ACKNOWLEDGE	SessionAcknowledgement Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
consumerCount (consumer)	Sets the number of consumer listeners used for this endpoint.	1	int
durable (consumer)	Sets topic consumer to durable.	false	boolean
durableSubscriptionId (consumer)	Sets the durable subscription Id required for durable topics.		String
shared (consumer)	Sets the consumer to shared.	false	boolean
subscriptionId (consumer)	Sets the subscription Id, required for durable or shared topics.		String
synchronous (consumer)	Sets whether synchronous processing should be strictly used or Camel is allowed to use asynchronous processing (if supported).	true	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
messageSelector (consumer)	Sets the JMS Message selector syntax.		String
namedReplyTo (producer)	Sets the reply to destination name used for InOut producer endpoints.		String
persistent (producer)	Flag used to enable/disable message persistence.	true	boolean
producerCount (producer)	Sets the number of producers used for this endpoint.	1	int
ttl (producer)	Flag used to adjust the Time To Live value of produced messages.	-1	long
allowNullBody (producer)	Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.	true	boolean
prefillPool (producer)	Whether to prefill the producer connection pool on startup, or create connections lazy when needed.	true	boolean
responseTimeOut (producer)	Sets the amount of time we should wait before timing out a InOut response.	5000	long
asyncStartListener (advanced)	Whether to startup the consumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.	false	boolean

Name	Description	Default	Type
asyncStopListener (advanced)	Whether to stop the consumer message listener asynchronously, when stopping a route.	false	boolean
connectionCount (advanced)	The maximum number of connections available to this endpoint		Integer
connectionFactory (advanced)	Initializes the connectionFactory for the endpoint, which takes precedence over the component's connectionFactory, if any		ConnectionFactory
connectionResource (advanced)	Initializes the connectionResource for the endpoint, which takes precedence over the component's connectionResource, if any		ConnectionResource
destinationCreationStrategy (advanced)	To use a custom DestinationCreationStrategy.		DestinationCreation Strategy
exceptionListener (advanced)	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.		ExceptionListener
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
includeAllJMSXProperties (advanced)	Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.	false	boolean
jmsKeyFormatStrategy (advanced)	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification.Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the org.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the notation.		JmsKeyFormatStrategy
mapJmsMessage (advanced)	Specifies whether Camel should auto map the received JMS message to a suited payload type, such as javax.jms.TextMessage to a String etc. See section about how mapping works below for more details.	true	boolean

Name	Description	Default	Type
messageCreatedStrategy (advanced)	To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms.Message objects when Camel is sending a JMS message.		MessageCreatedStrategy
errorHandlerLoggingLevel (logging)	Allows to configure the default errorHandler logging level for logging uncaught exceptions.	WARN	LoggingLevel
errorHandlerLogStackTrace (logging)	Allows to control whether stacktraces should be logged or not, by the default errorHandler.	true	boolean
transacted (transaction)	Specifies whether to use transacted mode	false	boolean
transactionBatchCount (transaction)	If transacted sets the number of messages to process before committing a transaction.	-1	int
transactionBatchTimeout (transaction)	Sets timeout (in millis) for batch transactions, the value should be 1000 or higher.	5000	long
transactionCommitStrategy (transaction)	Sets the commit strategy.		TransactionCommit Strategy
sharedJMSSession (transaction)	Specifies whether to share JMS session with other SJMS endpoints. Turn this off if your route is accessing to multiple JMS providers. If you need transaction against multiple JMS providers, use jms component to leverage XA transaction.	true	boolean

Below is an example of how to configure the **Sjms2Component** with its required **ConnectionFactory** provider. It will create a single connection by default and store it using the component's internal pooling APIs to ensure that it is able to service Session creation requests in a thread safe manner.

```
Sjms2Component component = new Sjms2Component();
component.setConnectionFactory(new ActiveMQConnectionFactory("tcp://localhost:61616"));
getContext().addComponent("sjms2", component);
```

For a SJMS2 component that is required to support a durable subscription, you can override the default **ConnectionFactoryResource** instance and set the **clientId** property.

```
ConnectionFactoryResource connectionResource = new ConnectionFactoryResource();
```

```
connectionResource.setConnectionFactory(new
ActiveMQConnectionFactory("tcp://localhost:61616"));
connectionResource.setClientId("myclient-id");

Sjms2Component component = new Sjms2Component();
component.setConnectionResource(connectionResource);
component.setMaxConnections(1);
```

287.3. PRODUCER USAGE

287.3.1. InOnly Producer - (Default)

The *InOnly* producer is the default behavior of the SJMS2 Producer Endpoint.

```
from("direct:start")
    .to("sjms2:queue:bar");
```

287.3.2. InOut Producer

To enable *InOut* behavior append the **exchangePattern** attribute to the URI. By default it will use a dedicated TemporaryQueue for each consumer.

```
from("direct:start")
    .to("sjms2:queue:bar?exchangePattern=InOut");
```

You can specify a **namedReplyTo** though which can provide a better monitor point.

```
from("direct:start")
    .to("sjms2:queue:bar?exchangePattern=InOut&namedReplyTo=my.reply.to.queue");
```

287.4. CONSUMER USAGE

287.4.1. Durable Shared Subscription

To create a durable subscription that can be shared between one or more consumers. Use a JMS 2.0 compliant connection factory and specify a common subscriptionId. Then set the subscription properties durable and shared to true.

```
from("sjms2:topic:foo?consumerCount=3&subscriptionId=bar&durable=true&shared=true")
    .to("mock:result");

from("sjms2:topic:foo?consumerCount=2&subscriptionId=bar&durable=true&shared=true")
    .to("mock:result");
```

287.4.2. InOnly Consumer - (Default)

The *InOnly* xonsumer is the default Exchange behavior of the SJMS2 Consumer Endpoint.

```
from("sjms2:queue:bar")
    .to("mock:result");
```

287.4.3. InOut Consumer

To enable *InOut* behavior append the **exchangePattern** attribute to the URI.

```
from("sjms2:queue:in.out.test?exchangePattern=InOut")
    .transform(constant("Bye Camel"));
```

287.5. ADVANCED USAGE NOTES

287.5.1. Plugable Connection Resource Management

SJMS2 provides JMS **Connection** resource management through built-in connection pooling. This eliminates the need to depend on third party API pooling logic. However there may be times that you are required to use an external Connection resource manager such as those provided by J2EE or OSGi containers. For this SJMS2 provides an interface that can be used to override the internal SJMS2 Connection pooling capabilities. This is accomplished through the **ConnectionResource** interface.

The **ConnectionResource** provides methods for borrowing and returning Connections as needed is the contract used to provide **Connection** pools to the SJMS2 component. A user should use when it is necessary to integrate SJMS2 with an external connection pooling manager.

It is recommended though that for standard **ConnectionFactory** providers you use the **ConnectionFactoryResource** implementation that is provided with SJMS2 as-is or extend as it is optimized for this component.

Below is an example of using the plugable ConnectionResource with the ActiveMQ **PooledConnectionFactory**:

```
public class AMQConnectionResource implements ConnectionResource {
    private PooledConnectionFactory pcf;

    public AMQConnectionResource(String connectString, int maxConnections) {
        super();
        pcf = new PooledConnectionFactory(connectString);
        pcf.setMaxConnections(maxConnections);
        pcf.start();
    }

    public void stop() {
        pcf.stop();
    }

    @Override
    public Connection borrowConnection() throws Exception {
        Connection answer = pcf.createConnection();
        answer.start();
        return answer;
    }

    @Override
    public Connection borrowConnection(long timeout) throws Exception {
        // SNIPPED...
    }
```

```
    @Override
    public void returnConnection(Connection connection) throws Exception {
        // Do nothing since there isn't a way to return a Connection
        // to the instance of PooledConnectionFactory
        log.info("Connection returned");
    }
}
```

Then pass in the **ConnectionResource** to the **Sjms2Component**:

```
CamelContext camelContext = new DefaultCamelContext();
AMQConnectionResource pool = new AMQConnectionResource("tcp://localhost:33333", 1);
Sjms2Component component = new Sjms2Component();
component.setConnectionResource(pool);
camelContext.addComponent("sjms2", component);
```

To see the full example of its usage please refer to the **ConnectionResourceIT**.

287.5.2. Session, Consumer, & Producer Pooling & Caching Management

Coming soon ...

287.5.3. Batch Message Support

The Sjms2Producer supports publishing a collection of messages by creating an Exchange that encapsulates a **List**. This Sjms2Producer will take then iterate through the contents of the List and publish each message individually.

If when producing a batch of messages there is the need to set headers that are unique to each message you can use the SJMS2 **BatchMessage** class. When the Sjms2Producer encounters a **BatchMessage** list it will iterate each **BatchMessage** and publish the included payload and headers.

Below is an example of using the BatchMessage class. First we create a list of **BatchMessage**:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Then publish the list:

```
template.sendBody("sjms2:queue:batch.queue", messages);
```

287.5.4. Customizable Transaction Commit Strategies (Local JMS Transactions only)

SJMS2 provides a developer the means to create a custom and plugable transaction strategy through the use of the **TransactionCommitStrategy** interface. This allows a user to define a unique set of circumstances that the **SessionTransactionSynchronization** will use to determine when to commit the Session. An example of its use is the **BatchTransactionCommitStrategy** which is detailed further in the next section.

287.5.5. Transacted Batch Consumers & Producers

The SJMS2 component has been designed to support the batching of local JMS transactions on both the Producer and Consumer endpoints. How they are handled on each is very different though.

The SJMS2 consumer endpoint is a straightforward implementation that will process X messages before committing them with the associated Session. To enable batched transaction on the consumer first enable transactions by setting the **transacted** parameter to true and then adding the **transactionBatchCount** and setting it to any value that is greater than 0. For example the following configuration will commit the Session every 10 messages:

```
sjms2:queue:transacted.batch.consumer?transacted=true&transactionBatchCount=10
```

If an exception occurs during the processing of a batch on the consumer endpoint, the Session rollback is invoked causing the messages to be redelivered to the next available consumer. The counter is also reset to 0 for the **BatchTransactionCommitStrategy** for the associated Session as well. It is the responsibility of the user to ensure they put hooks in their processors of batch messages to watch for messages with the JMSRedelivered header set to true. This is the indicator that messages were rolled back at some point and that a verification of a successful processing should occur.

A transacted batch consumer also carries with it an instance of an internal timer that waits a default amount of time (5000ms) between messages before committing the open transactions on the Session. The default value of 5000ms (minimum of 1000ms) should be adequate for most use-cases but if further tuning is necessary simply set the **transactionBatchTimeout** parameter.

```
sjms2:queue:transacted.batch.consumer?
transacted=true&transactionBatchCount=10&transactionBatchTimeout=2000
```

The minimal value that will be accepted is 1000ms as the amount of context switching may cause unnecessary performance impacts without gaining benefit.

The producer endpoint is handled much differently though. With the producer after each message is delivered to its destination the Exchange is closed and there is no longer a reference to that message. To make a available all the messages available for redelivery you simply enable transactions on a Producer Endpoint that is publishing BatchMessages. The transaction will commit at the conclusion of the exchange which includes all messages in the batch list. Nothing additional need be configured. For example:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Now publish the List with transactions enabled:

```
template.sendBody("sjms2:queue:batch.queue?transacted=true", messages);
```

287.6. ADDITIONAL NOTES

287.6.1. Message Header Format

The SJMS2 Component uses the same header format strategy that is used in the Camel JMS Component. This plugable strategy ensures that messages sent over the wire conform to the JMS Message spec.

For the **exchange.in.header** the following rules apply for the header keys:

- Keys starting with **JMS** or **JMSX** are reserved.
- **exchange.in.headers** keys must be literals and all be valid Java identifiers (do not use dots in the key name).
- Camel replaces dots & hyphens and the reverse when when consuming JMS messages:
  - is replaced by *DOT* and the reverse replacement when Camel consumes the message.
  - is replaced by *HYPHEN* and the reverse replacement when Camel consumes the message. See also the option **jmsKeyFormatStrategy**, which allows use of your own custom strategy for formatting keys.

For the **exchange.in.header**, the following rules apply for the header values:

287.6.2. Message Content

To deliver content over the wire we must ensure that the body of the message that is being delivered adheres to the JMS Message Specification. Therefore, all that are produced must either be primitives or their counter objects (such as **Integer**, **Long**, **Character**). The types, **String**, **CharSequence**, **Date**, **BigDecimal** and **BigInteger** are all converted to their **toString()** representation. All other types are dropped.

287.6.3. Clustering

When using *InOut* with SJMS2 in a clustered environment you must either use TemporaryQueue destinations or use a unique named reply to destination per InOut producer endpoint. Message correlation is handled by the endpoint, not with message selectors at the broker. The InOut Producer Endpoint uses Java Concurrency Exchangers cached by the Message **JMSCorrelationID**. This provides a nice performance increase while reducing the overhead on the broker since all the messages are consumed from the destination in the order they are produced by the interested consumer.

Currently the only correlation strategy is to use the **JMSCorrelationId**. The *InOut* Consumer uses this strategy as well ensuring that all responses messages to the included **JMSReplyTo** destination also have the **JMSCorrelationId** copied from the request as well.

287.7. TRANSACTION SUPPORT

SJMS2 currently only supports the use of internal JMS Transactions. There is no support for the Camel Transaction Processor or the Java Transaction API (JTA).

287.7.1. Does Springless Mean I Can't Use Spring?

Not at all. Below is an example of the SJMS2 component using the Spring DSL:

```
<route
    id="inout.named.reply.to.producer.route">
    <from
        uri="direct:invoke.named.reply.to.queue" />
```

```
    <to
        uri="sjms2:queue:named.reply.to.queue?
namedReplyTo=my.response.queue&amp;exchangePattern=InOut" />
</route>
```

Springless refers to moving away from the dependency on the Spring JMS API. A new JMS client API is being developed from the ground up to power SJMS2.

CHAPTER 288. SLACK COMPONENT

Available as of Camel version 2.16

The **slack** component allows you to connect to an instance of Slack and delivers a message contained in the message body via a pre established Slack incoming webhook.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-slack</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

288.1. URI FORMAT

To send a message to a channel.

```
slack:#channel[?options]
```

To send a direct message to a slackuser.

```
slack:@username[?options]
```

288.2. OPTIONS

The Slack component supports 2 options which are listed below.

Name	Description	Default	Type
webhookUrl (producer)	The incoming webhook URL		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Slack endpoint is configured using URI syntax:

```
slack:channel
```

with the following path and query parameters:

288.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
channel	**Required** The channel name (syntax name) or slackuser (syntax userName) to send a message directly to an user.		String

288.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
iconEmoji (producer)	Use a Slack emoji as an avatar		String
iconUrl (producer)	The avatar that the component will use when sending message to a channel or user.		String
username (producer)	This is the username that the bot will have when sending messages to a channel or user.		String
webhookUrl (producer)	The incoming webhook URL		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

288.3. SLACKCOMPONENT

The SlackComponent with XML must be configured as a Spring or Blueprint bean that contains the incoming webhook url for the integration as a parameter.

```
<bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
    <property name="webhookUrl"
value="https://hooks.slack.com/services/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"/>
</bean>
```

For Java you can configure this using Java code.

288.4. EXAMPLE

A CamelContext with Blueprint could be as:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" default-activation="lazy">

    <bean id="slack" class="org.apache.camel.component.slack.SlackComponent">
        <property name="webhookUrl"
```

```
value="https://hooks.slack.com/services/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"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="direct:test"/>
      <to uri="slack:#channel?iconEmoji=:camel:&amp;username=CamelTest"/>
    </route>
  </camelContext>

</blueprint>
```

288.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 289. SMPP COMPONENT

Available as of Camel version 2.2

This component provides access to an SMSC (Short Message Service Center) over the SMPP protocol to send and receive SMS. The JSMPP library is used for the protocol implementation.

The Camel component currently operates as an ESME (External Short Messaging Entity) and not as an SMSC itself.

Starting with*Camel 2.9* you are also able to execute ReplaceSm, QuerySm, SubmitMulti, CancelSm and DataSm.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-smpp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

289.1. SMS LIMITATIONS

SMS is neither reliable or secure. Users who require reliable and secure delivery may want to consider using the XMPP or SIP components instead, combined with a smartphone app supporting the chosen protocol.

- Reliability: although the SMPP standard offers a range of feedback mechanisms to indicate errors, non-delivery and confirmation of delivery it is not uncommon for mobile networks to hide or simulate these responses. For example, some networks automatically send a delivery confirmation for every message even if the destination number is invalid or not switched on. Some networks silently drop messages if they think they are spam. Spam detection rules in the network may be very crude, sometimes more than 100 messages per day from a single sender may be considered spam.

- Security: there is basic encryption for the last hop from the radio tower down to the recipient handset. SMS messages are not encrypted or authenticated in any other part of the network. Some operators allow staff in retail outlets or call centres to browse through the SMS message histories of their customers. Message sender identity can be easily forged. Regulators and even the mobile telephone industry itself has cautioned against the use of SMS in two-factor authentication schemes and other purposes where security is important.

While the Camel component makes it as easy as possible to send messages to the SMS network, it can not offer an easy solution to these problems.

289.2. DATA CODING, ALPHABET AND INTERNATIONAL CHARACTER SETS

Data coding and alphabet can be specified on a per-message basis. Default values can be specified for the endpoint. It is important to understand the relationship between these options and the way the component acts when more than one value is set.

Data coding is an 8 bit field in the SMPP wire format.

Alphabet corresponds to bits 0-3 of the data coding field. For some types of message, where a message class is used (by setting bit 5 of the data coding field), the lower two bits of the data coding field are not interpreted as alphabet and only bits 2 and 3 impact the alphabet.

Furthermore, current version of the JSMPP library only seems to support bits 2 and 3, assuming that bits 0 and 1 are used for message class. This is why the Alphabet class in JSMPP doesn't support the value 3 (binary 0011) which indicates ISO-8859-1.

Although JSMPP provides a representation of the message class parameter, the Camel component doesn't currently provide a way to set it other than manually setting the corresponding bits in the data coding field.

When setting the data coding field in the outgoing message, the Camel component considers the following values and uses the first one it can find:

- the data coding specified in a header
- the alphabet specified in a header
- the data coding specified in the endpoint configuration (URI parameter)

Older versions of Camel had bugs in support for international character sets. This feature only worked when a single encoding was used for all messages and was troublesome when users wanted to change it on a per-message basis. Users who require this to work should ensure their version of Camel includes the fix for

JIRA Issues Macro: com.atlassian.sal.api.net.ResponseStatusException: Unexpected response received. Status code: 404

.

In addition to trying to send the data coding value to the SMSC, the Camel component also tries to analyze the message body, convert it to a Java String (Unicode) and convert that to a byte array in the corresponding alphabet When deciding which alphabet to use in the byte array, the Camel SMPP component does not consider the data coding value (header or configuration), it only considers the specified alphabet (from either the header or endpoint parameter).

If some characters in the String can't be represented in the chosen alphabet, they may be replaced by the question mark (?) symbol. Users of the API may want to consider checking if their message body can be converted to ISO-8859-1 before passing it to the component and if not, setting the alphabet header to request UCS-2 encoding. If the alphabet and data coding options are not specified at all then the component may try to detect the required encoding and set the data coding for you.

The list of alphabet codes are specified in the SMPP specification v3.4, section 5.2.19. One notable limitation of the SMPP specification is that there is no alphabet code for explicitly requesting use of the GSM 3.38 (7 bit) character set. Choosing the value 0 for the alphabet selects the SMSC *default* alphabet, this usually means GSM 3.38 but it is not guaranteed. The SMPP gateway Nexmo actually allows the default to be mapped to any other character set with a control panel option. It is suggested that users check with their SMSC operator to confirm exactly which character set is being used as the default.

289.3. MESSAGE SPLITTING AND THROTTLING

After transforming a message body from a String to a byte array, the Camel component is also responsible for splitting the message into parts (within the 140 byte SMS size limit) before passing it to JSMPP. This is completed automatically.

If the GSM 3.38 alphabet is used, the component will pack up to 160 characters into the 140 byte message body. If an 8 bit character set is used (e.g. ISO-8859-1 for western Europe) then 140 characters will be allowed within the 140 byte message body. If 16 bit UCS-2 encoding is used then just 70 characters fit into each 140 byte message.

Some SMSC providers implement throttling rules. Each part of a message that has been split may be counted separately by the provider's throttling mechanism. The Camel Throttler component can be useful for throttling messages in the SMPP route before handing them to the SMSC.

289.4. URI FORMAT

```
smpp://[username@]hostname[:port][?options]
smpps://[username@]hostname[:port][?options]
```

If no **username** is provided, then Camel will provide the default value **smppclient**.
If no **port** number is provided, then Camel will provide the default value **2775**.
Camel 2.3: If the protocol name is "smpps", camel-smpp with try to use SSLSocket to init a connection to the server.

You can append query options to the URI in the following format, **?option=value&option=value&…**

289.5. URI OPTIONS

The SMPP component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared SmppConfiguration as configuration.		SmppConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The SMPP endpoint is configured using URI syntax:

```
smpp:host:port
```

with the following path and query parameters:

289.5.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	Hostname for the SMSC server to use.	localhost	String

Name	Description	Default	Type
port	Port number for the SMSC server to use.	2775	Integer

289.5.2. Query Parameters (38 parameters):

Name	Description	Default	Type
initialReconnectDelay (common)	Defines the initial delay in milliseconds after the consumer/producer tries to reconnect to the SMSC, after the connection was lost.	5000	long
maxReconnect (common)	Defines the maximum number of attempts to reconnect to the SMSC, if SMSC returns a negative bind response	2147483647	int
reconnectDelay (common)	Defines the interval in milliseconds between the reconnect attempts, if the connection to the SMSC was lost and the previous was not succeed.	5000	long
splittingPolicy (common)	You can specify a policy for handling long messages: ALLOW - the default, long messages are split to 140 bytes per message TRUNCATE - long messages are split and only the first fragment will be sent to the SMSC. Some carriers drop subsequent fragments so this reduces load on the SMPP connection sending parts of a message that will never be delivered. REJECT - if a message would need to be split, it is rejected with an SMPP NegativeResponseException and the reason code signifying the message is too long.	ALLOW	SmppSplittingPolicy
systemType (common)	This parameter is used to categorize the type of ESME (External Short Message Entity) that is binding to the SMSC (max. 13 characters).	cp	String
addressRange (consumer)	You can specify the address range for the SmppConsumer as defined in section 5.2.7 of the SMPP 3.4 specification. The SmppConsumer will receive messages only from SMSC's which target an address (MSISDN or IP address) within this range.		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
destAddr (producer)	Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS. Only for SubmitSm, SubmitMulti, CancelSm and DataSm.	1717	String
destAddrNpi (producer)	Defines the type of number (TON) to be used in the SME destination address parameters. Only for SubmitSm, SubmitMulti, CancelSm and DataSm. The following NPI values are defined: 0: Unknown 1: ISDN (E163/E164) 2: Data (X.121) 3: Telex (F.69) 6: Land Mobile (E.212) 8: National 9: Private 10: ERMES 13: Internet (IP) 18: WAP Client Id (to be defined by WAP Forum)		byte
destAddrTon (producer)	Defines the type of number (TON) to be used in the SME destination address parameters. Only for SubmitSm, SubmitMulti, CancelSm and DataSm. The following TON values are defined: 0: Unknown 1: International 2: National 3: Network Specific 4: Subscriber Number 5: Alphanumeric 6: Abbreviated		byte
lazySessionCreation (producer)	Sessions can be lazily created to avoid exceptions, if the SMSC is not available when the Camel producer is started. Camel will check the in message headers 'CamelSmppSystemId' and 'CamelSmppPassword' of the first exchange. If they are present, Camel will use these data to connect to the SMSC.	false	boolean

Name	Description	Default	Type
numberingPlanIndicator (producer)	Defines the numeric plan indicator (NPI) to be used in the SME. The following NPI values are defined: 0: Unknown 1: ISDN (E163/E164) 2: Data (X.121) 3: Telex (F.69) 6: Land Mobile (E.212) 8: National 9: Private 10: ERMES 13: Internet (IP) 18: WAP Client Id (to be defined by WAP Forum)		byte
priorityFlag (producer)	Allows the originating SME to assign a priority level to the short message. Only for SubmitSm and SubmitMulti. Four Priority Levels are supported: 0: Level 0 (lowest) priority 1: Level 1 priority 2: Level 2 priority 3: Level 3 (highest) priority		byte
protocolId (producer)	The protocol id		byte
registeredDelivery (producer)	Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. The following values are defined: 0: No SMSC delivery receipt requested. 1: SMSC delivery receipt requested where final delivery outcome is success or failure. 2: SMSC delivery receipt requested where the final delivery outcome is delivery failure.		byte
replaceIfPresentFlag (producer)	Used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following replace if present flag values are defined: 0: Don't replace 1: Replace		byte
serviceType (producer)	The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: CMT: Cellular Messaging CPT: Cellular Paging VMN: Voice Mail Notification VMA: Voice Mail Alerting WAP: Wireless Application Protocol USSD: Unstructured Supplementary Services Data	CMT	String
sourceAddr (producer)	Defines the address of SME (Short Message Entity) which originated this message.	1616	String

Name	Description	Default	Type
sourceAddrNpi (producer)	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: 0: Unknown 1: ISDN (E163/E164) 2: Data (X.121) 3: Telex (F.69) 6: Land Mobile (E.212) 8: National 9: Private 10: ERMES 13: Internet (IP) 18: WAP Client Id (to be defined by WAP Forum)		byte
sourceAddrTon (producer)	Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: 0: Unknown 1: International 2: National 3: Network Specific 4: Subscriber Number 5: Alphanumeric 6: Abbreviated		byte
typeOfNumber (producer)	Defines the type of number (TON) to be used in the SME. The following TON values are defined: 0: Unknown 1: International 2: National 3: Network Specific 4: Subscriber Number 5: Alphanumeric 6: Abbreviated		byte
enquireLinkTimer (advanced)	Defines the interval in milliseconds between the confidence checks. The confidence check is used to test the communication path between an ESME and an SMSC.	5000	Integer
sessionStateListener (advanced)	You can refer to a org.jsmpp.session.SessionStateListener in the Registry to receive callbacks when the session state changed.		SessionStateListener
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
transactionTimer (advanced)	Defines the maximum period of inactivity allowed after a transaction, after which an SMPP entity may assume that the session is no longer active. This timer may be active on either communicating SMPP entity (i.e. SMSC or ESME).	10000	Integer
alphabet (codec)	Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. 0: SMSC Default Alphabet 4: 8 bit Alphabet 8: UCS2 Alphabet		byte

Name	Description	Default	Type
dataCoding (codec)	Defines the data coding according the SMPP 3.4 specification, section 5.2.19. Example data encodings are: 0: SMSC Default Alphabet 3: Latin 1 (ISO-8859-1) 4: Octet unspecified (8-bit binary) 8: UCS2 (ISO/IEC-10646) 13: Extended Kanji JIS(X 0212-1990)		byte
encoding (codec)	Defines the encoding scheme of the short message user data. Only for SubmitSm, ReplaceSm and SubmitMulti.	ISO-8859-1	String
httpProxyHost (proxy)	If you need to tunnel SMPP through a HTTP proxy, set this attribute to the hostname or ip address of your HTTP proxy.		String
httpProxyPassword (proxy)	If your HTTP proxy requires basic authentication, set this attribute to the password required for your HTTP proxy.		String
httpProxyPort (proxy)	If you need to tunnel SMPP through a HTTP proxy, set this attribute to the port of your HTTP proxy.	3128	Integer
httpProxyUsername (proxy)	If your HTTP proxy requires basic authentication, set this attribute to the username required for your HTTP proxy.		String
proxyHeaders (proxy)	These headers will be passed to the proxy server while establishing the connection.		Map
password (security)	The password for connecting to SMSC server.		String
systemId (security)	The system id (username) for connecting to SMSC server.	smppclient	String
usingSSL (security)	Whether using SSL with the smpps protocol	false	boolean

You can have as many of these options as you like.

```
smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer
```

289.6. PRODUCER MESSAGE HEADERS

The following message headers can be used to affect the behavior of the SMPP producer

Header	Type	Description
CamelSmppDestAddr	**List/String**	**only for SubmitSm, SubmitMulti, CancelSm and DataSm** Defines the destination SME address(es). For mobile terminated messages, this is the directory number of the recipient MS. Is must be a **List<String>** for SubmitMulti and a **String** otherwise.
CamelSmppDestAddrTon	**Byte**	**only for SubmitSm, SubmitMulti, CancelSm and DataSm** Defines the type of number (TON) to be used in the SME destination address parameters. Use the **sourceAddrTon** URI option values defined above.
CamelSmppDestAddrNpi	**Byte**	**only for SubmitSm, SubmitMulti, CancelSm and DataSm** Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. Use the URI option **sourceAddrNpi** values defined above.
CamelSmppSourceAddr	**String**	Defines the address of SME (Short Message Entity) which originated this message.
CamelSmppSourceAddrTon	**Byte**	Defines the type of number (TON) to be used in the SME originator address parameters. Use the **sourceAddrTon** URI option values defined above.
CamelSmppSourceAddrNpi	**Byte**	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Use the URI option **sourceAddrNpi** values defined above.
CamelSmppServiceType	**String**	The service type parameter can be used to indicate the SMS Application service associated with the message. Use the URI option **serviceType** settings above.
CamelSmppRegisteredDelivery	**Byte**	**only for SubmitSm, ReplaceSm, SubmitMulti and DataSm** Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. Use the URI option **registeredDelivery** settings above.
CamelSmppPriorityFlag	**Byte**	**only for SubmitSm and SubmitMulti** Allows the originating SME to assign a priority level to the short message. Use the URI option **priorityFlag** settings above.

Header	Type	Description
CamelSmppScheduleDeliveryTime	**Date**	**only for SubmitSm, SubmitMulti and ReplaceSm** This parameter specifies the scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in chapter 7.1.1. in the smpp specification v3.4.
CamelSmppValidityPeriod	**String / Date**	**only for SubmitSm, SubmitMulti and ReplaceSm** The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not delivered to the destination. If it's provided as **Date**, it's interpreted as absolute time. **Camel 2.9.1 onwards:** It can be defined in absolute time format or relative time format if you provide it as **String** as specified in chapter 7.1.1 in the smpp specification v3.4.
CamelSmppReplaceIfPresentFlag	**Byte**	**only for SubmitSm and SubmitMulti** The replace if present flag parameter is used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following values are defined: **0**, Don't replace and **1**, Replace
CamelSmppAlphabet / CamelSmppDataCoding	**Byte**	**Camel 2.5 For SubmitSm, SubmitMulti and ReplaceSm** (Prior to **Camel 2.9** use **CamelSmppDataCoding** instead of **CamelSmppAlphabet**.) The data coding according to the SMPP 3.4 specification, section 5.2.19. Use the URI option **alphabet** settings above.
CamelSmppOptionalParameters	**Map<String, String>**	**Deprecated and will be removed in Camel 2.13.0/3.0.0** **Camel 2.10.5 and 2.11.1 onwards and only for SubmitSm, SubmitMulti and DataSm** The optional parameters send back by the SMSC.
CamelSmppOptionalParameter	**Map<Short, Object>**	**Camel 2.10.7 and 2.11.2 onwards and only for SubmitSm, SubmitMulti and DataSm** The optional parameter which are send to the SMSC. The value is converted in the following way: **String → org.jsmpp.bean.OptionalParameter.COctetString**, **byte[] → org.jsmpp.bean.OptionalParameter.OctetString**, **Byte → org.jsmpp.bean.OptionalParameter.Byte**, **Integer → org.jsmpp.bean.OptionalParameter.Int**, **Short → org.jsmpp.bean.OptionalParameter.Short**, **null → org.jsmpp.bean.OptionalParameter.Null**

Header	Type	Description
CamelSmppEncoding	String	**Camel 2.14.1 and Camel 2.15.0 onwards and*** only for SubmitSm, SubmitMulti and DataSm*. Specifies the encoding (character set name) of the bytes in the message body. If the message body is a string then this is not relevant because Java Strings are always Unicode. If the body is a byte array then this header can be used to indicate that it is ISO-8859-1 or some other value. Default value is specified by the endpoint configuration parameter *encoding*
CamelSmppSplittingPolicy	String	**Camel 2.14.1 and Camel 2.15.0 onwards and*** only for SubmitSm, SubmitMulti and DataSm*. Specifies the policy for message splitting for this exchange. Possible values are described in the endpoint configuration parameter *splittingPolicy*

The following message headers are used by the SMPP producer to set the response from the SMSC in the message header

Header	Type	Description
CamelSmppId	**List<String>/String**	The id to identify the submitted short message(s) for later use. **From Camel 2.9.0**: In case of a ReplaceSm, QuerySm, CancelSm and DataSm this header vaule is a **String**. In case of a SubmitSm or SubmitMultiSm this header vaule is a **List<String>**.
CamelSmppSentMessageCount	**Integer**	**From Camel 2.9 onwards only for SubmitSm and SubmitMultiSm** The total number of messages which has been sent.
CamelSmppError	**Map<String, List<Map<String, Object>>>**	**From Camel 2.9 onwards only for SubmitMultiSm** The errors which occurred by sending the short message(s) the form **Map<String, List<Map<String, Object>>>** (messageID : (destAddr : address, error : errorCode)).
CamelSmppOptionalParameters	**Map<String, String>**	**Deprecated and will be removed in Camel 2.13.0/3.0.0From Camel 2.11.1 onwards only for DataSm** The optional parameters which are returned from the SMSC by sending the message.

Header	Type	Description
CamelSmppOptionalParameter	**Map<Short, Object>**	**From Camel 2.10.7, 2.11.2 onwards only for DataSm** The optional parameter which are returned from the SMSC by sending the message. The key is the **Short** code for the optional parameter. The value is converted in the following way: **org.jsmpp.bean.OptionalParameter.COctetString → String**, **org.jsmpp.bean.OptionalParameter.OctetString → byte[]**, **org.jsmpp.bean.OptionalParameter.Byte → Byte**, **org.jsmpp.bean.OptionalParameter.Int → Integer**, **org.jsmpp.bean.OptionalParameter.Short → Short**, **org.jsmpp.bean.OptionalParameter.Null → null**

289.7. CONSUMER MESSAGE HEADERS

The following message headers are used by the SMPP consumer to set the request data from the SMSC in the message header

Header	Type	Description
CamelSmppSequenceNumber	**Integer**	**only for AlertNotification, DeliverSm and DataSm** A sequence number allows a response PDU to be correlated with a request PDU. The associated SMPP response PDU must preserve this field.
CamelSmppCommandId	**Integer**	**only for AlertNotification, DeliverSm and DataSm** The command id field identifies the particular SMPP PDU. For the complete list of defined values see chapter 5.1.2.1 in the smpp specification v3.4.
CamelSmppSourceAddr	**String**	**only for AlertNotification, DeliverSm and DataSm** Defines the address of SME (Short Message Entity) which originated this message.
CamelSmppSourceAddrNpi	**Byte**	**only for AlertNotification and DataSm** Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Use the URI option **sourceAddrNpi** values defined above.
CamelSmppSourceAddrTon	**Byte**	**only for AlertNotification and DataSm** Defines the type of number (TON) to be used in the SME originator address parameters. Use the **sourceAddrTon** URI option values defined above.

Header	Type	Description
CamelSmppEsmeAddr	**String**	**only for AlertNotification** Defines the destination ESME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmppEsmeAddrNpi	**Byte**	**only for AlertNotification** Defines the numeric plan indicator (NPI) to be used in the ESME originator address parameters. Use the URI option **sourceAddrNpi** values defined above.
CamelSmppEsmeAddrTon	**Byte**	**only for AlertNotification** Defines the type of number (TON) to be used in the ESME originator address parameters. Use the **sourceAddrTon** URI option values defined above.
CamelSmppId	**String**	**only for smsc DeliveryReceipt and DataSm** The message ID allocated to the message by the SMSC when originally submitted.
CamelSmppDelivered	**Integer**	**only for smsc DeliveryReceipt** Number of short messages delivered. This is only relevant where the original message was submitted to a distribution list.The value is padded with leading zeros if necessary.
CamelSmppDoneDate	**Date**	**only for smsc DeliveryReceipt** The time and date at which the short message reached it's final state. The format is as follows: YYMMDDhhmm.
CamelSmppStatus	**DeliveryReceiptState**	**only for smsc DeliveryReceipt:** The final status of the message. The following values are defined: **DELIVRD**: Message is delivered to destination,**EXPIRED**: Message validity period has expired, **DELETED**: Message has been deleted,**UNDELIV**: Message is undeliverable, **ACCEPTD**: Message is in accepted state (i.e. has been manually read on behalf of the subscriber by customer service), **UNKNOWN**: Message is in invalid state, **REJECTD**: Message is in a rejected state
CamelSmppCommandStatus	**Integer**	**only for DataSm** The Command status of the message.
CamelSmppError	**String**	**only for smsc DeliveryReceipt** Where appropriate this may hold a Network specific error code or an SMSC error code for the attempted delivery of the message. These errors are Network or SMSC specific and are not included here.

Header	Type	Description
CamelSmppSubmitDate	**Date**	**only for smsc DeliveryReceipt** The time and date at which the short message was submitted. In the case of a message which has been replaced, this is the date that the original message was replaced. The format is as follows: YYMMDDhhmm.
CamelSmppSubmitted	**Integer**	**only for smsc DeliveryReceipt** Number of short messages originally submitted. This is only relevant when the original message was submitted to a distribution list.The value is padded with leading zeros if necessary.
CamelSmppDestAddr	**String**	**only for DeliverSm and DataSm:** Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmppScheduleDeliveryTime	**String**	**only for DeliverSm:** This parameter specifies the scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in Section 7.1.1. in the smpp specification v3.4.
CamelSmppValidityPeriod	**String**	**only for DeliverSm** The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not delivered to the destination. It can be defined in absolute time format or relative time format. The encoding of absolute and relative time format is specified in Section 7.1.1 in the smpp specification v3.4.
CamelSmppServiceType	**String**	**only for DeliverSm and DataSm** The service type parameter indicates the SMS Application service associated with the message.
CamelSmppRegisteredDelivery	**Byte**	**only for DataSm** Is used to request an delivery receipt and/or SME originated acknowledgements. Same values as in Producer header list above.
CamelSmppDestAddrNpi	**Byte**	**only for DataSm** Defines the numeric plan indicator (NPI) in the destination address parameters. Use the URI option **sourceAddrNpi** values defined above.
CamelSmppDestAddrTon	**Byte**	**only for DataSm** Defines the type of number (TON) in the destination address parameters. Use the **sourceAddrTon** URI option values defined above.

Header	Type	Description
CamelSmppMessageType	**String**	**Camel 2.6 onwards**: Identifies the type of an incoming message:**AlertNotification**: an SMSC alert notification, **DataSm**: an SMSC data short message,**DeliveryReceipt**: an SMSC delivery receipt, **DeliverSm**: an SMSC deliver short message
CamelSmppOptionalParameters	**Map<String, Object>**	**Deprecated and will be removed in Camel 2.13.0/3.0.0Camel 2.10.5 onwards and only for DeliverSm** The optional parameters send back by the SMSC.
CamelSmppOptionalParameter	**Map<Short, Object>**	**Camel 2.10.7, 2.11.2 onwards and only for DeliverSm**The optional parameters send back by the SMSC. The key is the **Short** code for the optional parameter. The value is converted in the following way: **org.jsmpp.bean.OptionalParameter.COctetString → String**, **org.jsmpp.bean.OptionalParameter.OctetString → byte[]**, **org.jsmpp.bean.OptionalParameter.Byte → Byte**, **org.jsmpp.bean.OptionalParameter.Int → Integer**, **org.jsmpp.bean.OptionalParameter.Short → Short**, **org.jsmpp.bean.OptionalParameter.Null → null**

TIP

JSMPP library See the documentation of the JSMPP Library for more details about the underlying library.

289.8. EXCEPTION HANDLING

This component supports the general Camel exception handling capabilities

When an error occurs sending a message with SubmitSm (the default action), the org.apache.camel.component.smpp.SmppException is thrown with a nested exception, org.jsmpp.extra.NegativeResponseException. Call NegativeResponseException.getCommandStatus() to obtain the exact SMPP negative response code, the values are explained in the SMPP specification 3.4, section 5.1.3.
Camel 2.8 onwards: When the SMPP consumer receives a **DeliverSm** or **DataSm** short message and the processing of these messages fails, you can also throw a **ProcessRequestException** instead of handle the failure. In this case, this exception is forwarded to the underlying JSMPP library which will return the included error code to the SMSC. This feature is useful to e.g. instruct the SMSC to resend the short message at a later time. This could be done with the following lines of code:

```
from("smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer")
  .doTry()
    .to("bean:dao?method=updateSmsState")
  .doCatch(Exception.class)
    .throwException(new ProcessRequestException("update of sms state failed", 100))
  .end();
```

Please refer to the SMPP specification for the complete list of error codes and their meanings.

289.9. SAMPLES

A route which sends an SMS using the Java DSL:

```
from("direct:start")
  .to("smpp://smppclient@localhost:2775?
     password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=producer");
```

A route which sends an SMS using the Spring XML DSL:

```
<route>
  <from uri="direct:start"/>
  <to uri="smpp://smppclient@localhost:2775?

password=password&amp;enquireLinkTimer=3000&amp;transactionTimer=5000&amp;systemType=pro
ducer"/>
</route>
```

A route which receives an SMS using the Java DSL:

```
from("smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer")
  .to("bean:foo");
```

A route which receives an SMS using the Spring XML DSL:

```
  <route>
    <from uri="smpp://smppclient@localhost:2775?

password=password&amp;enquireLinkTimer=3000&amp;transactionTimer=5000&amp;systemType=co
nsumer"/>
    <to uri="bean:foo"/>
  </route>
```

TIP

SMSC simulator If you need an SMSC simulator for your test, you can use the simulator provided by Logica.

289.10. DEBUG LOGGING

This component has log level **DEBUG**, which can be helpful in debugging problems. If you use log4j, you can add the following line to your configuration:

```
log4j.logger.org.apache.camel.component.smpp=DEBUG
```

289.11. SEE ALSO

- Configuring Camel

- Component
- Endpoint
- Getting Started

CHAPTER 290. SNMP COMPONENT

Available as of Camel version 2.1

The **snmp:** component gives you the ability to poll SNMP capable devices or receiving traps

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-snmp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

290.1. URI FORMAT

```
snmp://hostname[:port][?Options]
```

The component supports polling OID values from an SNMP enabled device and receiving traps.

You can append query options to the URI in the following format, **?option=value&option=value&…**

290.2. SNMP PRODUCER

Available from 2.18 release

It can also be used to request information using GET method.

The response body type is org.apache.camel.component.snmp.SnmpMessage

290.3. OPTIONS

The SNMP component has no options.

The SNMP endpoint is configured using URI syntax:

```
snmp:host:port
```

with the following path and query parameters:

290.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** Hostname of the SNMP enabled device		String
port	**Required** Port number of the SNMP enabled device		Integer

290.3.2. Query Parameters (34 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delay (consumer)	Sets update rate in seconds	60000	long
oids (consumer)	Defines which values you are interested in. Please have a look at the Wikipedia to get a better understanding. You may provide a single OID or a coma separated list of OIDs. Example: oids=1.3.6.1.2.1.1.3.0,1.3.6.1.2.1.25.3.2.1.5.1,1.3.6.1.2.1.25.3.5.1.1.1,1.3.6.1.2.1.43.5.1.1.11.1		String
protocol (consumer)	Here you can select which protocol to use. You can use either udp or tcp.	udp	String
retries (consumer)	Defines how often a retry is made before canceling the request.	2	int
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
snmpCommunity (consumer)	Sets the community octet string for the snmp request.	public	String
snmpContextEngineId (consumer)	Sets the context engine ID field of the scoped PDU.		String
snmpContextName (consumer)	Sets the context name field of this scoped PDU.		String
snmpVersion (consumer)	Sets the snmp version for the request. The value 0 means SNMPv1, 1 means SNMPv2c, and the value 3 means SNMPv3	0	int
timeout (consumer)	Sets the timeout value for the request in millis.	1500	int
type (consumer)	Which operation to perform such as poll, trap, etc.		SnmpActionType

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel

Name	Description	Default	Type
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
authenticationPassphrase (security)	The authentication passphrase. If not null, authenticationProtocol must also be not null. RFC3414 11.2 requires passphrases to have a minimum length of 8 bytes. If the length of authenticationPassphrase is less than 8 bytes an IllegalArgumentException is thrown.		String
authenticationProtocol (security)	Authentication protocol to use if security level is set to enable authentication The possible values are: MD5, SHA1		String
privacyPassphrase (security)	The privacy passphrase. If not null, privacyProtocol must also be not null. RFC3414 11.2 requires passphrases to have a minimum length of 8 bytes. If the length of authenticationPassphrase is less than 8 bytes an IllegalArgumentException is thrown.		String
privacyProtocol (security)	The privacy protocol ID to be associated with this user. If set to null, this user only supports unencrypted messages.		String

Name	Description	Default	Type
securityLevel (security)	Sets the security level for this target. The supplied security level must be supported by the security model dependent information associated with the security name set for this target. The value 1 means: No authentication and no encryption. Anyone can create and read messages with this security level The value 2 means: Authentication and no encryption. Only the one with the right authentication key can create messages with this security level, but anyone can read the contents of the message. The value 3 means: Authentication and encryption. Only the one with the right authentication key can create messages with this security level, and only the one with the right encryption/decryption key can read the contents of the message.	3	int
securityName (security)	Sets the security name to be used with this target.		String

290.4. THE RESULT OF A POLL

Given the situation, that I poll for the following OIDs:

OIDs

```
1.3.6.1.2.1.1.3.0
1.3.6.1.2.1.25.3.2.1.5.1
1.3.6.1.2.1.25.3.5.1.1.1
1.3.6.1.2.1.43.5.1.1.11.1
```

The result will be the following:

Result of toString conversion

```
<?xml version="1.0" encoding="UTF-8"?>
<snmp>
  <entry>
    <oid>1.3.6.1.2.1.1.3.0</oid>
    <value>6 days, 21:14:28.00</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.25.3.2.1.5.1</oid>
    <value>2</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.25.3.5.1.1.1</oid>
    <value>3</value>
  </entry>
  <entry>
```

```
    <oid>1.3.6.1.2.1.43.5.1.1.11.1</oid>
    <value>6</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.1.1.0</oid>
    <value>My Very Special Printer Of Brand Unknown</value>
  </entry>
</snmp>
```

As you maybe recognized there is one more result than requested....1.3.6.1.2.1.1.1.0.
This one is filled in by the device automatically in this special case. So it may absolutely happen, that you receive more than you requested...be prepared.

290.5. EXAMPLES

Polling a remote device:

```
snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0
```

Setting up a trap receiver (**Note that no OID info is needed here!**):

```
snmp:127.0.0.1:162?protocol=udp&type=TRAP
```

From Camel 2.10.0, you can get the community of SNMP TRAP with message header 'securityName', peer address of the SNMP TRAP with message header 'peerAddress'.

Routing example in Java: (converts the SNMP PDU to XML String)

```
from("snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0").
convertBodyTo(String.class).
to("activemq:snmp.states");
```

290.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 291. SOAP DATAFORMAT

Available as of Camel version 2.3

SOAP is a Data Format which uses JAXB2 and JAX-WS annotations to marshal and unmarshal SOAP payloads. It provides the basic features of Apache CXF without need for the CXF Stack.

Supported SOAP versions

SOAP 1.1 is supported by default. SOAP 1.2 is supported from Camel 2.11 onwards.

Namespace prefix mapping

See JAXB for details how you can control namespace prefix mappings when marshalling using SOAP data format.

291.1. SOAP OPTIONS

The SOAP dataformat supports 7 options which are listed below.

Name	Default	Java Type	Description
contextPath		**String**	Package name where your JAXB classes are located.
encoding		**String**	To overrule and use a specific encoding
elementNameStrategyRef		**String**	Refers to an element strategy to lookup from the registry. An element name strategy is used for two purposes. The first is to find a xml element name for a given object and soap action when marshaling the object into a SOAP message. The second is to find an Exception class for a given soap fault name. The following three element strategy class name is provided out of the box. QNameStrategy - Uses a fixed qName that is configured on instantiation. Exception lookup is not supported TypeNameStrategy - Uses the name and namespace from the XMLType annotation of the given type. If no namespace is set then package-info is used. Exception lookup is not supported ServiceInterfaceStrategy - Uses information from a webservice interface to determine the type name and to find the exception class for a SOAP fault All three classes is located in the package name org.apache.camel.dataformat.soap.name If you have generated the web service stub code with cxf-codegen or a similar tool then you probably will want to use the ServiceInterfaceStrategy. In the case you have no annotated service interface you should use QNameStrategy or TypeNameStrategy.
version	**1.1**	**String**	SOAP version should either be 1.1 or 1.2. Is by default 1.1

Name	Default	Java Type	Description
namespacePrefixRef		**String**	When marshalling using JAXB or SOAP then the JAXB implementation will automatic assign namespace prefixes, such as ns2, ns3, ns4 etc. To control this mapping, Camel allows you to refer to a map which contains the desired mapping.
schema		**String**	To validate against an existing schema. Your can use the prefix classpath:, file: or http: to specify how the resource should by resolved. You can separate multiple schema files by using the ',' character.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

291.2. ELEMENTNAMESTRATEGY

An element name strategy is used for two purposes. The first is to find a xml element name for a given object and soap action when marshaling the object into a SOAP message. The second is to find an Exception class for a given soap fault name.

Strategy	Usage
QNameStrategy	Uses a fixed qName that is configured on instantiation. Exception lookup is not supported
TypeNameStrategy	Uses the name and namespace from the @XMLType annotation of the given type. If no namespace is set then package-info is used. Exception lookup is not supported
ServiceInterfaceStrategy	Uses information from a webservice interface to determine the type name and to find the exception class for a SOAP fault

If you have generated the web service stub code with cxf-codegen or a similar tool then you probably will want to use the ServiceInterfaceStrategy. In the case you have no annotated service interface you should use QNameStrategy or TypeNameStrategy.

291.3. USING THE JAVA DSL

The following example uses a named DataFormat of *soap* which is configured with the package com.example.customerservice to initialize the JAXBContext. The second parameter is the ElementNameStrategy. The route is able to marshal normal objects as well as exceptions. (Note the

below just sends a SOAP Envelope to a queue. A web service provider would actually need to be listening to the queue for a SOAP call to actually occur, in which case it would be a one way SOAP request. If you need request reply then you should look at the next example.)

```
SoapJaxbDataFormat soap = new SoapJaxbDataFormat("com.example.customerservice", new
ServiceInterfaceStrategy(CustomerService.class));
from("direct:start")
  .marshal(soap)
  .to("jms:myQueue");
```

TIP

See also As the SOAP dataformat inherits from the JAXB dataformat most settings apply here as well

291.3.1. Using SOAP 1.2

Available as of Camel 2.11

```
SoapJaxbDataFormat soap = new SoapJaxbDataFormat("com.example.customerservice", new
ServiceInterfaceStrategy(CustomerService.class));
soap.setVersion("1.2");
from("direct:start")
  .marshal(soap)
  .to("jms:myQueue");
```

When using XML DSL there is a version attribute you can set on the <soapjaxb> element.

```
    <!-- Defining a ServiceInterfaceStrategy for retrieving the element name when marshalling -->
    <bean id="myNameStrategy"
class="org.apache.camel.dataformat.soap.name.ServiceInterfaceStrategy">
        <constructor-arg value="com.example.customerservice.CustomerService"/>
    <constructor-arg value="true"/>
    </bean>
```

And in the Camel route

```
<route>
  <from uri="direct:start"/>
  <marshal>
    <soapjaxb contentPath="com.example.customerservice" version="1.2"
elementNameStrategyRef="myNameStrategy"/>
  </marshal>
  <to uri="jms:myQueue"/>
</route>
```

291.4. MULTI-PART MESSAGES

Available as of Camel 2.8.1

Multi-part SOAP messages are supported by the ServiceInterfaceStrategy. The ServiceInterfaceStrategy must be initialized with a service interface definition that is annotated in accordance with JAX-WS 2.2 and meets the requirements of the Document Bare style. The target method must meet the following criteria, as per the JAX-WS specification: 1) it must have at most one **in**

or **in**/**out** non-header parameter, 2) if it has a return type other than **void** it must have no **in**/**out** or **out** non-header parameters, 3) if it it has a return type of **void** it must have at most one **in**/**out** or **out** non-header parameter.

The ServiceInterfaceStrategy should be initialized with a boolean parameter that indicates whether the mapping strategy applies to the request parameters or response parameters.

```
ServiceInterfaceStrategy strat = new
ServiceInterfaceStrategy(com.example.customerservice.multipart.MultiPartCustomerService.class,
true);
SoapJaxbDataFormat soapDataFormat = new
SoapJaxbDataFormat("com.example.customerservice.multipart", strat);
```

291.4.1. Multi-part Request

The payload parameters for a multi-part request are initiazlied using a **BeanInvocation** object that reflects the signature of the target operation. The camel-soap DataFormat maps the content in the **BeanInvocation** to fields in the SOAP header and body in accordance with the JAX-WS mapping when the **marshal()** processor is invoked.

```
BeanInvocation beanInvocation = new BeanInvocation();

// Identify the target method
beanInvocation.setMethod(MultiPartCustomerService.class.getMethod("getCustomersByName",
    GetCustomersByName.class, com.example.customerservice.multipart.Product.class));

// Populate the method arguments
GetCustomersByName getCustomersByName = new GetCustomersByName();
getCustomersByName.setName("Dr. Multipart");

Product product = new Product();
product.setName("Multiuse Product");
product.setDescription("Useful for lots of things.");

Object[] args = new Object[] {getCustomersByName, product};

// Add the arguments to the bean invocation
beanInvocation.setArgs(args);

// Set the bean invocation object as the message body
exchange.getIn().setBody(beanInvocation);
```

291.4.2. Multi-part Response

A multi-part soap response may include an element in the soap body and will have one or more elements in the soap header. The camel-soap DataFormat will unmarshall the element in the soap body (if it exists) and place it onto the body of the out message in the exchange. Header elements will **not** be marshaled into their JAXB mapped object types. Instead, these elements are placed into the camel out message header **org.apache.camel.dataformat.soap.UNMARSHALLED_HEADER_LIST**. The elements will appear either as element instance values, or as JAXBElement values, depending upon the setting for the **ignoreJAXBElement** property. This property is inherited from camel-jaxb.

You can also have the camel-soap DataFormate ignore header content all-together by setting the **ignoreUnmarshalledHeaders** value to **true**.

291.4.3. Holder Object mapping

JAX-WS specifies the use of a type-parameterized **javax.xml.ws.Holder** object for **In**/**Out** and **Out** parameters. A **Holder** object may be used when building the **BeanInvocation**, or you may use an instance of the parameterized-type directly. The camel-soap DataFormat marshals Holder values in accordance with the JAXB mapping for the class of the **Holder's value. No mapping is provided for `Holder** objects in an unmarshalled response.

291.5. EXAMPLES

291.5.1. Webservice client

The following route supports marshalling the request and unmarshalling a response or fault.

```
String WS_URI = "cxf://http://myserver/customerservice?
serviceClass=com.example.customerservice&dataFormat=MESSAGE";
SoapJaxbDataFormat soapDF = new SoapJaxbDataFormat("com.example.customerservice", new
ServiceInterfaceStrategy(CustomerService.class));
from("direct:customerServiceClient")
  .onException(Exception.class)
    .handled(true)
    .unmarshal(soapDF)
  .end()
  .marshal(soapDF)
  .to(WS_URI)
  .unmarshal(soapDF);
```

The below snippet creates a proxy for the service interface and makes a SOAP call to the above route.

```
import org.apache.camel.Endpoint;
import org.apache.camel.component.bean.ProxyHelper;
...

Endpoint startEndpoint = context.getEndpoint("direct:customerServiceClient");
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
// CustomerService below is the service endpoint interface, *not* the javax.xml.ws.Service subclass
CustomerService proxy = ProxyHelper.createProxy(startEndpoint, classLoader,
CustomerService.class);
GetCustomersByNameResponse response = proxy.getCustomersByName(new
GetCustomersByName());
```

291.5.2. Webservice Server

Using the following route sets up a webservice server that listens on jms queue customerServiceQueue and processes requests using the class CustomerServiceImpl. The customerServiceImpl of course should implement the interface CustomerService. Instead of directly instantiating the server class it could be defined in a spring context as a regular bean.

```
SoapJaxbDataFormat soapDF = new SoapJaxbDataFormat("com.example.customerservice", new
ServiceInterfaceStrategy(CustomerService.class));
CustomerService serverBean = new CustomerServiceImpl();
from("jms://queue:customerServiceQueue")
  .onException(Exception.class)
```

```
    .handled(true)
    .marshal(soapDF)
  .end()
  .unmarshal(soapDF)
  .bean(serverBean)
  .marshal(soapDF);
```

291.6. DEPENDENCIES

To use the SOAP dataformat in your camel routes you need to add the following dependency to your pom.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-soap</artifactId>
  <version>2.3.0</version>
</dependency>
```

CHAPTER 292. SOLR COMPONENT

Available as of Camel version 2.9

The Solr component allows you to interface with an Apache Lucene Solr server (based on SolrJ 3.5.0).

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-solr</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

292.1. URI FORMAT

NOTE: solrs and solrCloud are new added since **Camel 2.14**.

```
solr://host[:port]/solr?[options]
solrs://host[:port]/solr?[options]
solrCloud://host[:port]/solr?[options]
```

292.2. SOLR OPTIONS

The Solr component has no options.

The Solr endpoint is configured using URI syntax:

```
solr:url
```

with the following path and query parameters:

292.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
url	**Required** Hostname and port for the solr server		String

292.2.2. Query Parameters (13 parameters):

Name	Description	Default	Type
allowCompression (producer)	Server side must support gzip or deflate for this to have any effect		Boolean

Name	Description	Default	Type
connectionTimeout (producer)	connectionTimeout on the underlying HttpConnectionManager		Integer
defaultMaxConnectionsPerHost (producer)	maxConnectionsPerHost on the underlying HttpConnectionManager		Integer
followRedirects (producer)	indicates whether redirects are used to get to the Solr server		Boolean
maxRetries (producer)	Maximum number of retries to attempt in the event of transient errors		Integer
maxTotalConnections (producer)	maxTotalConnection on the underlying HttpConnectionManager		Integer
requestHandler (producer)	Set the request handler to be used		String
soTimeout (producer)	Read timeout on the underlying HttpConnectionManager. This is desirable for queries, but probably not for indexing		Integer
streamingQueueSize (producer)	Set the queue size for the StreamingUpdateSolrServer	10	int
streamingThreadCount (producer)	Set the number of threads for the StreamingUpdateSolrServer	2	int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
collection (solrCloud)	Set the collection name which the solrCloud server could use		String
zkHost (solrCloud)	Set the ZooKeeper host information which the solrCloud could use, such as zkhost=localhost:8123.		String

292.3. MESSAGE OPERATIONS

The following Solr operations are currently supported. Simply set an exchange header with a key of "SolrOperation" and a value set to one of the following. Some operations also require the message body to be set.

- the INSERT operations use the CommonsHttpSolrServer

- the INSERT_STREAMING operations use the StreamingUpdateSolrServer (**Camel 2.9.2**)

Operation	Message body	Description
INSERT/INSERT_STREAMING	n/a	adds an index using message headers (must be prefixed with "SolrField.")
INSERT/INSERT_STREAMING	File	adds an index using the given File (using ContentStreamUpdateRequest)
INSERT/INSERT_STREAMING	SolrInputDocument	**Camel 2.9.2** updates index based on the given SolrInputDocument
INSERT/INSERT_STREAMING	String XML	**Camel 2.9.2** updates index based on the given XML (must follow SolrInputDocument format)
ADD_BEAN	bean instance	adds an index based on values in an annotated bean
ADD_BEANS	collection<bean>	**Camel 2.15** adds index based on a collection ofannotated bean
DELETE_BY_ID	index id to delete	delete a record by ID
DELETE_BY_QUERY	query string	delete a record by a query
COMMIT	n/a	performs a commit on any pending index changes
ROLLBACK	n/a	performs a rollback on any pending index changes

Operation	Message body	Description
OPTIMIZE	n/a	performs a commit on any pending index changes and then runs the optimize command

292.4. EXAMPLE

Below is a simple INSERT, DELETE and COMMIT example

```
from("direct:insert")
    .setHeader(SolrConstants.OPERATION, constant(SolrConstants.OPERATION_INSERT))
    .setHeader(SolrConstants.FIELD + "id", body())
    .to("solr://localhost:8983/solr");

from("direct:delete")
    .setHeader(SolrConstants.OPERATION, constant(SolrConstants.OPERATION_DELETE_BY_ID))
    .to("solr://localhost:8983/solr");

from("direct:commit")
    .setHeader(SolrConstants.OPERATION, constant(SolrConstants.OPERATION_COMMIT))
    .to("solr://localhost:8983/solr");
```

```
<route>
    <from uri="direct:insert"/>
    <setHeader headerName="SolrOperation">
        <constant>INSERT</constant>
    </setHeader>
    <setHeader headerName="SolrField.id">
        <simple>${body}</simple>
    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>
<route>
    <from uri="direct:delete"/>
    <setHeader headerName="SolrOperation">
        <constant>DELETE_BY_ID</constant>
    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>
<route>
    <from uri="direct:commit"/>
    <setHeader headerName="SolrOperation">
        <constant>COMMIT</constant>
    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>
```

A client would simply need to pass a body message to the insert or delete routes and then call the commit route.

```
    template.sendBody("direct:insert", "1234");
```

```
    template.sendBody("direct:commit", null);
    template.sendBody("direct:delete", "1234");
    template.sendBody("direct:commit", null);
```

292.5. QUERYING SOLR

Currently, this component doesn't support querying data natively (may be added later). For now, you can query Solr using HTTP as follows:

```
//define the route to perform a basic query
from("direct:query")
    .recipientList(simple("http://localhost:8983/solr/select/?q=${body}"))
    .convertBodyTo(String.class);
...
//query for an id of '1234' (url encoded)
String responseXml = (String) template.requestBody("direct:query", "id%3A1234");
```

For more information, see these resources...

Solr Query Tutorial

Solr Query Syntax

292.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

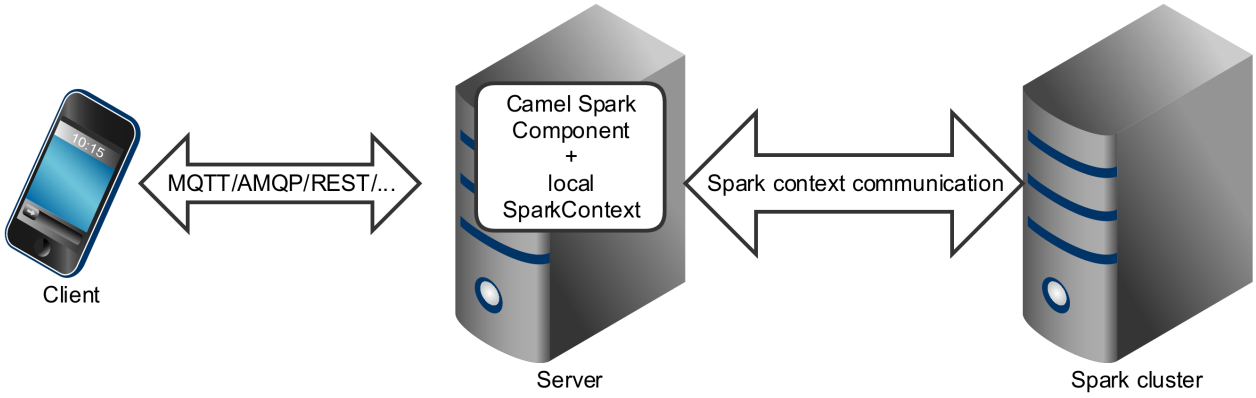
CHAPTER 293. APACHE SPARK COMPONENT

Available as of Camel version 2.17

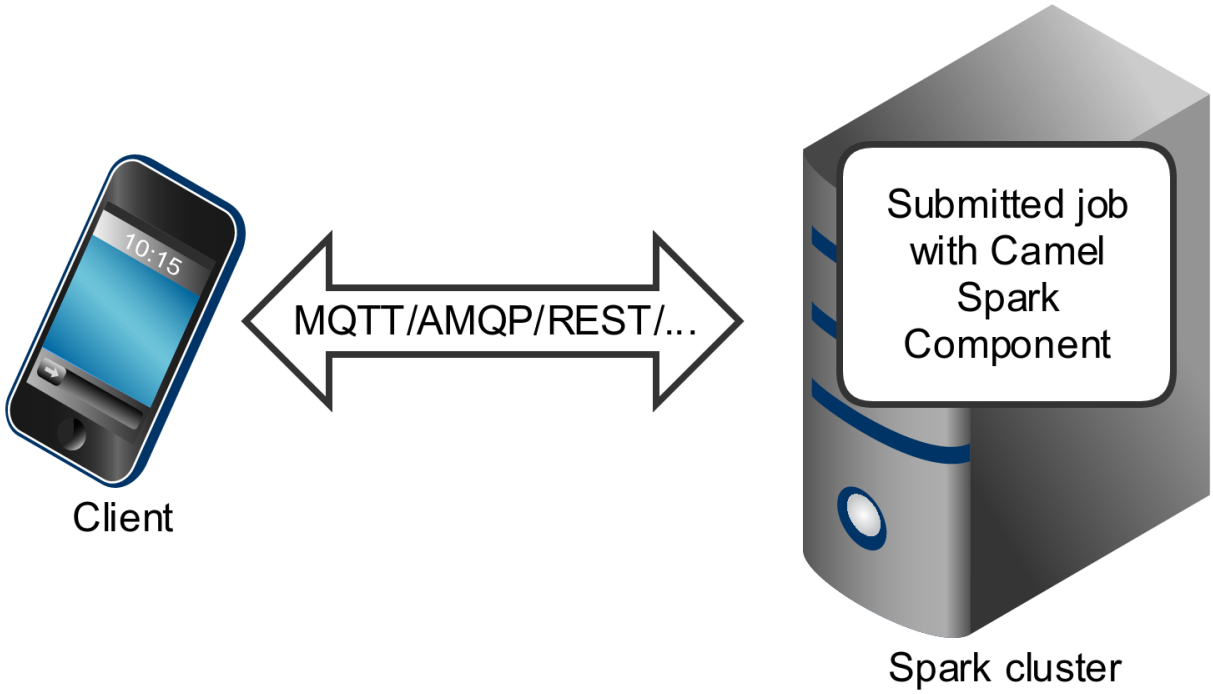
This documentation page covers the Apache Spark component for the Apache Camel. The main purpose of the Spark integration with Camel is to provide a bridge between Camel connectors and Spark tasks. In particular Camel connector provides a way to route message from various transports, dynamically choose a task to execute, use incoming message as input data for that task and finally deliver the results of the execution back to the Camel pipeline.

293.1. SUPPORTED ARCHITECTURAL STYLES

Spark component can be used as a driver application deployed into an application server (or executed as a fat jar).



Spark component can also be submitted as a job directly into the Spark cluster.



While Spark component is primary designed to work as a *long running job* serving as an bridge between Spark cluster and the other endpoints, you can also use it as a *fire-once* short job.

293.2. RUNNING SPARK IN OSGI SERVERS

Currently the Spark component doesn't support execution in the OSGi container. Spark has been designed to be executed as a fat jar, usually submitted as a job to a cluster. For those reasons running Spark in an OSGi server is at least challenging and is not support by Camel as well.

293.3. URI FORMAT

Currently the Spark component supports only producers - it it intended to invoke a Spark job and return results. You can call RDD, data frame or Hive SQL job.

Spark URI format

```
spark:{rdd|dataframe|hive}
```

293.3.1. Spark options

The Apache Spark component supports 3 options which are listed below.

Name	Description	Default	Type
rdd (producer)	RDD to compute against.		JavaRDDLike
rddCallback (producer)	Function performing action against an RDD.		RddCallback
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Apache Spark endpoint is configured using URI syntax:

```
spark:endpointType
```

with the following path and query parameters:

293.3.2. Path Parameters (1 parameters):

Name	Description	Default	Type
endpointType	**Required** Type of the endpoint (rdd, dataframe, hive).		EndpointType

293.3.3. Query Parameters (6 parameters):

Name	Description	Default	Type
collect (producer)	Indicates if results should be collected or counted.	true	boolean
dataFrame (producer)	DataFrame to compute against.		DataFrame
dataFrameCallback (producer)	Function performing action against an DataFrame.		DataFrameCallback
rdd (producer)	RDD to compute against.		JavaRDDLike
rddCallback (producer)	Function performing action against an RDD.		RddCallback
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

RDD jobs

To invoke an RDD job, use the following URI:

Spark RDD producer

```
spark:rdd?rdd=#testFileRdd&rddCallback=#transformation
```

Where **rdd** option refers to the name of an RDD instance (subclass of **org.apache.spark.api.java.JavaRDDLike**) from a Camel registry, while **rddCallback** refers to the implementation of **org.apache.camel.component.spark.RddCallback** interface (also from a registry). RDD callback provides a single method used to apply incoming messages against the given RDD. Results of callback computations are saved as a body to an exchange.

Spark RDD callback

```
public interface RddCallback<T> {
    T onRdd(JavaRDDLike rdd, Object... payloads);
}
```

The following snippet demonstrates how to send message as an input to the job and return results:

Calling spark job

```
String pattern = "job input";
long linesCount = producerTemplate.requestBody("spark:rdd?
rdd=#myRdd&rddCallback=#countLinesContaining", pattern, long.class);
```

The RDD callback for the snippet above registered as Spring bean could look as follows:

Spark RDD callback

-

```
@Bean
RddCallback<Long> countLinesContaining() {
   return new RddCallback<Long>() {
      Long onRdd(JavaRDDLike rdd, Object... payloads) {
         String pattern = (String) payloads[0];
         return rdd.filter({line -> line.contains(pattern)}).count();
      }
   }
}
```

The RDD definition in Spring could looks as follows:

Spark RDD definition

```
@Bean
JavaRDDLike myRdd(JavaSparkContext sparkContext) {
  return sparkContext.textFile("testrdd.txt");
}
```

293.3.4. Void RDD callbacks

If your RDD callback doesn't return any value back to a Camel pipeline, you can either return **null** value or use **VoidRddCallback** base class:

Spark RDD definition

```
@Bean
RddCallback<Void> rddCallback() {
  return new VoidRddCallback() {
      @Override
      public void doOnRdd(JavaRDDLike rdd, Object... payloads) {
         rdd.saveAsTextFile(output.getAbsolutePath());
      }
   };
}
```

293.3.5. Converting RDD callbacks

If you know what type of the input data will be sent to the RDD callback, you can use **ConvertingRddCallback** and let Camel to automatically convert incoming messages before inserting those into the callback:

Spark RDD definition

```
@Bean
RddCallback<Long> rddCallback(CamelContext context) {
  return new ConvertingRddCallback<Long>(context, int.class, int.class) {
         @Override
         public Long doOnRdd(JavaRDDLike rdd, Object... payloads) {
            return rdd.count() * (int) payloads[0] * (int) payloads[1];
         }
      };
   };
}
```

293.3.6. Annotated RDD callbacks

Probably the easiest way to work with the RDD callbacks is to provide class with method marked with **@RddCallback** annotation:

Annotated RDD callback definition

```
import static
org.apache.camel.component.spark.annotations.AnnotatedRddCallback.annotatedRddCallback;

@Bean
RddCallback<Long> rddCallback() {
   return annotatedRddCallback(new MyTransformation());
}

...

import org.apache.camel.component.spark.annotation.RddCallback;

public class MyTransformation {

   @RddCallback
   long countLines(JavaRDD<String> textFile, int first, int second) {
      return textFile.count() * first * second;
   }

}
```

If you will pass CamelContext to the annotated RDD callback factory method, the created callback will be able to convert incoming payloads to match the parameters of the annotated method:

Body conversions for annotated RDD callbacks

```
import static
org.apache.camel.component.spark.annotations.AnnotatedRddCallback.annotatedRddCallback;

@Bean
RddCallback<Long> rddCallback(CamelContext camelContext) {
   return annotatedRddCallback(new MyTransformation(), camelContext);
}

...


import org.apache.camel.component.spark.annotation.RddCallback;

public class MyTransformation {

   @RddCallback
   long countLines(JavaRDD<String> textFile, int first, int second) {
      return textFile.count() * first * second;
   }

}
```

```
...

// Convert String "10" to integer
long result = producerTemplate.requestBody("spark:rdd?rdd=#rdd&rddCallback=#rddCallback"
Arrays.asList(10, "10"), long.class);
```

293.4. DATAFRAME JOBS

Instead of working with RDDs Spark component can work with DataFrames as well.

To invoke an DataFrame job, use the following URI:

Spark RDD producer

```
spark:dataframe?dataFrame=#testDataFrame&dataFrameCallback=#transformation
```

Where **dataFrame** option refers to the name of an DataFrame instance (**instance of of org.apache.spark.sql.DataFrame**) from a Camel registry, while **dataFrameCallback** refers to the implementation of **org.apache.camel.component.spark.DataFrameCallback** interface (also from a registry). DataFrame callback provides a single method used to apply incoming messages against the given DataFrame. Results of callback computations are saved as a body to an exchange.

Spark RDD callback

```
public interface DataFrameCallback<T> {
    T onDataFrame(DataFrame dataFrame, Object... payloads);
}
```

The following snippet demonstrates how to send message as an input to a job and return results:

Calling spark job

```
String model = "Micra";
long linesCount = producerTemplate.requestBody("spark:dataFrame?
dataFrame=#cars&dataFrameCallback=#findCarWithModel", model, long.class);
```

The DataFrame callback for the snippet above registered as Spring bean could look as follows:

Spark RDD callback

```
@Bean
RddCallback<Long> findCarWithModel() {
    return new DataFrameCallback<Long>() {
        @Override
        public Long onDataFrame(DataFrame dataFrame, Object... payloads) {
            String model = (String) payloads[0];
            return dataFrame.where(dataFrame.col("model").eqNullSafe(model)).count();
        }
    };
}
```

The DataFrame definition in Spring could looks as follows:

Spark RDD definition

```
@Bean
DataFrame cars(HiveContext hiveContext) {
    DataFrame jsonCars = hiveContext.read().json("/var/data/cars.json");
    jsonCars.registerTempTable("cars");
    return jsonCars;
}
```

293.5. HIVE JOBS

Instead of working with RDDs or DataFrame Spark component can also receive Hive SQL queries as payloads. To send Hive query to Spark component, use the following URI:

Spark RDD producer

```
spark:hive
```

The following snippet demonstrates how to send message as an input to a job and return results:

Calling spark job

```
long carsCount = template.requestBody("spark:hive?collect=false", "SELECT * FROM cars",
Long.class);
List<Row> cars = template.requestBody("spark:hive", "SELECT * FROM cars", List.class);
```

The table we want to execute query against should be registered in a HiveContext before we query it. For example in Spring such registration could look as follows:

Spark RDD definition

```
@Bean
DataFrame cars(HiveContext hiveContext) {
    DataFrame jsonCars = hiveContext.read().json("/var/data/cars.json");
    jsonCars.registerTempTable("cars");
    return jsonCars;
}
```

293.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 294. SPARK REST COMPONENT

Available as of Camel version 2.14

The Spark-rest component allows to define REST endpoints using the Spark REST Java library using the Rest DSL.

INFO: Spark Java requires Java 8 runtime.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spark-rest</artifactId>
    <version>${camel-version}</version>
  </dependency>
```

294.1. URI FORMAT

```
spark-rest://verb:path?[options]
```

294.2. URI OPTIONS

The Spark Rest component supports 12 options which are listed below.

Name	Description	Default	Type
port (consumer)	Port number. Will by default use 4567	4567	int
ipAddress (consumer)	Set the IP address that Spark should listen on. If not called the default address is '0.0.0.0'.	0.0.0.0	String
minThreads (advanced)	Minimum number of threads in Spark thread-pool (shared globally)		int
maxThreads (advanced)	Maximum number of threads in Spark thread-pool (shared globally)		int
timeOutMillis (advanced)	Thread idle timeout in millis where threads that has been idle for a longer period will be terminated from the thread pool		int
keystoreFile (security)	Configures connection to be secure to use the keystore file		String
keystorePassword (security)	Configures connection to be secure to use the keystore password		String

Name	Description	Default	Type
truststoreFile (security)	Configures connection to be secure to use the truststore file		String
truststorePassword (security)	Configures connection to be secure to use the truststore password		String
sparkConfiguration (advanced)	To use the shared SparkConfiguration		SparkConfiguration
sparkBinding (advanced)	To use a custom SparkBinding to map to/from Camel message.		SparkBinding
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Spark Rest endpoint is configured using URI syntax:

```
spark-rest:verb:path
```

with the following path and query parameters:

294.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
verb	**Required** get, post, put, patch, delete, head, trace, connect, or options.		String
path	**Required** The content path which support Spark syntax.		String

294.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
accept (consumer)	Accept type such as: 'text/xml', or 'application/json'. By default we accept all kinds of types.		String

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
disableStreamCache (consumer)	Determines whether or not the raw input stream from Spark HttpRequestgetContent() is cached or not (Camel will read the stream into a in light-weight memory based Stream caching) cache. By default Camel will cache the Netty input stream to support reading it multiple times to ensure Camel can retrieve all data from the stream. However you can set this option to true when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. Mind that if you enable this option, then you cannot read the Netty stream multiple times out of the box, and you would need manually to reset the reader index on the Spark raw stream.	false	boolean
mapHeaders (consumer)	If this option is enabled, then during binding from Spark to Camel Message then the headers will be mapped as well (eg added as header to the Camel Message as well). You can turn off this option to disable this. The headers can still be accessed from the org.apache.camel.component.sparkrest.SparkMessage message with the method getRequest() that returns the Spark HTTP request instance.	true	boolean
transferException (consumer)	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	boolean
urlDecodeHeaders (consumer)	If this option is enabled, then during binding from Spark to Camel Message then the header values will be URL decoded (eg %20 will be a space character.)	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
matchOnUriPrefix (advanced)	Whether or not the consumer should try to find a target consumer by matching the URI prefix if no exact match is found.	false	boolean
sparkBinding (advanced)	To use a custom SparkBinding to map to/from Camel message.		SparkBinding
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

294.3. PATH USING SPARK SYNTAX

The path option is defined using a Spark REST syntax where you define the REST context path using support for parameters and splat. See more details at the Spark Java Route documentation.

The following is a Camel route using a fixed path

```
from("spark-rest:get:hello")
  .transform().constant("Bye World");
```

And the following route uses a parameter which is mapped to a Camel header with the key "me".

```
from("spark-rest:get:hello/:me")
  .transform().simple("Bye ${header.me}");
```

294.4. MAPPING TO CAMEL MESSAGE

The Spark Request object is mapped to a Camel Message as a **org.apache.camel.component.sparkrest.SparkMessage** which has access to the raw Spark request using the getRequest method. By default the Spark body is mapped to Camel message body, and any HTTP headers / Spark parameters is mapped to Camel Message headers. There is special support for the Spark splat syntax, which is mapped to the Camel message header with key splat.

For example the given route below uses Spark splat (the asterisk sign) in the context path which we can access as a header form the Simple language to construct a response message.

```
from("spark-rest:get:/hello/*/to/*")
  .transform().simple("Bye big ${header.splat[1]} from ${header.splat[0]}");
```

294.5. REST DSL

Apache Camel provides a new Rest DSL that allow to define the REST services in a nice REST style. For example we can define a REST hello service as shown below:

```
return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
         rest("/hello/{me}").get()
             .route().transform().simple("Bye ${header.me}");
     }
 };
```

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <rest uri="/hello/{me}">
    <get>
      <route>
        <transform>
          <simple>Bye ${header.me}</simple>
        </transform>
      </route>
    </get>
  </rest>
</camelContext>
```

See more details at the Rest DSL.

294.6. MORE EXAMPLES

There is a **camel-example-spark-rest-tomcat** example in the Apache Camel distribution, that demonstrates how to use camel-spark-rest in a web application that can be deployed on Apache Tomcat, or similar web containers.

CHAPTER 295. SPEL LANGUAGE

Available as of Camel version 2.7

Camel allows Spring Expression Language (SpEL) to be used as an Expression or Predicate in the DSL or XML Configuration.

NOTE

It is recommended to use SpEL in Spring runtimes. However from Camel 2.21 onwards you can use SpEL in other runtimes (there may be functionality SpEL cannot do when not running in a Spring runtime)

295.1. VARIABLES

The following variables are available in expressions and predicates written in SpEL:

Variable	Type	Description
this	Exchange	the Exchange is the root object
exchange	Exchange	the Exchange object
exception	Throwable	the Exchange exception (if any)
exchangeId	String	the exchange id
fault	Message	the Fault message (if any)
body	Object	The IN message body.
request	Message	the exchange.in message
response	Message	the exchange.out message (if any)
properties	Map	the exchange properties
property(name)	Object	the property by the given name

Variable	Type	Description
property(name, type)	Type	the property by the given name as the given type

295.2. OPTIONS

The SpEL language supports 1 options which are listed below.

Name	Default	Java Type	Description
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

295.3. SAMPLES

295.3.1. Expression templating

SpEL expressions need to be surrounded by **#{ }** delimiters since expression templating is enabled. This allows you to combine SpEL expressions with regular text and use this as extremely lightweight template language.

For example if you construct the following route:

```
from("direct:example")
    .setBody(spel("Hello #{request.body}! What a beautiful #{request.headers['dayOrNight']}"))
    .to("mock:result");
```

In the route above, notice spel is a static method which we need to import from **org.apache.camel.language.spel.SpelExpression.spel**, as we use spel as an Expression passed in as a parameter to the **setBody** method. Though if we use the fluent API we can do this instead:

```
from("direct:example")
    .setBody().spel("Hello #{request.body}! What a beautiful #{request.headers['dayOrNight']}")
    .to("mock:result");
```

Notice we now use the **spel** method from the **setBody()** method. And this does not require us to static import the spel method from **org.apache.camel.language.spel.SpelExpression.spel**.

And sent a message with the string "World" in the body, and a header "dayOrNight" with value "day":

```
template.sendBodyAndHeader("direct:example", "World", "dayOrNight", "day");
```

The output on **mock:result** will be *"Hello World! What a beautiful day"*

295.3.2. Bean integration

You can reference beans defined in the Registry (most likely an **ApplicationContext**) in your SpEL expressions. For example if you have a bean named "foo" in your **ApplicationContext** you can invoke the "bar" method on this bean like this:

```
#{@foo.bar == 'xyz'}
```

295.3.3. SpEL in enterprise integration patterns

You can use SpEL as an expression for Recipient List or as a predicate inside a Message Filter:

```
<route>
  <from uri="direct:foo"/>
  <filter>
    <spel>#{request.headers['foo'] == 'bar'}</spel>
    <to uri="direct:bar"/>
  </filter>
</route>
```

And the equivalent in Java DSL:

```
from("direct:foo")
    .filter().spel("#{request.headers['foo'] == 'bar'}")
    .to("direct:bar");
```

295.4. LOADING SCRIPT FROM EXTERNAL RESOURCE

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as **"classpath:"**, **"file:"**, or **"http:"**.
This is done using the following syntax: **"resource:scheme:location"**, eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").spel("resource:classpath:myspel.txt")
```

CHAPTER 296. SPLUNK COMPONENT

Available as of Camel version 2.13

The Splunk component provides access to Splunk using the Splunk provided client api, and it enables you to publish and search for events in Splunk.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-splunk</artifactId>
    <version>${camel-version}</version>
  </dependency>
```

296.1. URI FORMAT

```
splunk://[endpoint]?[options]
```

296.2. PRODUCER ENDPOINTS:

Endpoint	Description
stream	Streams data to a named index or the default if not specified. When using stream mode be aware of that Splunk has some internal buffer (about 1MB or so) before events gets to the index. If you need realtime, better use submit or tcp mode.
submit	submit mode. Uses Splunk rest api to publish events to a named index or the default if not specified.
tcp	tcp mode. Streams data to a tcp port, and requires a open receiver port in Splunk.

When publishing events the message body should contain a SplunkEvent. See comment under message body.

Example

```
    from("direct:start").convertBodyTo(SplunkEvent.class)
        .to("splunk://submit?
username=user&password=123&index=myindex&sourceType=someSourceType&source=mySource"
)...
```

In this example a converter is required to convert to a SplunkEvent class.

296.3. CONSUMER ENDPOINTS:

Endpoint	Description
normal	Performs normal search and requires a search query in the search option.
savedsearch	Performs search based on a search query saved in splunk and requires the name of the query in the savedSearch option.

Example

```
    from("splunk://normal?delay=5s&username=user&password=123&initEarliestTime=-10s&search=search index=myindex sourcetype=someSourcetype")
        .to("direct:search-result");
```

camel-splunk creates a route exchange per search result with a SplunkEvent in the body.

296.4. URI OPTIONS

The Splunk component supports 2 options which are listed below.

Name	Description	Default	Type
splunkConfigurationFactory (advanced)	To use the SplunkConfigurationFactory		SplunkConfigurationFactory
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Splunk endpoint is configured using URI syntax:

```
splunk:name
```

with the following path and query parameters:

296.4.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name has no purpose		String

296.4.2. Query Parameters (42 parameters):

Name	Description	Default	Type
app (common)	Splunk app		String
connectionTimeout (common)	Timeout in MS when connecting to Splunk server	5000	int
host (common)	Splunk host.	localhost	String
owner (common)	Splunk owner		String
port (common)	Splunk port	8089	int
scheme (common)	Splunk scheme	https	String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
count (consumer)	A number that indicates the maximum number of entities to return.		int
earliestTime (consumer)	Earliest time of the search time window.		String
initEarliestTime (consumer)	Initial start offset of the first search		String
latestTime (consumer)	Latest time of the search time window.		String
savedSearch (consumer)	The name of the query saved in Splunk to run		String
search (consumer)	The Splunk query to run		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean

Name	Description	Default	Type
streaming (consumer)	Sets streaming mode. Streaming mode sends exchanges as they are received, rather than in a batch.		Boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
eventHost (producer)	Override the default Splunk event host field		String
index (producer)	Splunk index to write to		String
raw (producer)	Should the payload be inserted raw	false	boolean
source (producer)	Splunk source argument		String
sourceType (producer)	Splunk sourcetype argument		String
tcpReceiverPort (producer)	Splunk tcp receiver port		int
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

Name	Description	Default	Type
password (security)	Password for Splunk		String
sslProtocol (security)	Set the ssl protocol to use	TLSv1.2	SSLSecurityProtocol
username (security)	Username for Splunk		String
useSunHttpsHandler (security)	Use sun.net.www.protocol.https.Handler Https handler to establish the Splunk Connection. Can be useful when running in application servers to avoid app. server https handling.	false	boolean

296.5. MESSAGE BODY

Splunk operates on data in key/value pairs. The SplunkEvent class is a placeholder for such data, and should be in the message body for the producer. Likewise it will be returned in the body per search result for the consumer.

As of Camel 2.16.0 you can send raw data to Splunk by setting the raw option on the producer endpoint. This is useful for eg. json/xml and other payloads where Splunk has build in support.

296.6. USE CASES

Search Twitter for tweets with music and publish events to Splunk

```
    from("twitter://search?
type=polling&keywords=music&delay=10&consumerKey=abc&consumerSecret=def&accessToken=hij&
ccessTokenSecret=xxx")
        .convertBodyTo(SplunkEvent.class)
        .to("splunk://submit?username=foo&password=bar&index=camel-
tweets&sourceType=twitter&source=music-tweets");
```

To convert a Tweet to a SplunkEvent you could use a converter like

```
@Converter
public class Tweet2SplunkEvent {
  @Converter
  public static SplunkEvent convertTweet(Status status) {
    SplunkEvent data = new SplunkEvent("twitter-message", null);
    //data.addPair("source", status.getSource());
    data.addPair("from_user", status.getUser().getScreenName());
    data.addPair("in_reply_to", status.getInReplyToScreenName());
    data.addPair(SplunkEvent.COMMON_START_TIME, status.getCreatedAt());
    data.addPair(SplunkEvent.COMMON_EVENT_ID, status.getId());
    data.addPair("text", status.getText());
    data.addPair("retweet_count", status.getRetweetCount());
    if (status.getPlace() != null) {
```

```
            data.addPair("place_country", status.getPlace().getCountry());
            data.addPair("place_name", status.getPlace().getName());
            data.addPair("place_street", status.getPlace().getStreetAddress());
        }
        if (status.getGeoLocation() != null) {
            data.addPair("geo_latitude", status.getGeoLocation().getLatitude());
            data.addPair("geo_longitude", status.getGeoLocation().getLongitude());
        }
        return data;
    }
}
```

Search Splunk for tweets

```
    from("splunk://normal?username=foo&password=bar&initEarliestTime=-2m&search=search
index=camel-tweets sourcetype=twitter")
        .log("${body}");
```

296.7. OTHER COMMENTS

Splunk comes with a variety of options for leveraging machine generated data with prebuilt apps for analyzing and displaying this.
For example the jmx app. could be used to publish jmx attributes, eg. route and jvm metrics to Splunk, and displaying this on a dashboard.

296.8. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 297. SPRING SUPPORT

Apache Camel is designed to work nicely with the Spring Framework in a number of ways.

- Camel uses Spring Transactions as the default transaction handling in components like JMS and JPA
- Camel works with Spring 2 XML processing with the Xml Configuration
- Camel Spring XML Schema's is defined at Xml Reference
- Camel supports a powerful version of Spring Remoting which can use powerful routing between the client and server side along with using all of the available Components for the transport
- Camel provides powerful Bean Integration with any bean defined in a Spring ApplicationContext
- Camel integrates with various Spring helper classes; such as providing Type Converter support for Spring Resources etc
- Allows Spring to dependency inject Component instances or the CamelContext instance itself and auto-expose Spring beans as components and endpoints.
- Allows you to reuse the Spring Testing framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's powerful Mock and Test endpoints
- From **Camel 2.15** onwards Camel supports Spring Boot using the **camel-spring-boot** component.

297.1. USING SPRING TO CONFIGURE THE CAMELCONTEXT

You can configure a CamelContext inside any spring.xml using the CamelContextFactoryBean. This will automatically start the CamelContext along with any referenced Routes along any referenced Component and Endpoint instances.

- Adding Camel schema
- Configure Routes in two ways:
  - Using Java Code
  - Using Spring XML

297.2. ADDING CAMEL SCHEMA

For Camel 1.x you need to use the following namespace:

```
http://activemq.apache.org/camel/schema/spring
```

with the following schema location:

```
http://activemq.apache.org/camel/schema/spring/camel-spring.xsd
```

You need to add Camel to the **schemaLocation** declaration

```
http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
```

So the XML file looks like this:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">
```

297.2.1. Using camel: namespace

Or you can refer to camel XSD in the XML declaration:

```
xmlns:camel="http://camel.apache.org/schema/spring"
```

i. so the declaration is:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:camel="http://camel.apache.org/schema/spring"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">
```

i. and then use the camel: namespace prefix, and you can omit the inline namespace declaration:

```
<camel:camelContext id="camel5">
  <camel:package>org.apache.camel.spring.example</camel:package>
</camel:camelContext>
```

297.2.2. Advanced configuration using Spring

See more details at Advanced configuration of CamelContext using Spring

$# Using Java Code

You can use Java Code to define your RouteBuilder implementations. These can be defined as beans in spring and then referenced in your camel context e.g.

297.2.3. Using <package>

Camel also provides a powerful feature that allows for the automatic discovery and initialization of routes in given packages. This is configured by adding tags to the camel context in your spring context definition, specifying the packages to be recursively searched for RouteBuilder implementations. To use this feature in 1.X, requires a <package></package> tag specifying a comma separated list of packages that should be searched e.g.

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <package>org.apache.camel.spring.config.scan.route</package>
</camelContext>
```

WARNING:Use caution when specifying the package name as **org.apache.camel** or a sub package of this. This causes Camel to search in its own packages for your routes which could cause problems.

INFO:*Will ignore already instantiated classes*. The <package> and <packageScan> will skip any classes which has already been created by Spring etc. So if you define a route builder as a spring bean tag then that class will be skipped. You can include those beans using **<routeBuilder ref="theBeanId"/>** or the **<contextScan>** feature.

297.2.4. Using <packageScan>

In Camel 2.0 this has been extended to allow selective inclusion and exclusion of discovered route classes using Ant like path matching. In spring this is specified by adding a <packageScan/> tag. The tag must contain one or more 'package' elements (similar to 1.x), and optionally one or more 'includes' or 'excludes' elements specifying patterns to be applied to the fully qualified names of the discovered classes. e.g.

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>org.example.routes</package>
    <excludes>**.*Excluded*</excludes>
    <includes>**.*</includes>
  </packageScan>
</camelContext>
```

Exclude patterns are applied before the include patterns. If no include or exclude patterns are defined then all the Route classes discovered in the packages will be returned.

In the above example, camel will scan all the 'org.example.routes' package and any subpackages for RouteBuilder classes. Say the scan finds two RouteBuilders, one in org.example.routes called 'MyRoute" and another 'MyExcludedRoute' in a subpackage 'excluded'. The fully qualified names of each of the classes are extracted (org.example.routes.MyRoute, org.example.routes.excluded.MyExcludedRoute) and the include and exclude patterns are applied.

The exclude pattern ***.*Excluded** is going to match the fqcn 'org.example.routes.excluded.MyExcludedRoute' and veto camel from initializing it.

Under the covers, this is using Spring's AntPatternMatcher implementation, which matches as follows

```
? matches one character
* matches zero or more characters
** matches zero or more segments of a fully qualified name
```

For example:

***.*Excluded** would match org.simple.Excluded, org.apache.camel.SomeExcludedRoute or org.example.RouteWhichIsExcluded

***.??cluded** would match org.simple.IncludedRoute, org.simple.Excluded but not match org.simple.PrecludedRoute

297.2.5. Using contextScan

Available as of Camel 2.4

You can allow Camel to scan the container context, e.g. the Spring **ApplicationContext** for route builder instances. This allow you to use the Spring **<component-scan>** feature and have Camel pickup any RouteBuilder instances which was created by Spring in its scan process.

This allows you to just annotate your routes using the Spring **@Component** and have those routes included by Camel

```
@Component
public class MyRoute extends SpringRouteBuilder {

    @Override
    public void configure() throws Exception {
        from("direct:start").to("mock:result");
    }
}
```

You can also use the ANT style for inclusion and exclusion, as mentioned above in the **<packageScan>** documentation.

297.3. HOW DO I IMPORT ROUTES FROM OTHER XML FILES

Available as of Camel 2.3

When defining routes in Camel using Xml Configuration you may want to define some routes in other XML files. For example you may have many routes and it may help to maintain the application if some of the routes are in separate XML files. You may also want to store common and reusable routes in other XML files, which you can simply import when needed.

In **Camel 2.3** it is now possible to define routes outside **<camelContext/>** which you do in a new **<routeContext/>** tag.

Notice: When you use <routeContext> then they are separated, and cannot reuse existing <onException>, <intercept>, <dataFormats> and similar cross cutting functionality defined in the <camelContext>. In other words the <routeContext> is currently isolated. This may change in Camel 3.x.

For example we could have a file named **myCoolRoutes.xml** which contains a couple of routes as shown:

myCoolRoutes.xml

Then in your XML file which contains the CamelContext you can use Spring to import the **myCoolRoute.xml** file.
And then inside **<camelContext/>** you can refer to the **<routeContext/>** by its id as shown below:

Also notice that you can mix and match, having routes inside CamelContext and also externalized in RouteContext.

You can have as many **<routeContextRef/>** as you like.

Reusable routes

The routes defined in **<routeContext/>** can be reused by multiple **<camelContext/>**. However its only the definition which is reused. At runtime each CamelContext will create its own instance of the route based on the definition.

297.3.1. Test time exclusion.

At test time it is often desirable to be able to selectively exclude matching routes from being initalized that are not applicable or useful to the test scenario. For instance you might a spring context file routes-context.xml and three Route builders RouteA, RouteB and RouteC in the 'org.example.routes' package. The packageScan definition would discover all three of these routes and initialize them.

Say RouteC is not applicable to our test scenario and generates a lot of noise during test. It would be nice to be able to exclude this route from this specific test. The SpringTestSupport class has been modified to allow this. It provides two methods (excludedRoute and excludedRoutes) that may be overridden to exclude a single class or an array of classes.

```
public class RouteAandRouteBOnlyTest extends SpringTestSupport {
    @Override
    protected Class excludeRoute() {
        return RouteC.class;
    }
}
```

In order to hook into the camelContext initialization by spring to exclude the MyExcludedRouteBuilder.class we need to intercept the spring context creation. When overriding createApplicationContext to create the spring context, we call the getRouteExcludingApplicationContext() method to provide a special parent spring context that takes care of the exclusion.

```
@Override
protected AbstractXmlApplicationContext createApplicationContext() {
    return new ClassPathXmlApplicationContext(new String[] {"routes-context.xml"},
getRouteExcludingApplicationContext());
}
```

RouteC will now be excluded from initialization. Similarly, in another test that is testing only RouteC, we could exclude RouteB and RouteA by overriding

```
@Override
protected Class[] excludeRoutes() {
    return new Class[]{RouteA.class, RouteB.class};
}
```

297.4. USING SPRING XML

You can use Spring 2.0 XML configuration to specify your Xml Configuration for Routes such as in the following example.

297.5. CONFIGURING COMPONENTS AND ENDPOINTS

You can configure your Component or Endpoint instances in your Spring XML as follows in this example.

Which allows you to configure a component using some name (activemq in the above example), then

you can refer to the component using **activemq:[queue:|topic:]destinationName**. This works by the SpringCamelContext lazily fetching components from the spring context for the scheme name you use for Endpoint URIs.

For more detail see Configuring Endpoints and Components.

297.6. CAMELCONTEXTAWARE

If you want to be injected with the CamelContext in your POJO just implement the CamelContextAware interface; then when Spring creates your POJO the CamelContext will be injected into your POJO. Also see the Bean Integration for further injections.

297.7. INTEGRATION TESTING

To avoid a hung route when testing using Spring Transactions see the note about Spring Integration Testing under Transactional Client.

297.8. SEE ALSO

- Spring JMS Tutorial
- Creating a new Spring based Camel Route
- Spring example
- Xml Reference
- Advanced configuration of CamelContext using Spring
- How do I import routes from other XML files

CHAPTER 298. SPRING BATCH COMPONENT

Available as of Camel version 2.10

The **spring-batch:** component and support classes provide integration bridge between Camel and Spring Batch infrastructure.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-batch</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

298.1. URI FORMAT

```
spring-batch:jobName[?options]
```

Where **jobName** represents the name of the Spring Batch job located in the Camel registry. Alternatively if a JobRegistry is provided it will be used to locate the job instead.

WARNING:This component can only be used to define producer endpoints, which means that you cannot use the Spring Batch component in a **from()** statement.

298.2. OPTIONS

The Spring Batch component supports 3 options which are listed below.

Name	Description	Default	Type
jobLauncher (producer)	Explicitly specifies a JobLauncher to be used.		JobLauncher
jobRegistry (producer)	Explicitly specifies a JobRegistry to be used.		JobRegistry
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Spring Batch endpoint is configured using URI syntax:

```
spring-batch:jobName
```

with the following path and query parameters:

298.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
jobName	**Required** The name of the Spring Batch job located in the registry.		String

298.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
jobFromHeader (producer)	Explicitly defines if the jobName should be taken from the headers instead of the URI.	false	boolean
jobLauncher (producer)	Explicitly specifies a JobLauncher to be used.		JobLauncher
jobRegistry (producer)	Explicitly specifies a JobRegistry to be used.		JobRegistry
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

298.3. USAGE

When Spring Batch component receives the message, it triggers the job execution. The job will be executed using the **org.springframework.batch.core.launch.JobLaucher** instance resolved according to the following algorithm:

- if **JobLauncher** is manually set on the component, then use it.
- if **jobLauncherRef** option is set on the component, then search Camel Registry for the **JobLauncher** with the given name. **Deprecated and will be removed in Camel 3.0!**
- if there is **JobLauncher** registered in the Camel Registry under **jobLauncher** name, then use it.
- if none of the steps above allow to resolve the **JobLauncher** and there is exactly one **JobLauncher** instance in the Camel Registry, then use it.

All headers found in the message are passed to the **JobLauncher** as job parameters. **String**, **Long**, **Double** and **java.util.Date** values are copied to the **org.springframework.batch.core.JobParametersBuilder** - other data types are converted to Strings.

298.4. EXAMPLES

Triggering the Spring Batch job execution:

```
from("direct:startBatch").to("spring-batch:myJob");
```

Triggering the Spring Batch job execution with the **JobLauncher** set explicitly.

```
from("direct:startBatch").to("spring-batch:myJob?jobLauncherRef=myJobLauncher");
```

Starting from the Camel **2.11.1** **JobExecution** instance returned by the **JobLauncher** is forwarded by the **SpringBatchProducer** as the output message. You can use the **JobExecution** instance to perform some operations using the Spring Batch API directly.

```
from("direct:startBatch").to("spring-batch:myJob").to("mock:JobExecutions");
...
MockEndpoint mockEndpoint = ...;
JobExecution jobExecution =
mockEndpoint.getExchanges().get(0).getIn().getBody(JobExecution.class);
BatchStatus currentJobStatus = jobExecution.getStatus();
```

298.5. SUPPORT CLASSES

Apart from the Component, Camel Spring Batch provides also support classes, which can be used to hook into Spring Batch infrastructure.

298.5.1. CamelItemReader

CamelItemReader can be used to read batch data directly from the Camel infrastructure.

For example the snippet below configures Spring Batch to read data from JMS queue.

```
<bean id="camelReader"
class="org.apache.camel.component.spring.batch.support.CamelItemReader">
  <constructor-arg ref="consumerTemplate"/>
  <constructor-arg value="jms:dataQueue"/>
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="camelReader" writer="someWriter" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

298.5.2. CamelItemWriter

CamelItemWriter has similar purpose as **CamelItemReader**, but it is dedicated to write chunk of the processed data.

For example the snippet below configures Spring Batch to read data from JMS queue.

```
<bean id="camelwriter" class="org.apache.camel.component.spring.batch.support.CamelItemWriter">
  <constructor-arg ref="producerTemplate"/>
  <constructor-arg value="jms:dataQueue"/>
</bean>
```

```
<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="someReader" writer="camelwriter" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

298.5.3. CamelItemProcessor

CamelItemProcessor is the implementation of Spring Batch **org.springframework.batch.item.ItemProcessor** interface. The latter implementation relays on Request Reply pattern to delegate the processing of the batch item to the Camel infrastructure. The item to process is sent to the Camel endpoint as the body of the message.

For example the snippet below performs simple processing of the batch item using the Direct endpoint and the Simple expression language.

```
<camel:camelContext>
  <camel:route>
    <camel:from uri="direct:processor"/>
    <camel:setExchangePattern pattern="InOut"/>
    <camel:setBody>
      <camel:simple>Processed ${body}</camel:simple>
    </camel:setBody>
  </camel:route>
</camel:camelContext>

<bean id="camelProcessor"
class="org.apache.camel.component.spring.batch.support.CamelItemProcessor">
  <constructor-arg ref="producerTemplate"/>
  <constructor-arg value="direct:processor"/>
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="someReader" writer="someWriter" processor="camelProcessor" commit-
interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

298.5.4. CamelJobExecutionListener

CamelJobExecutionListener is the implementation of the **org.springframework.batch.core.JobExecutionListener** interface sending job execution events to the Camel endpoint.

The **org.springframework.batch.core.JobExecution** instance produced by the Spring Batch is sent as a body of the message. To distinguish between before- and after-callbacks **SPRING_BATCH_JOB_EVENT_TYPE** header is set to the **BEFORE** or **AFTER** value.

The example snippet below sends Spring Batch job execution events to the JMS queue.

```
<bean id="camelJobExecutionListener"
class="org.apache.camel.component.spring.batch.support.CamelJobExecutionListener">
  <constructor-arg ref="producerTemplate"/>
  <constructor-arg value="jms:batchEventsBus"/>
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="someReader" writer="someWriter" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
  <batch:listeners>
    <batch:listener ref="camelJobExecutionListener"/>
  </batch:listeners>
</batch:job>
```

298.6. SPRING CLOUD

Available as of Camel 2.19

Spring Cloud component

Maven users will need to add the following dependency to their **pom.xml** in order to use this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-cloud</artifactId>
    <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>
```

camel-spring-cloud jar comes with the **spring.factories** file, so as soon as you add that dependency into your classpath, Spring Boot will automatically auto-configure Camel for you.

298.6.1. Camel Spring Cloud Starter

Available as of Camel 2.19

To use the starter, add the following to your spring boot pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-cloud-starter</artifactId>
    <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>
```

298.7. SPRING CLOUD NETFLIX

Available as of Camel 2.19

The Spring Cloud Netflix component bridges Camel Cloud and Spring Cloud Netflix so you can leverage Spring Cloud Netflix service discovery and load balance features in Camel and/or you can use Camel

Service Discovery implementations as ServerList source for Spring Cloud Netflix's Ribbon load balabncer.

Maven users will need to add the following dependency to their **pom.xml** in order to use this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-cloud-netflix</artifactId>
  <version>${camel.version}</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

camel-spring-cloud-netflix jar comes with the **spring.factories** file, so as soon as you add that dependency into your classpath, Spring Boot will automatically auto-configure Camel for you.

You can disable Camel Spring Cloud Netflix with the following properties:

```
# Enable/Disable the whole integration, default true
camel.cloud.netflix = true

# Enable/Disable the integration with Ribbon, default true
camel.cloud.netflix.ribbon = true
```

298.8. SPRING CLOUD NETFLIX STARTER

Available as of Camel 2.19

To use the starter, add the following to your spring boot pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-cloud-netflix-starter</artifactId>
  <version>${camel.version}</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 299. SPRING EVENT COMPONENT

Available as of Camel version 1.4

The **spring-event:** component provides access to the Spring **ApplicationEvent** objects. This allows you to publish **ApplicationEvent** objects to a Spring **ApplicationContext** or to consume them. You can then use Enterprise Integration Patterns to process them such as Message Filter.

299.1. URI FORMAT

```
spring-event://default[?options]
```

Note, at the moment there are no options for this component. That can easily change in future releases, so please check back.

299.2. SPRING EVENT OPTIONS

The Spring Event component has no options.

The Spring Event endpoint is configured using URI syntax:

```
spring-event:name
```

with the following path and query parameters:

299.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	Name of endpoint		String

299.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

299.3. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 300. SPRING INTEGRATION COMPONENT

Available as of Camel version 1.4

The **spring-integration:** component provides a bridge for Camel components to talk to spring integration endpoints.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-integration</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

300.1. URI FORMAT

```
spring-integration:defaultChannelName[?options]
```

Where **defaultChannelName** represents the default channel name which is used by the Spring Integration Spring context. It will equal to the **inputChannel** name for the Spring Integration consumer and the **outputChannel** name for the Spring Integration provider.

You can append query options to the URI in the following format, **?option=value&option=value&…**

300.2. OPTIONS

The Spring Integration component has no options.

The Spring Integration endpoint is configured using URI syntax:

```
spring-integration:defaultChannel
```

with the following path and query parameters:

300.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
defaultChannel	**Required** The default channel name which is used by the Spring Integration Spring context. It will equal to the inputChannel name for the Spring Integration consumer and the outputChannel name for the Spring Integration provider.		String

300.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
inOut (common)	The exchange pattern that the Spring integration endpoint should use. If inOut=true then a reply channel is expected, either from the Spring Integration Message header or configured on the endpoint.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
inputChannel (consumer)	The Spring integration input channel name that this endpoint wants to consume from Spring integration.		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
outputChannel (producer)	The Spring integration output channel name that is used to send messages to Spring integration.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

300.3. USAGE

The Spring integration component is a bridge that connects Camel endpoints with Spring integration endpoints through the Spring integration's input channels and output channels. Using this component, we can send Camel messages to Spring Integration endpoints or receive messages from Spring integration endpoints in a Camel routing context.

300.4. EXAMPLES

300.4.1. Using the Spring integration endpoint

You can set up a Spring integration endpoint using a URI, as follows:

Or directly using a Spring integration channel name:

300.4.2. The Source and Target adapter

Spring integration also provides the Spring integration's source and target adapters, which can route messages from a Spring integration channel to a Camel endpoint or from a Camel endpoint to a Spring integration channel.

This example uses the following namespaces:

You can bind your source or target to a Camel endpoint as follows:

300.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

300.6. SPRING JAVA CONFIG

Spring started life using XML Config to wire beans together. However some folks don't like using XML and would rather use Java code which led to the creation of Guice along with the Spring JavaConfig project.

You can use either the XML or Java config approaches with Camel; its your choice really on which you prefer.

300.6.1. Using Spring Java Config

To use Spring Java Config in your Camel project the easiest thing to do is add the following to your pom.xml

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-javaconfig</artifactId>
  <version>${camel-version}</version>
</dependency>
```

This will then add the dependencies on the Spring JavaConfig library along with some helper classes for configuring Camel inside Spring.

Note that this library is totally optional; you could just wire Camel together yourself with Java Config.

300.6.2. Configuration

The most common case of using JavaConfig with Camel would be to create configuration with defined list of routes to be used by router.

```
@Configuration
public class MyRouteConfiguration extends CamelConfiguration {
```

```
    @Autowire
    private MyRouteBuilder myRouteBuilder;

    @Autowire
    private MyAnotherRouteBuilder myAnotherRouteBuilder;

    @Override
    public List<RouteBuilder> routes() {
        return Arrays.asList(myRouteBuilder, myAnotherRouteBuilder);
    }

}
```

Starting from Camel 2.13.0 you can skip the **routes()** definition, and fall back to the **RouteBuilder** instances located in the Spring context.

```
@Configuration
@ComponentScan("com.example.routes")
public class MyRouteConfiguration extends CamelConfiguration {
}
```

300.6.3. Testing

Since **Camel 2.11.0** you can use the **CamelSpringJUnit4ClassRunner** with **CamelSpringDelegatingTestContextLoader**. This is the recommended way to test Java Config and Camel integration.

If you wish to create a collection of **RouteBuilder** instances then derive from the **CamelConfiguration** helper class and implement the **routes()** method. Keep in mind that (starting from the Camel 2.13.0) if you don't override **routes()** method, then **CamelConfiguration** will use all **RouteBuilder** instances available in the Spring context.

The following example using Java Config demonstrates how to test Java Config integration with Camel 2.10 and lower. Keep in mind that **JavaConfigContextLoader** is deprecated and could be removed in the future versions of Camel on the behalf of the **CamelSpringDelegatingTestContextLoader**.

The **@ContextConfiguration** annotation tells the Spring Testing framework to load the **ContextConfig** class as the configuration to use. This class derives from **SingleRouteCamelConfiguration** which is a helper Spring Java Config class which will configure the CamelContext for us and then register the RouteBuilder we create.

CHAPTER 301. SPRING LDAP COMPONENT

Available as of Camel version 2.11

The **spring-ldap:** component provides a Camel wrapper for Spring LDAP.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-ldap</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

301.1. URI FORMAT

```
spring-ldap:springLdapTemplate[?options]
```

Where **springLdapTemplate** is the name of the Spring LDAP Template bean. In this bean, you configure the URL and the credentials for your LDAP access.

301.2. OPTIONS

The Spring LDAP component has no options.

The Spring LDAP endpoint is configured using URI syntax:

```
spring-ldap:templateName
```

with the following path and query parameters:

301.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
templateName	**Required** Name of the Spring LDAP Template bean		String

301.2.2. Query Parameters (3 parameters):

Name	Description	Default	Type
operation (producer)	**Required** The LDAP operation to be performed.		LdapOperation
scope (producer)	The scope of the search operation.	subtree	String

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

301.3. USAGE

The component supports producer endpoint only. An attempt to create a consumer endpoint will result in an **UnsupportedOperationException**.
The body of the message must be a map (an instance of **java.util.Map**). Unless a base DN is specified by in the configuration of your ContextSource, this map must contain at least an entry with the key **dn** (not needed for function_driven operation) that specifies the root node for the LDAP operation to be performed. Other entries of the map are operation-specific (see below).

The body of the message remains unchanged for the **bind** and **unbind** operations. For the **search** and **function_driven** operations, the body is set to the result of the search, see http://static.springsource.org/spring-ldap/site/apidocs/org/springframework/ldap/core/LdapTemplate.html#search%28java.lang.String,%20ja

301.3.1. Search

The message body must have an entry with the key **filter**. The value must be a **String** representing a valid LDAP filter, see http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare.

301.3.2. Bind

The message body must have an entry with the key **attributes**. The value must be an instance of javax.naming.directory.Attributes This entry specifies the LDAP node to be created.

301.3.3. Unbind

No further entries necessary, the node with the specified **dn** is deleted.

301.3.4. Authenticate

The message body must have entries with the keys **filter** and **password**. The values must be an instance of **String** representing a valid LDAP filter and a user password, respectively.

301.3.5. Modify Attributes

The message body must have an entry with the key **modificationItems**. The value must be an instance of any array of type javax.naming.directory.ModificationItem

301.3.6. Function-Driven

The message body must have entries with the keys **function** and **request**. The **function** value must be of type **java.util.function.BiFunction<L, Q, S>**. The **L** type parameter must be of type **org.springframework.ldap.core.LdapOperations**. The **request** value must be the same type as the **Q** type parameter in the **function** and it must encapsulate the parameters expected by the LdapTemplate

method being invoked within the **function**. The **S** type parameter represents the response type as returned by the LdapTemplate method being invoked. This operation allows dynamic invocation of LdapTemplate methods that are not covered by the operations mentioned above.

Key definitions

In order to avoid spelling errors, the following constants are defined in **org.apache.camel.springldap.SpringLdapProducer**:

- public static final String DN = "dn"
- public static final String FILTER = "filter"
- public static final String ATTRIBUTES = "attributes"
- public static final String PASSWORD = "password";
- public static final String MODIFICATION_ITEMS = "modificationItems";
- public static final String FUNCTION = "function";
- public static final String REQUEST = "request";

CHAPTER 302. SPRING REDIS COMPONENT

Available as of Camel version 2.11

This component allows sending and receiving messages from Redis. Redis is advanced key-value store where keys can contain strings, hashes, lists, sets and sorted sets. In addition it provides pub/sub functionality for inter-app communications.
Camel provides a producer for executing commands, consumer for subscribing to pub/sub messages an idempotent repository for filtering out duplicate messages.

INFO:*Prerequisites* In order to use this component, you must have a Redis server running.

302.1. URI FORMAT

```
spring-redis://host:port[?options]
```

You can append query options to the URI in the following format, **?options=value&option2=value&…**

302.2. URI OPTIONS

The Spring Redis component has no options.

The Spring Redis endpoint is configured using URI syntax:

```
spring-redis:host:port
```

with the following path and query parameters:

302.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** The host where Redis server is running.		String
port	**Required** Redis server port number		Integer

302.2.2. Query Parameters (10 parameters):

Name	Description	Default	Type
channels (common)	List of topic names or name patterns to subscribe to. Multiple names can be separated by comma.		String
command (common)	Default command, which can be overridden by message header. Notice the consumer only supports the following commands: PSUBSCRIBE and SUBSCRIBE	SET	Command

Name	Description	Default	Type
connectionFactory (common)	Reference to a pre-configured RedisConnectionFactory instance to use.		RedisConnectionFactory
redisTemplate (common)	Reference to a pre-configured RedisTemplate instance to use.		RedisTemplate
serializer (common)	Reference to a pre-configured RedisSerializer instance to use.		RedisSerializer
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
listenerContainer (consumer)	Reference to a pre-configured RedisMessageListenerContainer instance to use.		RedisMessageListener Container
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

302.3. USAGE

See also the unit tests available at https://github.com/apache/camel/tree/master/components/camel-spring-redis/src/test/java/org/apache/camel/component/redis.

302.3.1. Message headers evaluated by the Redis producer

The producer issues commands to the server and each command has different set of parameters with specific types. The result from the command execution is returned in the message body.

Hash Commands	Description	Parameters	Result
HSET	Set the string value of a hash field	CamelRedis.Key (String), CamelRedis.Field (String), CamelRedis.Value (Object)	void
HGET	Get the value of a hash field	CamelRedis.Key (String), CamelRedis.Field (String)	String
HSETNX	Set the value of a hash field, only if the field does not exist	CamelRedis.Key (String), CamelRedis.Field (String), CamelRedis.Value (Object)	void
HMSET	Set multiple hash fields to multiple values	CamelRedis.Key (String), CamelRedis.Values(Map<String, Object>)	void
HMGET	Get the values of all the given hash fields	CamelRedis.Key (String), CamelRedis.Fields (Collection<String>)	Collection<Object>
HINCRBY	Increment the integer value of a hash field by the given number	CamelRedis.Key (String), CamelRedis.Field (String), CamelRedis.Value (Long)	Long
HEXISTS	Determine if a hash field exists	CamelRedis.Key (String), CamelRedis.Field (String)	Boolean
HDEL	Delete one or more hash fields	CamelRedis.Key (String), CamelRedis.Field (String)	void

Hash Commands	Description	Parameters	Result
HLEN	Get the number of fields in a hash	CamelRedis.Key (String)	Long
HKEYS	Get all the fields in a hash	CamelRedis.Key (String)	Set<String>
HVALS	Get all the values in a hash	CamelRedis.Key (String)	Collection<Object>
HGETALL	Get all the fields and values in a hash	CamelRedis.Key (String)	Map<String, Object>

List Commands	Description	Parameters	Result
RPUSH	Append one or multiple values to a list	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
RPUSHX	Append a value to a list, only if the list exists	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
LPUSH	Prepend one or multiple values to a list	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
LLEN	Get the length of a list	CamelRedis.Key (String)	Long
LRANGE	Get a range of elements from a list	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	List<Object>

List Commands	Description	Parameters	Result
LTRIM	Trim a list to the specified range	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	void
LINDEX	Get an element from a list by its index	CamelRedis.Key (String), CamelRedis.Index (Long)	String
LINSERT	Insert an element before or after another element in a list	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Pivot (String), CamelRedis.Position (String)	Long
LSET	Set the value of an element in a list by its index	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Index (Long)	void
LREM	Remove elements from a list	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Count (Long)	Long
LPOP	Remove and get the first element in a list	CamelRedis.Key (String)	Object
RPOP	Remove and get the last element in a list	CamelRedis.Key (String)	String

List Commands	Description	Parameters	Result
RPOPLPUSH	Remove the last element in a list, append it to another list and return it	CamelRedis.Key (String), CamelRedis.Destination (String)	Object
BRPOPLPUSH	Pop a value from a list, push it to another list and return it; or block until one is available	CamelRedis.Key (String), CamelRedis.Destination (String), CamelRedis.Timeout (Long)	Object
BLPOP	Remove and get the first element in a list, or block until one is available	CamelRedis.Key (String), CamelRedis.Timeout (Long)	Object
BRPOP	Remove and get the last element in a list, or block until one is available	CamelRedis.Key (String), CamelRedis.Timeout (Long)	String

Set Commands	Description	Parameters	Result
SADD	Add one or more members to a set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean
SMEMBERS	Get all the members in a set	CamelRedis.Key (String)	Set<Object>
SREM	Remove one or more members from a set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean

Set Commands	Description	Parameters	Result
SPOP	Remove and return a random member from a set	CamelRedis.Key (String)	String
SMOVE	Move a member from one set to another	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Destination (String)	Boolean
SCARD	Get the number of members in a set	CamelRedis.Key (String)	Long
SISMEMBER	Determine if a given value is a member of a set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean
SINTER	Intersect multiple sets	CamelRedis.Key (String), CamelRedis.Keys (String)	Set<Object>
SINTERSTORE	Intersect multiple sets and store the resulting set in a key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void
SUNION	Add multiple sets	CamelRedis.Key (String), CamelRedis.Keys (String)	Set<Object>
SUNIONSTORE	Add multiple sets and store the resulting set in a key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void

Set Commands	Description	Parameters	Result
SDIFF	Subtract multiple sets	CamelRedis.Key (String), CamelRedis.Keys (String)	Set<Object>
SDIFFSTORE	Subtract multiple sets and store the resulting set in a key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void
SRANDMEMBER	Get one or multiple random members from a set	CamelRedis.Key (String)	String

Ordered set Commands	Description	Parameters	Result
ZADD	Add one or more members to a sorted set, or update its score if it already exists	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Score (Double)	Boolean
ZRANGE	Return a range of members in a sorted set, by index	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long), CamelRedis.WithScore (Boolean)	Object
ZREM	Remove one or more members from a sorted set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean

Ordered set Commands	Description	Parameters	Result
ZINCRBY	Increment the score of a member in a sorted set	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Increment (Double)	Double
ZRANK	Determine the index of a member in a sorted set	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
ZREVRANK	Determine the index of a member in a sorted set, with scores ordered from high to low	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
ZREVRANGE	Return a range of members in a sorted set, by index, with scores ordered from high to low	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long), CamelRedis.WithScore (Boolean)	Object
ZCARD	Get the number of members in a sorted set	CamelRedis.Key (String)	Long
ZCOUNT	Count the members in a sorted set with scores within the given values	CamelRedis.Key (String), CamelRedis.Min (Double), CamelRedis.Max (Double)	Long

Ordered set Commands	Description	Parameters	Result
ZRANGEBYSCORE	Return a range of members in a sorted set, by score	CamelRedis.Key (String), CamelRedis.Min (Double), CamelRedis.Max (Double)	Set<Object>
ZREVRANGEBYSCORE	Return a range of members in a sorted set, by score, with scores ordered from high to low	CamelRedis.Key (String), CamelRedis.Min (Double), CamelRedis.Max (Double)	Set<Object>
ZREMRANGEBYRANK	Remove all members in a sorted set within the given indexes	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	void
ZREMRANGEBYSCORE	Remove all members in a sorted set within the given scores	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	void
ZUNIONSTORE	Add multiple sorted sets and store the resulting sorted set in a new key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void
ZINTERSTORE	Intersect multiple sorted sets and store the resulting sorted set in a new key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void

String Commands	Description	Parameters	Result
SET	Set the string value of a key	CamelRedis.Key (String), CamelRedis.Value (Object)	void
GET	Get the value of a key	CamelRedis.Key (String)	Object
STRLEN	Get the length of the value stored in a key	CamelRedis.Key (String)	Long
APPEND	Append a value to a key	CamelRedis.Key (String), CamelRedis.Value (String)	Integer
SETBIT	Sets or clears the bit at offset in the string value stored at key	CamelRedis.Key (String), CamelRedis.Offset (Long), CamelRedis.Value (Boolean)	void
GETBIT	Returns the bit value at offset in the string value stored at key	CamelRedis.Key (String), CamelRedis.Offset (Long)	Boolean
SETRANGE	Overwrite part of a string at key starting at the specified offset	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Offset (Long)	void
GETRANGE	Get a substring of the string stored at a key	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	String
SETNX	Set the value of a key, only if the key does not exist	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean

String Commands	Description	Parameters	Result
SETEX	Set the value and expiration of a key	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Timeout (Long), SECONDS	void
DECRBY	Decrement the integer value of a key by the given number	CamelRedis.Key (String), CamelRedis.Value (Long)	Long
DECR	Decrement the integer value of a key by one	CamelRedis.Key (String),	Long
INCRBY	Increment the integer value of a key by the given amount	CamelRedis.Key (String), CamelRedis.Value (Long)	Long
INCR	Increment the integer value of a key by one	CamelRedis.Key (String)	Long
MGET	Get the values of all the given keys	CamelRedis.Fields (Collection<String>)	List<Object>
MSET	Set multiple keys to multiple values	CamelRedis.Values(Map<String, Object>)	void
MSETNX	Set multiple keys to multiple values, only if none of the keys exist	CamelRedis.Key (String), CamelRedis.Value (Object)	void
GETSET	Set the string value of a key and return its old value	CamelRedis.Key (String), CamelRedis.Value (Object)	Object

Key Commands	Description	Parameters	Result
EXISTS	Determine if a key exists	CamelRedis.Key (String)	Boolean
DEL	Delete a key	CamelRedis.Keys (String)	void
TYPE	Determine the type stored at key	CamelRedis.Key (String)	DataType
KEYS	Find all keys matching the given pattern	CamelRedis.Pattern (String)	Collection<String>
RANDOMKEY	Return a random key from the keyspace	CamelRedis.Pattern (String), CamelRedis.Value (String)	String
RENAME	Rename a key	CamelRedis.Key (String)	void
RENAMENX	Rename a key, only if the new key does not exist	CamelRedis.Key (String), CamelRedis.Value (String)	Boolean
EXPIRE	Set a key's time to live in seconds	CamelRedis.Key (String), CamelRedis.Timeout (Long)	Boolean
SORT	Sort the elements in a list, set or sorted set	CamelRedis.Key (String)	List<Object>
PERSIST	Remove the expiration from a key	CamelRedis.Key (String)	Boolean
EXPIREAT	Set the expiration for a key as a UNIX timestamp	CamelRedis.Key (String), CamelRedis.Timestamp (Long)	Boolean

Key Commands	Description	Parameters	Result
PEXPIRE	Set a key's time to live in milliseconds	CamelRedis.Key (String), CamelRedis.Timeout (Long)	Boolean
PEXPIREAT	Set the expiration for a key as a UNIX timestamp specified in milliseconds	CamelRedis.Key (String), CamelRedis.Timestamp (Long)	Boolean
TTL	Get the time to live for a key	CamelRedis.Key (String)	Long
MOVE	Move a key to another database	CamelRedis.Key (String), CamelRedis.Db (Integer)	Boolean

Other Command	Description	Parameters	Result
MULTI	Mark the start of a transaction block	none	void
DISCARD	Discard all commands issued after MULTI	none	void
EXEC	Execute all commands issued after MULTI	none	void
WATCH	Watch the given keys to determine execution of the MULTI/EXEC block	CamelRedis.Keys (String)	void

Other Command	Description	Parameters	Result
UNWATCH	Forget about all watched keys	none	void
ECHO	Echo the given string	CamelRedis.Value (String)	String
PING	Ping the server	none	String
QUIT	Close the connection	none	void
PUBLISH	Post a message to a channel	CamelRedis.Channel (String), CamelRedis.Message (Object)	void

302.4. DEPENDENCIES

Maven users will need to add the following dependency to their pom.xml.

pom.xml

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-redis</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where **${camel-version**} must be replaced by the actual version of Camel (2.11 or higher).

302.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 303. SPRING SECURITY

Available as of Camel 2.3

The **camel-spring-security** component provides role-based authorization for Camel routes. It leverages the authentication and user services provided by Spring Security (formerly Acegi Security) and adds a declarative, role-based policy system to control whether a route can be executed by a given principal.

If you are not familiar with the Spring Security authentication and authorization system, please review the current reference documentation on the SpringSource web site linked above.

303.1. CREATING AUTHORIZATION POLICIES

Access to a route is controlled by an instance of a **SpringSecurityAuthorizationPolicy** object. A policy object contains the name of the Spring Security authority (role) required to run a set of endpoints and references to Spring Security **AuthenticationManager** and **AccessDecisionManager** objects used to determine whether the current principal has been assigned that role. Policy objects may be configured as Spring beans or by using an **<authorizationPolicy>** element in Spring XML.

The **<authorizationPolicy>** element may contain the following attributes:

Name	Default Value	Description
id	**null**	The unique Spring bean identifier which is used to reference the policy in routes (required)
access	**null**	The Spring Security authority name that is passed to the access decision manager (required)
authenticationManager	**authenticationManager**	The name of the Spring Security **AuthenticationManager** object in the context
accessDecisionManager	**accessDecisionManager**	The name of the Spring Security **AccessDecisionManager** object in the context
authenticationAdapter	Default Authentication Adapter	**Camel 2.4** The name of a**camel-spring-security** **AuthenticationAdapter** object in the context that is used to convert a **javax.security.auth.Subject** into a Spring Security **Authentication** instance.

Name	Default Value	Description
useThreadSecurityContext	**true**	If a **javax.security.auth.Subject** cannot be found in the In message header under Exchange.AUTHENTICATION, check the Spring Security **SecurityContextHolder** for an **Authentication** object.
alwaysReauthenticate	**false**	If set to true, the **SpringSecurityAuthorizationPolicy** will always call **AuthenticationManager.authenticate()** each time the policy is accessed.

303.2. CONTROLLING ACCESS TO CAMEL ROUTES

A Spring Security **AuthenticationManager** and **AccessDecisionManager** are required to use this component. Here is an example of how to configure these objects in Spring XML using the Spring Security namespace:

Now that the underlying security objects are set up, we can use them to configure an authorization policy and use that policy to control access to a route:

In this example, the endpoint **mock:end** will not be executed unless a Spring Security **Authentication** object that has been or can be authenticated and contains the **ROLE_ADMIN** authority can be located by the *admin* **SpringSecurityAuthorizationPolicy**.

303.3. AUTHENTICATION

The process of obtaining security credentials that are used for authorization is not specified by this component. You can write your own processors or components which get authentication information from the exchange depending on your needs. For example, you might create a processor that gets credentials from an HTTP request header originating in the Jetty component. No matter how the credentials are collected, they need to be placed in the In message or the **SecurityContextHolder** so the Camel Spring Security component can access them:

```
import javax.security.auth.Subject;
import org.apache.camel.*;
import org.apache.commons.codec.binary.Base64;
import org.springframework.security.authentication.*;


public class MyAuthService implements Processor {
   public void process(Exchange exchange) throws Exception {
      // get the username and password from the HTTP header
      // http://en.wikipedia.org/wiki/Basic_access_authentication
      String userpass = new
String(Base64.decodeBase64(exchange.getIn().getHeader("Authorization", String.class)));
      String[] tokens = userpass.split(":");

      // create an Authentication object
      UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(tokens[0], tokens[1]);
```

```
        // wrap it in a Subject
        Subject subject = new Subject();
        subject.getPrincipals().add(authToken);

        // place the Subject in the In message
        exchange.getIn().setHeader(Exchange.AUTHENTICATION, subject);

        // you could also do this if useThreadSecurityContext is set to true
        // SecurityContextHolder.getContext().setAuthentication(authToken);
    }
}
```

The **SpringSecurityAuthorizationPolicy** will automatically authenticate the **Authentication** object if necessary.

There are two issues to be aware of when using the **SecurityContextHolder** instead of or in addition to the **Exchange.AUTHENTICATION** header. First, the context holder uses a thread-local variable to hold the **Authentication** object. Any routes that cross thread boundaries, like **seda** or **jms**, will lose the **Authentication** object. Second, the Spring Security system appears to expect that an **Authentication** object in the context is already authenticated and has roles (see the Technical Overview section 5.3.1 for more details).

The default behavior of **camel-spring-security** is to look for a **Subject** in the **Exchange.AUTHENTICATION** header. This **Subject** must contain at least one principal, which must be a subclass of **org.springframework.security.core.Authentication**. You can customize the mapping of **Subject** to **Authentication** object by providing an implementation of the **org.apache.camel.component.spring.security.AuthenticationAdapter** to your **<authorizationPolicy>** bean. This can be useful if you are working with components that do not use Spring Security but do provide a **Subject**. At this time, only the CXF component populates the **Exchange.AUTHENTICATION** header.

303.4. HANDLING AUTHENTICATION AND AUTHORIZATION ERRORS

If authentication or authorization fails in the **SpringSecurityAuthorizationPolicy**, a **CamelAuthorizationException** will be thrown. This can be handled using Camel's standard exception handling methods, like the Exception Clause. The **CamelAuthorizationException** will have a reference to the ID of the policy which threw the exception so you can handle errors based on the policy as well as the type of exception:

```
<onException>
  <exception>org.springframework.security.authentication.AccessDeniedException</exception>
  <choice>
    <when>
      <simple>${exception.policyId} == 'user'</simple>
      <transform>
        <constant>You do not have ROLE_USER access!</constant>
      </transform>
    </when>
    <when>
      <simple>${exception.policyId} == 'admin'</simple>
      <transform>
        <constant>You do not have ROLE_ADMIN access!</constant>
      </transform>
```

```
    </when>
  </choice>
</onException>
```

303.5. DEPENDENCIES

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-security</artifactId>
  <version>2.4.0</version>
</dependency>
```

This dependency will also pull in **org.springframework.security:spring-security-core:3.0.3.RELEASE** and **org.springframework.security:spring-security-config:3.0.3.RELEASE**.

303.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Components

CHAPTER 304. SPRING WEBSERVICE COMPONENT

Available as of Camel version 2.6

The **spring-ws:** component allows you to integrate with Spring Web Services. It offers both *client*-side support, for accessing web services, and *server*-side support for creating your own contract-first web services.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-ws</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

INFO:*Dependencies* As of Camel 2.8 this component ships with Spring-WS 2.0.x which (like the rest of Camel) requires Spring 3.0.x. Earlier Camel versions shipped Spring-WS 1.5.9 which is compatible with Spring 2.5.x and 3.0.x. In order to run earlier versions of **camel-spring-ws** on Spring 2.5.x you need to add the **spring-webmvc** module from Spring 2.5.x. In order to run Spring-WS 1.5.9 on Spring 3.0.x you need to exclude the OXM module from Spring 3.0.x as this module is also included in Spring-WS 1.5.9 (see this post)

304.1. URI FORMAT

The URI scheme for this component is as follows

```
spring-ws:[mapping-type:]address[?options]
```

To expose a web service **mapping-type** needs to be set to any of the following:

Mapping type	Description
rootqname	Offers the option to map web service requests based on the qualified name of the root element contained in the message.
soapaction	Used to map web service requests based on the SOAP action specified in the header of the message.
uri	In order to map web service requests that target a specific URI.
xpathresult	Used to map web service requests based on the evaluation of an XPath **expression** against the incoming message. The result of the evaluation should match the XPath result specified in the endpoint URI.

Mapping type	Description
beanname	Allows you to reference an **org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher** object in order to integrate with existing (legacy) endpoint mappings like **PayloadRootQNameEndpointMapping**, **SoapActionEndpointMapping**, etc

As a consumer the **address** should contain a value relevant to the specified mapping-type (e.g. a SOAP action, XPath expression). As a producer the address should be set to the URI of the web service your calling upon.

You can append query **options** to the URI in the following format, **?option=value&option=value&...**

304.2. OPTIONS

The Spring WebService component supports 2 options which are listed below.

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Spring WebService endpoint is configured using URI syntax:

```
spring-ws:type:lookupKey:webServiceEndpointUri
```

with the following path and query parameters:

304.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
type	Endpoint mapping type if endpoint mapping is used. rootqname - Offers the option to map web service requests based on the qualified name of the root element contained in the message. soapaction - Used to map web service requests based on the SOAP action specified in the header of the message. uri - In order to map web service requests that target a specific URI. xpathresult - Used to map web service requests based on the evaluation of an XPath expression against the incoming message. The result of the evaluation should match the XPath result specified in the endpoint URI. beanname - Allows you to reference an org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher object in order to integrate with existing (legacy) endpoint mappings like PayloadRootQNameEndpointMapping, SoapActionEndpointMapping, etc		EndpointMappingType
lookupKey	Endpoint mapping key if endpoint mapping is used		String
webServiceEndpointUri	The default Web Service endpoint uri to use for the producer.		String

304.2.2. Query Parameters (22 parameters):

Name	Description	Default	Type
messageFilter (common)	Option to provide a custom MessageFilter. For example when you want to process your headers or attachments by your own.		MessageFilter
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
endpointDispatcher (consumer)	Spring org.springframework.ws.server.endpoint.MessageEndpoint for dispatching messages received by Spring-WS to a Camel endpoint, to integrate with existing (legacy) endpoint mappings like PayloadRootQNameEndpointMapping, SoapActionEndpointMapping, etc.		CamelEndpointDispatcher
endpointMapping (consumer)	Reference to an instance of org.apache.camel.component.spring.ws.bean.CamelEndpointMapping in the Registry/ApplicationContext. Only one bean is required in the registry to serve all Camel/Spring-WS endpoints. This bean is auto-discovered by the MessageDispatcher and used to map requests to Camel endpoints based on characteristics specified on the endpoint (like root QName, SOAP action, etc)		CamelSpringWSEndpoint Mapping
expression (consumer)	The XPath expression to use when option type=xpathresult. Then this option is required to be configured.		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
allowResponseAttachment Override (producer)	Option to override soap response attachments in in/out exchange with attachments from the actual service layer. If the invoked service appends or rewrites the soap attachments this option when set to true, allows the modified soap attachments to be overwritten in in/out message attachments	false	boolean
allowResponseHeaderOverride (producer)	Option to override soap response header in in/out exchange with header info from the actual service layer. If the invoked service appends or rewrites the soap header this option when set to true, allows the modified soap header to be overwritten in in/out message headers	false	boolean

Name	Description	Default	Type
faultAction (producer)	Signifies the value for the faultAction response WS-Addressing Fault Action header that is provided by the method.		URI
faultTo (producer)	Signifies the value for the faultAction response WS-Addressing FaultTo header that is provided by the method.		URI
messageFactory (producer)	Option to provide a custom WebServiceMessageFactory. For example when you want Apache Axiom to handle web service messages instead of SAAJ.		WebServiceMessage Factory
messageIdStrategy (producer)	Option to provide a custom MessageIdStrategy to control generation of unique message ids.		MessageIdStrategy
messageSender (producer)	Option to provide a custom WebServiceMessageSender. For example to perform authentication or use alternative transports		WebServiceMessage Sender
outputAction (producer)	Signifies the value for the response WS-Addressing Action header that is provided by the method.		URI
replyTo (producer)	Signifies the value for the replyTo response WS-Addressing ReplyTo header that is provided by the method.		URI
soapAction (producer)	SOAP action to include inside a SOAP request when accessing remote web services		String

Name	Description	Default	Type
timeout (producer)	Sets the socket read timeout (in milliseconds) while invoking a webservice using the producer, see URLConnection.setReadTimeout() and CommonsHttpMessageSender.setReadTimeout(). This option works when using the built-in message sender implementations: CommonsHttpMessageSender and HttpUrlConnectionMessageSender. One of these implementations will be used by default for HTTP based services unless you customize the Spring WS configuration options supplied to the component. If you are using a non-standard sender, it is assumed that you will handle your own timeout configuration. The built-in message sender HttpComponentsMessageSender is considered instead of CommonsHttpMessageSender which has been deprecated, see HttpComponentsMessageSender.setReadTimeout().		int
webServiceTemplate (producer)	Option to provide a custom WebServiceTemplate. This allows for full control over client-side web services handling; like adding a custom interceptor or specifying a fault resolver, message sender or message factory.		WebServiceTemplate
wsAddressingAction (producer)	WS-Addressing 1.0 action header to include when accessing web services. The To header is set to the address of the web service as specified in the endpoint URI (default Spring-WS behavior).		URI
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters

304.2.3. Message headers

Name	Type	Description
CamelSpringWebserviceEndpointUri	String	URI of the web service your accessing as a client, overrides *address* part of the endpoint URI

Name	Type	Description
CamelSpringWebserviceSoapAction	String	Header to specify the SOAP action of the message, overrides **soapAction** option if present
CamelSpringWebserviceSoapHeader	Source	**Camel 2.11.1:** Use this header to specify/access the SOAP headers of the message.
CamelSpringWebserviceAddressingAction	URI	Use this header to specify the WS-Addressing action of the message, overrides **wsAddressingAction** option if present
CamelSpringWebserviceAddressingFaultTo	URI	Use this header to specify the WS-Addressing FaultTo , overrides faultTo option if present
CamelSpringWebserviceAddressingReplyTo	URI	Use this header to specify the WS-Addressing ReplyTo , overrides replyTo option if present
CamelSpringWebserviceAddressingOutputAction	URI	Use this header to specify the WS-Addressing Action , overrides outputAction option if present

Name	Type	Description
CamelSpringWebserviceAddressingFaultAction	URI	Use this header to specify the WS-Addressing Fault Action , overrides faultAction option if present

304.3. ACCESSING WEB SERVICES

To call a web service at **http://foo.com/bar** simply define a route:

```
from("direct:example").to("spring-ws:http://foo.com/bar")
```

And sent a message:

```
template.requestBody("direct:example", "<foobar xmlns=\"http://foo.com\"><msg>test
message</msg></foobar>");
```

Remember if it's a SOAP service you're calling you don't have to include SOAP tags. Spring-WS will perform the XML-to-SOAP marshaling.

304.4. SENDING SOAP AND WS-ADDRESSING ACTION HEADERS

When a remote web service requires a SOAP action or use of the WS-Addressing standard you define your route as:

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?soapAction=http://foo.com&wsAddressingAction=http://bar.com")
```

Optionally you can override the endpoint options with header values:

```
template.requestBodyAndHeader("direct:example",
"<foobar xmlns=\"http://foo.com\"><msg>test message</msg></foobar>",
SpringWebserviceConstants.SPRING_WS_SOAP_ACTION, "http://baz.com");
```

304.5. USING SOAP HEADERS

Available as of Camel 2.11.1

You can provide the SOAP header(s) as a Camel Message header when sending a message to a spring-ws endpoint, for example given the following SOAP header in a String

```
String body = ...
String soapHeader = "<h:Header xmlns:h=\"http://www.webserviceX.NET/\">
<h:MessageID>1234567890</h:MessageID><h:Nested><h:NestedID>1111</h:NestedID>
</h:Nested></h:Header>";
```

We can set the body and header on the Camel Message as follows:

```
exchange.getIn().setBody(body);
exchange.getIn().setHeader(SpringWebserviceConstants.SPRING_WS_SOAP_HEADER,
soapHeader);
```

And then send the Exchange to a **spring-ws** endpoint to call the Web Service.

Likewise the spring-ws consumer will also enrich the Camel Message with the SOAP header.

For an example see this unit test.

304.6. THE HEADER AND ATTACHMENT PROPAGATION

Spring WS Camel supports propagation of the headers and attachments into Spring-WS WebServiceMessage response since version **2.10.3**. The endpoint will use so called "hook" the MessageFilter (default implementation is provided by BasicMessageFilter) to propagate the exchange headers and attachments into WebServiceMessage response. Now you can use

```
exchange.getOut().getHeaders().put("myCustom","myHeaderValue")
exchange.getIn().addAttachment("myAttachment", new DataHandler(...))
```

Note: If the exchange header in the pipeline contains text, it generates Qname(key)=value attribute in the soap header. Recommended is to create a QName class directly and put into any key into header.

304.7. HOW TO TRANSFORM THE SOAP HEADER USING A STYLESHEET

The header transformation filter (HeaderTransformationMessageFilter.java) can be used to transform the soap header for a soap request. If you want to use the header transformation filter, see the below example:

```
<bean id="headerTransformationFilter"
class="org.apache.camel.component.spring.ws.filter.impl.HeaderTransformationMessageFilter">
    <constructor-arg index="0" value="org/apache/camel/component/spring/ws/soap-header-
transform.xslt"/>
</bean
```

Use the bead defined above in the camel endpoint

```
<route>
    <from uri="direct:stockQuoteWebserviceHeaderTransformation"/>
    <to uri="spring-ws:http://localhost?
webServiceTemplate=#webServiceTemplate&amp;soapAction=http://www.stockquotes.edu/GetQuote&
amp;messageFilter=#headerTransformationFilter"/>
</route>
```

304.8. HOW TO USE MTOM ATTACHMENTS

The BasicMessageFilter provides all required information for Apache Axiom in order to produce MTOM message. If you want to use Apache Camel Spring WS within Apache Axiom, here is an example: - Simply define the messageFactory as is bellow and Spring-WS will use MTOM strategy to populate your SOAP

message with optimized attachments.

```
<bean id="axiomMessageFactory"
class="org.springframework.ws.soap.axiom.AxiomSoapMessageFactory">
<property name="payloadCaching" value="false" />
<property name="attachmentCaching" value="true" />
<property name="attachmentCacheThreshold" value="1024" />
</bean>
```

- Add into your pom.xml the following dependencies

```
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-api</artifactId>
<version>1.2.13</version>
</dependency>
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-impl</artifactId>
<version>1.2.13</version>
<scope>runtime</scope>
</dependency>
```

- Add your attachment into the pipeline, for example using a Processor implementation.

```
private class Attachement implements Processor {
public void process(Exchange exchange) throws Exception
{ exchange.getOut().copyFrom(exchange.getIn()); File file = new File("testAttachment.txt");
exchange.getOut().addAttachment("test", new DataHandler(new FileDataSource(file)));  }
}
```

- Define endpoint (producer) as ussual, for example like this:

```
from("direct:send")
.process(new Attachement())
.to("spring-ws:http://localhost:8089/mySoapService?
soapAction=mySoap&messageFactory=axiomMessageFactory");
```

- Now, your producer will generate MTOM message with otpmized attachments.

304.9. THE CUSTOM HEADER AND ATTACHMENT FILTERING

If you need to provide your custom processing of either headers or attachments, extend existing BasicMessageFilter and override the appropriate methods or write a brand new implementation of the MessageFilter interface.
To use your custom filter, add this into your spring context:

You can specify either a global a or a local message filter as follows: a) the global custom filter that provides the global configuration for all Spring-WS endpoints

```
<bean id="messageFilter" class="your.domain.myMessageFiler" scope="singleton" />
```

or b) the local messageFilter directly on the endpoint as follows:

-

```
to("spring-ws:http://yourdomain.com?messageFilter=#myEndpointSpecificMessageFilter");
```

For more information see CAMEL-5724

If you want to create your own MessageFilter, consider overriding the following methods in the default implementation of MessageFilter in class BasicMessageFilter:

```
protected void doProcessSoapHeader(Message inOrOut, SoapMessage soapMessage)
{your code /*no need to call super*/ }

protected void doProcessSoapAttachements(Message inOrOut, SoapMessage response)
{ your code /*no need to call super*/ }
```

304.10. USING A CUSTOM MESSAGESENDER AND MESSAGEFACTORY

A custom message sender or factory in the registry can be referenced like this:

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?
messageFactory=#messageFactory&messageSender=#messageSender")
```

Spring configuration:

```
<!-- authenticate using HTTP Basic Authentication -->
<bean id="messageSender"
class="org.springframework.ws.transport.http.HttpComponentsMessageSender">
    <property name="credentials">
        <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
            <constructor-arg index="0" value="admin"/>
            <constructor-arg index="1" value="secret"/>
        </bean>
    </property>
</bean>

<!-- force use of Sun SAAJ implementation, http://static.springsource.org/spring-
ws/sites/1.5/faq.html#saaj-jboss -->
<bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory">
    <property name="messageFactory">
        <bean
class="com.sun.xml.messaging.saaj.soap.ver1_1.SOAPMessageFactory1_1Impl"></bean>
    </property>
</bean>
```

304.11. EXPOSING WEB SERVICES

In order to expose a web service using this component you first need to set-up a MessageDispatcher to look for endpoint mappings in a Spring XML file. If you plan on running inside a servlet container you probably want to use a **MessageDispatcherServlet** configured in **web.xml**.

By default the **MessageDispatcherServlet** will look for a Spring XML named **/WEB-INF/spring-ws-servlet.xml**. To use Camel with Spring-WS the only mandatory bean in that XML file is **CamelEndpointMapping**. This bean allows the **MessageDispatcher** to dispatch web service requests

to your routes.

web.xml

```
<web-app>
   <servlet>
      <servlet-name>spring-ws</servlet-name>
      <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>spring-ws</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

spring-ws-servlet.xml

```
<bean id="endpointMapping"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointMapping" />

<bean id="wsdl" class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">
   <property name="schema">
      <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
         <property name="xsd" value="/WEB-INF/foobar.xsd"/>
      </bean>
   </property>
   <property name="portTypeName" value="FooBar"/>
   <property name="locationUri" value="/"/>
   <property name="targetNamespace" value="http://example.com/"/>
</bean>
```

More information on setting up Spring-WS can be found in Writing Contract-First Web Services. Basically paragraph 3.6 "Implementing the Endpoint" is handled by this component (specifically paragraph 3.6.2 "Routing the Message to the Endpoint" is where **CamelEndpointMapping** comes in). Also don't forget to check out the Spring Web Services Example included in the Camel distribution.

304.12. ENDPOINT MAPPING IN ROUTES

With the XML configuration in-place you can now use Camel's DSL to define what web service requests are handled by your endpoint:

The following route will receive all web service requests that have a root element named "GetFoo" within the **http://example.com/** namespace.

```
from("spring-ws:rootqname:{http://example.com/}GetFoo?endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

The following route will receive web service requests containing the **http://example.com/GetFoo** SOAP action.

```
from("spring-ws:soapaction:http://example.com/GetFoo?endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

The following route will receive all requests sent to **http://example.com/foobar**.

```
from("spring-ws:uri:http://example.com/foobar?endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

The route below will receive requests that contain the element **<foobar>abc</foobar>** anywhere inside the message (and the default namespace).

```
from("spring-ws:xpathresult:abc?expression=//foobar&endpointMapping=#endpointMapping")
.convertBodyTo(String.class).to(mock:example)
```

304.13. ALTERNATIVE CONFIGURATION, USING EXISTING ENDPOINT MAPPINGS

For every endpoint with mapping-type **beanname** one bean of type **CamelEndpointDispatcher** with a corresponding name is required in the Registry/ApplicationContext. This bean acts as a bridge between the Camel endpoint and an existing endpoint mapping like **PayloadRootQNameEndpointMapping**.

NOTE:The use of the **beanname** mapping-type is primarily meant for (legacy) situations where you're already using Spring-WS and have endpoint mappings defined in a Spring XML file. The **beanname** mapping-type allows you to wire your Camel route into an existing endpoint mapping. When you're starting from scratch it's recommended to define your endpoint mappings as Camel URI's (as illustrated above with **endpointMapping**) since it requires less configuration and is more expressive. Alternatively you could use vanilla Spring-WS with the help of annotations.

An example of a route using **beanname**:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="spring-ws:beanname:QuoteEndpointDispatcher" />
        <to uri="mock:example" />
    </route>
</camelContext>

<bean id="legacyEndpointMapping"
class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
    <property name="mappings">
        <props>
            <prop key="{http://example.com/}GetFuture">FutureEndpointDispatcher</prop>
            <prop key="{http://example.com/}GetQuote">QuoteEndpointDispatcher</prop>
        </props>
    </property>
</bean>

<bean id="QuoteEndpointDispatcher"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />
<bean id="FutureEndpointDispatcher"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />
```

304.14. POJO (UN)MARSHALLING

Camel's pluggable data formats offer support for pojo/xml marshalling using libraries such as JAXB, XStream, JibX, Castor and XMLBeans. You can use these data formats in your route to sent and receive pojo's, to and from web services.

When *accessing* web services you can marshal the request and unmarshal the response message:

```
JaxbDataFormat jaxb = new JaxbDataFormat(false);
jaxb.setContextPath("com.example.model");

from("direct:example").marshal(jaxb).to("spring-ws:http://foo.com/bar").unmarshal(jaxb);
```

Similarly when *providing* web services, you can unmarshal XML requests to POJO's and marshal the response message back to XML:

```
from("spring-ws:rootqname:{http://example.com/}GetFoo?
endpointMapping=#endpointMapping").unmarshal(jaxb)
.to("mock:example").marshal(jaxb);
```

304.15. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 305. SQL COMPONENT

Available as of Camel version 1.4

The **sql:** component allows you to work with databases using JDBC queries. The difference between this component and JDBC component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

This component uses **spring-jdbc** behind the scenes for the actual SQL handling.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sql</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

The SQL component also supports:

- JDBC-based repository for the Idempotent Consumer EIP pattern. See Section 305.10, "Using the JDBC-based idempotent repository".
- JDBC-based repository for the Aggregator EIP pattern. See Section 305.11, "Using the JDBC-based aggregation repository".

305.1. URI FORMAT

IMPORTANT

From Camel 2.11 onwards this component can create both consumer (e.g. **from()**) and producer endpoints (e.g. **to()**). In previous versions, it could only act as a producer.

NOTE

This component can be used as a Transactional Client.

The SQL component uses the following endpoint URI notation:

```
sql:select * from table where id=# order by name[?options]
```

From Camel 2.11 onwards you can use named parameters by using **#name_of_the_parameter** style as shown:

```
sql:select * from table where id=:#myId order by name[?options]
```

When using named parameters, Camel will lookup the names from, in the given precedence:
1. From message body if its a **java.util.Map**
2. From message headers

If a named parameter cannot be resolved, an exception is thrown.

From **Camel 2.14** onward you can use Simple expressions as parameters as shown:

```
sql:select * from table where id=:#${property.myId} order by name[?options]
```

Notice that the standard **?** symbol that denotes the parameters to an SQL query is substituted with the **#** symbol, because the **?** symbol is used to specify options for the endpoint. The **?** symbol replacement can be configured on endpoint basis.

From **Camel 2.17** onwards you can externalize your SQL queries to files in the classpath or file system as shown:

```
sql:classpath:sql/myquery.sql[?options]
```

And the **myquery.sql** file is in the classpath and is just a plain text:

```
-- this is a comment
select *
from table
where
  id = :#${property.myId}
order by
  name
```

In the file you can use multilines and format the SQL as you wish. And also use comments such as the – dash line.

You can append query options to the URI in the following format, **?option=value&option=value&…**

305.2. OPTIONS

The SQL component supports 3 options which are listed below.

Name	Description	Default	Type
dataSource (common)	Sets the DataSource to use to communicate with the database.		DataSource
usePlaceholder (advanced)	Sets whether to use placeholder and replace all placeholder characters with sign in the SQL queries. This option is default true	true	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The SQL endpoint is configured using URI syntax:

```
sql:query
```

with the following path and query parameters:

305.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
query	**Required** Sets the SQL query to perform. You can externalize the query by using file: or classpath: as prefix and specify the location of the file.		String

305.2.2. Query Parameters (45 parameters):

Name	Description	Default	Type
allowNamedParameters (common)	Whether to allow using named parameters in the queries.	true	boolean
dataSource (common)	Sets the DataSource to use to communicate with the database.		DataSource
dataSourceRef (common)	**Deprecated** Sets the reference to a DataSource to lookup from the registry, to use for communicating with the database.		String
outputClass (common)	Specify the full package and class name to use as conversion when outputType=SelectOne.		String
outputHeader (common)	Store the query result in a header instead of the message body. By default, outputHeader == null and the query result is stored in the message body, any existing content in the message body is discarded. If outputHeader is set, the value is used as the name of the header to store the query result and the original message body is preserved.		String

Name	Description	Default	Type
outputType (common)	Make the output of consumer or producer to SelectList as List of Map, or SelectOne as single Java object in the following way: a) If the query has only single column, then that JDBC Column object is returned. (such as SELECT COUNT( ) FROM PROJECT will return a Long object. b) If the query has more than one column, then it will return a Map of that result. c) If the outputClass is set, then it will convert the query result into an Java bean object by calling all the setters that match the column names. It will assume your class has a default constructor to create instance with. d) If the query resulted in more than one rows, it throws an non-unique result exception.	SelectList	SqlOutputType
separator (common)	The separator to use when parameter values is taken from message body (if the body is a String type), to be inserted at placeholders. Notice if you use named parameters, then a Map type is used instead. The default value is comma.	,	char
breakBatchOnConsumeFail (consumer)	Sets whether to break batch if onConsume failed.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
expectedUpdateCount (consumer)	Sets an expected update count to validate when using onConsume.	-1	int
maxMessagesPerPoll (consumer)	Sets the maximum number of messages to poll		int
onConsume (consumer)	After processing each row then this query can be executed, if the Exchange was processed successfully, for example to mark the row as processed. The query can have parameter.		String

Name	Description	Default	Type
onConsumeBatchComplete (consumer)	After processing the entire batch, this query can be executed to bulk update rows etc. The query cannot have parameters.		String
onConsumeFailed (consumer)	After processing each row then this query can be executed, if the Exchange failed, for example to mark the row as failed. The query can have parameter.		String
routeEmptyResultSet (consumer)	Sets whether empty resultset should be allowed to be sent to the next hop. Defaults to false. So the empty resultset will be filtered out.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
transacted (consumer)	Enables or disables transaction. If enabled then if processing an exchange failed then the consumer break out processing any further exchanges to cause a rollback eager	false	boolean
useIterator (consumer)	Sets how resultset should be delivered to route. Indicates delivery as either a list or individual object. defaults to true.	true	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
processingStrategy (consumer)	Allows to plugin to use a custom org.apache.camel.component.sql.SqlProcessingStrategy to execute queries when the consumer has processed the rows/batch.		SqlProcessingStrategy

Name	Description	Default	Type
batch (producer)	Enables or disables batch mode	false	boolean
noop (producer)	If set, will ignore the results of the SQL query and use the existing IN message as the OUT message for the continuation of processing	false	boolean
useMessageBodyForSql (producer)	Whether to use the message body as the SQL and then headers for parameters. If this option is enabled then the SQL in the uri is not used.	false	boolean
alwaysPopulateStatement (producer)	If enabled then the populateStatement method from org.apache.camel.component.sql.SqlPrepareStatementStrategy is always invoked, also if there is no expected parameters to be prepared. When this is false then the populateStatement is only invoked if there is 1 or more expected parameters to be set; for example this avoids reading the message body/headers for SQL queries with no parameters.	false	boolean
parametersCount (producer)	If set greater than zero, then Camel will use this count value of parameters to replace instead of querying via JDBC metadata API. This is useful if the JDBC vendor could not return correct parameters count, then user may override instead.		int
placeholder (advanced)	Specifies a character that will be replaced to in SQL query. Notice, that it is simple String.replaceAll() operation and no SQL parsing is involved (quoted strings will also change).	#	String
prepareStatementStrategy (advanced)	Allows to plugin to use a custom org.apache.camel.component.sql.SqlPrepareStatementStrategy to control preparation of the query and prepared statement.		SqlPrepareStatement Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
templateOptions (advanced)	Configures the Spring JdbcTemplate with the key/values from the Map		Map
usePlaceholder (advanced)	Sets whether to use placeholder and replace all placeholder characters with sign in the SQL queries. This option is default true	true	boolean

Name	Description	Default	Type
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean

Name	Description	Default	Type
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

305.3. TREATMENT OF THE MESSAGE BODY

The SQL component tries to convert the message body to an object of **java.util.Iterator** type and then uses this iterator to fill the query parameters (where each query parameter is represented by a **#** symbol (or configured placeholder) in the endpoint URI). If the message body is not an array or collection, the conversion results in an iterator that iterates over only one object, which is the body itself.

For example, if the message body is an instance of **java.util.List**, the first item in the list is substituted into the first occurrence of **#** in the SQL query, the second item in the list is substituted into the second occurrence of **#**, and so on.

If **batch** is set to **true**, then the interpretation of the inbound message body changes slightly – instead of an iterator of parameters, the component expects an iterator that contains the parameter iterators; the size of the outer iterator determines the batch size.

From Camel 2.16 onwards you can use the option useMessageBodyForSql that allows to use the message body as the SQL statement, and then the SQL parameters must be provided in a header with the key SqlConstants.SQL_PARAMETERS. This allows the SQL component to work more dynamic as the SQL query is from the message body.

305.4. RESULT OF THE QUERY

For **select** operations, the result is an instance of **List<Map<String, Object>>** type, as returned by the JdbcTemplate.queryForList() method. For **update** operations, the result is the number of updated rows, returned as an **Integer**.

By default, the result is placed in the message body. If the outputHeader parameter is set, the result is placed in the header. This is an alternative to using a full message enrichment pattern to add headers, it provides a concise syntax for querying a sequence or some other small value into a header. It is convenient to use outputHeader and outputType together:

```
from("jms:order.inbox")
    .to("sql:select order_seq.nextval from dual?outputHeader=OrderId&outputType=SelectOne")
    .to("jms:order.booking");
```

305.5. USING STREAMLIST

From **Camel 2.18** onwards the producer supports **outputType=StreamList** that uses an iterator to stream the output of the query. This allows to process the data in a streaming fashion which for example can be used by the Splitter EIP to process each row one at a time, and load data from the database as needed.

```
from("direct:withSplitModel")
        .to("sql:select * from projects order by id?
outputType=StreamList&outputClass=org.apache.camel.component.sql.ProjectModel")
        .to("log:stream")
        .split(body()).streaming()
            .to("log:row")
            .to("mock:result")
        .end();
```

305.6. HEADER VALUES

When performing **update** operations, the SQL Component stores the update count in the following message headers:

Header	Description
CamelSqlUpdateCount	The number of rows updated for **update** operations, returned as an **Integer** object. This header is not provided when using outputType=StreamList.
CamelSqlRowCount	The number of rows returned for **select** operations, returned as an **Integer** object. This header is not provided when using outputType=StreamList.
CamelSqlQuery	**Camel 2.8:** Query to execute. This query takes precedence over the query specified in the endpoint URI. Note that query parameters in the header *are* represented by a **?** instead of a **#** symbol

When performing **insert** operations, the SQL Component stores the rows with the generated keys and number of these rown in the following message headers (**Available as of Camel 2.12.4, 2.13.1**):

Header	Description
CamelSqlGeneratedKeysRowCount	The number of rows in the header that contains generated keys.
CamelSqlGeneratedKeyRows	Rows that contains the generated keys (a list of maps of keys).

305.7. GENERATED KEYS

Available as of Camel 2.12.4, 2.13.1 and 2.14

If you insert data using SQL INSERT, the RDBMS may support auto generated keys. You can instruct the SQL producer to return the generated keys in headers.
To do this, set the header **CamelSqlRetrieveGeneratedKeys=true**. Then the generated keys will be provided as headers with the keys listed in the table above.

You can see more details in this unit test.

305.8. CONFIGURATION

You can now set a reference to a **DataSource** in the URI directly:

```
sql:select * from table where id=# order by name?dataSource=myDS
```

305.9. SAMPLE

In the sample below we execute a query and retrieve the result as a **List** of rows, where each row is a **Map<String, Object** and the key is the column name.

First, we set up a table to use for our sample. As this is based on an unit test, we do it in java:

The SQL script **createAndPopulateDatabase.sql** we execute looks like as described below:

Then we configure our route and our **sql** component. Notice that we use a **direct** endpoint in front of the **sql** endpoint. This allows us to send an exchange to the **direct** endpoint with the URI, **direct:simple**, which is much easier for the client to use than the long **sql:** URI. Note that the **DataSource** is looked up up in the registry, so we can use standard Spring XML to configure our **DataSource**.

And then we fire the message into the **direct** endpoint that will route it to our **sql** component that queries the database.

We could configure the **DataSource** in Spring XML as follows:

```
<jee:jndi-lookup id="myDS" jndi-name="jdbc/myDataSource"/>
```

305.9.1. Using named parameters

Available as of Camel 2.11

In the given route below, we want to get all the projects from the projects table. Notice the SQL query has 2 named parameters, :#lic and :#min.
Camel will then lookup for these parameters from the message body or message headers. Notice in the example above we set two headers with constant value
for the named parameters:

```
from("direct:projects")
  .setHeader("lic", constant("ASF"))
  .setHeader("min", constant(123))
  .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

Though if the message body is a **java.util.Map** then the named parameters will be taken from the body.

```
from("direct:projects")
  .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

305.9.2. Using expression parameters

Available as of Camel 2.14

In the given route below, we want to get all the project from the database. It uses the body of the exchange for defining the license and uses the value of a property as the second parameter.

```
from("direct:projects")
  .setBody(constant("ASF"))
  .setProperty("min", constant(123))
  .to("sql:select * from projects where license = :#${body} and id > :#${property.min} order by id")
```

305.9.3. Using IN queries with dynamic values

Available as of Camel 2.17

From Camel 2.17 onwards the SQL producer allows to use SQL queries with IN statements where the IN values is dynamic computed. For example from the message body or a header etc.

To use IN you need to:

- Prefix the parameter name with **in:**
- Add **( )** around the parameter

An example explains this better. The following query is used:

```
-- this is a comment
select *
from projects
where project in (:#in:names)
order by id
```

In the following route:

```
from("direct:query")
    .to("sql:classpath:sql/selectProjectsIn.sql")
    .to("log:query")
    .to("mock:query");
```

Then the IN query can use a header with the key names with the dynamic values such as:

```
// use an array
template.requestBodyAndHeader("direct:query", "Hi there!", "names", new String[]{"Camel", "AMQ"});


// use a list
List<String> names = new ArrayList<String>();
names.add("Camel");
names.add("AMQ");
```

```
template.requestBodyAndHeader("direct:query", "Hi there!", "names", names);


// use a string separated values with comma
template.requestBodyAndHeader("direct:query", "Hi there!", "names", "Camel,AMQ");
```

The query can also be specified in the endpoint instead of being externalized (notice that externalizing makes maintaining the SQL queries easier)

```
from("direct:query")
    .to("sql:select * from projects where project in (:#in:names) order by id")
    .to("log:query")
    .to("mock:query");
```

305.10. USING THE JDBC-BASED IDEMPOTENT REPOSITORY

Available as of Camel 2.7: In this section we will use the JDBC based idempotent repository.

TIP

Abstract class From Camel 2.9 onwards there is an abstract class **org.apache.camel.processor.idempotent.jdbc.AbstractJdbcMessageIdRepository** you can extend to build custom JDBC idempotent repository.

First we have to create the database table which will be used by the idempotent repository. For **Camel 2.7**, we use the following schema:

```
CREATE TABLE CAMEL_MESSAGEPROCESSED ( processorName VARCHAR(255),
messageId VARCHAR(100) )
```

In **Camel 2.8**, we added the createdAt column:

```
CREATE TABLE CAMEL_MESSAGEPROCESSED ( processorName VARCHAR(255),
messageId VARCHAR(100), createdAt TIMESTAMP )
```

WARNING

The SQL Server **TIMESTAMP** type is a fixed-length binary-string type. It does not map to any of the JDBC time types: **DATE**, **TIME**, or **TIMESTAMP**.

305.10.1. Customize the JdbcMessageIdRepository

Starting with **Camel 2.9.1** you have a few options to tune the **org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository** for your needs:

Parameter	Default Value	Description
createTableIfNotExists	true	Defines whether or not Camel should try to create the table if it doesn't exist.
tableExistsString	SELECT 1 FROM CAMEL_MESSAGEPROCESSED WHERE 1 = 0	This query is used to figure out whether the table already exists or not. It must throw an exception to indicate the table doesn't exist.
createString	CREATE TABLE CAMEL_MESSAGEPROCESSED (processorName VARCHAR(255), messageId VARCHAR(100), createdAt TIMESTAMP)	The statement which is used to create the table.

Parameter	Default Value	Description
queryString	SELECT COUNT(*) FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ? AND messageId = ?	The query which is used to figure out whether the message already exists in the repository (the result is not equals to '0'). It takes two parameters. This first one is the processor name (**String**) and the second one is the message id (**String**).
insertString	INSERT INTO CAMEL_MESSAGEPROCESSED (processorName, messageId, createdAt) VALUES (?, ?, ?)	The statement which is used to add the entry into the table. It takes three parameter. The first one is the processor name (**String**), the second one is the message id (**String**) and the third one is the timestamp (**java.sql.Timestamp**) when this entry was added to the repository.

Parameter	Default Value	Description
delete String	DELETE FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ? AND messageId = ?	The statement which is used to delete the entry from the database. It takes two parameter. This first one is the processor name (**String**) and the second one is the message id (**String**).

305.11. USING THE JDBC-BASED AGGREGATION REPOSITORY

Available as of Camel 2.6

NOTE

Using JdbcAggregationRepository in Camel 2.6

In Camel 2.6, the JdbcAggregationRepository is provided in the **camel-jdbc-aggregator** component. From Camel 2.7 onwards, the **JdbcAggregationRepository** is provided in the **camel-sql** component.

JdbcAggregationRepository is an **AggregationRepository** which on the fly persists the aggregated messages. This ensures that you will not lose messages, as the default aggregator will use an in memory only **AggregationRepository**. The **JdbcAggregationRepository** allows together with Camel to provide persistent support for the Aggregator.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the **confirm** method is invoked on the **AggregationRepository**. This means if the same Exchange fails again it will be retried until successful.

You can use option **maximumRedeliveries** to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the **deadLetterUri** option so Camel knows where to send the Exchange when the **maximumRedeliveries** was hit.

You can see some examples in the unit tests of camel-sql, for example this test.

305.11.1. Database

To be operational, each aggregator uses two table: the aggregation and completed one. By convention the completed has the same name as the aggregation one suffixed with **"_COMPLETED"**. The name must be configured in the Spring bean with the **RepositoryName** property. In the following example aggregation will be used.

The table structure definition of both table are identical: in both case a String value is used as key (**id**)

whereas a Blob contains the exchange serialized in byte array.
However one difference should be remembered: the **id** field does not have the same content depending on the table.
In the aggregation table **id** holds the correlation Id used by the component to aggregate the messages.
In the completed table, **id** holds the id of the exchange stored in corresponding the blob field.

Here is the SQL query used to create the tables, just replace **"aggregation"** with your aggregator repository name.

```
CREATE TABLE aggregation ( id varchar(255) NOT NULL, exchange blob NOT
NULL, constraint aggregation_pk PRIMARY KEY (id) ); CREATE TABLE
aggregation_completed ( id varchar(255) NOT NULL, exchange blob NOT
NULL, constraint aggregation_completed_pk PRIMARY KEY (id) );
```

305.11.2. Storing body and headers as text

Available as of Camel 2.11

You can configure the **JdbcAggregationRepository** to store message body and select(ed) headers as String in separate columns. For example to store the body, and the following two headers **companyName** and **accountName** use the following SQL:

```
CREATE TABLE aggregationRepo3 ( id varchar(255) NOT NULL, exchange blob
NOT NULL, body varchar(1000), companyName varchar(1000), accountName
varchar(1000), constraint aggregationRepo3_pk PRIMARY KEY (id) ); CREATE
TABLE aggregationRepo3_completed ( id varchar(255) NOT NULL, exchange
blob NOT NULL, body varchar(1000), companyName varchar(1000),
accountName varchar(1000), constraint aggregationRepo3_completed_pk
PRIMARY KEY (id) );
```

And then configure the repository to enable this behavior as shown below:

```
<bean id="repo3"
class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
<property name="repositoryName" value="aggregationRepo3"/> <property
name="transactionManager" ref="txManager3"/> <property name="dataSource"
ref="dataSource3"/> <!-- configure to store the message body and
following headers as text in the repo --> <property
name="storeBodyAsText" value="true"/> <property
name="headersToStoreAsText"> <list> <value>companyName</value>
<value>accountName</value> </list> </property> </bean>
```

305.11.3. Codec (Serialization)

Because they can contain any type of payload, Exchanges are not serializable by design. It is converted into a byte array to be stored in a database BLOB field. All those conversions are handled by the **JdbcCodec** class. One detail of the code requires your attention: the **ClassLoadingAwareObjectInputStream**.

The **ClassLoadingAwareObjectInputStream** has been reused from the Apache ActiveMQ project. It wraps an **ObjectInputStream** and use it with the **ContextClassLoader** rather than the **currentThread** one. The benefit is to be able to load classes exposed by other bundles. This allows the exchange body

and headers to have custom types object references.

305.11.4. Transaction

A Spring **PlatformTransactionManager** is required to orchestrate transactions.

305.11.4.1. Service (Start/Stop)

The **start** method verifies the connection of the database and the presence of the required tables. If anything is wrong it will fail during starting.

305.11.5. Aggregator configuration

Depending on the targeted environment, the aggregator might need some configuration. As you already know, each aggregator should have its own repository (with the corresponding pair of table created in the database) and a data source. If the default lobHandler is not adapted to your database system, it can be injected with the **lobHandler** property.

Here is the declaration for Oracle:

```
<bean id="lobHandler"
class="org.springframework.jdbc.support.lob.OracleLobHandler"> <property
name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/> </bean> <bean
id="nativeJdbcExtractor"
class="org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"/>
<bean id="repo"
class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
<property name="transactionManager" ref="transactionManager"/> <property
name="repositoryName" value="aggregation"/> <property name="dataSource"
ref="dataSource"/> <!-- Only with Oracle, else use default --> <property
name="lobHandler" ref="lobHandler"/> </bean>
```

305.11.6. Optimistic locking

From **Camel 2.12** onwards you can turn on **optimisticLocking** and use this JDBC based aggregation repository in a clustered environment where multiple Camel applications shared the same database for the aggregation repository. If there is a race condition there JDBC driver will throw a vendor specific exception which the **JdbcAggregationRepository** can react upon. To know which caused exceptions from the JDBC driver is regarded as an optimistick locking error we need a mapper to do this. Therefore there is a **org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionMapper** allows you to implement your custom logic if needed. There is a default implementation **org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLockingExceptionMapper** which works as follows:

The following check is done:

- If the caused exception is an **SQLException** then the SQLState is checked if starts with 23.
- If the caused exception is a **DataIntegrityViolationException**
- If the caused exception class name has "ConstraintViolation" in its name.
- Optional checking for FQN class name matches if any class names has been configured

You can in addition add FQN classnames, and if any of the caused exception (or any nested) equals any of the FQN class names, then its an optimistick locking error.

Here is an example, where we define 2 extra FQN class names from the JDBC vendor:

```
<bean id="repo"
class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
<property name="transactionManager" ref="transactionManager"/> <property
name="repositoryName" value="aggregation"/> <property name="dataSource"
ref="dataSource"/> <property name"jdbcOptimisticLockingExceptionMapper"
ref="myExceptionMapper"/> </bean> <!-- use the default mapper with extra
FQN class names from our JDBC driver --> <bean id="myExceptionMapper"
class="org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLockingExceptionMapper">
<property name="classNames"> <util:set>
<value>com.foo.sql.MyViolationExceptoion</value>
<value>com.foo.sql.MyOtherViolationExceptoion</value> </util:set>
</property> </bean>
```

305.12. CAMEL SQL STARTER

A starter module is available to spring-boot users. When using the starter, the **DataSource** can be directly configured using spring-boot properties.

```
# Example for a mysql datasource
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

To use this feature, add the following dependencies to your spring boot pom.xml file:

```
<dependency>
   <groupId>org.apache.camel</groupId>
   <artifactId>camel-sql-starter</artifactId>
   <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>

<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-jdbc</artifactId>
   <version>${spring-boot-version}</version>
</dependency>
```

You should also include the specific database driver, if needed.

305.13. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

- SQL Stored Procedure
- JDBC

CHAPTER 306. SQL STORED PROCEDURE COMPONENT

Available as of Camel version 2.17

The **sql-stored:** component allows you to work with databases using JDBC Stored Procedure queries. This component is an extension to the SQL Component but specialized for calling stored procedures.

This component uses **spring-jdbc** behind the scenes for the actual SQL handling.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sql</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

306.1. URI FORMAT

The SQL component uses the following endpoint URI notation:

```
sql-stored:template[?options]
```

Where template is the stored procedure template, where you declare the name of the stored procedure and the IN, INOUT, and OUT arguments.

You can also refer to the template in a external file on the file system or classpath such as:

```
sql-stored:classpath:sql/myprocedure.sql[?options]
```

Where sql/myprocedure.sql is a plain text file in the classpath with the template, as show:

```
SUBNUMBERS(
  INTEGER ${headers.num1},
  INTEGER ${headers.num2},
  INOUT INTEGER ${headers.num3} out1,
  OUT INTEGER out2
)
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

306.2. OPTIONS

The SQL Stored Procedure component supports 2 options which are listed below.

Name	Description	Default	Type
dataSource (producer)	Sets the DataSource to use to communicate with the database.		DataSource

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The SQL Stored Procedure endpoint is configured using URI syntax:

```
sql-stored:template
```

with the following path and query parameters:

306.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
template	**Required** Sets the StoredProcedure template to perform		String

306.2.2. Query Parameters (7 parameters):

Name	Description	Default	Type
batch (producer)	Enables or disables batch mode	false	boolean
dataSource (producer)	Sets the DataSource to use to communicate with the database.		DataSource
function (producer)	Whether this call is for a function.	false	boolean
noop (producer)	If set, will ignore the results of the template and use the existing IN message as the OUT message for the continuation of processing	false	boolean
outputHeader (producer)	Store the template result in a header instead of the message body. By default, outputHeader == null and the template result is stored in the message body, any existing content in the message body is discarded. If outputHeader is set, the value is used as the name of the header to store the template result and the original message body is preserved.		String

Name	Description	Default	Type
useMessageBodyForTemplate (producer)	Whether to use the message body as the template and then headers for parameters. If this option is enabled then the template in the uri is not used.	false	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

306.3. DECLARING THE STORED PROCEDURE TEMPLATE

The template is declared using a syntax that would be similar to a Java method signature. The name of the stored procedure, and then the arguments enclosed in parenthesis. An example explains this well:

```
<to uri="sql-stored:STOREDSAMPLE(INTEGER ${headers.num1},INTEGER
${headers.num2},INOUT INTEGER ${headers.num3} result1,OUT INTEGER result2)"/>
```

The arguments are declared by a type and then a mapping to the Camel message using simple expression. So, in this example the first two parameters are IN values of INTEGER type, mapped to the message headers. The third parameter is INOUT, meaning it accepts an INTEGER and then returns a different INTEGER result. The last parameter is the OUT value, also an INTEGER type.

In SQL term the stored procedure could be declared as:

```
CREATE PROCEDURE STOREDSAMPLE(VALUE1 INTEGER, VALUE2 INTEGER, INOUT
RESULT1 INTEGER, OUT RESULT2 INTEGER)
```

306.3.1. IN Parameters

IN parameters take four parts separated by a space: parameter name, SQL type (with scale), type name and value source.

Parameter name is optional and will be auto generated if not provided. It must be given between quotes(').

SQL type is required and can be an integer (positive or negative) or reference to integer field in some class. If SQL type contains a dot then component tries resolve that class and read the given field. For example SQL type com.Foo.INTEGER is read from the field INTEGER of class com.Foo. If the type doesn't contain comma then class to resolve the integer value will be java.sql.Types. Type can be postfixed by scale for example DECIMAL(10) would mean java.sql.Types.DECIMAL with scale 10.

Type name is optional and must be given between quotes(').

Value source is required. Value source populates the parameter value from the Exchange. It can be either a Simple expression or header location i.e. :#<header name>. For example Simple expression ${header.val} would mean that parameter value will be read from the header "val". Header location expression :#val would have identical effect.

```
<to uri="sql-stored:MYFUNC('param1' org.example.Types.INTEGER(10) ${header.srcValue})"/>
```

URI means that the stored procedure will be called with parameter name "param1", it's SQL type is read from field INTEGER of class org.example.Types and scale will be set to 10. Input value for the parameter is passed from the header "srcValue".

```
<to uri="sql-stored:MYFUNC('param1' 100 'mytypename' ${header.srcValue})"/>
```

URI is identical to previous on except SQL-type is 100 and type name is "mytypename".

Actual call will be done using org.springframework.jdbc.core.SqlParameter.

306.3.2. OUT Parameters

OUT parameters work similarly IN parameters and contain three parts: SQL type(with scale), type name and output parameter name.

SQL type works the same as IN parameters.

Type name is optional and also works the same as IN parameters.

Output parameter name is used for the OUT parameter name, as well as the header name where the result will be stored.

```
<to uri="sql-stored:MYFUNC(OUT org.example.Types.DECIMAL(10) outheader1)"/>
```

URI means that OUT parameter's name is "outheader1" and result will be but into header "outheader1".

```
<to uri="sql-stored:MYFUNC(OUT org.example.Types.NUMERIC(10) 'mytype' outheader1)"/>
```

This is identical to previous one but type name will be "mytype".

Actual call will be done using org.springframework.jdbc.core.SqlOutParameter.

306.3.3. INOUT Parameters

INOUT parameters are a combination of all of the above. They receive a value from the exchange, as well as store a result as a message header. The only caveat is that the IN parameter's "name" is skipped. Instead, the OUT parameter's "name" defines both the SQL parameter name, as well as the result header name.

```
<to uri="sql-stored:MYFUNC(INOUT DECIMAL(10) ${headers.inheader} outheader)"/>
```

Actual call will be done using org.springframework.jdbc.core.SqlInOutParameter.

306.4. CAMEL SQL STARTER

A starter module is available to spring-boot users. When using the starter, the **DataSource** can be directly configured using spring-boot properties.

```
# Example for a mysql datasource
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
```

```
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

To use this feature, add the following dependencies to your spring boot pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sql-starter</artifactId>
  <version>${camel.version}</version> <!-- use the same version as your Camel core version -->
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
  <version>${spring-boot-version}</version>
</dependency>
```

You should also include the specific database driver, if needed.

306.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- SQL Component

CHAPTER 307. SSH COMPONENT

Available as of Camel version 2.10

The SSH component enables access to SSH servers such that you can send an SSH command, and process the response.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ssh</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

307.1. URI FORMAT

```
ssh:[username[:password]@]host[:port][?options]
```

307.2. OPTIONS

The SSH component supports 12 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared SSH configuration		SshConfiguration
host (common)	Sets the hostname of the remote SSH server.		String
port (common)	Sets the port number for the remote SSH server.		int
username (security)	Sets the username to use in logging into the remote SSH server.		String
password (security)	Sets the password to use in connecting to remote SSH server. Requires keyPairProvider to be set to null.		String
pollCommand (common)	Sets the command string to send to the remote SSH server during every poll cycle. Only works with camel-ssh component being used as a consumer, i.e. from(ssh://...). You may need to end your command with a newline, and that must be URL encoded %0A		String
keyPairProvider (security)	Sets the KeyPairProvider reference to use when connecting using Certificates to the remote SSH Server.		KeyPairProvider

Name	Description	Default	Type
keyType (security)	Sets the key type to pass to the KeyPairProvider as part of authentication. KeyPairProvider.loadKey(...) will be passed this value. Defaults to ssh-rsa.		String
timeout (common)	Sets the timeout in milliseconds to wait in establishing the remote SSH server connection. Defaults to 30000 milliseconds.		long
certFilename (security)	**Deprecated** Sets the resource path of the certificate to use for Authentication.		String
certResource (security)	Sets the resource path of the certificate to use for Authentication. Will use ResourceHelperKeyPairProvider to resolve file based certificate, and depends on keyType setting.		String
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The SSH endpoint is configured using URI syntax:

```
ssh:host:port
```

with the following path and query parameters:

307.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
host	**Required** Sets the hostname of the remote SSH server.		String
port	Sets the port number for the remote SSH server.	22	int

307.2.2. Query Parameters (28 parameters):

Name	Description	Default	Type
failOnUnknownHost (common)	Specifies whether a connection to an unknown host should fail or not. This value is only checked when the property knownHosts is set.	false	boolean

Name	Description	Default	Type
knownHostsResource (common)	Sets the resource path for a known_hosts file		String
timeout (common)	Sets the timeout in milliseconds to wait in establishing the remote SSH server connection. Defaults to 30000 milliseconds.	30000	long
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
pollCommand (consumer)	Sets the command string to send to the remote SSH server during every poll cycle. Only works with camel-ssh component being used as a consumer, i.e. from(ssh://...) You may need to end your command with a newline, and that must be URL encoded %0A		String
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Name	Description	Default	Type
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean

Name	Description	Default	Type
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
certResource (security)	Sets the resource path of the certificate to use for Authentication. Will use ResourceHelperKeyPairProvider to resolve file based certificate, and depends on keyType setting.		String
keyPairProvider (security)	Sets the KeyPairProvider reference to use when connecting using Certificates to the remote SSH Server.		KeyPairProvider
keyType (security)	Sets the key type to pass to the KeyPairProvider as part of authentication. KeyPairProvider.loadKey(...) will be passed this value. Defaults to ssh-rsa.	ssh-rsa	String
password (security)	Sets the password to use in connecting to remote SSH server. Requires keyPairProvider to be set to null.		String
username (security)	Sets the username to use in logging into the remote SSH server.		String

307.3. USAGE AS A PRODUCER ENDPOINT

When the SSH Component is used as a Producer (**.to("ssh://...")**), it will send the message body as the command to execute on the remote SSH server.

Here is an example of this within the XML DSL. Note that the command has an XML encoded newline (**
**).

```
<route id="camel-example-ssh-producer">
  <from uri="direct:exampleSshProducer"/>
  <setBody>
    <constant>features:list&#10;</constant>
  </setBody>
  <to uri="ssh://karaf:karaf@localhost:8101"/>
  <log message="${body}"/>
</route>
```

307.4. AUTHENTICATION

The SSH Component can authenticate against the remote SSH server using one of two mechanisms: Public Key certificate or username/password. Configuring how the SSH Component does authentication is based on how and which options are set.

1. First, it will look to see if the **certResource** option has been set, and if so, use it to locate the referenced Public Key certificate and use that for authentication.

2. If **certResource** is not set, it will look to see if a **keyPairProvider** has been set, and if so, it will use that to for certificate based authentication.

3. If neither **certResource** nor **keyPairProvider** are set, it will use the **username** and **password** options for authentication. Even though the **username** and **password** are provided in the endpoint configuration and headers set with **SshConstants.USERNAME_HEADER** (**CamelSshUsername**) and **SshConstants.PASSWORD_HEADER** (**CamelSshPassword**), the endpoint configuration is surpassed and credentials set in the headers are used.

The following route fragment shows an SSH polling consumer using a certificate from the classpath.

In the XML DSL,

```
<route>
  <from uri="ssh://scott@localhost:8101?
certResource=classpath:test_rsa&amp;useFixedDelay=true&amp;delay=5000&amp;pollCommand=feat
ures:list%0A"/>
  <log message="${body}"/>
</route>
```

In the Java DSL,

```
from("ssh://scott@localhost:8101?
certResource=classpath:test_rsa&useFixedDelay=true&delay=5000&pollCommand=features:list%0A"
)
    .log("${body}");
```

An example of using Public Key authentication is provided in **examples/camel-example-ssh-security**.

Certificate Dependencies

You will need to add some additional runtime dependencies if you use certificate based authentication. The dependency versions shown are as of Camel 2.11, you may need to use later versions depending what version of Camel you are using.

```
<dependency>
  <groupId>org.apache.sshd</groupId>
  <artifactId>sshd-core</artifactId>
  <version>0.8.0</version>
</dependency>
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcpg-jdk15on</artifactId>
  <version>1.47</version>
</dependency>
<dependency>
  <groupId>org.bouncycastle</groupId>
```

```
  <artifactId>bcpkix-jdk15on</artifactId>
  <version>1.47</version>
</dependency>
```

307.5. EXAMPLE

See the **examples/camel-example-ssh** and **examples/camel-example-ssh-security** in the Camel distribution.

307.6. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 308. STAX COMPONENT

Available as of Camel version 2.9

The StAX component allows messages to be process through a SAX ContentHandler.
Another feature of this component is to allow to iterate over JAXB records using StAX, for example using the Splitter EIP.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stax</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

308.1. URI FORMAT

```
stax:content-handler-class
```

example:

```
stax:org.superbiz.FooContentHandler
```

From **Camel 2.11.1** onwards you can lookup a **org.xml.sax.ContentHandler** bean from the Registry using the # syntax as shown:

```
stax:#myHandler
```

308.2. OPTIONS

The StAX component has no options.

The StAX endpoint is configured using URI syntax:

```
stax:contentHandlerClass
```

with the following path and query parameters:

308.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
contentHandlerClass	**Required** The FQN class name for the ContentHandler implementation to use.		String

308.2.2. Query Parameters (1 parameters):

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

308.3. USAGE OF A CONTENT HANDLER AS STAX PARSER

The message body after the handling is the handler itself.

Here an example:

```
from("file:target/in")
  .to("stax:org.superbiz.handler.CountingHandler")
  // CountingHandler implements org.xml.sax.ContentHandler or extends
org.xml.sax.helpers.DefaultHandler
  .process(new Processor() {
    @Override
    public void process(Exchange exchange) throws Exception {
        CountingHandler handler = exchange.getIn().getBody(CountingHandler.class);
        // do some great work with the handler
    }
  });
```

308.4. ITERATE OVER A COLLECTION USING JAXB AND STAX

First we suppose you have JAXB objects.

For instance a list of records in a wrapper object:

```
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "records")
public class Records {
    @XmlElement(required = true)
    protected List<Record> record;

    public List<Record> getRecord() {
        if (record == null) {
            record = new ArrayList<Record>();
        }
        return record;
    }
}
```

and

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "record", propOrder = { "key", "value" })
public class Record {
    @XmlAttribute(required = true)
    protected String key;

    @XmlAttribute(required = true)
    protected String value;

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

Then you get a XML file to process:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<records>
  <record value="v0" key="0"/>
  <record value="v1" key="1"/>
  <record value="v2" key="2"/>
  <record value="v3" key="3"/>
  <record value="v4" key="4"/>
  <record value="v5" key="5"/>
</record>
```

The StAX component provides an **StAXBuilder** which can be used when iterating XML elements with the Camel Splitter

```
from("file:target/in")
    .split(stax(Record.class)).streaming()
        .to("mock:records");
```

Where **stax** is a static method on **org.apache.camel.component.stax.StAXBuilder** which you can static import in the Java code. The stax builder is by default namespace aware on the XMLReader it uses. From **Camel 2.11.1** onwards you can turn this off by setting the boolean parameter to false, as

shown below:

```
from("file:target/in")
    .split(stax(Record.class, false)).streaming()
        .to("mock:records");
```

308.4.1. The previous example with XML DSL

The example above could be implemented as follows in XML DSL

308.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 309. STOMP COMPONENT

Available as of Camel version 2.12

The **stomp:** component is used for communicating with Stomp compliant message brokers, like Apache ActiveMQ or ActiveMQ Apollo

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stomp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

309.1. URI FORMAT

```
stomp:queue:destination[?options]
```

Where **destination** is the name of the queue.

309.2. OPTIONS

The Stomp component supports 8 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared stomp configuration		StompConfiguration
brokerURL (common)	The URI of the Stomp broker to connect to		String
login (security)	The username		String
passcode (security)	The password		String
host (common)	The virtual host		String
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
headerFilterStrategy (filter)	To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Stomp endpoint is configured using URI syntax:

```
stomp:destination
```

with the following path and query parameters:

309.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
destination	**Required** Name of the queue		String

309.2.2. Query Parameters (10 parameters):

Name	Description	Default	Type
brokerURL (common)	**Required** The URI of the Stomp broker to connect to	tcp://localhost:61613	String
host (common)	The virtual host name		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
login (security)	The username		String
passcode (security)	The password		String
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters

You can append query options to the URI in the following format, **?option=value&option=value&…**

309.3. SAMPLES

Sending messages:

```
from("direct:foo").to("stomp:queue:test");
```

Consuming messages:

```
from("stomp:queue:test").transform(body().convertToString()).to("mock:result")
```

309.4. ENDPOINTS

Camel supports the Message Endpoint pattern using the Endpoint interface. Endpoints are usually created by a Component and Endpoints are usually referred to in the DSL via their URIs.

From an Endpoint you can use the following methods

* createProducer() will create a Producer for sending message exchanges to the endpoint * createConsumer() implements the Event Driven Consumer pattern for consuming message exchanges from the endpoint via a Processor when creating a Consumer * createPollingConsumer() implements

the Polling Consumer pattern for consuming message exchanges from the endpoint via a PollingConsumer

309.5. SEE ALSO

- Configuring Camel
- Message Endpoint pattern
- URIs
- Writing Components

CHAPTER 310. STREAM COMPONENT

Available as of Camel version 1.3

The **stream:** component provides access to the **System.in**, **System.out** and **System.err** streams as well as allowing streaming of file and URL.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stream</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

310.1. URI FORMAT

```
stream:in[?options]
stream:out[?options]
stream:err[?options]
stream:header[?options]
```

In addition, the **file** and **url** endpoint URIs are supported:

```
stream:file?fileName=/foo/bar.txt
stream:url[?options]
```

If the **stream:header** URI is specified, the **stream** header is used to find the stream to write to. This option is available only for stream producers (that is, it cannot appear in **from()**).

You can append query options to the URI in the following format, **?option=value&option=value&…**

310.2. OPTIONS

The Stream component has no options.

The Stream endpoint is configured using URI syntax:

```
stream:kind
```

with the following path and query parameters:

310.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
kind	**Required** Kind of stream to use such as System.in or System.out.		String

310.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
encoding (common)	You can configure the encoding (is a charset name) to use text-based streams (for example, message body is a String object). If not provided, Camel uses the JVM default Charset.		String
fileName (common)	When using the stream:file URI format, this option specifies the filename to stream to/from.		String
url (common)	When using the stream:url URI format, this option specifies the URL to stream to/from. The input/output stream will be opened using the JDK URLConnection facility.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
groupLines (consumer)	To group X number of lines in the consumer. For example to group 10 lines and therefore only spit out an Exchange with 10 lines, instead of 1 Exchange per line.		int
groupStrategy (consumer)	Allows to use a custom GroupStrategy to control how to group lines.		GroupStrategy
initialPromptDelay (consumer)	Initial delay in milliseconds before showing the message prompt. This delay occurs only once. Can be used during system startup to avoid message prompts being written while other logging is done to the system out.	2000	long
promptDelay (consumer)	Optional delay in milliseconds before showing the message prompt.		long
promptMessage (consumer)	Message prompt to use when reading from stream:in; for example, you could set this to Enter a command:		String
retry (consumer)	Will retry opening the file if it's overwritten, somewhat like tail --retry	false	boolean

Name	Description	Default	Type
scanStream (consumer)	To be used for continuously reading a stream such as the unix tail command.	false	boolean
scanStreamDelay (consumer)	Delay in milliseconds between read attempts when using scanStream.		long
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
autoCloseCount (producer)	Number of messages to process before closing stream on Producer side. Never close stream by default (only when Producer is stopped). If more messages are sent, the stream is reopened for another autoCloseCount batch.		int
closeOnDone (producer)	This option is used in combination with Splitter and streaming to the same file. The idea is to keep the stream open and only close when the Splitter is done, to improve performance. Mind this requires that you only stream to the same file, and not 2 or more files.	false	boolean
delay (producer)	Initial delay in milliseconds before producing the stream.		long
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

310.3. MESSAGE CONTENT

The **stream:** component supports either **String** or **byte[]** for writing to streams. Just add either **String** or **byte[]** content to the **message.in.body**. Messages sent to the **stream:** producer in binary mode are not followed by the newline character (as opposed to the **String** messages). Message with **null** body will not be appended to the output stream.
The special **stream:header** URI is used for custom output streams. Just add a **java.io.OutputStream** object to **message.in.header** in the key **header**.
See samples for an example.

310.4. SAMPLES

In the following sample we route messages from the **direct:in** endpoint to the **System.out** stream:

```
// Route messages to the standard output.
from("direct:in").to("stream:out");

// Send String payload to the standard output.
// Message will be followed by the newline.
template.sendBody("direct:in", "Hello Text World");

// Send byte[] payload to the standard output.
// No newline will be added after the message.
template.sendBody("direct:in", "Hello Bytes World".getBytes());
```

The following sample demonstrates how the header type can be used to determine which stream to use. In the sample we use our own output stream, **MyOutputStream**.

The following sample demonstrates how to continuously read a file stream (analogous to the UNIX **tail** command):

```
from("stream:file?
fileName=/server/logs/server.log&scanStream=true&scanStreamDelay=1000").to("bean:logService?
method=parseLogLine");
```

One gotcha with scanStream (pre Camel 2.7) or scanStream + retry is the file will be re-opened and scanned with each iteration of scanStreamDelay. Until NIO2 is available we cannot reliably detect when a file is deleted/recreated.

310.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 311. STRING ENCODING DATAFORMAT

Available as of Camel version 2.12

The String Data Format is a textual based format that supports encoding.

311.1. OPTIONS

The String Encoding dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
charset		**String**	Sets an encoding to use. Will by default use the JVM platform default charset.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

311.2. MARSHAL

In this example we marshal the file content to String object in UTF-8 encoding.

```
from("file://data.csv").marshal().string("UTF-8").to("jms://myqueue");
```

311.3. UNMARSHAL

In this example we unmarshal the payload from the JMS queue to a String object using UTF-8 encoding, before its processed by the newOrder processor.

```
from("jms://queue/order").unmarshal().string("UTF-8").processRef("newOrder");
```

311.4. DEPENDENCIES

This data format is provided in **camel-core** so no additional dependencies is needed.

CHAPTER 312. STRING TEMPLATE COMPONENT

Available as of Camel version 1.2

The **string-template:** component allows you to process a message using a String Template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stringtemplate</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

312.1. URI FORMAT

```
string-template:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

You can append query options to the URI in the following format, **?option=value&option=value&...**

312.2. OPTIONS

The String Template component has no options.

The String Template endpoint is configured using URI syntax:

```
string-template:resourceUri
```

with the following path and query parameters:

312.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

312.2.2. Query Parameters (4 parameters):

Name	Description	Default	Type
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
delimiterStart (producer)	The variable start delimiter	<	char
delimiterStop (producer)	The variable end delimiter	>	char
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

312.3. HEADERS

Camel will store a reference to the resource in the message header with key, **org.apache.camel.stringtemplate.resource**. The Resource is an **org.springframework.core.io.Resource** object.

312.4. HOT RELOADING

The string template resource is by default hot-reloadable for both file and classpath resources (expanded jar). If you set **contentCache=true**, Camel loads the resource only once and hot-reloading is not possible. This scenario can be used in production when the resource never changes.

312.5. STRINGTEMPLATE ATTRIBUTES

Since Camel 2.14, you can define the custom context map by setting the message header "**CamelStringTemplateVariableMap**" just like the below code.

```
Map<String, Object> variableMap = new HashMap<String, Object>();
Map<String, Object> headersMap = new HashMap<String, Object>();
headersMap.put("name", "Willem");
variableMap.put("headers", headersMap);
variableMap.put("body", "Monday");
variableMap.put("exchange", exchange);
exchange.getIn().setHeader("CamelStringTemplateVariableMap", variableMap);
```

312.6. SAMPLES

For example you could use a string template as follows in order to formulate a response to a message:

```
from("activemq:My.Queue").
  to("string-template:com/acme/MyResponse.tm");
```

312.7. THE EMAIL SAMPLE

In this sample we want to use a string template to send an order confirmation email. The email template is laid out in **StringTemplate** as: This example works for **camel 2.11.0**. If your camel version is less than 2.11.0, the variables should be started and ended with $.

```
Dear <headers.lastName>, <headers.firstName>

Thanks for the order of <headers.item>.

Regards Camel Riders Bookstore
<body>
```

And the java code is as follows:

312.8. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 313. STUB COMPONENT

Available as of Camel version 2.10

The **stub:** component provides a simple way to stub out any physical endpoints while in development or testing, allowing you for example to run a route without needing to actually connect to a specific SMTP or Http endpoint. Just add **stub:** in front of any endpoint URI to stub out the endpoint.

Internally the Stub component creates VM endpoints. The main difference between Stub and VM is that VM will validate the URI and parameters you give it, so putting vm: in front of a typical URI with query arguments will usually fail. Stub won't though, as it basically ignores all query parameters to let you quickly stub out one or more endpoints in your route temporarily.

313.1. URI FORMAT

```
stub:someUri
```

Where **someUri** can be any URI with any query parameters.

313.2. OPTIONS

The Stub component supports 4 options which are listed below.

Name	Description	Default	Type
queueSize (advanced)	Sets the default maximum capacity of the SEDA queue (i.e., the number of messages it can hold).		int
concurrentConsumers (consumer)	Sets the default number of concurrent threads processing exchanges.	1	int
defaultQueueFactory (advanced)	Sets the default queue factory.		Exchange>
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Stub endpoint is configured using URI syntax:

```
stub:name
```

with the following path and query parameters:

313.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of queue		String

313.2.2. Query Parameters (16 parameters):

Name	Description	Default	Type
size (common)	The maximum capacity of the SEDA queue (i.e., the number of messages it can hold).	2147483647	int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
concurrentConsumers (consumer)	Number of concurrent threads processing exchanges.	1	int
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
limitConcurrentConsumers (consumer)	Whether to limit the number of concurrentConsumers to the maximum of 500. By default, an exception will be thrown if an endpoint is configured with a greater number. You can disable that check by turning this option off.	true	boolean
multipleConsumers (consumer)	Specifies whether multiple consumers are allowed. If enabled, you can use SEDA for Publish-Subscribe messaging. That is, you can send a message to the SEDA queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.	false	boolean

Name	Description	Default	Type
pollTimeout (consumer)	The timeout used when polling. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
purgeWhenStopping (consumer)	Whether to purge the task queue when stopping the consumer/route. This allows to stop faster, as any pending messages on the queue is discarded.	false	boolean
blockWhenFull (producer)	Whether a thread that sends messages to a full SEDA queue will block until the queue's capacity is no longer exhausted. By default, an exception will be thrown stating that the queue is full. By enabling this option, the calling thread will instead block and wait until the message can be accepted.	false	boolean
discardIfNoConsumers (producer)	Whether the producer should discard the message (do not add the message to the queue), when sending to a queue with no active consumers. Only one of the options discardIfNoConsumers and failIfNoConsumers can be enabled at the same time.	false	boolean
failIfNoConsumers (producer)	Whether the producer should fail by throwing an exception, when sending to a queue with no active consumers. Only one of the options discardIfNoConsumers and failIfNoConsumers can be enabled at the same time.	false	boolean
timeout (producer)	Timeout (in milliseconds) before a SEDA producer will stop waiting for an asynchronous task to complete. You can disable timeout by using 0 or a negative value.	30000	long
waitForTaskToComplete (producer)	Option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: Always, Never or IfReplyExpected. The first two values are self-explanatory. The last value, IfReplyExpected, will only wait if the message is Request Reply based. The default option is IfReplyExpected.	IfReplyExpected	WaitForTaskToComplete
queue (advanced)	Define the queue instance which will be used by the endpoint. This option is only for rare use-cases where you want to use a custom queue instance.		BlockingQueue
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

313.3. EXAMPLES

Here are a few samples of stubbing endpoint uris

```
stub:smtp://somehost.foo.com?user=whatnot&something=else
stub:http://somehost.bar.com/something
```

CHAPTER 314. SWAGGER JAVA COMPONENT

Available as of Camel 2.16

The Rest DSL can be integrated with the **camel-swagger-java** module which is used for exposing the REST services and their APIs using Swagger.

Maven users will need to add the following dependency to their **pom.xml** for this component:

From **Camel 2.16** onwards the swagger component is purely Java based, and its

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-swagger-java</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

The camel-swagger-java module can be used from the REST components (without the need for servlet)

For an example see the **camel-example-swagger-cdi** in the examples directory of the Apache Camel distribution.

314.1. USING SWAGGER IN REST-DSL

You can enable the swagger api from the rest-dsl by configuring the **apiContextPath** dsl as shown below:

```
public class UserRouteBuilder extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    // configure we want to use servlet as the component for the rest DSL
    // and we enable json binding mode
    restConfiguration().component("netty4-http").bindingMode(RestBindingMode.json)
      // and output using pretty print
      .dataFormatProperty("prettyPrint", "true")
      // setup context path and port number that netty will use
      .contextPath("/").port(8080)
      // add swagger api-doc out of the box
      .apiContextPath("/api-doc")
        .apiProperty("api.title", "User API").apiProperty("api.version", "1.2.3")
        // and enable CORS
        .apiProperty("cors", "true");

    // this user REST service is json only
    rest("/user").description("User rest service")
      .consumes("application/json").produces("application/json")
      .get("/{id}").description("Find user by id").outType(User.class)
        .param().name("id").type(path).description("The id of the user to
get").dataType("int").endParam()
        .to("bean:userService?method=getUser(${header.id})")
      .put().description("Updates or create a user").type(User.class)
        .param().name("body").type(body).description("The user to update or create").endParam()
        .to("bean:userService?method=updateUser")
      .get("/findAll").description("Find all users").outTypeList(User.class)
```

```
            .to("bean:userService?method=listUsers");
    }
}
```

314.2. OPTIONS

The swagger module can be configured using the following options. To configure using a servlet you use the init-param as shown above. When configuring directly in the rest-dsl, you use the appropriate method, such as **enableCORS**, **host,contextPath**, dsl. The options with **api.xxx** is configured using **apiProperty** dsl.

Option	Type	Description
cors	Boolean	Whether to enable CORS. Notice this only enables CORS for the api browser, and not the actual access to the REST services. Is default false. Instead of using this option is recommended to use the CorsFilte, see further below.
swagger.version	String	Swagger spec version. Is default 2.0.
host	String	To setup the hostname. If not configured camel-swagger-java will calculate the name as localhost based.
schemas	String	The protocol schemes to use. Multiple values can be separated by comma such as "http,https". The default value is "http". This option is **deprecated** from Camel 2.17 onwards due it should have been named schemes.
schemes	String	**Camel 2.17:** The protocol schemes to use. Multiple values can be separated by comma such as "http,https". The default value is "http".
base.path	String	**Required**: To setup the base path where the REST services is available. The path is relative (eg do not start with http/https) and camel-swagger-java will calculate the absolute base path at runtime, which will be **protocol://host:port/context-path/base.path**
api.path	String	To setup the path where the API is available (eg /api-docs). The path is relative (eg do not start with http/https) and camel-swagger-java will calculate the absolute base path at runtime, which will be **protocol://host:port/context-path/api.path** So using relative paths is much easier. See above for an example.
api.version	String	The version of the api. Is default 0.0.0.
api.title	String	The title of the application.
api.description	String	A short description of the application.

Option	Type	Description
api.termsOfService	String	A URL to the Terms of Service of the API.
api.contact.name	String	Name of person or organization to contact
api.contact.email	String	An email to be used for API-related correspondence.
api.contact.url	String	A URL to a website for more contact information.
api.license.name	String	The license name used for the API.
api.license.url	String	A URL to the license used for the API.
apiContextIdListing	boolean	Whether to allow listing all the CamelContext names in the JVM that has REST services. When enabled then the root path of the api-doc will list all the contexts. When disabled then no context ids is listed and the root path of the api-doc lists the current CamelContext. Is default false.
apiContextIdPattern	String	A pattern that allows to filter which CamelContext names is shown in the context listing. The pattern is using regular expression and * as wildcard. Its the same pattern matching as used by Intercept

314.3. CONTEXTIDLISTING ENABLED

When contextIdListing is enabled then its detecting all the running CamelContexts in the same JVM. These contexts are listed in the root path, eg /**api-docs** as a simple list of names in json format. To access the swagger documentation then the context-path must be appended with the Camel context id, such as **api-docs/myCamel**. The option apiContextIdPattern can be used to filter the names in this list.

314.4. JSON OR YAML

Available as of Camel 2.17

The camel-swagger-java module supports both JSon and Yaml out of the box. You can specify in the request url what you want returned by using /swagger.json or /swagger.yaml for either one. If none is specified then the HTTP Accept header is used to detect if json or yaml can be accepted. If either both is accepted or none was set as accepted then json is returned as the default format.

314.5. EXAMPLES

In the Apache Camel distribution we ship the **camel-example-swagger-cdi** and **camel-example-swagger-java** which demonstrates using this Swagger component.

CHAPTER 315. SYSLOG DATAFORMAT

Available as of Camel version 2.6

The **syslog** dataformat is used for working with RFC3164 and RFC5424 messages.

This component supports the following:

- UDP consumption of syslog messages
- Agnostic data format using either plain String objects or SyslogMessage model objects.
- Type Converter from/to SyslogMessage and String
- Integration with the camel-mina component.
- Integration with the camel-netty component.
- **Camel 2.14:** Encoder and decoder for the camel-netty component.
- **Camel 2.14:** Support for RFC5424 also.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-syslog</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

315.1. RFC3164 SYSLOG PROTOCOL

Syslog uses the user datagram protocol (UDP) 1 as its underlying transport layer mechanism. The UDP port that has been assigned to syslog is 514.

To expose a Syslog listener service we reuse the existing camel-mina component or camel-netty where we just use the **Rfc3164SyslogDataFormat** to marshal and unmarshal messages. Notice that from **Camel 2.14** onwards the syslog dataformat is renamed to **SyslogDataFormat**.

315.2. OPTIONS

The Syslog dataformat supports 1 options which are listed below.

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

315.3. RFC5424 SYSLOG PROTOCOL

Available as of Camel 2.14

To expose a Syslog listener service we reuse the existing camel-mina component or camel-netty where we just use the **SyslogDataFormat** to marshal and unmarshal messages

315.3.1. Exposing a Syslog listener

In our Spring XML file, we configure an endpoint to listen for udp messages on port 10514, note that in netty we disable the defaultCodec, this
will allow a fallback to a NettyTypeConverter and delivers the message as an InputStream:

```
<camelContext id="myCamel" xmlns="http://camel.apache.org/schema/spring">

  <dataFormats>
      <syslog id="mySyslog"/>
  </dataFormats>

  <route>
      <from uri="netty:udp://localhost:10514?sync=false&amp;allowDefaultCodec=false"/>
      <unmarshal ref="mySyslog"/>
      <to uri="mock:stop1"/>
  </route>

</camelContext>
```

The same route using camel-mina

```
<camelContext id="myCamel" xmlns="http://camel.apache.org/schema/spring">

  <dataFormats>
      <syslog id="mySyslog"/>
  </dataFormats>

  <route>
      <from uri="mina:udp://localhost:10514"/>
      <unmarshal ref="mySyslog"/>
      <to uri="mock:stop1"/>
  </route>

</camelContext>
```

315.3.2. Sending syslog messages to a remote destination

```
<camelContext id="myCamel" xmlns="http://camel.apache.org/schema/spring">

  <dataFormats>
     <syslog id="mySyslog"/>
  </dataFormats>

  <route>
     <from uri="direct:syslogMessages"/>
     <marshal ref="mySyslog"/>
```

```
        <to uri="mina:udp://remotehost:10514"/>
    </route>

</camelContext>
```

315.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 316. TAR FILE DATAFORMAT

Available as of Camel version 2.16

The Tar File Data Format is a message compression and de-compression format. Messages can be marshalled (compressed) to Tar Files containing a single entry, and Tar Files containing a single entry can be unmarshalled (decompressed) to the original file contents.

There is also a aggregation strategy that can aggregate multiple messages into a single Tar File.

316.1. TARFILE OPTIONS

The Tar File dataformat supports 4 options which are listed below.

Name	Default	Java Type	Description
usingIterator	**false**	**Boolean**	If the tar file has more then one entry, the setting this option to true, allows to work with the splitter EIP, to split the data using an iterator in a streaming mode.
allowEmptyDirectory	**false**	**Boolean**	If the tar file has more then one entry, setting this option to true, allows to get the iterator even if the directory is empty
preservePathElements	**false**	**Boolean**	If the file name contains path elements, setting this option to true, allows the path to be maintained in the tar file.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

316.2. MARSHAL

In this example we marshal a regular text/XML payload to a compressed payload using Tar File compression, and send it to an ActiveMQ queue called MY_QUEUE.

```
from("direct:start").marshal().tarFile().to("activemq:queue:MY_QUEUE");
```

The name of the Tar entry inside the created Tar File is based on the incoming **CamelFileName** message header, which is the standard message header used by the file component. Additionally, the outgoing **CamelFileName** message header is automatically set to the value of the incoming **CamelFileName** message header, with the ".tar" suffix. So for example, if the following route finds a file named "test.txt" in the input directory, the output will be a Tar File named "test.txt.tar" containing a single Tar entry named "test.txt":

```
from("file:input/directory?antInclude=*/.txt").marshal().tarFile().to("file:output/directory");
```

If there is no incoming **CamelFileName** message header (for example, if the file component is not the consumer), then the message ID is used by default, and since the message ID is normally a unique

generated ID, you will end up with filenames like **ID-MACHINENAME-2443-1211718892437-1-0.tar**. If you want to override this behavior, then you can set the value of the **CamelFileName** header explicitly in your route:

```
from("direct:start").setHeader(Exchange.FILE_NAME,
constant("report.txt")).marshal().tarFile().to("file:output/directory");
```

This route would result in a Tar File named "report.txt.tar" in the output directory, containing a single Tar entry named "report.txt".

316.3. UNMARSHAL

In this example we unmarshal a Tar File payload from an ActiveMQ queue called MY_QUEUE to its original format, and forward it for processing to the **UnTarpedMessageProcessor**.

```
from("activemq:queue:MY_QUEUE").unmarshal().tarFile().process(new
UnTarpedMessageProcessor());
```

If the Tar File has more then one entry, the usingIterator option of TarFileDataFormat to be true, and you can use splitter to do the further work.

```
  TarFileDataFormat tarFile = new TarFileDataFormat();
  tarFile.setUsingIterator(true);
  from("file:src/test/resources/org/apache/camel/dataformat/tarfile/?
consumer.delay=1000&noop=true")
    .unmarshal(tarFile)
    .split(body(Iterator.class))
        .streaming()
          .process(new UnTarpedMessageProcessor())
    .end();
```

Or you can use the TarSplitter as an expression for splitter directly like this

```
   from("file:src/test/resources/org/apache/camel/dataformat/tarfile?
consumer.delay=1000&noop=true")
     .split(new TarSplitter())
        .streaming()
        .process(new UnTarpedMessageProcessor())
     .end();
```

316.4. AGGREGATE

INFO:Please note that this aggregation strategy requires eager completion check to work properly.

In this example we aggregate all text files found in the input directory into a single Tar File that is stored in the output directory.

```
   from("file:input/directory?antInclude=*/.txt")
     .aggregate(new TarAggregationStrategy())
       .constant(true)
       .completionFromBatchConsumer()
       .eagerCheckCompletion()
    .to("file:output/directory");
```

The outgoing **CamelFileName** message header is created using java.io.File.createTempFile, with the ".tar" suffix. If you want to override this behavior, then you can set the value of the **CamelFileName** header explicitly in your route:

```
from("file:input/directory?antInclude=*/.txt")
  .aggregate(new TarAggregationStrategy())
    .constant(true)
    .completionFromBatchConsumer()
    .eagerCheckCompletion()
  .setHeader(Exchange.FILE_NAME, constant("reports.tar"))
.to("file:output/directory");
```

316.5. DEPENDENCIES

To use Tar Files in your camel routes you need to add a dependency on **camel-tarfile** which implements this data format.

If you use Maven you can just add the following to your **pom.xml**, substituting the version number for the latest & greatest release (see the download page for the latest versions).

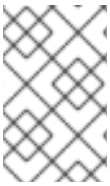
```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-tarfile</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 317. TELEGRAM COMPONENT

Available as of Camel version 2.18

The **Telegram** component provides access to the Telegram Bot API. It allows a Camel-based application to send and receive messages by acting as a Bot, participating in direct conversations with normal users, private and public groups or channels.

A Telegram Bot must be created before using this component, following the instructions at the Telegram Bot developers home. When a new Bot is created, the BotFather provides an **authorization token** corresponding to the Bot. The authorization token is a mandatory parameter for the camel-telegram endpoint.



NOTE

In order to allow the Bot to receive all messages exchanged within a group or channel (not just the ones starting with a '/' character), ask the BotFather to **disable the privacy mode**, using the **/setprivacy** command.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-telegram</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

317.1. URI FORMAT

```
telegram:type/authorizationToken[?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&...**

317.2. OPTIONS

The Telegram component supports 2 options which are listed below.

Name	Description	Default	Type
authorizationToken (security)	The default Telegram authorization token to be used when the information is not provided in the endpoints.		String
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Telegram endpoint is configured using URI syntax:

```
telegram:type/authorizationToken
```

with the following path and query parameters:

317.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
type	**Required** The endpoint type. Currently, only the 'bots' type is supported.		String
authorizationToken	**Required** The authorization token for using the bot (ask the BotFather), eg. 654321531:HGF_dTra456323dHuOedsE343211fqr3t-H.		String

317.2.2. Query Parameters (22 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
limit (consumer)	Limit on the number of updates that can be received in a single polling request.	100	Integer
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
timeout (consumer)	Timeout in seconds for long polling. Put 0 for short polling or a bigger number for long polling. Long polling produces shorter response time.	30	Integer
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
chatId (producer)	The identifier of the chat that will receive the produced messages. Chat ids can be first obtained from incoming messages (eg. when a telegram user starts a conversation with a bot, its client sends automatically a '/start' message containing the chat id). It is an optional parameter, as the chat id can be set dynamically for each outgoing message (using body or headers).		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean

Name	Description	Default	Type
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

317.3. MESSAGE HEADERS

Name	Description
CamelTelegramChatId	This header is used by the producer endpoint in order to resolve the chat id that will receive the message. The recipient chat id can be placed (in order of priority) in message body, in the **CamelTelegramChatId** header or in the endpoint configuration (**chatId** option). This header is also present in all incoming messages.
CamelTelegramMediaType	This header is used to identify the media type when the outgoing message is composed of pure binary data. Possible values are strings or enum values belonging to the **org.apache.camel.component.telegram.TelegramMediaType** enumeration.
CamelTelegramMediaTitleCaption	This header is used to provide a caption or title for outgoing binary messages.

Name	Description
CamelTelegramParseMode	This header is used to format text messages using HTML or Markdown (see **org.apache.camel.component.telegram.TelegramParseMode**).

317.4. USAGE

The Telegram component supports both consumer and producer endpoints. It can also be used in **reactive chat-bot mode** (to consume, then produce messages).

317.5. PRODUCER EXAMPLE

The following is a basic example of how to send a message to a Telegram chat through the Telegram Bot API.

in Java DSL

```
from("direct:start").to("telegram:bots/123456789:insertAuthorizationTokenHere");
```

or in Spring XML

```
<route>
   <from uri="direct:start"/>
   <to uri="telegram:bots/123456789:insertAuthorizationTokenHere"/>
<route>
```

The code **123456789:insertAuthorizationTokenHere** is the **authorization token** corresponding to the Bot.

When using the producer endpoint without specifying the **chat id** option, the target chat will be identified using information contained in the body or headers of the message. The following message bodies are allowed for a producer endpoint (messages of type **OutgoingXXXMessage** belong to the package **org.apache.camel.component.telegram.model**)

Java Type	Description
OutgoingTextMessage	To send a text message to a chat
OutgoingPhotoMessage	To send a photo (JPG, PNG) to a chat
OutgoingAudioMessage	To send a mp3 audio to a chat
OutgoingVideoMessage	To send a mp4 video to a chat
OutgoingDocumentMessage	To send a file to a chat (any media type)
byte[]	To send any media type supported. It requires the **CamelTelegramMediaType** header to be set to the appropriate media type

Java Type	Description
String	To send a text message to a chat. It gets converted automatically into a **OutgoingTextMessage**

317.6. CONSUMER EXAMPLE

The following is a basic example of how to receive all messages that telegram users are sending to the configured Bot. In Java DSL

```
from("telegram:bots/123456789:insertAuthorizationTokenHere")
.bean(ProcessorBean.class)
```

or in Spring XML

```
<route>
   <from uri="telegram:bots/123456789:insertAuthorizationTokenHere"/>
   <bean ref="myBean" />
<route>

<bean id="myBean" class="com.example.MyBean"/>
```

The **MyBean** is a simple bean that will receive the messages

```
public class MyBean {

   public void process(String message) {
      // or Exchange, or org.apache.camel.component.telegram.model.IncomingMessage (or both)

      // do process
   }

}
```

Supported types for incoming messages are

Java Type	Description
IncomingMessage	The full object representation of an incoming message
String	The content of the message, for text messages only

317.7. REACTIVE CHAT-BOT EXAMPLE

The reactive chat-bot mode is a simple way of using the Camel component to build a simple chat bot that replies directly to chat messages received from the Telegram users.

The following is a basic configuration of the chat-bot in Java DSL

```
from("telegram:bots/123456789:insertAuthorizationTokenHere")
.bean(ChatBotLogic.class)
.to("telegram:bots/123456789:insertAuthorizationTokenHere");
```

or in Spring XML

```
<route>
   <from uri="telegram:bots/123456789:insertAuthorizationTokenHere"/>
   <bean ref="chatBotLogic" />
   <to uri="telegram:bots/123456789:insertAuthorizationTokenHere"/>
<route>

<bean id="chatBotLogic" class="com.example.ChatBotLogic"/>
```

The **ChatBotLogic** is a simple bean that implements a generic String-to-String method.

```
public class ChatBotLogic {

   public String chatBotProcess(String message) {
      if( "do-not-reply".equals(message) ) {
         return null; // no response in the chat
      }

      return "echo from the bot: " + message; // echoes the message
   }

}
```

Every non-null string returned by the **chatBotProcess** method is automatically routed to the chat that originated the request (as the **CamelTelegramChatId** header is used to route the message).

317.8. GETTING THE CHAT ID

If you want to push messages to a specific Telegram chat when an event occurs, you need to retrieve the corresponding chat ID. The chat ID is not currently shown in the telegram client, but you can obtain it using a simple route.

First, add the bot to the chat where you want to push messages, then run a route like the following one.

```
from("telegram:bots/123456789:insertAuthorizationTokenHere")
.to("log:INFO?showHeaders=true");
```

Any message received by the bot will be dumped to your log together with information about the chat (**CamelTelegramChatId** header).

Once you get the chat ID, you can use the following sample route to push message to it.

```
from("timer:tick")
.setBody().constant("Hello")
to("telegram:bots/123456789:insertAuthorizationTokenHere?chatId=123456")
```

Note that the corresponding URI parameter is simply **chatId**.

CHAPTER 318. TEST COMPONENT

Available as of Camel version 1.3

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Camel Testing Framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Bean Integration.

The **test** component extends the Mock component to support pulling messages from another endpoint on startup to set the expected message bodies on the underlying Mock endpoint. That is, you use the test endpoint in a route and messages arriving on it will be implicitly compared to some expected messages extracted from some other location.

So you can use, for example, an expected set of message bodies as files. This will then set up a properly configured Mock endpoint, which is only valid if the received messages match the number of expected messages and their message payloads are equal.

Maven users will need to add the following dependency to their **pom.xml** for this component when using **Camel 2.8** or older:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

From Camel 2.9 onwards the Test component is provided directly in the camel-core.

318.1. URI FORMAT

```
test:expectedMessagesEndpointUri
```

Where **expectedMessagesEndpointUri** refers to some other Component URI that the expected message bodies are pulled from before starting the test.

318.2. URI OPTIONS

The Test component has no options.

The Test endpoint is configured using URI syntax:

```
test:name
```

with the following path and query parameters:

318.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of endpoint to lookup in the registry to use for polling messages used for testing		String

318.2.2. Query Parameters (14 parameters):

Name	Description	Default	Type
anyOrder (producer)	Whether the expected messages should arrive in the same order or can be in any order.	false	boolean
assertPeriod (producer)	Sets a grace period after which the mock endpoint will re-assert to ensure the preliminary assertion is still valid. This is used for example to assert that exactly a number of messages arrives. For example if link expectedMessageCount(int) was set to 5, then the assertion is satisfied when 5 or more message arrives. To ensure that exactly 5 messages arrives, then you would need to wait a little period to ensure no further message arrives. This is what you can use this link setAssertPeriod(long) method for. By default this period is disabled.	0	long
delimiter (producer)	The split delimiter to use when split is enabled. By default the delimiter is new line based. The delimiter can be a regular expression.		String
expectedCount (producer)	Specifies the expected number of message exchanges that should be received by this endpoint. Beware: If you want to expect that 0 messages, then take extra care, as 0 matches when the tests starts, so you need to set a assert period time to let the test run for a while to make sure there are still no messages arrived; for that use link setAssertPeriod(long). An alternative is to use NotifyBuilder, and use the notifier to know when Camel is done routing some messages, before you call the link assertIsSatisfied() method on the mocks. This allows you to not use a fixed assert period, to speedup testing times. If you want to assert that exactly n'th message arrives to this mock endpoint, then see also the link setAssertPeriod(long) method for further details.	-1	int
reportGroup (producer)	A number that is used to turn on throughput logging based on groups of the size.		int

Name	Description	Default	Type
resultMinimumWaitTime (producer)	Sets the minimum expected amount of time (in millis) the link assertIsSatisfied() will wait on a latch until it is satisfied	0	long
resultWaitTime (producer)	Sets the maximum amount of time (in millis) the link assertIsSatisfied() will wait on a latch until it is satisfied	0	long
retainFirst (producer)	Specifies to only retain the first n'th number of received Exchanges. This is used when testing with big data, to reduce memory consumption by not storing copies of every Exchange this mock endpoint receives. Important: When using this limitation, then the link getReceivedCounter() will still return the actual number of received Exchanges. For example if we have received 5000 Exchanges, and have configured to only retain the first 10 Exchanges, then the link getReceivedCounter() will still return 5000 but there is only the first 10 Exchanges in the link getExchanges() and link getReceivedExchanges() methods. When using this method, then some of the other expectation methods is not supported, for example the link expectedBodiesReceived(Object...) sets a expectation on the first number of bodies received. You can configure both link setRetainFirst(int) and link setRetainLast(int) methods, to limit both the first and last received.	-1	int
retainLast (producer)	Specifies to only retain the last n'th number of received Exchanges. This is used when testing with big data, to reduce memory consumption by not storing copies of every Exchange this mock endpoint receives. Important: When using this limitation, then the link getReceivedCounter() will still return the actual number of received Exchanges. For example if we have received 5000 Exchanges, and have configured to only retain the last 20 Exchanges, then the link getReceivedCounter() will still return 5000 but there is only the last 20 Exchanges in the link getExchanges() and link getReceivedExchanges() methods. When using this method, then some of the other expectation methods is not supported, for example the link expectedBodiesReceived(Object...) sets a expectation on the first number of bodies received. You can configure both link setRetainFirst(int) and link setRetainLast(int) methods, to limit both the first and last received.	-1	int

Name	Description	Default	Type
sleepForEmptyTest (producer)	Allows a sleep to be specified to wait to check that this endpoint really is empty when link expectedMessageCount(int) is called with zero	0	long
split (producer)	If enabled the the messages loaded from the test endpoint will be split using new line delimiters so each line is an expected message. For example to use a file endpoint to load a file where each line is an expected message.	false	boolean
timeout (producer)	The timeout to use when polling for message bodies from the URI	2000	long
copyOnExchange (producer)	Sets whether to make a deep copy of the incoming Exchange when received at this mock endpoint. Is by default true.	true	boolean
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

318.3. EXAMPLE

For example, you could write a test case as follows:

```
from("seda:someEndpoint").
  to("test:file://data/expectedOutput?noop=true");
```

If your test then invokes the MockEndpoint.assertIsSatisfied(camelContext) method, your test case will perform the necessary assertions.

To see how you can set other expectations on the test endpoint, see the Mock component.

318.4. SEE ALSO

- Spring Testing

CHAPTER 319. THRIFT COMPONENT

Available as of Camel version 2.20

The Thrift component allows you to call or expose Remote Procedure Call (RPC) services using Apache Thrift binary communication protocol and serialization mechanism.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-thrift</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

319.1. URI FORMAT

```
thrift://service[?options]
```

319.2. ENDPOINT OPTIONS

The Thrift component supports 2 options which are listed below.

Name	Description	Default	Type
useGlobalSslContext Parameters (security)	Determine if the thrift component is using global SSL context parameters	false	boolean
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Thrift endpoint is configured using URI syntax:

```
thrift:host:port/service
```

with the following path and query parameters:

319.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
host	The Thrift server host name. This is localhost or 0.0.0.0 (if not defined) when being a consumer or remote server host name when using producer.		String
port	**Required** The Thrift server port		int
service	**Required** Fully qualified service name from the thrift descriptor file (package dot service definition name)		String

319.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type
compressionType (common)	Protocol compression mechanism type	NONE	ThriftCompressionType
exchangeProtocol (common)	Exchange protocol serialization type	BINARY	ThriftExchangeProtocol
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
clientTimeout (consumer)	Client timeout for consumers		int
maxPoolSize (consumer)	The Thrift server consumer max thread pool size	10	int
poolSize (consumer)	The Thrift server consumer initial thread pool size	1	int
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler

Name	Description	Default	Type
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
method (producer)	The Thrift invoked method name		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
negotiationType (security)	Security negotiation type	PLAIN TEXT	ThriftNegotiationType
sslParameters (security)	Configuration parameters for SSL/TLS security negotiation		SSLContextParameters

319.3. THRIFT METHOD PARAMETERS MAPPING

Parameters in the called procedure must be passed as a list of objects inside the message body. The primitives are converted from the objects on the fly. In order to correctly find the corresponding method, all types must be transmitted regardless of the values. Please see an exmaple below, how to pass different parameters to the method with the Camel body

```
List requestBody = new ArrayList();

requestBody.add((boolean)true);
requestBody.add((byte)THRIFT_TEST_NUM1);
requestBody.add((short)THRIFT_TEST_NUM1);
requestBody.add((int)THRIFT_TEST_NUM1);
requestBody.add((long)THRIFT_TEST_NUM1);
requestBody.add((double)THRIFT_TEST_NUM1);
requestBody.add("empty"); // String parameter
requestBody.add(ByteBuffer.allocate(10)); // binary parameter
requestBody.add(new Work(THRIFT_TEST_NUM1, THRIFT_TEST_NUM2, Operation.MULTIPLY));
// Struct parameter
requestBody.add(new ArrayList<Integer>()); // list paramater
requestBody.add(new HashSet<String>()); // set parameter
requestBody.add(new HashMap<String, Long>()); // map parameter

Object responseBody = template.requestBody("direct:thrift-alltypes", requestBody);
```

Incoming parameters in the service consumer will also be passed to the message body as a list of objects.

319.4. THRIFT CONSUMER HEADERS (WILL BE INSTALLED AFTER THE CONSUMER INVOCATION)

Header name	Description	Possible values
CamelThriftMethodName	Method name handled by the consumer service	

319.5. EXAMPLES

Below is a simple synchronous method invoke with host and port parameters

```
from("direct:thrift-calculate")
.to("thrift://localhost:1101/org.apache.camel.component.thrift.generated.Calculator?
method=calculate&synchronous=true");
```

Below is a simple synchronous method invoke for the XML DSL configuration

```
<route>
    <from uri="direct:thrift-add" />
    <to uri="thrift://localhost:1101/org.apache.camel.component.thrift.generated.Calculator?
method=add&synchronous=true"/>
</route>
```

Thrift service consumer with asynchronous communication

```
from("thrift://localhost:1101/org.apache.camel.component.thrift.generated.Calculator")
.to("direct:thrift-service");
```

It's possible to automate Java code generation for .thrift files using **thrift-maven-plugin**, but before start the thrift compiler binary distribution for your operating system must be present on the running host.

319.6. FOR MORE INFORMATION, SEE THESE RESOURCES

Thrift project GitHubhttps://thrift.apache.org/tutorial/java [Apache Thrift Java tutorial]

319.7. SEE ALSO

- Getting Started
- Configuring Camel
- Component
- Endpoint

CHAPTER 320. THRIFT DATAFORMAT

Available as of Camel version 2.20

Camel provides a Data Format to serialize between Java and the Apache Thrift . The project's site details why you may wish to https://thrift.apache.org/. Apache Thrift is language-neutral and platform-neutral, so messages produced by your Camel routes may be consumed by other language implementations.

Apache Thrift Implementation

320.1. THRIFT OPTIONS

The Thrift dataformat supports 3 options which are listed below.

Name	Default	Java Type	Description
instanceClass		**String**	Name of class to use when unarmshalling
contentTypeFormat	**binary**	**String**	Defines a content type format in which thrift message will be serialized/deserialized from(to) the Java been. The format can either be native or json for either native binary thrift, json or simple json fields representation. The default value is binary.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

320.2. CONTENT TYPE FORMAT

It's possible to parse JSON message to convert it to the Thrift format and unparse it back using native util converter. To use this option, set contentTypeFormat value to 'json' or call thrift with second parameter. If default instance is not specified, always use native binary Thrift format. The simple JSON format is write-only (marshal) and produces a simple output format suitable for parsing by scripting languages. The sample code shows below:

```
from("direct:marshal")
    .unmarshal()
    .thrift("org.apache.camel.dataformat.thrift.generated.Work", "json")
    .to("mock:reverse");
```

320.3. THRIFT OVERVIEW

This quick overview of how to use Thrift. For more detail see the complete tutorial

320.4. DEFINING THE THRIFT FORMAT

The first step is to define the format for the body of your exchange. This is defined in a .thrift file as so:

tutorial.thrift

```
namespace java org.apache.camel.dataformat.thrift.generated

enum Operation {
  ADD = 1,
  SUBTRACT = 2,
  MULTIPLY = 3,
  DIVIDE = 4
}

struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
  4: optional string comment,
}
```

320.5. GENERATING JAVA CLASSES

The Apache Thrift provides a compiler which will generate the Java classes for the format we defined in our .thrift file.

You can also run the compiler for any additional supported languages you require manually.

thrift -r --gen java -out ../java/ ./tutorial-dataformat.thrift

This will generate separate Java class for each type defined in .thrift file, i.e. struct or enum. The generated classes implement org.apache.thrift.TBase which is required by the serialization mechanism. For this reason it important that only these classes are used in the body of your exchanges. Camel will throw an exception on route creation if you attempt to tell the Data Format to use a class that does not implement org.apache.thrift.TBase.

320.6. JAVA DSL

You can use create the ThriftDataFormat instance and pass it to Camel DataFormat marshal and unmarshal API like this.

```
ThriftDataFormat format = new ThriftDataFormat(new Work());

from("direct:in").marshal(format);
from("direct:back").unmarshal(format).to("mock:reverse");
```

Or use the DSL thrift() passing the unmarshal default instance or default instance class name like this.

```
// You don't need to specify the default instance for the thrift marshaling
from("direct:marshal").marshal().thrift();
from("direct:unmarshalA").unmarshal()
    .thrift("org.apache.camel.dataformat.thrift.generated.Work")
    .to("mock:reverse");

from("direct:unmarshalB").unmarshal().thrift(new Work()).to("mock:reverse");
```

320.7. SPRING DSL

The following example shows how to use Thrift to unmarshal using Spring configuring the thrift data type

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <unmarshal>
      <thrift instanceClass="org.apache.camel.dataformat.thrift.generated.Work" />
    </unmarshal>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

320.8. DEPENDENCIES

To use Thrift in your camel routes you need to add the a dependency on **camel-thrift** which implements this data format.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-thrift</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 321. TIDYMARKUP DATAFORMAT

Available as of Camel version 2.0

TidyMarkup is a Data Format that uses the TagSoup to tidy up HTML. It can be used to parse ugly HTML and return it as pretty wellformed HTML.

Camel eats our own -dog food- soap

We had some issues in our pdf Manual where we had some strange symbols. So Jonathan used this data format to tidy up the wiki html pages that are used as base for rendering the pdf manuals. And then the mysterious symbols vanished.

TidyMarkup only supports the **unmarshal** operation as we really don't want to turn well formed HTML into ugly HTML.

321.1. TIDYMARKUP OPTIONS

The TidyMarkup dataformat supports 3 options which are listed below.

Name	Default	Java Type	Description
dataObjectType	**org.w3c.dom.Node**	**String**	What data type to unmarshal as, can either be org.w3c.dom.Node or java.lang.String. Is by default org.w3c.dom.Node
omitXmlDeclaration	**false**	**Boolean**	When returning a String, do we omit the XML declaration in the top.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

321.2. JAVA DSL EXAMPLE

An example where the consumer provides some HTML

```
from("file://site/inbox").unmarshal().tidyMarkup().to("file://site/blogs");
```

321.3. SPRING XML EXAMPLE

The following example shows how to use TidyMarkup to unmarshal using Spring

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://site/inbox"/>
    <unmarshal>
```

```
      <tidyMarkup/>
    </unmarshal>
    <to uri="file://site/blogs"/>
  </route>
</camelContext>
```

321.4. DEPENDENCIES

To use TidyMarkup in your camel routes you need to add the a dependency on **camel-tagsoup** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-tagsoup</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 322. TIKA COMPONENT

Available as of Camel version 2.19

The **Tika**: components provides the ability to detect and parse documents with Apache Tika. This component uses Apache Tika as underlying library to work with documents.

In order to use the Tika component, Maven users will need to add the following dependency to their **pom.xml**:

pom.xml

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-tika</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

The TIKA component only supports producer endpoints.

322.1. OPTIONS

The Tika component has no options.

The Tika endpoint is configured using URI syntax:

```
tika:operation
```

with the following path and query parameters:

322.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
operation	**Required** Tika Operation. parse or detect		TikaOperation

322.1.2. Query Parameters (5 parameters):

Name	Description	Default	Type
tikaConfig (producer)	Tika Config		TikaConfig
tikaConfigUri (producer)	Tika Config Uri: The URI of tika-config.xml		String

Name	Description	Default	Type
tikaParseOutputEncoding (producer)	Tika Parse Output Encoding - Used to specify the character encoding of the parsed output. Defaults to Charset.defaultCharset() .		String
tikaParseOutputFormat (producer)	Tika Output Format. Supported output formats. xml: Returns Parsed Content as XML. html: Returns Parsed Content as HTML. text: Returns Parsed Content as Text. textMain: Uses the boilerpipe library to automatically extract the main content from a web page.	xml	TikaParseOutputFormat
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

322.2. TO DETECT A FILE'S MIME TYPE

The file should be placed in the Body.

```
from("direct:start")
        .to("tika:detect");
```

322.3. TO PARSE A FILE

The file should be placed in the Body.

```
from("direct:start")
        .to("tika:parse");
```

CHAPTER 323. TIMER COMPONENT

Available as of Camel version 1.0

The **timer:** component is used to generate message exchanges when a timer fires You can only consume events from this endpoint.

323.1. URI FORMAT

```
timer:name[?options]
```

Where **name** is the name of the **Timer** object, which is created and shared across endpoints. So if you use the same name for all your timer endpoints, only one **Timer** object and thread will be used.

You can append query options to the URI in the following format, **?option=value&option=value&…**

Note: The IN body of the generated exchange is **null**. So **exchange.getIn().getBody()** returns **null**.

TIP:*Advanced Scheduler* See also the Quartz component that supports much more advanced scheduling.

TIP:*Specify time in human friendly format* In **Camel 2.3** onwards you can specify the time in human friendly syntax.

323.2. OPTIONS

The Timer component has no options.

The Timer endpoint is configured using URI syntax:

```
timer:timerName
```

with the following path and query parameters:

323.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
timerName	**Required** The name of the timer		String

323.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
delay (consumer)	The number of milliseconds to wait before the first event is generated. Should not be used in conjunction with the time option. The default value is 1000. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
fixedRate (consumer)	Events take place at approximately regular intervals, separated by the specified period.	false	boolean
period (consumer)	If greater than 0, generate periodic events every period milliseconds. The default value is 1000. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
repeatCount (consumer)	Specifies a maximum limit of number of fires. So if you set it to 1, the timer will only fire once. If you set it to 5, it will only fire five times. A value of zero or negative means fire forever.	0	long
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
daemon (advanced)	Specifies whether or not the thread associated with the timer endpoint runs as a daemon. The default value is true.	true	boolean
pattern (advanced)	Allows you to specify a custom Date pattern to use for setting the time option using URI syntax.		String

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
time (advanced)	A java.util.Date the first event should be generated. If using the URI, the pattern expected is: yyyy-MM-dd HH:mm:ss or yyyy-MM-dd'T'HH:mm:ss.		Date
timer (advanced)	To use a custom Timer		Timer

323.3. EXCHANGE PROPERTIES

When the timer is fired, it adds the following information as properties to the **Exchange**:

Name	Type	Description
Exchange.TIMER_NAME	**String**	The value of the **name** option.
Exchange.TIMER_TIME	**Date**	The value of the **time** option.
Exchange.TIMER_PERIOD	**long**	The value of the **period** option.
Exchange.TIMER_FIRED_TIME	**Date**	The time when the consumer fired.
Exchange.TIMER_COUNTER	**Long**	**Camel 2.8:** The current fire counter. Starts from 1.

323.4. SAMPLE

To set up a route that generates an event every 60 seconds:

```
from("timer://foo?fixedRate=true&period=60000").to("bean:myBean?method=someMethodName");
```

TIP

Instead of 60000 you can use period=60s which is more friendly to read.

The above route will generate an event and then invoke the **someMethodName** method on the bean called **myBean** in the Registry such as JNDI or Spring.

And the route in Spring DSL:

```
<route>
  <from uri="timer://foo?fixedRate=true&amp;period=60000"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

323.5. FIRING AS SOON AS POSSIBLE

Available as of Camel 2.17

You may want to fire messages in a Camel route as soon as possible you can use a negative delay:

```
<route>
  <from uri="timer://foo?delay=-1"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

In this way the timer will fire messages immediately.

You can also specify a repeatCount parameter in conjunction with a negative delay to stop firing messages after a fixed number has been reached.

If you don't specify a repeatCount then the timer will continue firing messages until the route will be stopped.

323.6. FIRING ONLY ONCE

Available as of Camel 2.8

You may want to fire a message in a Camel route only once, such as when starting the route. To do that you use the repeatCount option as shown:

```
<route>
  <from uri="timer://foo?repeatCount=1"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

323.7. SEE ALSO

- Scheduler

- Quartz

CHAPTER 324. TWILIO COMPONENT

Available as of Camel version 2.20

The Twilio component provides access to Version 2010-04-01 of Twilio REST APIs accessible using Twilio Java SDK.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-twilio</artifactId>
  <version>${camel-version}</version>
</dependency>
```

324.1. TWILIO OPTIONS

The Twilio component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use the shared configuration		TwilioConfiguration
restClient (advanced)	To use the shared REST client		TwilioRestClient
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Twilio endpoint is configured using URI syntax:

```
twilio:apiName/methodName
```

with the following path and query parameters:

324.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
apiName	**Required** What kind of operation to perform		TwilioApiName
methodName	**Required** What sub operation to use for the selected operation		String

324.1.2. Query Parameters (8 parameters):

Name	Description	Default	Type
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
accountSid (security)	The account SID to use.		String
password (security)	Auth token for the account.		String
username (security)	The account to use.		String

324.2. URI FORMAT

```
twilio://endpoint-prefix/endpoint?[options]
```

Endpoint prefix can be one of:

- account
- address
- address-dependent-phone-number

- application
- available-phone-number-country
- available-phone-number-country-local
- available-phone-number-country-mobile
- available-phone-number-country-toll-free
- call
- call-feedback
- call-feedback-summary
- call-notification
- call-recording
- conference
- conference-participant
- connect-app
- incoming-phone-number
- incoming-phone-number-local
- incoming-phone-number-mobile
- incoming-phone-number-toll-free
- key
- message
- message-feedback
- message-media
- new-key
- new-signing-key
- notification
- outgoing-caller-id
- queue
- queue-member
- recording
- recording-add-on-result

- recording-add-on-result-payload
- recording-transcription
- short-code
- signing-key
- sip-credential-list
- sip-credential-list-credential
- sip-domain
- sip-domain-credential-list-mapping
- sip-domain-ip-access-control-list-mapping
- sip-ip-access-control-list
- sip-ip-access-control-list-ip-address
- token
- transcription
- usage-record
- usage-record-all-time
- usage-record-daily
- usage-record-last-month
- usage-record-monthly
- usage-record-this-month
- usage-record-today
- usage-record-yearly
- usage-record-yesterday
- usage-trigger
- validation-request

324.3. PRODUCER ENDPOINTS:

Producer endpoints can use endpoint prefixes followed by endpoint names and associated options described next. A shorthand alias can be used for all of the endpoints. The endpoint URI MUST contain a prefix.

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelTwilio.<option>**. Note that the **inBody** option overrides message header, i.e. the endpoint option **inBody=option** would override

a **CamelTwilio.option** header.

Endpoint can be one of:

Endpoint	Shorthand Alias	Description
creator	create	Make the request to the Twilio API to perform the create
deleter	delete	Make the request to the Twilio API to perform the delete
fetcher	fetch	Make the request to the Twilio API to perform the fetch
reader	read	Make the request to the Twilio API to perform the read
updater	update	Make the request to the Twilio API to perform the update

Available endpoints differ depending on the endpoint prefixes.

For more information on the endpoints and options see API documentation at:
https://www.twilio.com/docs/libraries/reference/twilio-java/index.html

324.4. CONSUMER ENDPOINTS:

Any of the producer endpoints can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. Consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange.

If you want to receive calls or messages from Twilio and respond to them using a Camel consumer endpoint, you can use other HTTP-based components such as **camel-servlet**, **camel-undertow**, **camel-jetty**, and **camel-netty-http** to respond with TwiML.

324.5. MESSAGE HEADER

Any of the options can be provided in a message header for producer endpoints with **CamelTwilio.** prefix.

324.6. MESSAGE BODY

All result message bodies utilize objects provided by the Twilio Java SDK. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint parameter.

CHAPTER 325. TWITTER COMPONENTS

Available as of Camel version 2.10

The camel-twitter consists of 4 components:

- Twitter Direct Message
- Twitter Search
- Twitter Streaming
- Twitter Timeline

The Twitter components enable the most useful features of the Twitter API by encapsulating Twitter4J. It allows direct, polling, or event-driven consumption of timelines, users, trends, and direct messages. Also, it supports producing messages as status updates or direct messages.

Twitter now requires the use of OAuth for all client application authentication. In order to use camel-twitter with your account, you'll need to create a new application within Twitter at https://dev.twitter.com/apps/new and grant the application access to your account. Finally, generate your access token and secret.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-twitter</artifactId>
  <version>${camel-version}</version>
</dependency>
```

325.1. CONSUMER ENDPOINTS

Rather than the endpoints returning a List through one single route exchange, camel-twitter creates one route exchange per returned object. As an example, if "timeline/home" results in five statuses, the route will be executed five times (one for each Status).

Endpoint	Context	Body Type	Notice
twitter-directmessage	direct, polling	twitter4j.DirectMessage	
twitter-search	direct, polling	twitter4j.Status	
twitter-streaming	event, polling	twitter4j.Status	

Endpoint	Context	Body Type	Notice
twitter-timeline	direct, polling	twitter4j.Status	

325.2. PRODUCER ENDPOINTS

Endpoint	Body Type	Notice
twitter-directmessage	String	
twitter-search	List<twitter4j.Status>	
twitter-timeline	String	Only 'user' timelineType is supported for producer

325.3. MESSAGE HEADERS

Name	Description
CamelTwitterKeywords	This header is used by the search producer to change the search key words dynamically.
CamelTwitterSearchLanguage	**Camel 2.11.0:** This header can override the option of**lang** which set the search language for the search endpoint dynamically
CamelTwitterCount	**Camel 2.11.0** This header can override the option of**count** which sets the max twitters that will be returned.
CamelTwitterNumberOfPages	**Camel 2.11.0** This header can override the option of**numberOfPages** which sets how many pages we want to twitter returns.

325.4. MESSAGE BODY

All message bodies utilize objects provided by the Twitter4J API.

325.5. USE CASES

NOTE

API Rate Limits: Twitter REST APIs encapsulated by Twitter4J are subjected to API Rate Limiting. You can find the per method limits in the API Rate Limits documentation. Note that endpoints/resources not listed in that page are default to 15 requests per allotted user per window.

325.5.1. To create a status update within your Twitter profile, send this producer a String body:

```
from("direct:foo")
  .to("twitter-timeline://user?consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]);
```

325.5.2. To poll, every 60 sec., all statuses on your home timeline:

```
from("twitter-timeline://home?type=polling&delay=60&consumerKey=[s]&consumerSecret=
[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

325.5.3. To search for all statuses with the keyword 'camel' only once:

```
from("twitter-search://foo?type=polling&keywords=camel&consumerKey=[s]&consumerSecret=
[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

325.5.4. Searching using a producer with static keywords:

```
from("direct:foo")
  .to("twitter-search://foo?keywords=camel&consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]");
```

325.5.5. Searching using a producer with dynamic keywords from header:

In the **bar** header we have the keywords we want to search, so we can assign this value to the **CamelTwitterKeywords** header:

```
from("direct:foo")
  .setHeader("CamelTwitterKeywords", header("bar"))
  .to("twitter-search://foo?consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]");
```

325.6. EXAMPLE

See also the Twitter Websocket Example.

325.7. SEE ALSO

- Configuring Camel

- Component
- Endpoint
- Getting Started
- Twitter Websocket Example

CHAPTER 326. TWITTER DIRECT MESSAGE COMPONENT

Available as of Camel version 2.10

The Twitter Direct Message Component consumes/produces a user's direct messages.

326.1. COMPONENT OPTIONS

The Twitter Direct Message component supports 9 options which are listed below.

Name	Description	Default	Type
accessToken (security)	The access token		String
accessTokenSecret (security)	The access token secret		String
consumerKey (security)	The consumer key		String
consumerSecret (security)	The consumer secret		String
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter.		String
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter.		int
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

326.2. ENDPOINT OPTIONS

The Twitter Direct Message endpoint is configured using URI syntax:

```
twitter-directmessage:user
```

with the following path and query parameters:

326.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
user	**Required** The user name to send a direct message. This will be ignored for consumer.		String

326.2.2. Query Parameters (42 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
type (consumer)	Endpoint type to use. Only streaming supports event type.	polling	EndpointType
distanceMetric (consumer)	Used by the non-stream geography search, to search by radius using the configured metrics. The unit can either be mi for miles, or km for kilometers. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.	km	String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
extendedMode (consumer)	Used for enabling full text from twitter (eg receive tweets that contains more than 140 characters).	true	boolean

Name	Description	Default	Type
latitude (consumer)	Used by the non-stream geography search to search by latitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
locations (consumer)	Bounding boxes, created by pairs of lat/lons. Can be used for streaming/filter. A pair is defined as lat,lon. And multiple paris can be separated by semi colon.		String
longitude (consumer)	Used by the non-stream geography search to search by longitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
radius (consumer)	Used by the non-stream geography search to search by radius. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
twitterStream (consumer)	To use a custom instance of TwitterStream		TwitterStream
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
count (filter)	Limiting number of results per page.		Integer
filterOld (filter)	Filter out old tweets, that has previously been polled. This state is stored in memory only, and based on last tweet id.	true	boolean
lang (filter)	The lang string ISO_639-1 which will be used for searching		String
numberOfPages (filter)	The number of pages result which you want camel-twitter to consume.	1	Integer

Name	Description	Default	Type
sinceId (filter)	The last tweet id which will be used for pulling the tweets. It is useful when the camel route is restarted after a long running.	1	long
userIds (filter)	To filter by user ids for streaming/filter. Multiple values can be separated by comma.		String
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll.	30000	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map

Name	Description	Default	Type
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
sortById (sort)	Sorts by id, so the oldest are first, and newest last.	true	boolean
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		Integer
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
accessToken (security)	The access token. Can also be configured on the TwitterComponent level instead.		String
accessTokenSecret (security)	The access secret. Can also be configured on the TwitterComponent level instead.		String
consumerKey (security)	The consumer key. Can also be configured on the TwitterComponent level instead.		String
consumerSecret (security)	The consumer secret. Can also be configured on the TwitterComponent level instead.		String

CHAPTER 327. TWITTER SEARCH COMPONENT

Available as of Camel version 2.10

The Twitter Search component consumes search results.

327.1. COMPONENT OPTIONS

The Twitter Search component supports 9 options which are listed below.

Name	Description	Default	Type
accessToken (security)	The access token		String
accessTokenSecret (security)	The access token secret		String
consumerKey (security)	The consumer key		String
consumerSecret (security)	The consumer secret		String
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter.		String
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter.		int
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

327.2. ENDPOINT OPTIONS

The Twitter Search endpoint is configured using URI syntax:

```
twitter-search:keywords
```

with the following path and query parameters:

327.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
keywords	**Required** The search keywords. Multiple values can be separated with comma.		String

327.2.2. Query Parameters (42 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
type (consumer)	Endpoint type to use. Only streaming supports event type.	polling	EndpointType
distanceMetric (consumer)	Used by the non-stream geography search, to search by radius using the configured metrics. The unit can either be mi for miles, or km for kilometers. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.	km	String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
extendedMode (consumer)	Used for enabling full text from twitter (eg receive tweets that contains more than 140 characters).	true	boolean

Name	Description	Default	Type
latitude (consumer)	Used by the non-stream geography search to search by latitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
locations (consumer)	Bounding boxes, created by pairs of lat/lons. Can be used for streaming/filter. A pair is defined as lat,lon. And multiple paris can be separated by semi colon.		String
longitude (consumer)	Used by the non-stream geography search to search by longitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
radius (consumer)	Used by the non-stream geography search to search by radius. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
twitterStream (consumer)	To use a custom instance of TwitterStream		TwitterStream
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
count (filter)	Limiting number of results per page.		Integer
filterOld (filter)	Filter out old tweets, that has previously been polled. This state is stored in memory only, and based on last tweet id.	true	boolean
lang (filter)	The lang string ISO_639-1 which will be used for searching		String
numberOfPages (filter)	The number of pages result which you want camel-twitter to consume.	1	Integer

Name	Description	Default	Type
sinceId (filter)	The last tweet id which will be used for pulling the tweets. It is useful when the camel route is restarted after a long running.	1	long
userIds (filter)	To filter by user ids for streaming/filter. Multiple values can be separated by comma.		String
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll.	30000	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map

Name	Description	Default	Type
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
sortById (sort)	Sorts by id, so the oldest are first, and newest last.	true	boolean
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		Integer
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
accessToken (security)	The access token. Can also be configured on the TwitterComponent level instead.		String
accessTokenSecret (security)	The access secret. Can also be configured on the TwitterComponent level instead.		String
consumerKey (security)	The consumer key. Can also be configured on the TwitterComponent level instead.		String
consumerSecret (security)	The consumer secret. Can also be configured on the TwitterComponent level instead.		String

CHAPTER 328. TWITTER STREAMING COMPONENT

Available as of Camel version 2.10

The Twitter Streaming component consumes twitter statuses using Streaming API.

328.1. COMPONENT OPTIONS

The Twitter Streaming component supports 9 options which are listed below.

Name	Description	Default	Type
accessToken (security)	The access token		String
accessTokenSecret (security)	The access token secret		String
consumerKey (security)	The consumer key		String
consumerSecret (security)	The consumer secret		String
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter.		String
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter.		int
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

328.2. ENDPOINT OPTIONS

The Twitter Streaming endpoint is configured using URI syntax:

```
twitter-streaming:streamingType
```

with the following path and query parameters:

328.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
streamingType	**Required** The streaming type to consume.		StreamingType

328.2.2. Query Parameters (43 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
type (consumer)	Endpoint type to use. Only streaming supports event type.	polling	EndpointType
distanceMetric (consumer)	Used by the non-stream geography search, to search by radius using the configured metrics. The unit can either be mi for miles, or km for kilometers. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.	km	String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
extendedMode (consumer)	Used for enabling full text from twitter (eg receive tweets that contains more than 140 characters).	true	boolean

Name	Description	Default	Type
latitude (consumer)	Used by the non-stream geography search to search by latitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
locations (consumer)	Bounding boxes, created by pairs of lat/lons. Can be used for streaming/filter. A pair is defined as lat,lon. And multiple paris can be separated by semi colon.		String
longitude (consumer)	Used by the non-stream geography search to search by longitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
radius (consumer)	Used by the non-stream geography search to search by radius. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
twitterStream (consumer)	To use a custom instance of TwitterStream		TwitterStream
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
count (filter)	Limiting number of results per page.		Integer
filterOld (filter)	Filter out old tweets, that has previously been polled. This state is stored in memory only, and based on last tweet id.	true	boolean
keywords (filter)	Can be used for a streaming filter. Multiple values can be separated with comma.		String
lang (filter)	The lang string ISO_639-1 which will be used for searching		String
numberOfPages (filter)	The number of pages result which you want camel-twitter to consume.	1	Integer

Name	Description	Default	Type
sinceId (filter)	The last tweet id which will be used for pulling the tweets. It is useful when the camel route is restarted after a long running.	1	long
userIds (filter)	To filter by user ids for streaming/filter. Multiple values can be separated by comma.		String
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll.	30000	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map

Name	Description	Default	Type
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
sortById (sort)	Sorts by id, so the oldest are first, and newest last.	true	boolean
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		Integer
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
accessToken (security)	The access token. Can also be configured on the TwitterComponent level instead.		String
accessTokenSecret (security)	The access secret. Can also be configured on the TwitterComponent level instead.		String
consumerKey (security)	The consumer key. Can also be configured on the TwitterComponent level instead.		String
consumerSecret (security)	The consumer secret. Can also be configured on the TwitterComponent level instead.		String

CHAPTER 329. TWITTER TIMELINE COMPONENT

Available as of Camel version 2.10

The Twitter Timeline component consumes twitter timeline or update the status of specific user.

329.1. COMPONENT OPTIONS

The Twitter Timeline component supports 9 options which are listed below.

Name	Description	Default	Type
accessToken (security)	The access token		String
accessTokenSecret (security)	The access token secret		String
consumerKey (security)	The consumer key		String
consumerSecret (security)	The consumer secret		String
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter.		String
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter.		int
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

329.2. ENDPOINT OPTIONS

The Twitter Timeline endpoint is configured using URI syntax:

```
twitter-timeline:timelineType
```

with the following path and query parameters:

329.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
timelineType	**Required** The timeline type to produce/consume.		TimelineType

329.2.2. Query Parameters (43 parameters):

Name	Description	Default	Type
user (common)	The username when using timelineType=user		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
type (consumer)	Endpoint type to use. Only streaming supports event type.	polling	EndpointType
distanceMetric (consumer)	Used by the non-stream geography search, to search by radius using the configured metrics. The unit can either be mi for miles, or km for kilometers. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.	km	String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
extendedMode (consumer)	Used for enabling full text from twitter (eg receive tweets that contains more than 140 characters).	true	boolean

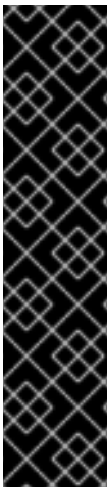
Name	Description	Default	Type
latitude (consumer)	Used by the non-stream geography search to search by latitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
locations (consumer)	Bounding boxes, created by pairs of lat/lons. Can be used for streaming/filter. A pair is defined as lat,lon. And multiple paris can be separated by semi colon.		String
longitude (consumer)	Used by the non-stream geography search to search by longitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
radius (consumer)	Used by the non-stream geography search to search by radius. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
twitterStream (consumer)	To use a custom instance of TwitterStream		TwitterStream
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
count (filter)	Limiting number of results per page.		Integer
filterOld (filter)	Filter out old tweets, that has previously been polled. This state is stored in memory only, and based on last tweet id.	true	boolean
lang (filter)	The lang string ISO_639-1 which will be used for searching		String
numberOfPages (filter)	The number of pages result which you want camel-twitter to consume.	1	Integer

Name	Description	Default	Type
sinceId (filter)	The last tweet id which will be used for pulling the tweets. It is useful when the camel route is restarted after a long running.	1	long
userIds (filter)	To filter by user ids for streaming/filter. Multiple values can be separated by comma.		String
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll.	30000	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler

Name	Description	Default	Type
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
sortById (sort)	Sorts by id, so the oldest are first, and newest last.	true	boolean
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		Integer
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
accessToken (security)	The access token. Can also be configured on the TwitterComponent level instead.		String
accessTokenSecret (security)	The access secret. Can also be configured on the TwitterComponent level instead.		String
consumerKey (security)	The consumer key. Can also be configured on the TwitterComponent level instead.		String
consumerSecret (security)	The consumer secret. Can also be configured on the TwitterComponent level instead.		String

CHAPTER 330. TWITTER COMPONENT (DEPRECATED)

Available as of Camel version 2.10



IMPORTANT

The composite twitter component has been deprecated. Use individual component for directmessage, search, streaming and timeline.

- Twitter Components
  - Twitter Direct Message
  - Twitter Search
  - Twitter Streaming
  - Twitter Timeline

The Twitter component enables the most useful features of the Twitter API by encapsulating Twitter4J. It allows direct, polling, or event-driven consumption of timelines, users, trends, and direct messages. Also, it supports producing messages as status updates or direct messages.

Twitter now requires the use of OAuth for all client application authentication. In order to use camel-twitter with your account, you'll need to create a new application within Twitter at https://dev.twitter.com/apps/new and grant the application access to your account. Finally, generate your access token and secret.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-twitter</artifactId>
  <version>${camel-version}</version>
</dependency>
```

330.1. URI FORMAT

```
twitter://endpoint[?options]
```

330.2. TWITTER COMPONENT

The twitter component can be configured with the Twitter account settings which is mandatory to configure before using.

The Twitter component supports 9 options which are listed below.

Name	Description	Default	Type
accessToken (security)	The access token		String

Name	Description	Default	Type
accessTokenSecret (security)	The access token secret		String
consumerKey (security)	The consumer key		String
consumerSecret (security)	The consumer secret		String
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter.		String
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter.		int
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

You can also configure these options directly in the endpoint.

330.3. CONSUMER ENDPOINTS

Rather than the endpoints returning a List through one single route exchange, camel-twitter creates one route exchange per returned object. As an example, if "timeline/home" results in five statuses, the route will be executed five times (one for each Status).

Endpoint	Context	Body Type	Notice
directmessage	direct, polling	twitter4j.DirectMessage	
search	direct, polling	twitter4j.Status	

Endpoint	Context	Body Type	Notice
streaming/filter	event, polling	twitter4j.Status	
streaming/sample	event, polling	twitter4j.Status	
streaming/user	event, polling	twitter4j.Status	**Camel 2.16**: To receive tweets from protected users and accounts.
timeline/home	direct, polling	twitter4j.Status	
timeline/mentions	direct, polling	twitter4j.Status	
timeline/public	**direct, polling**	**twitter4j.Status**	**@deprecated. Use timeline/home or direct/home instead. Removed from *Camel 2.11** onwards.*
timeline/retweetsofme	direct, polling	twitter4j.Status	
timeline/user	direct, polling	twitter4j.Status	
trends/daily	***Camel 2.10.1:** direct, polling*	**twitter4j.Status**	**@deprecated. Removed from Camel 2.11 onwards.**
trends/weekly	***Camel 2.10.1:** direct, polling*	**twitter4j.Status**	**@deprecated. Removed from Camel 2.11 onwards.**

330.4. PRODUCER ENDPOINTS

Endpoint	Body Type
directmessage	String
search	List<twitter4j.Status>
timeline/user	String

330.5. URI OPTIONS

The Twitter endpoint is configured using URI syntax:

```
twitter:kind
```

with the following path and query parameters:

330.5.1. Path Parameters (1 parameters):

Name	Description	Default	Type
kind	**Required** The kind of endpoint		String

330.5.2. Query Parameters (44 parameters):

Name	Description	Default	Type
user (common)	Username, used for user timeline consumption, direct message production, etc.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
type (consumer)	Endpoint type to use. Only streaming supports event type.	polling	EndpointType

Name	Description	Default	Type
distanceMetric (consumer)	Used by the non-stream geography search, to search by radius using the configured metrics. The unit can either be mi for miles, or km for kilometers. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.	km	String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
extendedMode (consumer)	Used for enabling full text from twitter (eg receive tweets that contains more than 140 characters).	true	boolean
latitude (consumer)	Used by the non-stream geography search to search by latitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
locations (consumer)	Bounding boxes, created by pairs of lat/lons. Can be used for streaming/filter. A pair is defined as lat,lon. And multiple paris can be separated by semi colon.		String
longitude (consumer)	Used by the non-stream geography search to search by longitude. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
radius (consumer)	Used by the non-stream geography search to search by radius. You need to configure all the following options: longitude, latitude, radius, and distanceMetric.		Double
twitterStream (consumer)	To use a custom instance of TwitterStream		TwitterStream

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
count (filter)	Limiting number of results per page.		Integer
filterOld (filter)	Filter out old tweets, that has previously been polled. This state is stored in memory only, and based on last tweet id.	true	boolean
keywords (filter)	Can be used for search and streaming/filter. Multiple values can be separated with comma.		String
lang (filter)	The lang string ISO_639-1 which will be used for searching		String
numberOfPages (filter)	The number of pages result which you want camel-twitter to consume.	1	Integer
sinceId (filter)	The last tweet id which will be used for pulling the tweets. It is useful when the camel route is restarted after a long running.	1	long
userIds (filter)	To filter by user ids for streaming/filter. Multiple values can be separated by comma.		String
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll.	30000	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean

Name	Description	Default	Type
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
sortById (sort)	Sorts by id, so the oldest are first, and newest last.	true	boolean
httpProxyHost (proxy)	The http proxy host which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPassword (proxy)	The http proxy password which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
httpProxyPort (proxy)	The http proxy port which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		Integer

Name	Description	Default	Type
httpProxyUser (proxy)	The http proxy user which can be used for the camel-twitter. Can also be configured on the TwitterComponent level instead.		String
accessToken (security)	The access token. Can also be configured on the TwitterComponent level instead.		String
accessTokenSecret (security)	The access secret. Can also be configured on the TwitterComponent level instead.		String
consumerKey (security)	The consumer key. Can also be configured on the TwitterComponent level instead.		String
consumerSecret (security)	The consumer secret. Can also be configured on the TwitterComponent level instead.		String

330.6. MESSAGE HEADERS

Name	Description
CamelTwitterKeywords	This header is used by the search producer to change the search key words dynamically.
CamelTwitterSearchLanguage	**Camel 2.11.0:** This header can override the option of**lang** which set the search language for the search endpoint dynamically
CamelTwitterCount	**Camel 2.11.0** This header can override the option of**count** which sets the max twitters that will be returned.
CamelTwitterNumberOfPages	**Camel 2.11.0** This header can override the option of**numberOfPages** which sets how many pages we want to twitter returns.

330.7. MESSAGE BODY

All message bodies utilize objects provided by the Twitter4J API.

330.8. USE CASES

NOTE

API Rate Limits: Twitter REST APIs encapsulated by Twitter4J are subjected to API Rate Limiting. You can find the per method limits in the API Rate Limits documentation. Note that endpoints/resources not listed in that page are default to 15 requests per allotted user per window.

330.8.1. To create a status update within your Twitter profile, send this producer a String body:

```
from("direct:foo")
  .to("twitter://timeline/user?consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]);
```

330.8.2. To poll, every 60 sec., all statuses on your home timeline:

```
from("twitter://timeline/home?type=polling&delay=60&consumerKey=[s]&consumerSecret=
[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

330.8.3. To search for all statuses with the keyword 'camel' only once:

```
from("twitter://search?type=polling&keywords=camel&consumerKey=[s]&consumerSecret=
[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

330.8.4. Searching using a producer with static keywords:

```
from("direct:foo")
  .to("twitter://search?keywords=camel&consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]");
```

330.8.5. Searching using a producer with dynamic keywords from header:

In the **bar** header we have the keywords we want to search, so we can assign this value to the **CamelTwitterKeywords** header:

```
from("direct:foo")
  .setHeader("CamelTwitterKeywords", header("bar"))
  .to("twitter://search?consumerKey=[s]&consumerSecret=[s]&accessToken=
[s]&accessTokenSecret=[s]");
```

330.9. EXAMPLE

See also the Twitter Websocket Example.

330.10. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started
- Twitter Websocket Example

CHAPTER 331. UNDERTOW COMPONENT

Available as of Camel version 2.16

The **undertow** component provides HTTP and WebSocket based endpoints for consuming and producing HTTP/WebSocket requests.

That is, the Undertow component behaves as a simple Web server. Undertow can also be used as a http client which mean you can also use it with Camel as a producer.

TIP

Since Camel version 2.21, the **undertow** component also supports WebSocket connections and can thus serve as a drop-in replacement for Camel websocket component or atmosphere-websocket component.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-undertow</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

331.1. URI FORMAT

```
undertow:http://hostname[:port][/resourceUri][?options]
undertow:https://hostname[:port][/resourceUri][?options]
undertow:ws://hostname[:port][/resourceUri][?options]
undertow:wss://hostname[:port][/resourceUri][?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

331.2. OPTIONS

The Undertow component supports 5 options which are listed below.

Name	Description	Default	Type
undertowHttpBinding (advanced)	To use a custom HttpBinding to control the mapping between Camel message and HttpClient.		UndertowHttpBinding
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean

Name	Description	Default	Type
hostOptions (advanced)	To configure common options, such as thread pools		UndertowHostOptions
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Undertow endpoint is configured using URI syntax:

```
undertow:httpURI
```

with the following path and query parameters:

331.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
httpURI	**Required** The url of the HTTP endpoint to use.		URI

331.2.2. Query Parameters (21 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
httpMethodRestrict (consumer)	Used to only allow consuming if the HttpMethod matches, such as GET/POST/PUT etc. Multiple methods can be specified separated by comma.		String
matchOnUriPrefix (consumer)	Whether or not the consumer should try to find a target consumer by matching the URI prefix if no exact match is found.	false	Boolean

Name	Description	Default	Type
optionsEnabled (consumer)	Specifies whether to enable HTTP OPTIONS for this Servlet consumer. By default OPTIONS is turned off.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
cookieHandler (producer)	Configure a cookie handler to maintain a HTTP session		CookieHandler
keepAlive (producer)	Setting to ensure socket is not closed due to inactivity	true	Boolean
options (producer)	Sets additional channel options. The options that can be used are defined in org.xnio.Options. To configure from endpoint uri, then prefix each option with option., such as option.close-abort=true&option.send-buffer=8192		Map
reuseAddresses (producer)	Setting to facilitate socket multiplexing	true	Boolean
tcpNoDelay (producer)	Setting to improve TCP protocol performance	true	Boolean
throwExceptionOnFailure (producer)	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.	true	Boolean
transferException (producer)	If enabled and an Exchange failed processing on the consumer side and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type. On the producer side the exception will be deserialized and thrown as is instead of the HttpOperationFailedException. The caused exception is required to be serialized. This is by default turned off. If you enable this then be aware that Java will deserialize the incoming data from the request to Java and that can be a potential security risk.	false	Boolean

Name	Description	Default	Type
headerFilterStrategy (advanced)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
undertowHttpBinding (advanced)	To use a custom UndertowHttpBinding to control the mapping between Camel message and undertow.		UndertowHttpBinding
fireWebSocketChannelEvents (websocket)	if true, the consumer will post notifications to the route when a new WebSocket peer connects, disconnects, etc. See UndertowConstants.EVENT_TYPE and EventType.	false	boolean
sendTimeout (websocket)	Timeout in milliseconds when sending to a websocket channel. The default timeout is 30000 (30 seconds).	30000	Integer
sendToAll (websocket)	To send to all websocket subscribers. Can be used to configure on endpoint level, instead of having to use the UndertowConstants.SEND_TO_ALL header on the message.		Boolean
useStreaming (websocket)	if true, text and binary messages coming through a WebSocket will be wrapped as java.io.Reader and java.io.InputStream respectively before they are passed to an Exchange; otherwise they will be passed as String and byte respectively.	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters

331.3. MESSAGE HEADERS

Camel uses the same message headers as the HTTP component. From Camel 2.2, it also uses **Exchange.HTTP_CHUNKED,CamelHttpChunked** header to turn on or turn off the chuched encoding on the camel-undertow consumer.

Camel also populates **all** request.parameter and request.headers. For example, given a client request with the URL, **http://myserver/myserver?orderid=123**, the exchange will contain a header named **orderid** with the value 123.

331.4. HTTP PRODUCER EXAMPLE

The following is a basic example of how to send an HTTP request to an existing HTTP endpoint.

in Java DSL

```
from("direct:start")
    .to("undertow:http://www.google.com");
```

or in XML

```
<route>
   <from uri="direct:start"/>
   <to uri="undertow:http://www.google.com"/>
<route>
```

331.5. HTTP CONSUMER EXAMPLE

In this sample we define a route that exposes a HTTP service at **http://localhost:8080/myapp/myservice**:

```
<route>
  <from uri="undertow:http://localhost:8080/myapp/myservice"/>
  <to uri="bean:myBean"/>
</route>
```

331.6. WEBSOCKET EXAMPLE

In this sample we define a route that exposes a WebSocket service at **http://localhost:8080/myapp/mysocket** and returns back a response to the same channel:

```
<route>
  <from uri="undertow:ws://localhost:8080/myapp/mysocket"/>
  <transform><simple>Echo ${body}</simple></transform>
  <to uri="undertow:ws://localhost:8080/myapp/mysocket"/>
</route>
```

331.7. USING LOCALHOST AS HOST

When you specify **localhost** in a URL, Camel exposes the endpoint only on the local TCP/IP network interface, so it cannot be accessed from outside the machine it operates on.

If you need to expose a Jetty endpoint on a specific network interface, the numerical IP address of this interface should be used as the host. If you need to expose a Jetty endpoint on all network interfaces, the **0.0.0.0** address should be used.

To listen across an entire URI prefix, see How do I let Jetty match wildcards .

If you actually want to expose routes by HTTP and already have a Servlet, you should instead refer to the Servlet Transport.

331.8. UNDERTOW CONSUMERS ON {WILDFLY}

The configuration of camel-undertow consumers on {wildfly} is different to that of standalone Camel. Producer endpoints work as per normal.

On {wildfly}, camel-undertow consumers leverage the default Undertow HTTP server provided by the container. The server is defined within the undertow subsystem configuration. Here's an excerpt of the default configuration from standalone.xml:

```
<subsystem xmlns="urn:jboss:domain:undertow:4.0">
    <buffer-cache name="default" />
    <server name="default-server">
        <http-listener name="default" socket-binding="http" redirect-socket="https" enable-http2="true" />
        <https-listener name="https" socket-binding="https" security-realm="ApplicationRealm" enable-
http2="true" />
        <host name="default-host" alias="localhost">
            <location name="/" handler="welcome-content" />
            <filter-ref name="server-header" />
            <filter-ref name="x-powered-by-header" />
            <http-invoker security-realm="ApplicationRealm" />
        </host>
    </server>
</subsystem>
```

In this instance, Undertow is configured to listen on interfaces / ports specified by the http and https socket-binding. By default this is port 8080 for http and 8443 for https.

This has the following implications:

- camel-undertow consumers will only bind to localhost:8080 or localhost:8443
- Some endpoint consumer configuration options have no effect (see below), since these settings are managed by the {wildfly} container

For example, if you configure an endpoint consumer using different host or port combinations, a warning will appear within the server log file. For example the following host & port configurations would be ignored:

```
from("undertow:http://somehost:1234/path/to/resource")
```

```
[org.wildfly.extension.camel] (pool-2-thread-1) Ignoring configured host:
http://somehost:1234/path/to/resource
```

However, the consumer is still available on the default host & port localhost:8080 or localhost:8443.

331.8.1. Configuring alternative ports

If alternative ports are to be accepted, then these must be configured via the {wildfly} subsystem configuration. This is explained in the server documentation:

https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.1/html/configuration_guide/configuring_the_web_ser

331.8.2. Ignored camel-undertow consumer configuration options on {wildfly}

hostOptions

Refer to the {wildfly} undertow configuration guide for how to configure server host options:

https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.1/html-single/how_to_configure_server_security/#configure_one_way_and_two_way_ssl_tls_for_application

sslContextParameters

To configure SSL, refer to the {wildfly} SSL configuration guide:

https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.1/html-single/how_to_configure_server_security/#configure_one_way_and_two_way_ssl_tls_for_application

CHAPTER 332. UNIVOCITY CSV DATAFORMAT

Available as of Camel version 2.15

This Data Format uses uniVocity-parsers for reading and writing 3 kinds of tabular data text files:

- CSV (Comma Separated Values), where the values are separated by a symbol (usually a comma)
- fixed-width, where the values have known sizes
- TSV (Tabular Separated Values), where the fields are separated by a tabulation

Thus there are 3 data formats based on uniVocity-parsers.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest and greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-univocity-parsers</artifactId>
  <version>x.x.x</version>
</dependency>
```

332.1. OPTIONS

Most configuration options of the uniVocity-parsers are available in the data formats. If you want more information about a particular option, please refer to their documentation page.

The 3 data formats share common options and have dedicated ones, this section presents them all.

332.2. OPTIONS

The uniVocity CSV dataformat supports 18 options which are listed below.

Name	Default	Java Type	Description
quoteAllFields	**false**	**Boolean**	Whether or not all values must be quoted when writing them.
quote	**"**	**String**	The quote symbol.
quoteEscape	**"**	**String**	The quote escape symbol
delimiter	**,**	**String**	The delimiter of values
nullValue		**String**	The string representation of a null value. The default value is null
skipEmptyLines	**true**	**Boolean**	Whether or not the empty lines must be ignored. The default value is true

Name	Default	Java Type	Description
ignoreTrailingWhitespaces	true	Boolean	Whether or not the trailing white spaces must ignored. The default value is true
ignoreLeadingWhitespaces	true	Boolean	Whether or not the leading white spaces must be ignored. The default value is true
headersDisabled	false	Boolean	Whether or not the headers are disabled. When defined, this option explicitly sets the headers as null which indicates that there is no header. The default value is false
headerExtractionEnabled	false	Boolean	Whether or not the header must be read in the first line of the test document The default value is false
numberOfRecordsToRead		Integer	The maximum number of record to read.
emptyValue		String	The String representation of an empty value
lineSeparator		String	The line separator of the files The default value is to use the JVM platform line separator
normalizedLineSeparator		String	The normalized line separator of the files The default value is a new line character.
comment	#	String	The comment symbol. The default value is
lazyLoad	false	Boolean	Whether the unmarshalling should produce an iterator that reads the lines on the fly or if all the lines must be read at one. The default value is false
asMap	false	Boolean	Whether the unmarshalling should produce maps for the lines values instead of lists. It requires to have header (either defined or collected). The default value is false
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

332.3. MARSHALLING USAGES

The marshalling accepts either:

- A list of maps (L`ist<Map<String, ?>>`), one for each line

- A single map (**Map<String, ?>**), for a single line

Any other body will throws an exception.

332.3.1. Usage example: marshalling a Map into CSV format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-csv/>
  </marshal>
  <to uri="mock:result"/>
</route>
```

332.3.2. Usage example: marshalling a Map into fixed-width format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-fixed padding="_">
      <univocity-header length="5"/>
      <univocity-header length="5"/>
      <univocity-header length="5"/>
    </univocity-fixed>
  </marshal>
  <to uri="mock:result"/>
</route>
```

332.3.3. Usage example: marshalling a Map into TSV format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-tsv/>
  </marshal>
  <to uri="mock:result"/>
</route>
```

332.4. UNMARSHALLING USAGES

The unmarshalling uses an **InputStream** in order to read the data.

Each row produces either:

- a list with all the values in it (**asMap** option with **false**);
- A map with all the values indexed by the headers (**asMap** option with **true**).

All the rows can either:

- be collected at once into a list (**lazyLoad** option with **false**);
- be read on the fly using an iterator (**lazyLoad** option with **true**).

332.4.1. Usage example: unmarshalling a CSV format into maps with automatic headers

```
<route>
  <from uri="direct:input"/>
  <unmarshal>
    <univocity-csv headerExtractionEnabled="true" asMap="true"/>
  </unmarshal>
  <to uri="mock:result"/>
</route>
```

332.4.2. Usage example: unmarshalling a fixed-width format into lists

```
<route>
  <from uri="direct:input"/>
  <unmarshal>
    <univocity-fixed>
      <univocity-header length="5"/>
      <univocity-header length="5"/>
      <univocity-header length="5"/>
    </univocity-fixed>
  </unmarshal>
  <to uri="mock:result"/>
</route>
```

CHAPTER 333. UNIVOCITY FIXED LENGTH DATAFORMAT

Available as of Camel version 2.15

This Data Format uses uniVocity-parsers for reading and writing 3 kinds of tabular data text files:

- CSV (Comma Separated Values), where the values are separated by a symbol (usually a comma)
- fixed-width, where the values have known sizes
- TSV (Tabular Separated Values), where the fields are separated by a tabulation

Thus there are 3 data formats based on uniVocity-parsers.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest and greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-univocity-parsers</artifactId>
  <version>x.x.x</version>
</dependency>
```

333.1. OPTIONS

Most configuration options of the uniVocity-parsers are available in the data formats. If you want more information about a particular option, please refer to their documentation page.

The 3 data formats share common options and have dedicated ones, this section presents them all.

333.2. OPTIONS

The uniVocity Fixed Length dataformat supports 17 options which are listed below.

Name	Default	Java Type	Description
skipTrailingCharsUntilNewline	**false**	**Boolean**	Whether or not the trailing characters until new line must be ignored. The default value is false
recordEndsOnNewline	**false**	**Boolean**	Whether or not the record ends on new line. The default value is false
padding		**String**	The padding character. The default value is a space
nullValue		**String**	The string representation of a null value. The default value is null
skipEmptyLines	**true**	**Boolean**	Whether or not the empty lines must be ignored. The default value is true

Name	Default	Java Type	Description
ignoreTrailingWhitespaces	true	Boolean	Whether or not the trailing white spaces must ignored. The default value is true
ignoreLeadingWhitespaces	true	Boolean	Whether or not the leading white spaces must be ignored. The default value is true
headersDisabled	false	Boolean	Whether or not the headers are disabled. When defined, this option explicitly sets the headers as null which indicates that there is no header. The default value is false
headerExtractionEnabled	false	Boolean	Whether or not the header must be read in the first line of the test document The default value is false
numberOfRecordsToRead		Integer	The maximum number of record to read.
emptyValue		String	The String representation of an empty value
lineSeparator		String	The line separator of the files The default value is to use the JVM platform line separator
normalizedLineSeparator		String	The normalized line separator of the files The default value is a new line character.
comment	#	String	The comment symbol. The default value is
lazyLoad	false	Boolean	Whether the unmarshalling should produce an iterator that reads the lines on the fly or if all the lines must be read at one. The default value is false
asMap	false	Boolean	Whether the unmarshalling should produce maps for the lines values instead of lists. It requires to have header (either defined or collected). The default value is false
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

333.3. MARSHALLING USAGES

The marshalling accepts either:

- A list of maps (L`ist<Map<String, ?>>`), one for each line
- A single map (**Map<String, ?>**), for a single line

Any other body will throws an exception.

333.3.1. Usage example: marshalling a Map into CSV format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-csv/>
  </marshal>
  <to uri="mock:result"/>
</route>
```

333.3.2. Usage example: marshalling a Map into fixed-width format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-fixed padding="_">
      <univocity-header length="5"/>
      <univocity-header length="5"/>
      <univocity-header length="5"/>
    </univocity-fixed>
  </marshal>
  <to uri="mock:result"/>
</route>
```

333.3.3. Usage example: marshalling a Map into TSV format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-tsv/>
  </marshal>
  <to uri="mock:result"/>
</route>
```

333.4. UNMARSHALLING USAGES

The unmarshalling uses an **InputStream** in order to read the data.

Each row produces either:

- a list with all the values in it (**asMap** option with **false**);
- A map with all the values indexed by the headers (**asMap** option with **true**).

All the rows can either:

- be collected at once into a list (**lazyLoad** option with **false**);
- be read on the fly using an iterator (**lazyLoad** option with **true**).

333.4.1. Usage example: unmarshalling a CSV format into maps with automatic headers

```
<route>
  <from uri="direct:input"/>
  <unmarshal>
    <univocity-csv headerExtractionEnabled="true" asMap="true"/>
  </unmarshal>
  <to uri="mock:result"/>
</route>
```

333.4.2. Usage example: unmarshalling a fixed-width format into lists

```
<route>
  <from uri="direct:input"/>
  <unmarshal>
    <univocity-fixed>
      <univocity-header length="5"/>
      <univocity-header length="5"/>
      <univocity-header length="5"/>
    </univocity-fixed>
  </unmarshal>
  <to uri="mock:result"/>
</route>
```

CHAPTER 334. UNIVOCITY TSV DATAFORMAT

Available as of Camel version 2.15

This Data Format uses uniVocity-parsers for reading and writing 3 kinds of tabular data text files:

- CSV (Comma Separated Values), where the values are separated by a symbol (usually a comma)
- fixed-width, where the values have known sizes
- TSV (Tabular Separated Values), where the fields are separated by a tabulation

Thus there are 3 data formats based on uniVocity-parsers.

If you use Maven you can just add the following to your pom.xml, substituting the version number for the latest and greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-univocity-parsers</artifactId>
  <version>x.x.x</version>
</dependency>
```

334.1. OPTIONS

Most configuration options of the uniVocity-parsers are available in the data formats. If you want more information about a particular option, please refer to their documentation page.

The 3 data formats share common options and have dedicated ones, this section presents them all.

334.2. OPTIONS

The uniVocity TSV dataformat supports 15 options which are listed below.

Name	Default	Java Type	Description
escapeChar	\	**String**	The escape character.
nullValue		**String**	The string representation of a null value. The default value is null
skipEmptyLines	**true**	**Boolean**	Whether or not the empty lines must be ignored. The default value is true
ignoreTrailingWhitespaces	**true**	**Boolean**	Whether or not the trailing white spaces must ignored. The default value is true
ignoreLeadingWhitespaces	**true**	**Boolean**	Whether or not the leading white spaces must be ignored. The default value is true

Name	Default	Java Type	Description
headersDisabled	false	Boolean	Whether or not the headers are disabled. When defined, this option explicitly sets the headers as null which indicates that there is no header. The default value is false
headerExtractionEnabled	false	Boolean	Whether or not the header must be read in the first line of the test document The default value is false
numberOfRecordsToRead		Integer	The maximum number of record to read.
emptyValue		String	The String representation of an empty value
lineSeparator		String	The line separator of the files The default value is to use the JVM platform line separator
normalizedLineSeparator		String	The normalized line separator of the files The default value is a new line character.
comment	#	String	The comment symbol. The default value is
lazyLoad	false	Boolean	Whether the unmarshalling should produce an iterator that reads the lines on the fly or if all the lines must be read at one. The default value is false
asMap	false	Boolean	Whether the unmarshalling should produce maps for the lines values instead of lists. It requires to have header (either defined or collected). The default value is false
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

334.3. MARSHALLING USAGES

The marshalling accepts either:

- A list of maps (L`ist<Map<String, ?>>`), one for each line
- A single map (**Map<String, ?>**), for a single line

Any other body will throws an exception.

334.3.1. Usage example: marshalling a Map into CSV format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-csv/>
  </marshal>
  <to uri="mock:result"/>
</route>
```

334.3.2. Usage example: marshalling a Map into fixed-width format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-fixed padding="_">
      <univocity-header length="5"/>
      <univocity-header length="5"/>
      <univocity-header length="5"/>
    </univocity-fixed>
  </marshal>
  <to uri="mock:result"/>
</route>
```

334.3.3. Usage example: marshalling a Map into TSV format

```
<route>
  <from uri="direct:input"/>
  <marshal>
    <univocity-tsv/>
  </marshal>
  <to uri="mock:result"/>
</route>
```

334.4. UNMARSHALLING USAGES

The unmarshalling uses an **InputStream** in order to read the data.

Each row produces either:

- a list with all the values in it (**asMap** option with **false**);
- A map with all the values indexed by the headers (**asMap** option with **true**).

All the rows can either:

- be collected at once into a list (**lazyLoad** option with **false**);
- be read on the fly using an iterator (**lazyLoad** option with **true**).

334.4.1. Usage example: unmarshalling a CSV format into maps with automatic headers

```
<route>
```

```
    <from uri="direct:input"/>
    <unmarshal>
        <univocity-csv headerExtractionEnabled="true" asMap="true"/>
    </unmarshal>
    <to uri="mock:result"/>
</route>
```

334.4.2. Usage example: unmarshalling a fixed-width format into lists

```
<route>
    <from uri="direct:input"/>
    <unmarshal>
        <univocity-fixed>
            <univocity-header length="5"/>
            <univocity-header length="5"/>
            <univocity-header length="5"/>
        </univocity-fixed>
    </unmarshal>
    <to uri="mock:result"/>
</route>
```

CHAPTER 335. VALIDATOR COMPONENT

Available as of Camel version 1.1

The Validation component performs XML validation of the message body using the JAXP Validation API and based on any of the supported XML schema languages, which defaults to XML Schema

Note that the Jing component also supports the following useful schema languages:

- RelaxNG Compact Syntax
- RelaxNG XML Syntax

The MSV component also supports RelaxNG XML Syntax.

335.1. URI FORMAT

```
validator:someLocalOrRemoteResource
```

Where **someLocalOrRemoteResource** is some URL to a local resource on the classpath or a full URL to a remote resource or resource on the file system which contains the XSD to validate against. For example:

- **msv:org/foo/bar.xsd**
- **msv:file:../foo/bar.xsd**
- **msv:http://acme.com/cheese.xsd**
- **validator:com/mypackage/myschema.xsd**

Maven users will need to add the following dependency to their **pom.xml** for this component when using **Camel 2.8** or older:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

From Camel 2.9 onwards the Validation component is provided directly in the camel-core.

335.2. OPTIONS

The Validator component supports 2 options which are listed below.

Name	Description	Default	Type
resourceResolverFactory (advanced)	To use a custom LSResourceResolver which depends on a dynamic endpoint resource URI		ValidatorResourceResolverFactory

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Validator endpoint is configured using URI syntax:

```
validator:resourceUri
```

with the following path and query parameters:

335.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** URL to a local resource on the classpath,or a reference to lookup a bean in the Registry, or a full URL to a remote resource or resource on the file system which contains the XSD to validate against.		String

335.2.2. Query Parameters (11 parameters):

Name	Description	Default	Type
failOnNullBody (producer)	Whether to fail if no body exists.	true	boolean
failOnNullHeader (producer)	Whether to fail if no header exists when validating against a header.	true	boolean
headerName (producer)	To validate against a header instead of the message body.		String
errorHandler (advanced)	To use a custom org.apache.camel.processor.validation.ValidatorErrorHandler. The default error handler captures the errors and throws an exception.		ValidatorErrorHandler
resourceResolver (advanced)	To use a custom LSResourceResolver. See also link setResourceResolverFactory(ValidatorResourceResolverFactory)		LSResourceResolver

Name	Description	Default	Type
resourceResolverFactory (advanced)	For creating a resource resolver which depends on the endpoint resource URI. Must not be used in combination with method link setResourceResolver(LSResourceResolver). If not set then DefaultValidatorResourceResolverFactory is used		ValidatorResourceResolverFactory
schemaFactory (advanced)	To use a custom javax.xml.validation.SchemaFactory		SchemaFactory
schemaLanguage (advanced)	Configures the W3C XML Schema Namespace URI.	http://www.w3.org/2001/XMLSchema	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
useDom (advanced)	Whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator.	false	boolean
useSharedSchema (advanced)	Whether the Schema instance should be shared or not. This option is introduced to work around a JDK 1.6.x bug. Xerces should not have this issue.	true	boolean

335.3. EXAMPLE

The following example shows how to configure a route from endpoint **direct:start** which then goes to one of two endpoints, either **mock:valid** or **mock:invalid** based on whether or not the XML matches the given schema (which is supplied on the classpath).

335.4. ADVANCED: JMX METHOD CLEARCACHEDSCHEMA

Since **Camel 2.17**, you can force that the cached schema in the validator endpoint is cleared and reread with the next process call with the JMX operation **clearCachedSchema. `You can also use this method to programmatically clear the cache. This method is available on the `ValidatorEndpoint `class**.`

CHAPTER 336. VELOCITY COMPONENT

Available as of Camel version 1.2

The **velocity:** component allows you to process a message using an Apache Velocity template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-velocity</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

336.1. URI FORMAT

```
velocity:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: file://folder/myfile.vm).

You can append query options to the URI in the following format, **?option=value&option=value&...**

336.2. OPTIONS

The Velocity component supports 2 options which are listed below.

Name	Description	Default	Type
velocityEngine (advanced)	To use the VelocityEngine otherwise a new engine is created		VelocityEngine
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Velocity endpoint is configured using URI syntax:

```
velocity:resourceUri
```

with the following path and query parameters:

336.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the resource. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod.		String

336.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
contentCache (producer)	Sets whether to use resource content cache or not	false	boolean
encoding (producer)	Character encoding of the resource content.		String
loaderCache (producer)	Enables / disables the velocity resource loader cache which is enabled by default	true	boolean
propertiesFile (producer)	The URI of the properties file which is used for VelocityEngine initialization.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

336.3. MESSAGE HEADERS

The velocity component sets a couple headers on the message (you can't set these yourself and from Camel 2.1 velocity component will not set these headers which will cause some side effect on the dynamic template support):

Header	Description
CamelVelocityResourceUri	The **templateName** as a **String** object.

Header	Description
CamelVelocitySupplementalContext	**Camel 2.16:** To add additional information to the used VelocityContext. The value of this header should be a **Map** with key/values that will added (override any existing key with the same name). This can be used to pre setup some common key/values you want to reuse in your velocity endpoints.

Headers set during the Velocity evaluation are returned to the message and added as headers. Then its kinda possible to return values from Velocity to the Message.

For example, to set the header value of **fruit** in the Velocity template **.tm**:

```
$in.setHeader("fruit", "Apple")
```

The **fruit** header is now accessible from the **message.out.headers**.

336.4. VELOCITY CONTEXT

Camel will provide exchange information in the Velocity context (just a **Map**). The **Exchange** is transfered as:

key	value
exchange	The **Exchange** itself.
exchange.properties	The **Exchange** properties.
headers	The headers of the In message.
camelContext	The Camel Context instance.
request	The In message.
in	The In message.
body	The In message body.
out	The Out message (only for InOut message exchange pattern).

key	value
response	The Out message (only for InOut message exchange pattern).

Since Camel-2.14, you can setup a custom Velocity Context yourself by setting the message header *CamelVelocityContext *just like this

```
VelocityContext velocityContext = new VelocityContext(variableMap);
exchange.getIn().setHeader("CamelVelocityContext", velocityContext);
```

336.5. HOT RELOADING

The Velocity template resource is, by default, hot reloadable for both file and classpath resources (expanded jar). If you set **contentCache=true**, Camel will only load the resource once, and thus hot reloading is not possible. This scenario can be used in production, when the resource never changes.

336.6. DYNAMIC TEMPLATES

Available as of Camel 2.1
Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description
CamelVelocityResourceUri	String	**Camel 2.1:** A URI for the template resource to use instead of the endpoint configured.
CamelVelocityTemplate	String	**Camel 2.1:** The template to use instead of the endpoint configured.

336.7. SAMPLES

For example you could use something like

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

To use a Velocity template to formulate a response to a message for InOut message exchanges (where there is a **JMSReplyTo** header).

If you want to use InOnly and consume the message and send it to another destination, you could use the following route:

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

And to use the content cache, e.g. for use in production, where the **.vm** template never changes:

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm?contentCache=true").
  to("activemq:Another.Queue");
```

And a file based resource:

```
from("activemq:My.Queue").
  to("velocity:file://myfolder/MyResponse.vm?contentCache=true").
  to("activemq:Another.Queue");
```

In **Camel 2.1** it's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelVelocityResourceUri").constant("path/to/my/template.vm").
  to("velocity:dummy");
```

In **Camel 2.1** it's possible to specify a template directly as a header the component should use dynamically via a header, so for example:

```
from("direct:in").
  setHeader("CamelVelocityTemplate").constant("Hi this is a velocity template that can do templating
${body}").
  to("velocity:dummy");
```

336.8. THE EMAIL SAMPLE

In this sample we want to use Velocity templating for an order confirmation email. The email template is laid out in Velocity as:

```
Dear ${headers.lastName}, ${headers.firstName}

Thanks for the order of ${headers.item}.

Regards Camel Riders Bookstore
${body}
```

And the java code:

336.9. SEE ALSO

- Configuring Camel

- Component
- Endpoint
- Getting Started

CHAPTER 337. VERT.X COMPONENT

Available as of Camel version 2.12

The **vertx** component is for working with the VertxEventBus.

The vertx EventBus sends and receives JSON events.

INFO:From **Camel 2.16** onwards vertx 3 is in use which requires Java 1.8 at runtime.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-vertx</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

337.1. URI FORMAT

```
vertx:channelName[?options]
```

337.2. OPTIONS

The Vert.x component supports 7 options which are listed below.

Name	Description	Default	Type
vertxFactory (advanced)	To use a custom VertxFactory implementation		VertxFactory
host (common)	Hostname for creating an embedded clustered EventBus		String
port (common)	Port for creating an embedded clustered EventBus		int
vertxOptions (common)	Options to use for creating vertx		VertxOptions
vertx (common)	To use the given vertx EventBus instead of creating a new embedded EventBus		Vertx
timeout (common)	Timeout in seconds to wait for clustered Vertx EventBus to be ready. The default value is 60.	60	int

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Vert.x endpoint is configured using URI syntax:

```
vertx:address
```

with the following path and query parameters:

337.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
address	**Required** Sets the event bus address used to communicate		String

337.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
pubSub (common)	Whether to use publish/subscribe instead of point to point when sending to a vertx endpoint.		Boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern

Name	Description	Default	Type
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Camel 2.12.3: Whether to use publish/subscribe instead of point to point when sending to a vertx endpoint.

> You can append query options to the URI in the following format, ?option=value&option=value&...

337.3. CONNECTING TO THE EXISTING VERT.X INSTANCE

If you would like to connect to the Vert.x instance already existing in your JVM, you can set the instance on the component level:

```
Vertx vertx = ...;
VertxComponent vertxComponent = new VertxComponent();
vertxComponent.setVertx(vertx);
camelContext.addComponent("vertx", vertxComponent);
```

337.4. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 338. VM COMPONENT

Available as of Camel version 1.1

The **vm:** component provides asynchronous SEDA behavior, exchanging messages on a BlockingQueue and invoking consumers in a separate thread pool.

This component differs from the Seda component in that VM supports communication across CamelContext instances - so you can use this mechanism to communicate across web applications (provided that **camel-core.jar** is on the **system/boot** classpath).

VM is an extension to the Seda component.

338.1. URI FORMAT

```
vm:queueName[?options]
```

Where **queueName** can be any string to uniquely identify the endpoint within the JVM (or at least within the classloader that loaded camel-core.jar)

You can append query options to the URI in the following format: **?option=value&option=value&…**

An exactly identical VM endpoint URI **must** be used for both the producer and the consumer endpoint. Otherwise, Camel will create a second VM endpoint despite that the **queueName** portion of the URI is identical. For example:

```
from("direct:foo").to("vm:bar?concurrentConsumers=5");

from("vm:bar?concurrentConsumers=5").to("file://output");
```

Notice that we have to use the full URI, including options in both the producer and consumer.

In Camel 2.4 this has been fixed so that only the queue name must match. Using the queue name **bar**, we could rewrite the previous exmple as follows:

```
from("direct:foo").to("vm:bar");

from("vm:bar?concurrentConsumers=5").to("file://output");
```

338.2. OPTIONS

The VM component supports 4 options which are listed below.

Name	Description	Default	Type
queueSize (advanced)	Sets the default maximum capacity of the SEDA queue (i.e., the number of messages it can hold).		int
concurrentConsumers (consumer)	Sets the default number of concurrent threads processing exchanges.	1	int

Name	Description	Default	Type
defaultQueueFactory (advanced)	Sets the default queue factory.		Exchange>
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The VM endpoint is configured using URI syntax:

```
vm:name
```

with the following path and query parameters:

338.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** Name of queue		String

338.2.2. Query Parameters (16 parameters):

Name	Description	Default	Type
size (common)	The maximum capacity of the SEDA queue (i.e., the number of messages it can hold).	2147483647	int
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN/ERROR level and ignored.	false	boolean
concurrentConsumers (consumer)	Number of concurrent threads processing exchanges.	1	int

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN/ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the default exchange pattern when creating an exchange.		ExchangePattern
limitConcurrentConsumers (consumer)	Whether to limit the number of concurrentConsumers to the maximum of 500. By default, an exception will be thrown if an endpoint is configured with a greater number. You can disable that check by turning this option off.	true	boolean
multipleConsumers (consumer)	Specifies whether multiple consumers are allowed. If enabled, you can use SEDA for Publish-Subscribe messaging. That is, you can send a message to the SEDA queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.	false	boolean
pollTimeout (consumer)	The timeout used when polling. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.	1000	int
purgeWhenStopping (consumer)	Whether to purge the task queue when stopping the consumer/route. This allows to stop faster, as any pending messages on the queue is discarded.	false	boolean
blockWhenFull (producer)	Whether a thread that sends messages to a full SEDA queue will block until the queue's capacity is no longer exhausted. By default, an exception will be thrown stating that the queue is full. By enabling this option, the calling thread will instead block and wait until the message can be accepted.	false	boolean
discardIfNoConsumers (producer)	Whether the producer should discard the message (do not add the message to the queue), when sending to a queue with no active consumers. Only one of the options discardIfNoConsumers and failIfNoConsumers can be enabled at the same time.	false	boolean

Name	Description	Default	Type
failIfNoConsumers (producer)	Whether the producer should fail by throwing an exception, when sending to a queue with no active consumers. Only one of the options discardIfNoConsumers and failIfNoConsumers can be enabled at the same time.	false	boolean
timeout (producer)	Timeout (in milliseconds) before a SEDA producer will stop waiting for an asynchronous task to complete. You can disable timeout by using 0 or a negative value.	30000	long
waitForTaskToComplete (producer)	Option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: Always, Never or IfReplyExpected. The first two values are self-explanatory. The last value, IfReplyExpected, will only wait if the message is Request Reply based. The default option is IfReplyExpected.	IfReplyExpected	WaitForTaskToComplete
queue (advanced)	Define the queue instance which will be used by the endpoint. This option is only for rare use-cases where you want to use a custom queue instance.		BlockingQueue
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

See the Seda component for options and other important usage details as the same rules apply to the Vm component.

338.3. SAMPLES

In the route below we send exchanges across CamelContext instances to a VM queue named **order.email**:

```
from("direct:in").bean(MyOrderBean.class).to("vm:order.email");
```

And then we receive exchanges in some other Camel context (such as deployed in another **.war** application):

```
from("vm:order.email").bean(MyOrderEmailSender.class);
```

338.4. SEE ALSO

- Seda

CHAPTER 339. WEATHER COMPONENT

Available as of Camel version 2.12

The **weather:** component is used for polling weather information from Open Weather Map – a site that provides free global weather and forecast information. The information is returned as a json String object.

Camel will poll for updates to the current weather and forecasts once per hour by default. It can also be used to query the weather api based on the parameters defined on the endpoint which is used as producer.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-weather</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

339.1. URI FORMAT

```
weather://<unused name>[?options]
```

339.2. REMARK

Since the 9th of October, an Api Key is required to access the openweather service. This key is passed as parameter to the URI definition of the weather endpoint using the appid param !

339.3. GEOLOCATION PROVIDER

Since July 2018 FreegeoIP is no longer available. The camel-weather component was using this API. We switch to IPstack so you'll need to specify and Access Key and the IP from where you're using the API now on.

339.4. OPTIONS

The Weather component supports 3 options, which are listed below.

Name	Description	Default	Type
geolocationAccessKey (common)	The geolocation service now needs an accessKey to be used		String
geolocationRequestHost IP (common)	The geolocation service now needs to specify the IP associated to the accessKey you're using		String

Name	Description	Default	Type
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Weather endpoint is configured using URI syntax:

```
weather:name
```

with the following path and query parameters:

339.4.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** The name value is not used.		String

339.4.2. Query Parameters (45 parameters):

Name	Description	Default	Type
appid (common)	**Required** APPID ID used to authenticate the user connected to the API Server		String
headerName (common)	To store the weather result in this header instead of the message body. This is useable if you want to keep current message body as-is.		String
language (common)	Language of the response.	en	WeatherLanguage
mode (common)	The output format of the weather data.	JSON	WeatherMode
period (common)	If null, the current weather will be returned, else use values of 5, 7, 14 days. Only the numeric value for the forecast period is actually parsed, so spelling, capitalisation of the time period is up to you (its ignored)		String
units (common)	The units for temperature measurement.		WeatherUnits

Name	Description	Default	Type
weatherApi (common)	The API to be use (current, forecast/3 hour, forecast daily, station)		WeatherApi
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
httpConnectionManager (advanced)	To use a custom HttpConnectionManager to manage connections		HttpConnectionManager
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

Name	Description	Default	Type
cnt (filter)	Number of results to be found		Integer
ids (filter)	List of id's of city/stations. You can separate multiple ids by comma.		String
lat (filter)	Latitude of location. You can use lat and lon options instead of location. For boxed queries this is the bottom latitude.		String
location (filter)	If null Camel will try and determine your current location using the geolocation of your ip address, else specify the city,country. For well known city names, Open Weather Map will determine the best fit, but multiple results may be returned. Hence specifying and country as well will return more accurate data. If you specify current as the location then the component will try to get the current latitude and longitude and use that to get the weather details. You can use lat and lon options instead of location.		String
lon (filter)	Longitude of location. You can use lat and lon options instead of location. For boxed queries this is the left longtitude.		String
rightLon (filter)	For boxed queries this is the right longtitude. Needs to be used in combination with topLat and zoom.		String
topLat (filter)	For boxed queries this is the top latitude. Needs to be used in combination with rightLon and zoom.		String
zip (filter)	Zip-code, e.g. 94040,us		String
zoom (filter)	For boxed queries this is the zoom. Needs to be used in combination with rightLon and topLat.		Integer
proxyAuthDomain (proxy)	Domain for proxy NTLM authentication		String
proxyAuthHost (proxy)	Optional host for proxy NTLM authentication		String
proxyAuthMethod (proxy)	Authentication method for proxy, either as Basic, Digest or NTLM.		String
proxyAuthPassword (proxy)	Password for proxy authentication		String

Name	Description	Default	Type
proxyAuthUsername (proxy)	Username for proxy authentication		String
proxyHost (proxy)	The proxy host name		String
proxyPort (proxy)	The proxy port number		Integer
geolocationAccessKey (security)	**Required** The geolocation service now needs an accessKey to be used		String
geolocationRequestHostIP (security)	**Required** The geolocation service now needs to specify the IP associated to the accessKey you're using		String

You can append query options to the URI in the following format, **?option=value&option=value&…**

339.5. EXCHANGE DATA FORMAT

Camel will deliver the body as a json formatted java.lang.String (see the **mode** option above).

339.6. MESSAGE HEADERS

Header	Description
CamelWeatherQuery	The original query URL sent to the Open Weather Map site
CamelWeatherLocation	Used by the producer to override the endpoint location and use the location from this header instead.

339.7. SAMPLES

In this sample we find the 7 day weather forecast for Madrid, Spain:

```
from("weather:foo?location=Madrid,Spain&period=7
days&appid=APIKEY&geolocationAccessKey=IPSTACK_ACCESS_KEY&geolocationRequestHostIP=L
OCAL_IP").to("jms:queue:weather");
```

To just find the current weather for your current location you can use this:

```
from("weather:foo?
appid=APIKEY&geolocationAccessKey=IPSTACK_ACCESS_KEY&geolocationRequestHostIP=LOCAL_
IP").to("jms:queue:weather");
```

And to find the weather using the producer we do:

```
from("direct:start")
  .to("weather:foo?
location=Madrid,Spain&appid=APIKEY&geolocationAccessKey=IPSTACK_ACCESS_KEY&geolocation
equestHostIP=LOCAL_IP");
```

And we can send in a message with a header to get the weather for any location as shown:

```
  String json = template.requestBodyAndHeader("direct:start", "", "CamelWeatherLocation",
"Paris,France&appid=APIKEY", String.class);
```

And to get the weather at the current location, then:

```
  String json = template.requestBodyAndHeader("direct:start", "", "CamelWeatherLocation",
"current&appid=APIKEY", String.class);
```

CHAPTER 340. JETTY WEBSOCKET COMPONENT

Available as of Camel version 2.10

The **websocket** component provides websocket endpoints for communicating with clients using websocket. The component uses Eclipse Jetty Server which implements the IETF specification (drafts and RFC 6455). It supports the protocols ws:// and wss://. To use wss:// protocol, the SSLContextParameters must be defined.

Version currently supported

Camel 2.18 uses Jetty 9

340.1. URI FORMAT

```
websocket://hostname[:port][/resourceUri][?options]
```

You can append query options to the URI in the following format, **?option=value&option=value&…**

340.2. WEBSOCKET OPTIONS

The Jetty Websocket component supports 14 options which are listed below.

Name	Description	Default	Type
staticResources (consumer)	Set a resource path for static resources (such as .html files etc). The resources can be loaded from classpath, if you prefix with classpath:, otherwise the resources is loaded from file system or from JAR files. For example to load from root classpath use classpath:., or classpath:WEB-INF/static If not configured (eg null) then no static resource is in use.		String
host (common)	The hostname. The default value is 0.0.0.0	0.0.0.0	String
port (common)	The port number. The default value is 9292	9292	Integer
sslKeyPassword (security)	The password for the keystore when using SSL.		String
sslPassword (security)	The password when using SSL.		String
sslKeystore (security)	The path to the keystore.		String
enableJmx (advanced)	If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.	false	boolean

Name	Description	Default	Type
minThreads (advanced)	To set a value for minimum number of threads in server thread pool. MaxThreads/minThreads or threadPool fields are required due to switch to Jetty9. The default values for minThreads is 1.		Integer
maxThreads (advanced)	To set a value for maximum number of threads in server thread pool. MaxThreads/minThreads or threadPool fields are required due to switch to Jetty9. The default values for maxThreads is 1 2 noCores.		Integer
threadPool (advanced)	To use a custom thread pool for the server. MaxThreads/minThreads or threadPool fields are required due to switch to Jetty9.		ThreadPool
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters
useGlobalSslContext Parameters (security)	Enable usage of global SSL context parameters.	false	boolean
socketFactory (common)	To configure a map which contains custom WebSocketFactory for sub protocols. The key in the map is the sub protocol. The default key is reserved for the default implementation.		Map
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Jetty Websocket endpoint is configured using URI syntax:

```
websocket:host:port/resourceUri
```

with the following path and query parameters:

340.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	The hostname. The default value is 0.0.0.0. Setting this option on the component will use the component configured value as default.	0.0.0.0	String

Name	Description	Default	Type
port	The port number. The default value is 9292. Setting this option on the component will use the component configured value as default.	9292	Integer
resourceUri	**Required** Name of the websocket channel to use		String

340.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
maxBinaryMessageSize (common)	Can be used to set the size in bytes that the websocket created by the websocketServlet may be accept before closing. (Default is -1 - or unlimited)	-1	Integer
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sessionSupport (consumer)	Whether to enable session support which enables HttpSession for each http request.	false	boolean
staticResources (consumer)	Set a resource path for static resources (such as .html files etc). The resources can be loaded from classpath, if you prefix with classpath:, otherwise the resources is loaded from file system or from JAR files. For example to load from root classpath use classpath:., or classpath:WEB-INF/static If not configured (eg null) then no static resource is in use.		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
sendTimeout (producer)	Timeout in millis when sending to a websocket channel. The default timeout is 30000 (30 seconds).	30000	Integer

Name	Description	Default	Type
sendToAll (producer)	To send to all websocket subscribers. Can be used to configure on endpoint level, instead of having to use the WebsocketConstants.SEND_TO_ALL header on the message.		Boolean
bufferSize (advanced)	Set the buffer size of the websocketServlet, which is also the max frame byte size (default 8192)	8192	Integer
maxIdleTime (advanced)	Set the time in ms that the websocket created by the websocketServlet may be idle before closing. (default is 300000)	300000	Integer
maxTextMessageSize (advanced)	Can be used to set the size in characters that the websocket created by the websocketServlet may be accept before closing.		Integer
minVersion (advanced)	Can be used to set the minimum protocol version accepted for the websocketServlet. (Default 13 - the RFC6455 version)	13	Integer
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
allowedOrigins (cors)	The CORS allowed origins. Use to allow all.		String
crossOriginFilterOn (cors)	Whether to enable CORS	false	boolean
filterPath (cors)	Context path for filtering CORS		String
enableJmx (monitoring)	If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.	false	boolean
sslContextParameters (security)	To configure security using SSLContextParameters		SSLContextParameters

340.3. MESSAGE HEADERS

The websocket component uses 2 headers to indicate to either send messages back to a single/current client, or to all clients.

WebsocketConstants.SEND_TO_ALL	Sends the message to all clients which are currently connected. You can use the **sendToAll** option on the endpoint instead of using this header.
WebsocketConstants.CONNECTION_KEY	Sends the message to the client with the given connection key.

340.4. USAGE

In this example we let Camel exposes a websocket server which clients can communicate with. The websocket server uses the default host and port, which would be **0.0.0.0:9292**.
The example will send back an echo of the input. To send back a message, we need to send the transformed message to the same endpoint **"websocket://echo"**. This is needed because by default the messaging is InOnly.

This example is part of an unit test, which you can find here. As a client we use the AHC library which offers support for web socket as well.

Here is another example where webapp resources location have been defined to allow the Jetty Application Server to not only register the WebSocket servlet but also to expose web resources for the browser. Resources should be defined under the webapp directory.

```
from("activemq:topic:newsTopic")
   .routeId("fromJMStoWebSocket")
   .to("websocket://localhost:8443/newsTopic?sendToAll=true&staticResources=classpath:webapp");
```

340.5. SETTING UP SSL FOR WEBSOCKET COMPONENT

340.5.1. Using the JSSE Configuration Utility

As of Camel 2.10, the WebSocket component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Cometd component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
```

```
kmp.setKeyPassword("keyPassword");

TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);
scp.setTrustManagers(tmp);

CometdComponent commetdComponent = getContext().getComponent("cometds",
CometdComponent.class);
commetdComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
     id="sslContextParameters">
    <camel:keyManagers
       keyPassword="keyPassword">
      <camel:keyStore
         resource="/users/home/server/keystore.jks"
         password="keystorePassword"/>
    </camel:keyManagers>
    <camel:trustManagers>
      <camel:keyStore
         resource="/users/home/server/keystore.jks"
         password="keystorePassword"/>
    </camel:trustManagers>
  </camel:sslContextParameters>...
...
  <to uri="websocket://127.0.0.1:8443/test?sslContextParameters=#sslContextParameters"/>...
```

Java DSL based configuration of endpoint

```
...
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() {

                String uri = "websocket://127.0.0.1:8443/test?
sslContextParameters=#sslContextParameters";

                from(uri)
                    .log(">>> Message received from WebSocket Client : ${body}")
                    .to("mock:client")
                    .loop(10)
                        .setBody().constant(">> Welcome on board!")
                        .to(uri);
...
```

340.6. SEE ALSO

- Configuring Camel

- Component
- Endpoint
- Getting Started
- AHC
- Jetty
- Twitter Websocket Example demonstrates how to poll a constant feed of twitter searches and publish results in real time using web socket to a web page.

CHAPTER 341. WORDPRESS COMPONENT

Available as of Camel version 2.21

Camel component for Wordpress API.

Currently only the **Posts** and **Users** operations are supported.

341.1. OPTIONS

The Wordpress component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	Wordpress component configuration		WordpressComponent Configuration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Wordpress endpoint is configured using URI syntax:

```
wordpress:operationDetail
```

with the following path and query parameters:

341.1.1. Path Parameters (2 parameters):

Name	Description	Default	Type
operation	**Required** The endpoint operation.		String
operationDetail	The second part of an endpoint operation. Needed only when endpoint semantic is not enough, like wordpress:post:delete		String

341.1.2. Query Parameters (11 parameters):

Name	Description	Default	Type
apiVersion (common)	The Wordpress REST API version	2	String

Name	Description	Default	Type
criteria (common)	The criteria to use with complex searches.		Map
force (common)	Whether to bypass trash and force deletion.	false	Boolean
id (common)	The entity id		Integer
password (common)	Password from authorized user		String
url (common)	**Required** The Wordpress API URL from your site, e.g. http://myblog.com/wp-json/		String
user (common)	Authorized user to perform writing operations		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

Most of parameters needed when performing a read operation mirrors from the official API. When performing searches operations, the **criteria.** suffix is needed. Take the following **Consumer** as example:

```
wordpress:post?criteria.perPage=10&criteria.orderBy=author&criteria.categories=camel,dozer,json
```

341.1.3. Configuring Wordpress component

The **WordpressConfiguration** class can be used to set initial properties configuration to the component instead of passing it as query parameter. The following listing shows how to set the component to be used in your routes.

```
public void configure() {
    final WordpressConfiguration configuration = new WordpressConfiguration();
    final WordpressComponentConfiguration component = new WordpressComponentConfiguration();
    configuration.setApiVersion("2");
    configuration.setUrl("http://yoursite.com/wp-json/");
    component.setConfiguration(configuration);
    getContext().addComponent("wordpress", component);

    from("wordpress:post?id=1")
      .to("mock:result");
}
```

341.1.4. Consumer Example

Consumer polls from the API from time to time domain objects from Wordpress. Following, an example using the **Post** operation:

- **wordpress:post** retrieves posts (defaults to 10 posts)
- **wordpress:post?id=1** search for a specific post

341.1.5. Producer Example

Producer performs write operations on Wordpress like adding a new user or update a post. To be able to write, you must have an authorized user credentials (see Authentication).

- **wordpress:post** creates a new post from the **org.apache.camel.component.wordpress.api.model.Post** class in the message body.
- **wordpress:post?id=1** updates a post based on data **org.apache.camel.component.wordpress.api.model.Post** from the message body.
- **wordpress:post:delete?id=1** deletes a specific post

341.2. AUTHENTICATION

Producers that perform write operations (e.g. create a new post) must have an authenticated user to do so. The standard authentication mechanism used by Wordpress is cookie. Unfortunately this method is not supported outside Wordpress environment because it's rely on nonce internal function.

There's some alternatives to use the Wordpress API without nonces, but requires specific plugin installations.

At this time, **camel-wordpress** only supports Basic Authentication (more to come). To configure it, you must install the Basic-Auth Wordpress plugin and pass the credentials to the endpoint:

from("direct:deletePost").to("wordpress:post:delete?id=9&user=ben&password=password123").to("mock:resultDelete");

It's not recommend to use Basic Authentication in production without TLS!!

CHAPTER 342. XCHANGE COMPONENT

Available as of Camel version 2.21

The **xchange:** component uses the XChange Java library to provide access to 60+ Bitcoin and Altcoin exchanges. It comes with a consistent interface for trading and accessing market data.

Camel can get crypto currency market data, query historical data, place market orders and much more.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xchange</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

342.1. URI FORMAT

```
xchange://exchange?options
```

342.2. OPTIONS

The XChange component has no options.

The XChange endpoint is configured using URI syntax:

```
xchange:name
```

with the following path and query parameters:

342.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
name	**Required** The exchange to connect to		String

342.2.2. Query Parameters (5 parameters):

Name	Description	Default	Type
currency (producer)	The currency		Currency
currencyPair (producer)	The currency pair		CurrencyPair

Name	Description	Default	Type
method (producer)	**Required** The method to execute		XChangeMethod
service (producer)	**Required** The service to call		XChangeService
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

342.3. AUTHENTICATION

This component communicates with supported crypto currency exchanges via REST API. Some API requests use simple unauthenticated GET request. For most of the interesting stuff however, you'd need an account with the exchange and have API access keys enabled.

These API access keys need to be guarded tightly, especially so when they also allow for the withdraw functionality. In which case, anyone who can get hold of your API keys can easily transfer funds from your account to some other address i.e. steal your money.

Your API access keys can be strored in an exchange specific properties file in your SSH directory. For Binance for example this would be: **~/.ssh/binance-secret.keys**

```
##
# This file MUST NEVER be commited to source control.
# It is therefore added to .gitignore.
#
apiKey = GuRW0*********
secretKey = nKLki************
```

342.4. MESSAGE HEADERS

<TODO><title>Samples</title>
In this sample we find the current Bitcoin market price in USDT:

```
from("direct:ticker").to("xchange:binance?service=market&method=ticker&currencyPair=BTC/USDT")
```

</TODO>

CHAPTER 343. XML BEANS DATAFORMAT (DEPRECATED)

Available as of Camel version 1.2

XmlBeans is a Data Format which uses the XmlBeans library to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

```
from("activemq:My.Queue").
  unmarshal().xmlBeans().
  to("mqseries:Another.Queue");
```

343.1. OPTIONS

The XML Beans dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
prettyPrint	**false**	**Boolean**	To enable pretty printing output nicely formatted. Is by default false.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

343.2. DEPENDENCIES

To use XmlBeans in your camel routes you need to add the dependency on **camel-xmlbeans** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xmlbeans</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 344. XML JSON DATAFORMAT (DEPRECATED)

Available as of Camel version 2.10

Camel already supports a number of data formats to perform XML and JSON-related conversions, but all of them require a POJO either as an input (for marshalling) or produce a POJO as output (for unmarshalling). This data format provides the capability to convert from XML to JSON and vice-versa directly, without stepping through intermediate POJOs.

This data format leverages the Json-lib library to achieve direct conversion. In this context, XML is considered the high-level format, while JSON is the low-level format. Hence, the marshal/unmarshal semantics are assigned as follows:

- marshalling ⇒ converting from XML to JSON
- unmarshalling ⇒ converting from JSON to XML.

344.1. OPTIONS

The XML JSon dataformat supports 13 options which are listed below.

Name	Default	Java Type	Description
encoding		**String**	Sets the encoding. Used for unmarshalling (JSON to XML conversion).
elementName		**String**	Specifies the name of the XML elements representing each array element. Used for unmarshalling (JSON to XML conversion).
arrayName		**String**	Specifies the name of the top-level XML element. Used for unmarshalling (JSON to XML conversion). For example, when converting 1, 2, 3, it will be output by default as 123. By setting this option or rootName, you can alter the name of element 'a'.
forceTopLevelObject	**false**	**Boolean**	Determines whether the resulting JSON will start off with a top-most element whose name matches the XML root element. Used for marshalling (XML to JSon conversion). If disabled, XML string 12 turns into 'x: '1', 'y': '2' . Otherwise, it turns into 'a': 'x: '1', 'y': '2' .
namespaceLenient	**false**	**Boolean**	Flag to be tolerant to incomplete namespace prefixes. Used for unmarshalling (JSON to XML conversion). In most cases, json-lib automatically changes this flag at runtime to match the processing.

Name	Default	Java Type	Description
rootName		String	Specifies the name of the top-level element. Used for unmarshalling (JSON to XML conversion). If not set, json-lib will use arrayName or objectName (default value: 'o', at the current time it is not configurable in this data format). If set to 'root', the JSON string 'x': 'value1', 'y' : 'value2' would turn into value1value2, otherwise the 'root' element would be named 'o'.
skipWhitespace	false	Boolean	Determines whether white spaces between XML elements will be regarded as text values or disregarded. Used for marshalling (XML to JSon conversion).
trimSpaces	false	Boolean	Determines whether leading and trailing white spaces will be omitted from String values. Used for marshalling (XML to JSon conversion).
skipNamespaces	false	Boolean	Signals whether namespaces should be ignored. By default they will be added to the JSON output using xmlns elements. Used for marshalling (XML to JSon conversion).
removeNamespacePrefixes	false	Boolean	Removes the namespace prefixes from XML qualified elements, so that the resulting JSON string does not contain them. Used for marshalling (XML to JSon conversion).
expandableProperties		List	With expandable properties, JSON array elements are converted to XML as a sequence of repetitive XML elements with the local name equal to the JSON key, for example: number: 1,2,3 , normally converted to: 123 (where e can be modified by setting elementName), would instead translate to 123, if number is set as an expandable property Used for unmarshalling (JSON to XML conversion).
typeHints		String	Adds type hints to the resulting XML to aid conversion back to JSON. Used for unmarshalling (JSON to XML conversion).
contentTypeHeader	false	Boolean	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

344.2. BASIC USAGE WITH JAVA DSL

344.2.1. Explicitly instantiating the data format

Just instantiate the **XmlJsonDataFormat** from package **org.apache.camel.dataformat.xmljson**. Make sure you have installed the **camel-xmljson** feature (if running on OSGi) or that you've included **camel-xmljson-7.3.jar** and its transitive dependencies in your classpath. Example initialization with a default configuration:

```
XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();
```

To tune the behaviour of the data format as per the options above, use the appropriate setters:

```
XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();
xmlJsonFormat.setEncoding("UTF-8");
xmlJsonFormat.setForceTopLevelObject(true);
xmlJsonFormat.setTrimSpaces(true);
xmlJsonFormat.setRootName("newRoot");
xmlJsonFormat.setSkipNamespaces(true);
xmlJsonFormat.setRemoveNamespacePrefixes(true);
xmlJsonFormat.setExpandableProperties(Arrays.asList("d", "e"));
```

Once you've instantiated the data format, the next step is to actually use the it from within the **marshal()** and **unmarshal()** DSL elements:

```
// from XML to JSON
from("direct:marshal").marshal(xmlJsonFormat).to("mock:json");
// from JSON to XML
from("direct:unmarshal").unmarshal(xmlJsonFormat).to("mock:xml");
```

344.2.2. Defining the data format in-line

Alternatively, you can define the data format inline by using the **xmljson()** DSL element:

```
// from XML to JSON - inline dataformat
from("direct:marshalInline").marshal().xmljson().to("mock:jsonInline");
// from JSON to XML - inline dataformat
from("direct:unmarshalInline").unmarshal().xmljson().to("mock:xmlInline");
```

If you wish, you can even pass in a **Map<String, String>** to the inline methods to provide custom options:

```
Map<String, String> xmlJsonOptions = new HashMap<String, String>();
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.ENCODING, "UTF-8");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.ROOT_NAME,
"newRoot");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.SKIP_NAMESPACES,
"true");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.REMOVE_NAMESPACE
_PREFIXES, "true");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.EXPANDABLE_PROPE
RTIES, "d e");

// from XML to JSON - inline dataformat w/ options
from("direct:marshalInlineOptions").marshal().xmljson(xmlJsonOptions).to("mock:jsonInlineOptions");
// form JSON to XML - inline dataformat w/ options
from("direct:unmarshalInlineOptions").unmarshal().xmljson(xmlJsonOptions).to("mock:xmlInlineOption
s");
```

344.3. BASIC USAGE WITH SPRING OR BLUEPRINT DSL

Within the **<dataFormats>** block, simply configure an **xmljson** element with unique IDs:

```
<dataFormats>
    <xmljson id="xmljson"/>
    <xmljson id="xmljsonWithOptions" forceTopLevelObject="true" trimSpaces="true"
rootName="newRoot" skipNamespaces="true"
            removeNamespacePrefixes="true" expandableProperties="d e"/>
</dataFormats>
```

Then you simply refer to the data format object within your **<marshal />** and **<unmarshal />** DSLs:

```
<route>
    <from uri="direct:marshal"/>
    <marshal ref="xmljson"/>
    <to uri="mock:json" />
</route>

<route>
    <from uri="direct:unmarshalWithOptions"/>
    <unmarshal ref="xmljsonWithOptions"/>
    <to uri="mock:xmlWithOptions"/>
</route>
```

Enabling XML DSL autocompletion for this component is easy: just refer to the appropriate Schema locations, depending on whether you're using Spring or Blueprint DSL. Remember that this data format is available from Camel 2.10 onwards, so only schemas from that version onwards will include these new XML elements and attributes.

The syntax with Blueprint is identical to that of the Spring DSL. Just ensure the correct namespaces and schemaLocations are in use.

344.4. NAMESPACE MAPPINGS

XML has namespaces to fully qualify elements and attributes; JSON doesn't. You need to take this into account when performing XML-JSON conversions.

To bridge the gap, Json-lib has an option to bind namespace declarations in the form of prefixes and namespace URIs to XML output elements while unmarshalling (i.e. converting from JSON to XML). For example, provided the following JSON string:

```
{ "pref1:a": "value1", "pref2:b": "value2" }
```

you can ask Json-lib to output namespace declarations on elements **pref1:a** and **pref2:b** to bind the prefixes **pref1** and **pref2** to specific namespace URIs.

To use this feature, simply create **XmlJsonDataFormat.NamespacesPerElementMapping** objects and add them to the **namespaceMappings** option (which is a **List**).

The **XmlJsonDataFormat.NamespacesPerElementMapping** holds an element name and a Map of [prefix ⇒ namespace URI]. To facilitate mapping multiple prefixes and namespace URIs, the **NamespacesPerElementMapping(String element, String pipeSeparatedMappings)** constructor takes a String-based pipe-separated sequence of [prefix, namespaceURI] pairs in the following way: **|ns2|http://camel.apache.org/personalData|ns3|http://camel.apache.org/personalData2|**.

In order to define a default namespace, just leave the corresponding key field empty: **|ns1|http://camel.apache.org/test1||http://camel.apache.org/default|**.

Binding namespace declarations to an element name = empty string will attach those namespaces to the root element.

The full code would look like that:

```
XmlJsonDataFormat namespacesFormat = new XmlJsonDataFormat();
List<XmlJsonDataFormat.NamespacesPerElementMapping> namespaces = new
ArrayList<XmlJsonDataFormat.NamespacesPerElementMapping>();
namespaces.add(new XmlJsonDataFormat.
                NamespacesPerElementMapping("",
"|ns1|http://camel.apache.org/test1||http://camel.apache.org/default|"));
namespaces.add(new XmlJsonDataFormat.
                NamespacesPerElementMapping("surname",
"|ns2|http://camel.apache.org/personalData|" +
                  "ns3|http://camel.apache.org/personalData2|"));
namespacesFormat.setNamespaceMappings(namespaces);
namespacesFormat.setRootElement("person");
```

And you can achieve the same in Spring DSL.

344.4.1. Example

Using the namespace bindings in the Java snippet above on the following JSON string:

```
{ "name": "Raul", "surname": "Kripalani", "f": true, "g": null}
```

Would yield the following XML:

```
<person xmlns="http://camel.apache.org/default" xmlns:ns1="http://camel.apache.org/test1">
   <f>true</f>
   <g null="true"/>
   <name>Raul</name>
   <surname xmlns:ns2="http://camel.apache.org/personalData"
xmlns:ns3="http://camel.apache.org/personalData2">Kripalani</surname>
</person>
```

Remember that the JSON spec defines a JSON object as follows:

> An object is an unordered set of name/value pairs. [...].

That's why the elements are in a different order in the output XML.

344.5. DEPENDENCIES

To use the XmlJson dataformat in your camel routes you need to add the following dependency to your pom:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xmljson</artifactId>
  <version>x.x.x</version>
  <!-- Use the same version as camel-core, but remember that this component is only available from
```

```
2.10 onwards -->
</dependency>

<!-- And also XOM must be included. XOM cannot be included by default due to an incompatible
license with ASF; so add this manually -->
<dependency>
  <groupId>xom</groupId>
  <artifactId>xom</artifactId>
  <version>1.2.5</version>
</dependency>
```

344.6. SEE ALSO

- Data Format
- json-lib

CHAPTER 345. XML SECURITY COMPONENT

Available as of Camel version 2.12

With this Apache Camel component, you can generate and validate XML signatures as described in the W3C standard XML Signature Syntax and Processing or as described in the successor version 1.1. For XML Encryption support, please refer to the XML Security Data Format.

You can find an introduction to XML signature here. The implementation of the component is based on JSR 105, the Java API corresponding to the W3C standard and supports the Apache Santuario and the JDK provider for JSR 105. The implementation will first try to use the Apache Santuario provider; if it does not find the Santuario provider, it will use the JDK provider. Further, the implementation is DOM based.

Since Camel 2.15.0 we also provide support for **XAdES-BES/EPES** for the signer endpoint; see subsection "XAdES-BES/EPES for the Signer Endpoint".

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xmlsecurity</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

345.1. XML SIGNATURE WRAPPING MODES

XML Signature differs between enveloped, enveloping, and detached XML signature. In the enveloped XML signature case, the XML Signature is wrapped by the signed XML Document; which means that the XML signature element is a child element of a parent element, which belongs to the signed XML Document. In the enveloping XML signature case, the XML Signature contains the signed content. All other cases are called detached XML signatures. A certain form of detached XML signature is supported since **2.14.0**.

In the **enveloped XML signature** case, the supported generated XML signature has the following structure (Variables are surrounded by **[]**).

```
<[parent element]>
  ... <!-- Signature element is added as last child of the parent element-->
  <Signature Id="generated_unique_signature_id">
    <SignedInfo>
      <Reference URI="">
        <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
        (<Transform>)* <!-- By default "http://www.w3.org/2006/12/xml-c14n11" is added to the
transforms -->
        <DigestMethod>
        <DigestValue>
      </Reference>
      (<Reference URI="#[keyinfo_Id]">
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
        <DigestMethod>
        <DigestValue>
      </Reference>)?
      <!-- further references possible, see option 'properties' below -->
```

```
    </SignedInfo>
    <SignatureValue>
    (<KeyInfo Id="[keyinfo_id]">)?
    <!-- Object elements possible, see option 'properties' below -->
  </Signature>
</[parent element]>
```

In the **enveloping XML signature** case, the supported generated XML signature has the structure:

```
<Signature Id="generated_unique_signature_id">
  <SignedInfo>
      <Reference URI="#generated_unique_object_id" type="[optional_type_value]">
          (<Transform>)* <!-- By default "http://www.w3.org/2006/12/xml-c14n11" is added to the
transforms -->
          <DigestMethod>
          <DigestValue>
      </Reference>
      (<Reference URI="#[keyinfo_id]">
          <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
          <DigestMethod>
          <DigestValue>
      </Reference>)?
       <!-- further references possible, see option 'properties' below  -->
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo Id="[keyinfo_id]">)?
  <Object Id="generated_unique_object_id"/> <!-- The Object element contains the in-message body;
the object ID can either be generated or set by the option parameter "contentObjectId" -->
  <!-- Further Object elements possible, see option 'properties' below -->
</Signature>
```

As of 2.14.0 **detached XML signatures** with the following structure are supported (see also sub-chapter XML Signatures as Siblings of Signed Elements):

```
(<[signed element] Id="[id_value]">
<!-- signed element must have an attribute of type ID -->
    ...

</[signed element]>
<other sibling/>*
<!-- between the signed element and the corresponding signature element, there can be other
siblings.
 Signature element is added as last sibling. -->
<Signature Id="generated_unique_ID">
  <SignedInfo>
    <CanonicalizationMethod>
    <SignatureMethod>
    <Reference URI="#[id_value]" type="[optional_type_value]">
    <!-- reference URI contains the ID attribute value of the signed element -->
        (<Transform>)* <!-- By default "http://www.w3.org/2006/12/xml-c14n11" is added to the
transforms -->
        <DigestMethod>
        <DigestValue>
    </Reference>
    (<Reference URI="#[generated_keyinfo_Id]">
```

```
        <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
        <DigestMethod>
        <DigestValue>
    </Reference>)?
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo Id="[generated_keyinfo_id]">)?
</Signature>)+
```

345.2. URI FORMAT

The camel component consists of two endpoints which have the following URI format:

```
xmlsecurity:sign:name[?options]
xmlsecurity:verify:name[?options]
```

- With the signer endpoint, you can generate a XML signature for the body of the in-message which can be either a XML document or a plain text. The enveloped, enveloping, or detached (as of 12.14) XML signature(s) will be set to the body of the out-message.
- With the verifier endpoint, you can validate an enveloped or enveloping XML signature or even several detached (as of 2.14.0) XML signatures contained in the body of the in-message; if the validation is successful, then the original content is extracted from the XML signature and set to the body of the out-message.
- The **name** part in the URI can be chosen by the user to distinguish between different signer/verifier endpoints within the camel context.

345.3. BASIC EXAMPLE

The following example shows the basic usage of the component.

```
from("direct:enveloping").to("xmlsecurity:sign://enveloping?keyAccessor=#accessor",
                             "xmlsecurity:verify://enveloping?keySelector=#selector",
                             "mock:result")
```

In Spring XML:

```
<from uri="direct:enveloping" />
   <to uri="xmlsecurity:sign://enveloping?keyAccessor=#accessor" />
   <to uri="xmlsecurity:verify://enveloping?keySelector=#selector" />
<to uri="mock:result" />
```

For the signing process, a private key is necessary. You specify a key accessor bean which provides this private key. For the validation, the corresponding public key is necessary; you specify a key selector bean which provides this public key.

The key accessor bean must implement the **KeyAccessor** interface. The package **org.apache.camel.component.xmlsecurity.api** contains the default implementation class **DefaultKeyAccessor** which reads the private key from a Java keystore.

The key selector bean must implement the **javax.xml.crypto.KeySelector** interface. The package **org.apache.camel.component.xmlsecurity.api** contains the default implementation class **DefaultKeySelector** which reads the public key from a keystore.

In the example, the default signature algorithm **http://www.w3.org/2000/09/xmldsig#rsa-sha1** is used. You can set the signature algorithm of your choice by the option **signatureAlgorithm** (see below). The signer endpoint creates an *enveloping* XML signature. If you want to create an *enveloped* XML signature then you must specify the parent element of the Signature element; see option **parentLocalName** for more details.

For creating *detached* XML signatures, see sub-chapter "Detached XML Signatures as Siblings of the Signed Elements".

345.4. COMPONENT OPTIONS

The XML Security component supports 3 options which are listed below.

Name	Description	Default	Type
signerConfiguration (advanced)	To use a shared XmlSignerConfiguration configuration to use as base for configuring endpoints.		XmlSignerConfiguration
verifierConfiguration (advanced)	To use a shared XmlVerifierConfiguration configuration to use as base for configuring endpoints.		XmlVerifierConfiguration
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

345.5. ENDPOINT OPTIONS

The XML Security endpoint is configured using URI syntax:

```
xmlsecurity:command:name
```

with the following path and query parameters:

345.5.1. Path Parameters (2 parameters):

Name	Description	Default	Type
command	**Required** Whether to sign or verify.		XmlCommand

Name	Description	Default	Type
name	**Required** The name part in the URI can be chosen by the user to distinguish between different signer/verifier endpoints within the camel context.		String

345.5.2. Query Parameters (35 parameters):

Name	Description	Default	Type
baseUri (common)	You can set a base URI which is used in the URI dereferencing. Relative URIs are then concatenated with the base URI.		String
clearHeaders (common)	Determines if the XML signature specific headers be cleared after signing and verification. Defaults to true.	true	Boolean
cryptoContextProperties (common)	Sets the crypto context properties. See link XMLCryptoContextsetProperty(String, Object). Possible properties are defined in XMLSignContext an XMLValidateContext (see Supported Properties). The following properties are set by default to the value link BooleanTRUE for the XML validation. If you want to switch these features off you must set the property value to link BooleanFALSE. org.jcp.xml.dsig.validateManifests javax.xml.crypto.dsig.cacheReference		Map
disallowDoctypeDecl (common)	Disallows that the incoming XML document contains DTD DOCTYPE declaration. The default value is link BooleanTRUE.	true	Boolean
omitXmlDeclaration (common)	Indicator whether the XML declaration in the outgoing message body should be omitted. Default value is false. Can be overwritten by the header link XmlSignatureConstantsHEADER_OMIT_XML_DECLARATION.	false	Boolean
outputXmlEncoding (common)	The character encoding of the resulting signed XML document. If null then the encoding of the original XML document is used.		String

Name	Description	Default	Type
schemaResourceUri (common)	Classpath to the XML Schema. Must be specified in the detached XML Signature case for determining the ID attributes, might be set in the enveloped and enveloping case. If set, then the XML document is validated with the specified XML schema. The schema resource URI can be overwritten by the header link XmlSignatureConstantsHEADER_SCHEMA_RESOURCE_URI.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
uriDereferencer (advanced)	If you want to restrict the remote access via reference URIs, you can set an own dereferencer. Optional parameter. If not set the provider default dereferencer is used which can resolve URI fragments, HTTP, file and XPpointer URIs. Attention: The implementation is provider dependent!		URIDereferencer
addKeyInfoReference (sign)	In order to protect the KeyInfo element from tampering you can add a reference to the signed info element so that it is protected via the signature value. The default value is true. Only relevant when a KeyInfo is returned by KeyAccessor. and link KeyInfogetId() is not null.	true	Boolean
canonicalizationMethod (sign)	Canonicalization method used to canonicalize the SignedInfo element before the digest is calculated. You can use the helper methods XmlSignatureHelper.getCanonicalizationMethod(String algorithm) or getCanonicalizationMethod(String algorithm, List inclusiveNamespacePrefixes) to create a canonicalization method.	http://www.w3.org/TR/2001/REC-xml-c14n-20010315	AlgorithmMethod
contentObjectId (sign)	Sets the content object Id attribute value. By default a UUID is generated. If you set the null value, then a new UUID will be generated. Only used in the enveloping case.		String
contentReferenceType (sign)	Type of the content reference. The default value is null. This value can be overwritten by the header link XmlSignatureConstantsHEADER_CONTENT_REFERENCE_TYPE.		String

Name	Description	Default	Type
contentReferenceUri (sign)	Reference URI for the content to be signed. Only used in the enveloped case. If the reference URI contains an ID attribute value, then the resource schema URI (link setSchemaResourceUri(String)) must also be set because the schema validator will then find out which attributes are ID attributes. Will be ignored in the enveloping or detached case.		String
digestAlgorithm (sign)	Digest algorithm URI. Optional parameter. This digest algorithm is used for calculating the digest of the input message. If this digest algorithm is not specified then the digest algorithm is calculated from the signature algorithm. Example: http://www.w3.org/2001/04/xmlencsha256		String
keyAccessor (sign)	For the signing process, a private key is necessary. You specify a key accessor bean which provides this private key. The key accessor bean must implement the KeyAccessor interface. The package org.apache.camel.component.xmlsecurity.api contains the default implementation class DefaultKeyAccessor which reads the private key from a Java keystore.		KeyAccessor
parentLocalName (sign)	Local name of the parent element to which the XML signature element will be added. Only relevant for enveloped XML signature. Alternatively you can also use link setParentXpath(XPathFilterParameterSpec). Default value is null. The value must be null for enveloping and detached XML signature. This parameter or the parameter link setParentXpath(XPathFilterParameterSpec) for enveloped signature and the parameter link setXpathsToIdAttributes(List) for detached signature must not be set in the same configuration. If the parameters parentXpath and parentLocalName are specified in the same configuration then an exception is thrown.		String
parentNamespace (sign)	Namespace of the parent element to which the XML signature element will be added.		String

Name	Description	Default	Type
parentXpath (sign)	Sets the XPath to find the parent node in the enveloped case. Either you specify the parent node via this method or the local name and namespace of the parent with the methods link setParentLocalName(String) and link setParentNamespace(String). Default value is null. The value must be null for enveloping and detached XML signature. If the parameters parentXpath and parentLocalName are specified in the same configuration then an exception is thrown.		XPathFilterParameter Spec
plainText (sign)	Indicator whether the message body contains plain text. The default value is false, indicating that the message body contains XML. The value can be overwritten by the header link XmlSignatureConstantsHEADER_MESSAGE_IS_PLAIN_TEXT.	false	Boolean
plainTextEncoding (sign)	Encoding of the plain text. Only relevant if the message body is plain text (see parameter link plainText. Default value is UTF-8.	UTF-8	String
prefixForXmlSignature Namespace (sign)	Namespace prefix for the XML signature namespace http://www.w3.org/2000/09/xmldsig. Default value is ds. If null or an empty value is set then no prefix is used for the XML signature namespace. See best practice http://www.w3.org/TR/xmldsig-bestpractices/signing-xml- without-namespaces	ds	String
properties (sign)	For adding additional References and Objects to the XML signature which contain additional properties, you can provide a bean which implements the XmlSignatureProperties interface.		XmlSignatureProperties
signatureAlgorithm (sign)	Signature algorithm. Default value is http://www.w3.org/2000/09/xmldsigrsa-sha1.	http://www.w3.org/2000/09/xmldsig#rsa-sha1	String
signatureId (sign)	Sets the signature Id. If this parameter is not set (null value) then a unique ID is generated for the signature ID (default). If this parameter is set to (empty string) then no Id attribute is created in the signature element.		String

Name	Description	Default	Type
transformMethods (sign)	Transforms which are executed on the message body before the digest is calculated. By default, C14n is added and in the case of enveloped signature (see option parentLocalName) also http://www.w3.org/2000/09/xmldsigenveloped-signature is added at position 0 of the list. Use methods in XmlSignatureHelper to create the transform methods.		List
xpathsToIdAttributes (sign)	Define the elements which are signed in the detached case via XPATH expressions to ID attributes (attributes of type ID). For each element found via the XPATH expression a detached signature is created whose reference URI contains the corresponding attribute value (preceded by ''). The signature becomes the last sibling of the signed element. Elements with deeper hierarchy level are signed first. You can also set the XPATH list dynamically via the header link XmlSignatureConstantsHEADER_XPATHS_TO_ID_ATTRIBUTES. The parameter link setParentLocalName(String) or link setParentXpath(XPathFilterParameterSpec) for enveloped signature and this parameter for detached signature must not be set in the same configuration.		List
keySelector (verify)	Provides the key for validating the XML signature.		KeySelector
outputNodeSearch (verify)	Sets the output node search value for determining the node from the XML signature document which shall be set to the output message body. The class of the value depends on the type of the output node search. The output node search is forwarded to XmlSignature2Message.		String
outputNodeSearchType (verify)	Determines the search type for determining the output node which is serialized into the output message bodyF. See link setOutputNodeSearch(Object). The supported default search types you can find in DefaultXmlSignature2Message.	Default	String

Name	Description	Default	Type
removeSignatureElements (verify)	Indicator whether the XML signature elements (elements with local name Signature and namesapce http://www.w3.org/2000/09/xmldsig) shall be removed from the document set to the output message. Normally, this is only necessary, if the XML signature is enveloped. The default value is link BooleanFALSE. This parameter is forwarded to XmlSignature2Message. This indicator has no effect if the output node search is of type link DefaultXmlSignature2MessageOUTPUT_NODE_SEARCH_TYPE_DEFAULT.F	false	Boolean
secureValidation (verify)	Enables secure validation. If true then secure validation is enabled.	true	Boolean
validationFailedHandler (verify)	Handles the different validation failed situations. The default implementation throws specific exceptions for the different situations (All exceptions have the package name org.apache.camel.component.xmlsecurity.api and are a sub-class of XmlSignatureInvalidException. If the signature value validation fails, a XmlSignatureInvalidValueException is thrown. If a reference validation fails, a XmlSignatureInvalidContentHashException is thrown. For more detailed information, see the JavaDoc.		ValidationFailedHandler

Name	Description	Default	Type
xmlSignature2Message (verify)	Bean which maps the XML signature to the output-message after the validation. How this mapping should be done can be configured by the options outputNodeSearchType, outputNodeSearch, and removeSignatureElements. The default implementation offers three possibilities which are related to the three output node search types Default, ElementName, and XPath. The default implementation determines a node which is then serialized and set to the body of the output message If the search type is ElementName then the output node (which must be in this case an element) is determined by the local name and namespace defined in the search value (see option outputNodeSearch). If the search type is XPath then the output node is determined by the XPath specified in the search value (in this case the output node can be of type Element, TextNode or Document). If the output node search type is Default then the following rules apply: In the enveloped XML signature case (there is a reference with URI= and transform http://www.w3.org/2000/09/xmldsigenveloped-signature), the incoming XML document without the Signature element is set to the output message body. In the non-enveloped XML signature case, the message body is determined from a referenced Object; this is explained in more detail in chapter Output Node Determination in Enveloping XML Signature Case.		XmlSignature2Message
xmlSignatureChecker (verify)	This interface allows the application to check the XML signature before the validation is executed. This step is recommended in http://www.w3.org/TR/xmldsig-bestpractices/check-what-is-signed		XmlSignatureChecker

345.5.3. Output Node Determination in Enveloping XML Signature Case

After the validation the node is extracted from the XML signature document which is finally returned to the output-message body. In the enveloping XML signature case, the default implementation **DefaultXmlSignature2Message** of **XmlSignature2Message** does this for the node search type **Default** in the following way (see option **xmlSignature2Message**):

- First an object reference is determined:
  - Only same document references are taken into account (URI must start with **#**)
  - Also indirect same document references to an object via manifest are taken into account.
  - The resulting number of object references must be 1.

- Then, the object is dereferenced and the object must only contain one XML element. This element is returned as output node.

This does mean that the enveloping XML signature must have either the structure:

```
<Signature>
    <SignedInfo>
      <Reference URI="#object"/>
      <!-- further references possible but they must not point to an Object or Manifest containing an object reference -->
       ...
    </SignedInfo>

    <Object Id="object">
        <!-- contains one XML element which is extracted to the message body -->
    <Object>
    <!-- further object elements possible which are not referenced-->
    ...
    (<KeyInfo>)?
</Signature>
```

or the structure:

```
<Signature>
    <SignedInfo>
      <Reference URI="#manifest"/>
      <!-- further references  are possible but they must not point to an Object or other manifest containing an object reference -->
       ...
    </SignedInfo>

    <Object >
      <Manifest Id="manifest">
        <Reference URI=#object/>
      </Manifest>
    </Objet>
    <Object Id="object">
       <!-- contains the DOM node which is extracted to the message body -->
    </Object>
     <!-- further object elements possible which are not referenced -->
    ...
    (<KeyInfo>)?
</Signature>
```

345.6. DETACHED XML SIGNATURES AS SIBLINGS OF THE SIGNED ELEMENTS

Since 2.14.0

You can create detached signatures where the signature is a sibling of the signed element. The following example contains two detached signatures. The first signature is for the element **C** and the second signature is for element **A**. The signatures are *nested*; the second signature is for the element **A** which also contains the first signature.

Example Detached XML Signatures

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
   <A ID="IDforA">
      <B>
         <C ID="IDforC">
            <D>dvalue</D>
         </C>
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
            Id="_6bf13099-0568-4d76-8649-faf5dcb313c0">
            <ds:SignedInfo>
               <ds:CanonicalizationMethod
                  Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
               <ds:SignatureMethod
                  Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
               <ds:Reference URI="#IDforC">
                  ...
               </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>aUDFmiG71</ds:SignatureValue>
         </ds:Signature>
      </B>
   </A>
   <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#"Id="_6b02fb8a-30df-42c6-ba25-
76eba02c8214">
      <ds:SignedInfo>
         <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
         <ds:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
         <ds:Reference URI="#IDforA">
            ...
         </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>q3tvRoGgc8cMUqUSzP6C21zb7tt04riPnDuk=</ds:SignatureValue>
   </ds:Signature>
<root>
```

The example shows that you can sign several elements and that for each element a signature is created as sibling. The elements to be signed must have an attribute of type ID. The ID type of the attribute must be defined in the XML schema (see option **schemaResourceUri**). You specify a list of XPATH expressions pointing to attributes of type ID (see option **xpathsToIdAttributes**). These attributes determine the elements to be signed. The elements are signed by the same key given by the **keyAccessor** bean. Elements with higher (i.e. deeper) hierarchy level are signed first. In the example, the element **C** is signed before the element **A**.

Java DSL Example

```
from("direct:detached")
  .to("xmlsecurity:sign://detached?
keyAccessor=#keyAccessorBeant&xpathsToIdAttributes=#xpathsToIdAttributesBean&schemaResource
Uri=Test.xsd")
  .to("xmlsecurity:verify://detached?
```

```
keySelector=#keySelectorBean&schemaResourceUri=org/apache/camel/component/xmlsecurity/Test.xs
d")
  .to("mock:result");
```

Spring Example

```
<bean id="xpathsToIdAttributesBean" class="java.util.ArrayList">
    <constructor-arg type="java.util.Collection">
      <list>
        <bean
          class="org.apache.camel.component.xmlsecurity.api.XmlSignatureHelper"
          factory-method="getXpathFilter">
          <constructor-arg type="java.lang.String"
            value="/ns:root/a/@ID" />
          <constructor-arg>
            <map key-type="java.lang.String" value-type="java.lang.String">
              <entry key="ns" value="http://test" />
            </map>
          </constructor-arg>
        </bean>
      </list>
    </constructor-arg>
  </bean>
...
 <from uri="direct:detached" />
    <to
      uri="xmlsecurity:sign://detached?
keyAccessor=#keyAccessorBean&amp;xpathsToIdAttributes=#xpathsToIdAttributesBean&amp;schema
ResourceUri=Test.xsd" />
    <to
      uri="xmlsecurity:verify://detached?
keySelector=#keySelectorBean&amp;schemaResourceUri=Test.xsd" />
    <to uri="mock:result" />
```

345.7. XADES-BES/EPES FOR THE SIGNER ENDPOINT

Available as of Camel 2.15.0

XML Advanced Electronic Signatures (XAdES) defines extensions to XML Signature. This standard was defined by the European Telecommunication Standards Institute and allows you to create signatures which are compliant to the European Union Directive (1999/93/EC) on a Community framework for electronic signatures. XAdES defines different sets of signature properties which are called signature forms. We support the signature forms **Basic Electronic Signature** (XAdES-BES) and **Explicit Policy Based Electronic Signature** (XAdES-EPES) for the Signer Endpoint. The forms **Electronic Signature with Validation Data** XAdES-T and XAdES-C are not supported.

We support the following properties of the XAdES-EPES form ("?" denotes zero or one occurrence):

Supported XAdES-EPES Properties

```
<QualifyingProperties Target>
  <SignedProperties>
    <SignedSignatureProperties>
      (SigningTime)?
```

```
            (SigningCertificate)?
            (SignaturePolicyIdentifier)
            (SignatureProductionPlace)?
            (SignerRole)?
        </SignedSignatureProperties>
        <SignedDataObjectProperties>
            (DataObjectFormat)?
            (CommitmentTypeIndication)?
        </SignedDataObjectProperties>
    </SignedProperties>
</QualifyingProperties>
```

The properties of the XAdES-BES form are the same except that the **SignaturePolicyIdentifier** property is not part of XAdES-BES.

You can configure the XAdES-BES/EPES properties via the bean **org.apache.camel.component.xmlsecurity.api.XAdESSignatureProperties** or **org.apache.camel.component.xmlsecurity.api.DefaultXAdESSignatureProperties. XAdESSignatureProperties** does support all properties mentioned above except the **SigningCertificate** property. To get the **SigningCertificate** property, you must overwrite either the method **XAdESSignatureProperties.getSigningCertificate()** or **XAdESSignatureProperties.getSigningCertificateChain()**. The class **DefaultXAdESSignatureProperties** overwrites the method **getSigningCertificate()** and allows you to specify the signing certificate via a keystore and alias. The following example shows all parameters you can specify. If you do not need certain parameters you can just omit them.

XAdES-BES/EPES Example in Java DSL

```
 Keystore keystore = ... // load a keystore
DefaultKeyAccessor accessor = new DefaultKeyAccessor();
accessor.setKeyStore(keystore);
accessor.setPassword("password");
accessor.setAlias("cert_alias"); // signer key alias

DefaultXAdESSignatureProperties props = new DefaultXAdESSignatureProperties();
props.setNamespace("http://uri.etsi.org/01903/v1.3.2#"); // sets the namespace for the XAdES
elements; the namspace is related to the XAdES version, default value is
"http://uri.etsi.org/01903/v1.3.2#", other possible values are "http://uri.etsi.org/01903/v1.1.1#" and
"http://uri.etsi.org/01903/v1.2.2#"
props.setPrefix("etsi"); // sets the prefix for the XAdES elements, default value is "etsi"

// signing certificate
props.setKeystore(keystore));
props.setAlias("cert_alias"); // specify the alias of the signing certificate in the keystore = signer key
alias
props.setDigestAlgorithmForSigningCertificate(DigestMethod.SHA256); // possible values for the
algorithm are "http://www.w3.org/2000/09/xmldsig#sha1",
"http://www.w3.org/2001/04/xmlenc#sha256", "http://www.w3.org/2001/04/xmldsig-more#sha384",
"http://www.w3.org/2001/04/xmlenc#sha512", default value is
"http://www.w3.org/2001/04/xmlenc#sha256"
props.setSigningCertificateURIs(Collections.singletonList("http://certuri"));

// signing time
props.setAddSigningTime(true);

// policy
```

```
props.setSignaturePolicy(XAdESSignatureProperties.SIG_POLICY_EXPLICIT_ID);
// also the values XAdESSignatureProperties.SIG_POLICY_NONE ("None"), and
XAdESSignatureProperties.SIG_POLICY_IMPLIED ("Implied")are possible, default value is
XAdESSignatureProperties.SIG_POLICY_EXPLICIT_ID ("ExplicitId")
// For "None" and "Implied" you must not specify any further policy parameters
props.setSigPolicyId("urn:oid:1.2.840.113549.1.9.16.6.1");
props.setSigPolicyIdQualifier("OIDAsURN"); //allowed values are empty string, "OIDAsURI",
"OIDAsURN"; default value is empty string
props.setSigPolicyIdDescription("invoice version 3.1");
props.setSignaturePolicyDigestAlgorithm(DigestMethod.SHA256);// possible values for the algorithm
are "http://www.w3.org/2000/09/xmldsig#sha1", http://www.w3.org/2001/04/xmlenc#sha256",
"http://www.w3.org/2001/04/xmldsig-more#sha384", "http://www.w3.org/2001/04/xmlenc#sha512",
default value is http://www.w3.org/2001/04/xmlenc#sha256"
props.setSignaturePolicyDigestValue("Ohixl6upD6av8N7pEvDABhEL6hM=");
// you can add  qualifiers for the signature policy either by specifying text or an XML fragment with the
root element "SigPolicyQualifier"
props.setSigPolicyQualifiers(Arrays
    .asList(new String[] {
        "<SigPolicyQualifier xmlns=\"http://uri.etsi.org/01903/v1.3.2#\">
<SPURI>http://test.com/sig.policy.pdf</SPURI><SPUserNotice><ExplicitText>display
text</ExplicitText>"
            + "</SPUserNotice></SigPolicyQualifier>", "category B" }));
props.setSigPolicyIdDocumentationReferences(Arrays.asList(new String[]
{"http://test.com/policy.doc.ref1.txt",
    "http://test.com/policy.doc.ref2.txt" }));

// production place
props.setSignatureProductionPlaceCity("Munich");
props.setSignatureProductionPlaceCountryName("Germany");
props.setSignatureProductionPlacePostalCode("80331");
props.setSignatureProductionPlaceStateOrProvince("Bavaria");

//role
// you can add claimed roles either by specifying text or an XML fragment with the root element
"ClaimedRole"
props.setSignerClaimedRoles(Arrays.asList(new String[] {"test",
    "<a:ClaimedRole xmlns:a=\"http://uri.etsi.org/01903/v1.3.2#\"><TestRole>TestRole</TestRole>
</a:ClaimedRole>" }));
props.setSignerCertifiedRoles(Collections.singletonList(new
XAdESEncapsulatedPKIData("Ahixl6upD6av8N7pEvDABhEL6hM=",
    "http://uri.etsi.org/01903/v1.2.2#DER", "IdCertifiedRole")));

// data object format
props.setDataObjectFormatDescription("invoice");
props.setDataObjectFormatMimeType("text/xml");
props.setDataObjectFormatIdentifier("urn:oid:1.2.840.113549.1.9.16.6.2");
props.setDataObjectFormatIdentifierQualifier("OIDAsURN"); //allowed values are empty string,
"OIDAsURI", "OIDAsURN"; default value is empty string
props.setDataObjectFormatIdentifierDescription("identifier desc");
props.setDataObjectFormatIdentifierDocumentationReferences(Arrays.asList(new String[] {
    "http://test.com/dataobject.format.doc.ref1.txt", "http://test.com/dataobject.format.doc.ref2.txt" }));

//commitment
props.setCommitmentTypeId("urn:oid:1.2.840.113549.1.9.16.6.4");
props.setCommitmentTypeIdQualifier("OIDAsURN"); //allowed values are empty string, "OIDAsURI",
"OIDAsURN"; default value is empty string
```

```
props.setCommitmentTypeIdDescription("description for commitment type ID");
props.setCommitmentTypeIdDocumentationReferences(Arrays.asList(new String[]
{"http://test.com/commitment.ref1.txt",
    "http://test.com/commitment.ref2.txt" }));
// you can specify a commitment type qualifier either by simple text or an XML fragment with root
element "CommitmentTypeQualifier"
props.setCommitmentTypeQualifiers(Arrays.asList(new String[] {"commitment qualifier",
    "<c:CommitmentTypeQualifier xmlns:c=\"http://uri.etsi.org/01903/v1.3.2#\"><C>c</C>
</c:CommitmentTypeQualifier>" }));

beanRegistry.bind("xmlSignatureProperties",props);
beanRegistry.bind("keyAccessorDefault",keyAccessor);

// you must reference the properties bean in the "xmlsecurity" URI
from("direct:xades").to("xmlsecurity:sign://xades?
keyAccessor=#keyAccessorDefault&properties=#xmlSignatureProperties")
            .to("mock:result");
```

XAdES-BES/EPES Example in Spring XML

```
...
<from uri="direct:xades" />
    <to
        uri="xmlsecurity:sign://xades?keyAccessor=#accessorRsa&amp;properties=#xadesProperties"
/>
    <to uri="mock:result" />
...
<bean id="xadesProperties"
    class="org.apache.camel.component.xmlsecurity.api.XAdESSignatureProperties">
    <!-- For more properties see the the previous Java DSL example.
         If you want to have a signing certificate then use the bean class
DefaultXAdESSignatureProperties (see the previous Java DSL example). -->
    <property name="signaturePolicy" value="ExplicitId" />
    <property name="sigPolicyId" value="http://www.test.com/policy.pdf" />
    <property name="sigPolicyIdDescription" value="factura" />
    <property name="signaturePolicyDigestAlgorithm"
value="http://www.w3.org/2000/09/xmldsig#sha1" />
    <property name="signaturePolicyDigestValue" value="Ohixl6upD6av8N7pEvDABhEL1hM=" />
    <property name="signerClaimedRoles" ref="signerClaimedRoles_XMLSigner" />
    <property name="dataObjectFormatDescription" value="Factura electrónica" />
    <property name="dataObjectFormatMimeType" value="text/xml" />
</bean>
<bean class="java.util.ArrayList" id="signerClaimedRoles_XMLSigner">
    <constructor-arg>
        <list>
            <value>Emisor</value>
            <value>&lt;ClaimedRole
                xmlns=&quot;http://uri.etsi.org/01903/v1.3.2#&quot;&gt;&lt;test
                xmlns=&quot;http://test.com/&quot;&gt;test&lt;/test&gt;&lt;/ClaimedRole&gt;</value>
        </list>
    </constructor-arg>
</bean>
```

345.7.1. Headers

Header	Type	Description
CamelXmlSignatureXAdESQualifyingPropertiesId	**String**	for the 'Id' attribute value of **QualifyingProperties** element
CamelXmlSignatureXAdESSignedDataObjectPropertiesId	**String**	for the 'Id' attribute value of **SignedDataObjectProperties** element
CamelXmlSignatureXAdESSignedSignaturePropertiesId	**String**	for the 'Id' attribute value of **SignedSignatureProperties** element
CamelXmlSignatureXAdESDataObjectFormatEncoding	**String**	for the value of the Encoding element of the **DataObjectFormat** element
CamelXmlSignatureXAdESNamespace	**String**	overwrites the XAdES namespace parameter value
CamelXmlSignatureXAdESPrefix	**String**	overwrites the XAdES prefix parameter value

345.7.2. Limitations with regard to XAdES version 1.4.2

- No support for signature form XAdES-T and XAdES-C
- Only signer part implemented. Verifier part currently not available.
- No support for the **QualifyingPropertiesReference** element (see section 6.3.2 of spec).
- No support for the **Transforms** element contained in the **SignaturePolicyId** element contained in the **SignaturePolicyIdentifier element**
- No support of the **CounterSignature** element → no support for the **UnsignedProperties** element
- At most one **DataObjectFormat** element. More than one **DataObjectFormat** element makes no sense because we have only one data object which is signed (this is the incoming message body to the XML signer endpoint).

- At most one **CommitmentTypeIndication** element. More than one **CommitmentTypeIndication** element makes no sense because we have only one data object which is signed (this is the incoming message body to the XML signer endpoint).
- A **CommitmentTypeIndication** element contains always the **AllSignedDataObjects** element. The **ObjectReference** element within **CommitmentTypeIndication** element is not supported.
- The **AllDataObjectsTimeStamp** element is not supported
- The **IndividualDataObjectsTimeStamp** element is not supported

345.8. SEE ALSO

- Best Practices

CHAPTER 346. XMPP COMPONENT

Available as of Camel version 1.0

The **xmpp:** component implements an XMPP (Jabber) transport.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xmpp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

346.1. URI FORMAT

```
xmpp://[login@]hostname[:port][/participant][?Options]
```

The component supports both room based and private person-person conversations.
The component supports both producer and consumer (you can get messages from XMPP or send messages to XMPP). Consumer mode supports rooms starting.

You can append query options to the URI in the following format, **?option=value&option=value&...**

346.2. OPTIONS

The XMPP component has no options.

The XMPP endpoint is configured using URI syntax:

```
xmpp:host:port/participant
```

with the following path and query parameters:

346.2.1. Path Parameters (3 parameters):

Name	Description	Default	Type
host	**Required** Hostname for the chat server		String
port	**Required** Port number for the chat server		int
participant	JID (Jabber ID) of person to receive messages. room parameter has precedence over participant.		String

346.2.2. Query Parameters (18 parameters):

Name	Description	Default	Type
login (common)	Whether to login the user.	true	boolean
nickname (common)	Use nickname when joining room. If room is specified and nickname is not, user will be used for the nickname.		String
pubsub (common)	Accept pubsub packets on input, default is false	false	boolean
room (common)	If this option is specified, the component will connect to MUC (Multi User Chat). Usually, the domain name for MUC is different from the login domain. For example, if you are supermanjabber.org and want to join the krypton room, then the room URL is kryptonconference.jabber.org. Note the conference part. It is not a requirement to provide the full room JID. If the room parameter does not contain the symbol, the domain part will be discovered and added by Camel		String
serviceName (common)	The name of the service you are connecting to. For Google Talk, this would be gmail.com.		String
testConnectionOnStartup (common)	Specifies whether to test the connection on startup. This is used to ensure that the XMPP client has a valid connection to the XMPP server when the route starts. Camel throws an exception on startup if a connection cannot be established. When this option is set to false, Camel will attempt to establish a lazy connection when needed by a producer, and will poll for a consumer connection until the connection is established. Default is true.	true	boolean
createAccount (common)	If true, an attempt to create an account will be made. Default is false.	false	boolean
resource (common)	XMPP resource. The default is Camel.	Camel	String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
connectionPollDelay (consumer)	The amount of time in seconds between polls (in seconds) to verify the health of the XMPP connection, or between attempts to establish an initial consumer connection. Camel will try to re-establish a connection if it has become inactive. Default is 10 seconds.	10	int
doc (consumer)	Set a doc header on the IN message containing a Document form of the incoming packet; default is true if presence or pubsub are true, otherwise false	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
connectionConfig (advanced)	To use an existing connection configuration. Currently org.jivesoftware.smack.tcp.XMPPTCPConnectionConfiguration is only supported (XMPP over TCP).		ConnectionConfiguration
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
headerFilterStrategy (filter)	To use a custom HeaderFilterStrategy to filter header to and from Camel message.		HeaderFilterStrategy
password (security)	Password for login		String
user (security)	User name (without server name). If not specified, anonymous login will be attempted.		String

346.3. HEADERS AND SETTING SUBJECT OR LANGUAGE

Camel sets the message IN headers as properties on the XMPP message. You can configure a **HeaderFilterStategy** if you need custom filtering of headers. The **Subject** and **Language** of the XMPP message are also set if they are provided as IN headers.

346.4. EXAMPLES

User **superman** to join room **krypton** at **jabber** server with password, **secret**:

```
xmpp://superman@jabber.org/?room=krypton@conference.jabber.org&password=secret
```

User **superman** to send messages to **joker**:

```
xmpp://superman@jabber.org/joker@jabber.org?password=secret
```

Routing example in Java:

```
from("timer://kickoff?period=10000").
setBody(constant("I will win!\n Your Superman.")).
to("xmpp://superman@jabber.org/joker@jabber.org?password=secret");
```

Consumer configuration, which writes all messages from **joker** into the queue, **evil.talk**.

```
from("xmpp://superman@jabber.org/joker@jabber.org?password=secret").
to("activemq:evil.talk");
```

Consumer configuration, which listens to room messages:

```
from("xmpp://superman@jabber.org/?password=secret&room=krypton@conference.jabber.org").
to("activemq:krypton.talk");
```

Room in short notation (no domain part):

```
from("xmpp://superman@jabber.org/?password=secret&room=krypton").
to("activemq:krypton.talk");
```

When connecting to the Google Chat service, you'll need to specify the **serviceName** as well as your credentials:

```
from("direct:start").
  to("xmpp://talk.google.com:5222/touser@gmail.com?
serviceName=gmail.com&user=fromuser&password=secret").
  to("mock:result");
```

346.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 347. XPATH LANGUAGE

Available as of Camel version 1.1

Camel supports XPath to allow an Expression or Predicate to be used in the DSL or Xml Configuration. For example you could use XPath to create an Predicate in a Message Filter or as an Expression for a Recipient List.

Streams

If the message body is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**. So often when you use XPath as Message Filter or Content Based Router then you need to access the data multiple times, and you should use Stream Caching or convert the message body to a **String** prior which is safe to be re-read multiple times.

```
from("queue:foo").
  filter().xpath("//foo")).
  to("queue:bar")
```

```
from("queue:foo").
  choice().xpath("//foo")).to("queue:bar").
  otherwise().to("queue:others");
```

347.1. XPATH LANGUAGE OPTIONS

The XPath language supports 9 options which are listed below.

Name	Default	Java Type	Description
documentType		**String**	Name of class for document type The default value is org.w3c.dom.Document
resultType	**NODESET**	**String**	Sets the class name of the result type (type from output) The default result type is NodeSet
saxon	**false**	**Boolean**	Whether to use Saxon.
factoryRef		**String**	References to a custom XPathFactory to lookup in the registry
objectModel		**String**	The XPath object model to use
logNamespaces	**false**	**Boolean**	Whether to log namespaces which can assist during trouble shooting
headerName		**String**	Name of header to use as input, instead of the message body

Name	Default	Java Type	Description
threadSafety	**false**	**Boolean**	Whether to enable thread-safety for the returned result of the xpath expression. This applies to when using NODESET as the result type, and the returned set has multiple elements. In this situation there can be thread-safety issues if you process the NODESET concurrently such as from a Camel Splitter EIP in parallel processing mode. This option prevents concurrency issues by doing defensive copies of the nodes. It is recommended to turn this option on if you are using camel-saxon or Saxon in your application. Saxon has thread-safety issues which can be prevented by turning this option on.
trim	**true**	**Boolean**	Whether to trim the value to remove leading and trailing whitespaces and line breaks

347.2. NAMESPACES

You can easily use namespaces with XPath expressions using the Namespaces helper class.

347.3. VARIABLES

Variables in XPath is defined in different namespaces. The default namespace is **http://camel.apache.org/schema/spring**.

Namespace URI	Local part	Type	Description
http://camel.apache.org/xml/in/	in	Message	the exchange.in message
http://camel.apache.org/xml/out/	out	Message	the exchange.out message
http://camel.apache.org/xml/function/	functions	Object	**Camel 2.5:** Additional functions

Namespace URI	Local part	Type	Description
http://camel.apache.org/xml/variables/environment-variables	env	Object	OS environment variables
http://camel.apache.org/xml/variables/system-properties	system	Object	Java System properties
http://camel.apache.org/xml/variables/exchange-property		Object	the exchange property

Camel will resolve variables according to either:

- namespace given
- no namespace given

347.3.1. Namespace given

If the namespace is given then Camel is instructed exactly what to return. However when resolving either **in** or **out** Camel will try to resolve a header with the given local part first, and return it. If the local part has the value **body** then the body is returned instead.

347.3.2. No namespace given

If there is no namespace given then Camel resolves only based on the local part. Camel will try to resolve a variable in the following steps:

- from **variables** that has been set using the **variable(name, value)** fluent builder

- from message.in.header if there is a header with the given key
- from exchange.properties if there is a property with the given key

347.4. FUNCTIONS

Camel adds the following XPath functions that can be used to access the exchange:

Function	Argument	Type	Description
in:body	none	Object	Will return the **in** message body.
in:header	the header name	Object	Will return the **in** message header.
out:body	none	Object	Will return the **out** message body.
out:header	the header name	Object	Will return the **out** message header.
function:properties	key for property	String	**Camel 2.5:** To lookup a property using the Properties component (property placeholders).
function:simple	simple expression	Object	**Camel 2.5:** To evaluate a Simple expression.

CAUTION

function:properties and **function:simple** is not supported when the return type is a **NodeSet**, such as when using with a Splitter EIP.

Here's an example showing some of these functions in use.

And the new functions introduced in Camel 2.5:

347.5. USING XML CONFIGURATION

If you prefer to configure your routes in your Spring XML file then you can use XPath expressions as follows

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring"
xmlns:foo="http://example.com/person">
    <route>
      <from uri="activemq:MyQueue"/>
      <filter>
        <xpath>/foo:person[@name='James']</xpath>
        <to uri="mqseries:SomeOtherQueue"/>
      </filter>
    </route>
  </camelContext>
</beans>
```

Notice how we can reuse the namespace prefixes, **foo** in this case, in the XPath expression for easier namespace based XPath expressions!

See also this discussion on the mailinglist about using your own namespaces with xpath

347.6. SETTING RESULT TYPE

The XPath expression will return a result type using native XML objects such as **org.w3c.dom.NodeList**. But many times you want a result type to be a String. To do this you have to instruct the XPath which result type to use.

In Java DSL:

```
xpath("/foo:person/@id", String.class)
```

In Spring DSL you use the **resultType** attribute to provide a fully qualified classname:

```
<xpath resultType="java.lang.String">/foo:person/@id</xpath>
```

In @XPath:
Available as of Camel 2.1

```
@XPath(value = "concat('foo-',//order/name/)", resultType = String.class) String name)
```

Where we use the xpath function concat to prefix the order name with **foo-**. In this case we have to specify that we want a String as result type so the concat function works.

347.7. USING XPATH ON HEADERS

Available as of Camel 2.11

Some users may have XML stored in a header. To apply an XPath to a header's value you can do this by defining the 'headerName' attribute.

And in Java DSL you specify the headerName as the 2nd parameter as shown:

```
  xpath("/invoice/@orderType = 'premium'", "invoiceDetails")
```

347.8. EXAMPLES

Here is a simple example using an XPath expression as a predicate in a Message Filter

If you have a standard set of namespaces you wish to work with and wish to share them across many different XPath expressions you can use the NamespaceBuilder as shown in this example

In this sample we have a choice construct. The first choice evaulates if the message has a header key **type** that has the value **Camel**.
The 2nd choice evaluates if the message body has a name tag **<name>** which values is **Kong**.
If neither is true the message is routed in the otherwise block:

And the spring XML equivalent of the route:

347.9. XPATH INJECTION

You can use Bean Integration to invoke a method on a bean and use various languages such as XPath to extract a value from the message and bind it to a method parameter.

The default XPath annotation has SOAP and XML namespaces available. If you want to use your own namespace URIs in an XPath expression you can use your own copy of the XPath annotation to create whatever namespace prefixes you want to use.

i.e. cut and paste upper code to your own project in a different package and/or annotation name then add whatever namespace prefix/uris you want in scope when you use your annotation on a method parameter. Then when you use your annotation on a method parameter all the namespaces you want will be available for use in your XPath expression.

For example

```
public class Foo {

    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@MyXPath("/ns1:foo/ns2:bar/text()") String correlationID, @Body String
body) {
        // process the inbound message here
    }
}
```

347.10. USING XPATHBUILDER WITHOUT AN EXCHANGE

Available as of Camel 2.3

You can now use the **org.apache.camel.builder.XPathBuilder** without the need for an Exchange. This comes handy if you want to use it as a helper to do custom xpath evaluations.

It requires that you pass in a CamelContext since a lot of the moving parts inside the XPathBuilder requires access to the Camel Type Converter and hence why CamelContext is needed.

For example you can do something like this:

```
boolean matches = XPathBuilder.xpath("/foo/bar/@xyz").matches(context, "<foo><bar xyz='cheese'/>
</foo>"));
```

This will match the given predicate.

You can also evaluate for example as shown in the following three examples:

```
String name = XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>cheese</bar></foo>",
String.class);
Integer number = XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>123</bar></foo>",
Integer.class);
Boolean bool = XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>true</bar></foo>",
Boolean.class);
```

Evaluating with a String result is a common requirement and thus you can do it a bit simpler:

```
String name = XPathBuilder.xpath("foo/bar").evaluate(context, "<foo><bar>cheese</bar></foo>");
```

347.11. USING SAXON WITH XPATHBUILDER

Available as of Camel 2.3

You need to add **camel-saxon** as dependency to your project.

Its now easier to use Saxon with the XPathBuilder which can be done in several ways as shown below. Where as the latter ones are the easiest ones.

- Using a factory
- Using ObjectModel

The easy one

347.12. SETTING A CUSTOM XPATHFACTORY USING SYSTEM PROPERTY

Available as of Camel 2.3

Camel now supports reading the JVM system property **javax.xml.xpath.XPathFactory** that can be used to set a custom XPathFactory to use.

This unit test shows how this can be done to use Saxon instead:

Camel will log at **INFO** level if it uses a non default XPathFactory such as:

```
XPathBuilder  INFO  Using system property
javax.xml.xpath.XPathFactory:http://saxon.sf.net/jaxp/xpath/om with value:
                net.sf.saxon.xpath.XPathFactoryImpl when creating XPathFactory
```

To use Apache Xerces you can configure the system property

```
-Djavax.xml.xpath.XPathFactory=org.apache.xpath.jaxp.XPathFactoryImpl
```

347.13. ENABLING SAXON FROM SPRING DSL

Available as of Camel 2.10

Similarly to Java DSL, to enable Saxon from Spring DSL you have three options:

Specifying the factory

```
<xpath factoryRef="saxonFactory" resultType="java.lang.String">current-dateTime()</xpath>
```

Specifying the object model

```
<xpath objectModel="http://saxon.sf.net/jaxp/xpath/om" resultType="java.lang.String">current-
dateTime()</xpath>
```

Shortcut

```
<xpath saxon="true" resultType="java.lang.String">current-dateTime()</xpath>
```

347.14. NAMESPACE AUDITING TO AID DEBUGGING

Available as of Camel 2.10

A large number of XPath-related issues that users frequently face are linked to the usage of namespaces. You may have some misalignment between the namespaces present in your message and those that your XPath expression is aware of or referencing. XPath predicates or expressions that are unable to locate the XML elements and attributes due to namespaces issues may simply look like "they are not working", when in reality all there is to it is a lack of namespace definition.

Namespaces in XML are completely necessary, and while we would love to simplify their usage by implementing some magic or voodoo to wire namespaces automatically, truth is that any action down this path would disagree with the standards and would greatly hinder interoperability.

Therefore, the utmost we can do is assist you in debugging such issues by adding two new features to the XPath Expression Language and are thus accesible from both predicates and expressions.

#=== Logging the Namespace Context of your XPath expression/predicate

Every time a new XPath expression is created in the internal pool, Camel will log the namespace context of the expression under the **org.apache.camel.builder.xml.XPathBuilder** logger. Since Camel represents Namespace Contexts in a hierarchical fashion (parent-child relationships), the entire tree is output in a recursive manner with the following format:

```
[me: {prefix -> namespace}, {prefix -> namespace}], [parent: [me: {prefix -> namespace}, {prefix ->
namespace}], [parent: [me: {prefix -> namespace}]]]
```

Any of these options can be used to activate this logging:

1. Enable TRACE logging on the **org.apache.camel.builder.xml.XPathBuilder** logger, or some parent logger such as **org.apache.camel** or the root logger
2. Enable the **logNamespaces** option as indicated in Auditing Namespaces, in which case the logging will occur on the INFO level

347.15. AUDITING NAMESPACES

Camel is able to discover and dump all namespaces present on every incoming message before evaluating an XPath expression, providing all the richness of information you need to help you analyse and pinpoint possible namespace issues.

To achieve this, it in turn internally uses another specially tailored XPath expression to extract all namespace mappings that appear in the message, displaying the prefix and the full namespace URI(s) for each individual mapping.

Some points to take into account:

- The implicit XML namespace (xmlns:xml="http://www.w3.org/XML/1998/namespace") is suppressed from the output because it adds no value
- Default namespaces are listed under the DEFAULT keyword in the output
- Keep in mind that namespaces can be remapped under different scopes. Think of a top-level 'a' prefix which in inner elements can be assigned a different namespace, or the default namespace changing in inner scopes. For each discovered prefix, all associated URIs are listed.

You can enable this option in Java DSL and Spring DSL.

Java DSL:

```
XPathBuilder.xpath("/foo:person/@id", String.class).logNamespaces()
```

Spring DSL:

```
<xpath logNamespaces="true" resultType="String">/foo:person/@id</xpath>
```

The result of the auditing will be appear at the INFO level under the **org.apache.camel.builder.xml.XPathBuilder** logger and will look like the following:

```
2012-01-16 13:23:45,878 [stSaxonWithFlag] INFO  XPathBuilder  - Namespaces discovered in
message:
{xmlns:a=[http://apache.org/camel], DEFAULT=[http://apache.org/default],
xmlns:b=[http://apache.org/camelA, http://apache.org/camelB]}
```

347.16. LOADING SCRIPT FROM EXTERNAL RESOURCE

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as **"classpath:"**, **"file:"**, or **"http:"**.
This is done using the following syntax: **"resource:scheme:location"**, eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").xpath("resource:classpath:myxpath.txt", String.class)
```

347.17. DEPENDENCIES

The XPath language is part of camel-core.

CHAPTER 348. XQUERY COMPONENT

Available as of Camel version 1.0

Camel supports XQuery to allow an Expression or Predicate to be used in the DSL or Xml Configuration. For example you could use XQuery to create an Predicate in a Message Filter or as an Expression for a Recipient List.

348.1. OPTIONS

The XQuery component supports 4 options which are listed below.

Name	Description	Default	Type
moduleURIResolver (advanced)	To use the custom ModuleURIResolver		ModuleURIResolver
configuration (advanced)	To use a custom Saxon configuration		Configuration
configurationProperties (advanced)	To set custom Saxon configuration properties		Map
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The XQuery endpoint is configured using URI syntax:

```
xquery:resourceUri
```

with the following path and query parameters:

348.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** The name of the template to load from classpath or file system		String

348.1.2. Query Parameters (31 parameters):

Name	Description	Default	Type
allowStAX (common)	Whether to allow using StAX mode	false	boolean
headerName (common)	To use a Camel Message header as the input source instead of Message body.		String
namespacePrefixes (common)	Allows to control which namespace prefixes to use for a set of namespace mappings		Map
resultsFormat (common)	What output result to use	DOM	ResultFormat
resultType (common)	What output result to use defined as a class		Class<?>
stripsAllWhiteSpace (common)	Whether to strip all whitespaces	true	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy

Name	Description	Default	Type
configuration (advanced)	To use a custom Saxon configuration		Configuration
configurationProperties (advanced)	To set custom Saxon configuration properties		Map
moduleURIResolver (advanced)	To use the custom ModuleURIResolver		ModuleURIResolver
parameters (advanced)	Additional parameters		Map
properties (advanced)	Properties to configure the serialization parameters		Properties
staticQueryContext (advanced)	To use a custom Saxon StaticQueryContext		StaticQueryContext
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
delay (scheduler)	Milliseconds before the next poll. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	500	long
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean

Name	Description	Default	Type
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean

348.2. EXAMPLES

```
from("queue:foo").filter().
  xquery("//foo").
  to("queue:bar")
```

You can also use functions inside your query, in which case you need an explicit type conversion (or you will get a org.w3c.dom.DOMException: HIERARCHY_REQUEST_ERR) by passing the Class as a second argument to the **xquery()** method.

```
from("direct:start").
  recipientList().xquery("concat('mock:foo.', /person/@city)", String.class);
```

348.3. VARIABLES

The IN message body will be set as the **contextItem**. Besides this these Variables is also added as parameters:

Variable	Type	Description
exchange	Exchange	The current Exchange
in.body	Object	The In message's body
out.body	Object	The OUT message's body (if any)
in.headers.*	Object	You can access the value of exchange.in.headers with key **foo** by using the variable which name is in.headers.foo
out.headers.*	Object	You can access the value of exchange.out.headers with key **foo** by using the variable which name is out.headers.foo variable
key name	Object	Any exchange.properties and exchange.in.headers and any additional parameters set using **setParameters(Map)**. These parameters is added with they own key name, for instance if there is an IN header with the key name **foo** then its added as **foo**.

348.4. USING XML CONFIGURATION

If you prefer to configure your routes in your Spring XML file then you can use XPath expressions as follows

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:foo="http://example.com/person"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-
spring.xsd">

  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
      <from uri="activemq:MyQueue"/>
      <filter>
        <xquery>/foo:person[@name='James']</xquery>
        <to uri="mqseries:SomeOtherQueue"/>
      </filter>
    </route>
  </camelContext>
</beans>
```

Notice how we can reuse the namespace prefixes, **foo** in this case, in the XPath expression for easier namespace based XQuery expressions!

When you use functions in your XQuery expression you need an explicit type conversion which is done in the xml configuration via the @**type** attribute:

```
    <xquery type="java.lang.String">concat('mock:foo.', /person/@city)</xquery>
```

348.5. USING XQUERY AS TRANSFORMATION

We can do a message translation using transform or setBody in the route, as shown below:

```
from("direct:start").
  transform().xquery("/people/person");
```

Notice that xquery will use DOMResult by default, so if we want to grab the value of the person node, using text() we need to tell xquery to use String as result type, as shown:

```
from("direct:start").
  transform().xquery("/people/person/text()", String.class);
```

348.6. USING XQUERY AS AN ENDPOINT

Sometimes an XQuery expression can be quite large; it can essentally be used for Templating. So you may want to use an XQuery Endpoint so you can route using XQuery templates.

The following example shows how to take a message of an ActiveMQ queue (MyQueue) and transform it using XQuery and send it to MQSeries.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="activemq:MyQueue"/>
    <to uri="xquery:com/acme/someTransform.xquery"/>
    <to uri="mqseries:SomeOtherQueue"/>
  </route>
</camelContext>
```

Currently custom functions in XQuery might result in a NullPointerException (Camel 2.18, 2.19 and 2.20). This is expected to be fixed in Camel 2.21

348.7. EXAMPLES

Here is a simple example using an XQuery expression as a predicate in a Message Filter

This example uses XQuery with namespaces as a predicate in a Message Filter

348.8. LEARNING XQUERY

XQuery is a very powerful language for querying, searching, sorting and returning XML. For help learning XQuery try these tutorials

- Mike Kay's XQuery Primer
- the W3Schools XQuery Tutorial

You might also find the XQuery function reference useful

348.9. LOADING SCRIPT FROM EXTERNAL RESOURCE

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as **"classpath:"**, **"file:"**, or **"http:"**.
This is done using the following syntax: **"resource:scheme:location"**, eg to refer to a file on the classpath you can do:

```
.setHeader("myHeader").xquery("resource:classpath:myxquery.txt", String.class)
```

348.10. DEPENDENCIES

To use XQuery in your camel routes you need to add the a dependency on **camel-saxon** which implements the XQuery language.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-saxon</artifactId>
  <version>x.x.x</version>
</dependency>
```

CHAPTER 349. XSLT COMPONENT

Available as of Camel version 1.3

The **xslt:** component allows you to process a message using an XSLT template. This can be ideal when using Templating to generate respopnses for requests.

349.1. URI FORMAT

```
xslt:templateName[?options]
```

The URI format contains **templateName**, which can be one of the following:

- the classpath-local URI of the template to invoke
- the complete URL of the remote template.

You can append query options to the URI in the following format:

?option=value&option=value&...

Refer to the Spring Documentation for more detail of the URI syntax.

Table 349.1. Example URIs

URI	Description
xslt:com/acme/mytransform.xsl	Refers to the file com/acme/mytransform.xsl on the classpath
xslt:file:///foo/bar.xsl	Refers to the file /foo/bar.xsl
xslt:http://acme.com/cheese/foo.xsl	Refers to the remote http resource

For **Camel 2.8** or older, Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

From **Camel 2.9** onwards the XSLT component is provided directly in the camel-core.

349.2. OPTIONS

The XSLT component supports 9 options which are listed below.

Name	Description	Default	Type
xmlConverter (advanced)	To use a custom implementation of org.apache.camel.converter.jaxp.XmlConverter		XmlConverter
uriResolverFactory (advanced)	To use a custom UriResolver which depends on a dynamic endpoint resource URI. Should not be used together with the option 'uriResolver'.		XsltUriResolverFactory
uriResolver (advanced)	To use a custom UriResolver. Should not be used together with the option 'uriResolverFactory'.		URIResolver
contentCache (producer)	Cache for the resource content (the stylesheet file) when it is loaded. If set to false Camel will reload the stylesheet file on each message processing. This is good for development. A cached stylesheet can be forced to reload at runtime via JMX using the clearCachedStylesheet operation.	true	boolean
saxon (producer)	Whether to use Saxon as the transformerFactoryClass. If enabled then the class net.sf.saxon.TransformerFactoryImpl. You would need to add Saxon to the classpath.	false	boolean
saxonExtensionFunctions (advanced)	Allows you to use a custom net.sf.saxon.lib.ExtensionFunctionDefinition. You would need to add camel-saxon to the classpath. The function is looked up in the registry, where you can comma to separate multiple values to lookup.		String
saxonConfiguration (advanced)	To use a custom Saxon configuration		Object
saxonConfiguration Properties (advanced)	To set custom Saxon configuration properties		Map
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The XSLT endpoint is configured using URI syntax:

```
xslt:resourceUri
```

with the following path and query parameters:

349.2.1. Path Parameters (1 parameters):

Name	Description	Default	Type
resourceUri	**Required** Path to the template. The following is supported by the default URIResolver. You can prefix with: classpath, file, http, ref, or bean. classpath, file and http loads the resource using these protocols (classpath is default). ref will lookup the resource in the registry. bean will call a method on a bean to be used as the resource. For bean you can specify the method name after dot, eg bean:myBean.myMethod		String

349.2.2. Query Parameters (17 parameters):

Name	Description	Default	Type
allowStAX (producer)	Whether to allow using StAX as the javax.xml.transform.Source.	true	boolean
contentCache (producer)	Cache for the resource content (the stylesheet file) when it is loaded. If set to false Camel will reload the stylesheet file on each message processing. This is good for development. A cached stylesheet can be forced to reload at runtime via JMX using the clearCachedStylesheet operation.	true	boolean
deleteOutputFile (producer)	If you have output=file then this option dictates whether or not the output file should be deleted when the Exchange is done processing. For example suppose the output file is a temporary file, then it can be a good idea to delete it after use.	false	boolean
failOnNullBody (producer)	Whether or not to throw an exception if the input body is null.	true	boolean
output (producer)	Option to specify which output type to use. Possible values are: string, bytes, DOM, file. The first three options are all in memory based, where as file is streamed directly to a java.io.File. For file you must specify the filename in the IN header with the key Exchange.XSLT_FILE_NAME which is also CamelXsltFileName. Also any paths leading to the filename must be created beforehand, otherwise an exception is thrown at runtime.	string	XsltOutput
saxon (producer)	Whether to use Saxon as the transformerFactoryClass. If enabled then the class net.sf.saxon.TransformerFactoryImpl. You would need to add Saxon to the classpath.	false	boolean

Name	Description	Default	Type
transformerCacheSize (producer)	The number of javax.xml.transform.Transformer object that are cached for reuse to avoid calls to Template.newTransformer().	0	int
converter (advanced)	To use a custom implementation of org.apache.camel.converter.jaxp.XmlConverter		XmlConverter
entityResolver (advanced)	To use a custom org.xml.sax.EntityResolver with javax.xml.transform.sax.SAXSource.		EntityResolver
errorListener (advanced)	Allows to configure to use a custom javax.xml.transform.ErrorListener. Beware when doing this then the default error listener which captures any errors or fatal errors and store information on the Exchange as properties is not in use. So only use this option for special use-cases.		ErrorListener
resultHandlerFactory (advanced)	Allows you to use a custom org.apache.camel.builder.xml.ResultHandlerFactory which is capable of using custom org.apache.camel.builder.xml.ResultHandler types.		ResultHandlerFactory
saxonConfiguration (advanced)	To use a custom Saxon configuration		Object
saxonExtensionFunctions (advanced)	Allows you to use a custom net.sf.saxon.lib.ExtensionFunctionDefinition. You would need to add camel-saxon to the classpath. The function is looked up in the registry, where you can comma to separate multiple values to lookup.		String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
transformerFactory (advanced)	To use a custom XSLT transformer factory		TransformerFactory
transformerFactoryClass (advanced)	To use a custom XSLT transformer factory, specified as a FQN class name		String
uriResolver (advanced)	To use a custom javax.xml.transform.URIResolver		URIResolver

349.3. USING XSLT ENDPOINTS

The following format is an expample of using an XSLT template to formulate a response for a message for InOut message exchanges (where there is a **JMSReplyTo** header)

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl");
```

If you want to use InOnly and consume the message and send it to another destination you could use the following route:

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl").
  to("activemq:Another.Queue");
```

349.4. GETTING USEABLE PARAMETERS INTO THE XSLT

By default, all headers are added as parameters which are then available in the XSLT.
To make the parameters useable, you will need to declare them.

```
<setHeader headerName="myParam"><constant>42</constant></setHeader>
<to uri="xslt:MyTransform.xsl"/>
```

The parameter also needs to be declared in the top level of the XSLT for it to be available:

```
<xsl: ..... >

  <xsl:param name="myParam"/>

   <xsl:template ...>
```

349.5. SPRING XML VERSIONS

To use the above examples in Spring XML you would use something like the following code:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="xslt:org/apache/camel/spring/processor/example.xsl"/>
    <to uri="activemq:Another.Queue"/>
  </route>
</camelContext>
```

To see an example, look at the test case along with its Spring XML.

349.6. USING XSL:INCLUDE

Camel 2.2 or older
If you use **xsl:include** in your XSL files in **Camel 2.2 or older**, the default **javax.xml.transform.URIResolver** is used. Files will be resolved relative to the JVM starting folder.

For example the following include statement will look up the **staff_template.xsl** file starting from the folder where the application was started.

```
<xsl:include href="staff_template.xsl"/>
```

Camel 2.3 or newer
For Camel 2.3 or newer, Camel provides its own implementation of **URIResolver**. This allows Camel to load included files from the classpath.

For example the include file in the following code will be located relative to the starting endpoint.

```
<xsl:include href="staff_template.xsl"/>
```

This means that Camel will locate the file in the **classpath** as **org/apache/camel/component/xslt/staff_template.xsl**

You can use **classpath:** or **file:** to instruct Camel to look either in the classpath or file system. If you omit the prefix then Camel uses the prefix from the endpoint configuration. If no prefix is specified in the endpoint configuration, the default is **classpath:**.

You can also refer backwards in the include paths. In the following example, the xsl file will be resolved under **org/apache/camel/component**.

```
    <xsl:include href="../staff_other_template.xsl"/>
```

349.7. USING XSL:INCLUDE AND DEFAULT PREFIX

In **Camel 2.10.3 and older**, **classpath:** is used as the default prefix.
If you configure the starting resource to load using **file:** then all subsequent incudes will have to be prefixed with **file:**.

From **Camel 2.10.4**, Camel will use the prefix from the endpoint configuration as the default prefix.

You can explicitly specify **file:** or **classpath:** loading. The two loading types can be mixed in a XSLT script, if necessary.

349.8. USING SAXON EXTENSION FUNCTIONS

Since Saxon 9.2, writing extension functions has been supplemented by a new mechanism, referred to as integrated extension functions you can now easily use camel as shown in the below example:

```
SimpleRegistry registry = new SimpleRegistry();
registry.put("function1", new MyExtensionFunction1());
registry.put("function2", new MyExtensionFunction2());

CamelContext context = new DefaultCamelContext(registry);
context.addRoutes(new RouteBuilder() {
   @Override
   public void configure() throws Exception {
      from("direct:start")
         .to("xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?
saxonExtensionFunctions=#function1,#function2");
   }
});
```

With Spring XML:

```
<bean id="function1" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction1"/>
<bean id="function2" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction2"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:extensions"/>
    <to uri="xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?
saxonExtensionFunctions=#function1,#function2"/>
  </route>
</camelContext>
```

349.9. DYNAMIC STYLESHEETS

To provide a dynamic stylesheet at runtime you can define a dynamic URI. See How to use a dynamic URI in to() for more information.

Available as of Camel 2.9 (removed in 2.11.4, 2.12.3 and 2.13.0)
Camel provides the **CamelXsltResourceUri** header which you can use to define an alternative stylesheet to that configured on the endpoint URI. This allows you to provide a dynamic stylesheet at runtime.

349.10. ACCESSING WARNINGS, ERRORS AND FATALERRORS FROM XSLT ERRORLISTENER

Available as of Camel 2.14

From **Camel 2.14**, any warning/error or fatalError is stored on the current Exchange as a property with the keys **Exchange.XSLT_ERROR**, **Exchange.XSLT_FATAL_ERROR**, or **Exchange.XSLT_WARNING** which allows end users to get hold of any errors happening during transformation.

For example in the stylesheet below, we want to terminate if a staff has an empty dob field. And to include a custom error message using xsl:message.

```
<xsl:template match="/">
  <html>
    <body>
      <xsl:for-each select="staff/programmer">
        <p>Name: <xsl:value-of select="name"/><br />
          <xsl:if test="dob=''">
            <xsl:message terminate="yes">Error: DOB is an empty string!</xsl:message>
          </xsl:if>
        </p>
      </xsl:for-each>
    </body>
  </html>
</xsl:template>
```

The exception is stored on the Exchange as a warning with the key **Exchange.XSLT_WARNING.**

349.11. NOTES ON USING XSLT AND JAVA VERSIONS

Here are some observations from Sameer, a Camel user, which he kindly shared with us:

In case anybody faces issues with the XSLT endpoint please review these points.

I was trying to use an xslt endpoint for a simple transformation from one xml to another using a simple xsl. The output xml kept appearing (after the xslt processor in the route) with outermost xml tag with no content within.

No explanations show up in the DEBUG logs. On the TRACE logs however I did find some error/warning indicating that the XMLConverter bean could no be initialized.

After a few hours of cranking my mind, I had to do the following to get it to work (thanks to some posts on the users forum that gave some clue):

1. Use the transformerFactory option in the route **("xslt:my-transformer.xsl?transformerFactory=tFactory")** with the **tFactory** bean having bean defined in the spring context for **class="org.apache.xalan.xsltc.trax.TransformerFactoryImpl"**.

2. Added the Xalan jar into my maven pom.

My guess is that the default xml parsing mechanism supplied within the JDK (I am using 1.6.0_03) does not work right in this context and does not throw up any error either. When I switched to Xalan this way it works. This is not a Camel issue, but might need a mention on the xslt component page.

Another note, jdk 1.6.0_03 ships with JAXB 2.0 while Camel needs 2.1. One workaround is to add the 2.1 jar to the **jre/lib/endorsed** directory for the jvm or as specified by the container.

Hope this post saves newbie Camel riders some time.

349.12. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 350. XSTREAM DATAFORMAT

Available as of Camel version 1.3

XStream is a Data Format which uses the XStream library to marshal and unmarshal Java objects to and from XML.

To use XStream in your camel routes you need to add the a dependency on **camel-xstream** which implements this data format.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xstream</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

350.1. OPTIONS

The XStream dataformat supports 10 options which are listed below.

Name	Default	Java Type	Description
permissions		**String**	Adds permissions that controls which Java packages and classes XStream is allowed to use during unmarshal from xml/json to Java beans. A permission must be configured either here or globally using a JVM system property. The permission can be specified in a syntax where a plus sign is allow, and minus sign is deny. Wildcards is supported by using . as prefix. For example to allow com.foo and all subpackages then specfy com.foo.. Multiple permissions can be configured separated by comma, such as com.foo.,-com.foo.bar.MySecretBean. The following default permission is always included: -,java.lang.,java.util. unless its overridden by specifying a JVM system property with they key org.apache.camel.xstream.permissions.
encoding		**String**	Sets the encoding to use
driver		**String**	To use a custom XStream driver. The instance must be of type com.thoughtworks.xstream.io.HierarchicalStreamDriver
driverRef		**String**	To refer to a custom XStream driver to lookup in the registry. The instance must be of type com.thoughtworks.xstream.io.HierarchicalStreamDriver

Name	Default	Java Type	Description
mode		**String**	Mode for dealing with duplicate references The possible values are: NO_REFERENCES ID_REFERENCES XPATH_RELATIVE_REFERENCES XPATH_ABSOLUTE_REFERENCES SINGLE_NODE_XPATH_RELATIVE_REFERENCES SINGLE_NODE_XPATH_ABSOLUTE_REFERENCES
converters		**List**	List of class names for using custom XStream converters. The classes must be of type com.thoughtworks.xstream.converters.Converter
aliases		**Map**	Alias a Class to a shorter name to be used in XML elements.
omitFields		**Map**	Prevents a field from being serialized. To omit a field you must always provide the declaring type and not necessarily the type that is converted.
implicitCollections		**Map**	Adds a default implicit collection which is used for any unmapped XML tag.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

350.2. USING THE JAVA DSL

```
// lets turn Object messages into XML then send to MQSeries
from("activemq:My.Queue").
  marshal().xstream().
  to("mqseries:Another.Queue");
```

If you would like to configure the **XStream** instance used by the Camel for the message transformation, you can simply pass a reference to that instance on the DSL level.

```
XStream xStream = new XStream();
xStream.aliasField("money", PurchaseOrder.class, "cash");
// new Added setModel option since Camel 2.14
xStream.setModel("NO_REFERENCES");
...

from("direct:marshal").
  marshal(new XStreamDataFormat(xStream)).
  to("mock:marshaled");
```

350.3. XMLINPUTFACTORY AND XMLOUTPUTFACTORY

The XStream library uses the **javax.xml.stream.XMLInputFactory** and **javax.xml.stream.XMLOutputFactory**, you can control which implementation of this factory should be used.

The Factory is discovered using this algorithm: 1. Use the **javax.xml.stream.XMLInputFactory** , **javax.xml.stream.XMLOutputFactory** system property. 2. Use the **lib/xml.stream.properties** file in the **JRE_HOME** directory. 3. Use the Services API, if available, to determine the classname by looking in the **META-INF/services/javax.xml.stream.XMLInputFactory**, **META-INF/services/javax.xml.stream.XMLOutputFactory** files in jars available to the JRE. 4. Use the platform default XMLInputFactory,XMLOutputFactory instance.

350.4. HOW TO SET THE XML ENCODING IN XSTREAM DATAFORMAT?

From Camel 2.2.0, you can set the encoding of XML in Xstream DataFormat by setting the Exchange's property with the key **Exchange.CHARSET_NAME**, or setting the encoding property on Xstream from DSL or Spring config.

```
from("activemq:My.Queue").
  marshal().xstream("UTF-8").
  to("mqseries:Another.Queue");
```

350.5. SETTING THE TYPE PERMISSIONS OF XSTREAM DATAFORMAT

In Camel, one can always use its own processing step in the route to filter and block certain XML documents to be routed to the XStream's unmarhall step. From Camel 2.16.1, 2.15.5, you can set XStream's type permissions to automatically allow or deny the instantiation of certain types.

The default type permissions setting used by Camel denies all types except for those from java.lang and java.util packages. This setting can be changed by setting System property org.apache.camel.xstream.permissions. Its value is a string of comma-separated permission terms, each representing a type being allowed or denied, depending on whether the term is prefixed with '' (note '' may be omitted) or with '-', respectively.

Each term may contain a wildcard character '**. For example, value "-**java.lang.**,java.util.**" indicates denying all types except for java.lang.* and java.util.* classes. Setting this value to an empty string "" reverts to the default XStream's type permissions handling which denies certain blacklisted classes and allow others.

The type permissions setting can be extended at an individual XStream DataFormat instance by setting its type permissions property.

```
<dataFormats>
   <xstream id="xstream-default"
            permissions="org.apache.camel.samples.xstream.*"/>
   ...
```

CHAPTER 351. YAML SNAKEYAML DATAFORMAT

Available as of Camel version 2.17

YAML is a Data Format to marshal and unmarshal Java objects to and from YAML.

For YAML to object marshalling, Camel provides integration with three popular YAML libraries:

- The SnakeYAML library

Every library requires adding the special camel component (see "Dependency..." paragraphs further down). By default Camel uses the SnakeYAML library.

351.1. YAML OPTIONS

The YAML SnakeYAML dataformat supports 11 options which are listed below.

Name	Default	Java Type	Description
library	**SnakeYAML**	**YAMLLibrary**	Which yaml library to use. By default it is SnakeYAML
unmarshalTypeName		**String**	Class name of the java type to use when unarmshalling
constructor		**String**	BaseConstructor to construct incoming documents.
representer		**String**	Representer to emit outgoing objects.
dumperOptions		**String**	DumperOptions to configure outgoing objects.
resolver		**String**	Resolver to detect implicit type
useApplicationContextClassLoader	**true**	**Boolean**	Use ApplicationContextClassLoader as custom ClassLoader
prettyFlow	**false**	**Boolean**	Force the emitter to produce a pretty YAML document when using the flow style.
allowAnyType	**false**	**Boolean**	Allow any class to be un-marshaled
typeFilter		**List**	Set the types SnakeYAML is allowed to un-marshall

Name	Default	Java Type	Description
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

WARNING

SnakeYAML can load any class from YAML definition which may lead to security breach so by default, SnakeYAML DataForma restrict the object it can load to standard Java objects like List or Long. If you want to load custom POJOs you need to add theirs type to SnakeYAML DataFormat type filter list. If your source is trusted, you can set the property allowAnyType to true so SnakeYAML DataForma won't perform any filter on the types.

351.2. USING YAML DATA FORMAT WITH THE SNAKEYAML LIBRARY

- Turn Object messages into yaml then send to MQSeries

```
from("activemq:My.Queue")
  .marshal().yaml()
  .to("mqseries:Another.Queue");
```

```
from("activemq:My.Queue")
  .marshal().yaml(YAMLLibrary.SnakeYAML)
  .to("mqseries:Another.Queue");
```

- Restrict classes to be loaded from YAML

```
// Creat a SnakeYAMLDataFormat instance
SnakeYAMLDataFormat yaml = new SnakeYAMLDataFormat();

// Restrict classes to be loaded from YAML
yaml.addTypeFilters(TypeFilters.types(MyPojo.class, MyOtherPojo.class));

from("activemq:My.Queue")
  .unmarshal(yaml)
  .to("mqseries:Another.Queue");
```

351.3. USING YAML IN SPRING DSL

When using Data Format in Spring DSL you need to declare the data formats first. This is done in the **DataFormats** XML tag.

```
<dataFormats>
  <!--
    here we define a YAML data format with the id snake and that it should use
    the TestPojo as the class type when doing unmarshal. The unmarshalTypeName
    is optional
  -->
  <yaml
    id="snake"
    library="SnakeYAML"
    unmarshalTypeName="org.apache.camel.component.yaml.model.TestPojo"/>

  <!--
    here we define a YAML data format with the id snake-safe which restricts the
    classes to be loaded from YAML to TestPojo and those belonging to package
    com.mycompany
  -->
  <yaml id="snake-safe">
    <typeFilter value="org.apache.camel.component.yaml.model.TestPojo"/>
    <typeFilter value="com.mycompany\..*" type="regexp"/>
  </yaml>
</dataFormats>
```

And then you can refer to those ids in the route:

```
<route>
  <from uri="direct:unmarshal"/>
  <unmarshal>
    <custom ref="snake"/>
  </unmarshal>
  <to uri="mock:unmarshal"/>
</route>
<route>
  <from uri="direct:unmarshal-safe"/>
  <unmarshal>
    <custom ref="snake-safe"/>
  </unmarshal>
  <to uri="mock:unmarshal-safe"/>
</route>
```

351.4. DEPENDENCIES FOR SNAKEYAML

To use YAML in your camel routes you need to add the a dependency on **camel-snakeyaml** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-snakeyaml</artifactId>
  <version>${camel-version}</version>
</dependency>
```

CHAPTER 352. YAMMER COMPONENT

Available as of Camel version 2.12

The Yammer component allows you to interact with the Yammer enterprise social network. Consuming messages, users, and user relationships is supported as well as creating new messages.

Yammer uses OAuth 2 for all client application authentication. In order to use camel-yammer with your account, you'll need to create a new application within Yammer and grant the application access to your account. Finally, generate your access token. More details are at https://developer.yammer.com/authentication/.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-yammer</artifactId>
    <version>${camel-version}</version>
</dependency>
```

352.1. URI FORMAT

```
yammer:[function]?[options]
```

352.2. COMPONENT OPTIONS

The Yammer component can be configured with the Yammer account settings which are mandatory to configure before using.

The Yammer component supports 5 options which are listed below.

Name	Description	Default	Type
consumerKey (security)	The consumer key		String
consumerSecret (security)	The consumer secret		String
accessToken (security)	The access token		String
config (advanced)	To use a shared yammer configuration		YammerConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

You can also configure these options directly in the endpoint.

352.3. ENDPOINT OPTIONS

The Yammer endpoint is configured using URI syntax:

```
yammer:function
```

with the following path and query parameters:

352.3.1. Path Parameters (1 parameters):

Name	Description	Default	Type
function	**Required** The function to use		YammerFunctionType

352.3.2. Query Parameters (28 parameters):

Name	Description	Default	Type
useJson (common)	Set to true if you want to use raw JSON rather than converting to POJOs.	false	boolean
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
delay (consumer)	Delay between polling in millis	5000	long
limit (consumer)	Return only the specified number of messages. Works for threaded=true and threaded=extended.	-1	int
newerThan (consumer)	Returns messages newer than the message ID specified as a numeric string. This should be used when polling for new messages. If you're looking at messages, and the most recent message returned is 3516, you can make a request with the parameter newerThan=3516 to ensure that you do not get duplicate copies of messages already on your page.	-1	int

Name	Description	Default	Type
olderThan (consumer)	Returns messages older than the message ID specified as a numeric string. This is useful for paginating messages. For example, if you're currently viewing 20 messages and the oldest is number 2912, you could append olderThan=2912 to your request to get the 20 messages prior to those you're seeing.	-1	int
sendEmptyMessageWhenIdle (consumer)	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.	false	boolean
threaded (consumer)	threaded=true will only return the first message in each thread. This parameter is intended for apps which display message threads collapsed. threaded=extended will return the thread starter messages in order of most recently active as well as the two most recent messages, as they are viewed in the default view on the Yammer web interface.		String
userId (consumer)	The user id		String
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
pollStrategy (consumer)	A pluggable org.apache.camel.PollingConsumerPollingStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation before an Exchange have been created and being routed in Camel.		PollingConsumerPoll Strategy
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
backoffErrorThreshold (scheduler)	The number of subsequent error polls (failed due some error) that should happen before the backoffMultipler should kick-in.		int
backoffIdleThreshold (scheduler)	The number of subsequent idle polls that should happen before the backoffMultipler should kick-in.		int

Name	Description	Default	Type
backoffMultiplier (scheduler)	To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then backoffIdleThreshold and/or backoffErrorThreshold must also be configured.		int
greedy (scheduler)	If greedy is enabled, then the ScheduledPollConsumer will run immediately again, if the previous run polled 1 or more messages.	false	boolean
initialDelay (scheduler)	Milliseconds before the first poll starts. You can also specify time values using units, such as 60s (60 seconds), 5m30s (5 minutes and 30 seconds), and 1h (1 hour).	1000	long
runLoggingLevel (scheduler)	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.	TRACE	LoggingLevel
scheduledExecutorService (scheduler)	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool.		ScheduledExecutor Service
scheduler (scheduler)	To use a cron scheduler from either camel-spring or camel-quartz2 component	none	ScheduledPollConsumer Scheduler
schedulerProperties (scheduler)	To configure additional properties when using a custom scheduler or any of the Quartz2, Spring based scheduler.		Map
startScheduler (scheduler)	Whether the scheduler should be auto started.	true	boolean
timeUnit (scheduler)	Time unit for initialDelay and delay options.	MILLISECONDS	TimeUnit
useFixedDelay (scheduler)	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.	true	boolean
accessToken (security)	**Required** The access token		String
consumerKey (security)	**Required** The consumer key		String

Name	Description	Default	Type
consumerSecret (security)	**Required** The consumer secret		String

352.4. CONSUMING MESSAGES

The Yammer component provides several endpoints for consuming messages:

URI	Description
yammer:messages?[options]	All public messages in the user's (whose access token is being used to make the API call) Yammer network. Corresponds to "All" conversations in the Yammer web interface.
yammer:my_feed?[options]	The user's feed, based on the selection they have made between "Following" and "Top" conversations.
yammer:algo?[options]	The algorithmic feed for the user that corresponds to "Top" conversations, which is what the vast majority of users will see in the Yammer web interface.
yammer:following?[options]	The "Following" feed which is conversations involving people, groups and topics that the user is following.
yammer:sent?[options]	All messages sent by the user.
yammer:private?[options]	Private messages received by the user.
yammer:received?[options]	**Camel 2.12.1:** All messages received by the user

352.4.1. Message format

All messages by default are converted to a POJO model provided in the **org.apache.camel.component.yammer.model** package. The original message coming from yammer is in JSON. For all message consuming and producing endpoints, a **Messages** object is returned. Take for example a route like:

```
from("yammer:messages?
consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAccessToken"
)
    .to("mock:result");
```

and lets say the yammer server returns:


```
{
  "messages":[
    {
      "replied_to_id":null,
      "network_id":7654,
      "url":"https://www.yammer.com/api/v1/messages/305298242",
      "thread_id":305298242,
      "id":305298242,
      "message_type":"update",
      "chat_client_sequence":null,
      "body":{
        "parsed":"Testing yammer API...",
        "plain":"Testing yammer API...",
        "rich":"Testing yammer API..."
      },
      "client_url":"https://www.yammer.com/",
      "content_excerpt":"Testing yammer API...",
      "created_at":"2013/06/25 18:14:45 +0000",
      "client_type":"Web",
      "privacy":"public",
      "sender_type":"user",
      "liked_by":{
        "count":1,
        "names":[
          {
            "permalink":"janstey",
            "full_name":"Jonathan Anstey",
            "user_id":1499642294
          }

        ]

      },
      "sender_id":1499642294,
      "language":null,
      "system_message":false,
      "attachments":[

      ],
      "direct_message":false,
      "web_url":"https://www.yammer.com/redhat.com/messages/305298242"
    },
    {
      "replied_to_id":null,
      "network_id":7654,
      "url":"https://www.yammer.com/api/v1/messages/294326302",
      "thread_id":294326302,
      "id":294326302,
      "message_type":"system",
      "chat_client_sequence":null,
      "body":{
        "parsed":"(Principal Software Engineer) has [[tag:14658]] the redhat.com network. Take a moment to welcome Jonathan.",
        "plain":"(Principal Software Engineer) has #joined the redhat.com network. Take a moment
```

```
to welcome Jonathan.",
                "rich":"(Principal Software Engineer) has #joined the redhat.com network. Take a moment
to welcome Jonathan."
            },
            "client_url":"https://www.yammer.com/",
            "content_excerpt":"(Principal Software Engineer) has #joined the redhat.com network. Take a
moment to welcome Jonathan.",
            "created_at":"2013/05/10 19:08:29 +0000",
            "client_type":"Web",
            "sender_type":"user",
            "privacy":"public",
            "liked_by":{
               "count":0,
               "names":[

               ]

            }
         }
         ]

   }
```

Camel will marshal that into a **Messages** object containing 2 **Message** objects. As shown below there is a rich object model that makes it easy to get any information you need:

```
Exchange exchange = mock.getExchanges().get(0);
Messages messages = exchange.getIn().getBody(Messages.class);

assertEquals(2, messages.getMessages().size());
assertEquals("Testing yammer API...", messages.getMessages().get(0).getBody().getPlain());
assertEquals("(Principal Software Engineer) has #joined the redhat.com network. Take a moment to
welcome Jonathan.", messages.getMessages().get(1).getBody().getPlain());
```

That said, marshaling this data into POJOs is not free so if you need you can switch back to using pure JSON by adding the **useJson=false** option to your URI.

352.5. CREATING MESSAGES

To create a new message in the account of the current user, you can use the following URI:

```
yammer:messages?[options]
```

The current Camel message body is what will be used to set the text of the Yammer message. The response body will include the new message formatted the same way as when you consume messages (i.e. as a **Messages** object by default).

Take this route for instance:

```
from("direct:start")
    .to("yammer:messages?
consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAccessToken"
)
    .to("mock:result");
```

By sending to the **direct:start** endpoint a **"Hi from Camel!"** message body:

```
template.sendBody("direct:start", "Hi from Camel!");
```

a new message will be created in the current user's account on the server and also this new message will be returned to Camel and converted into a **Messages** object. Like when consuming messages you can interrogate the **Messages** object:

```
Exchange exchange = mock.getExchanges().get(0);
Messages messages = exchange.getIn().getBody(Messages.class);

assertEquals(1, messages.getMessages().size());
assertEquals("Hi from Camel!", messages.getMessages().get(0).getBody().getPlain());
```

352.6. RETRIEVING USER RELATIONSHIPS

The Yammer component can retrieve user relationships:

```
yammer:relationships?[options]
```

352.7. RETRIEVING USERS

The Yammer component provides several endpoints for retrieving users:

URI	Description
`yammer:users?[options]`	Retrieve users in the current user's Yammer network.
`yammer:current?[options]`	View data about the current user.

352.8. USING AN ENRICHER

It is helpful sometimes (or maybe always in the case of users or relationship consumers) to use an enricher pattern rather than a route initiated with one of the polling consumers in camel-yammer. This is because the consumers will fire repeatedly, however often you set the delay for. If you just want to look up a user's data, or grab a message at a point in time, it is better to call that consumer once and then get one with your route.

Lets say you have a route that at some point needs to go out and fetch user data for the current user. Rather than polling for this user over and over again, use the **pollEnrich** DSL method:

```
from("direct:start")
    .pollEnrich("yammer:current?
consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAccessToken"
)
    .to("mock:result");
```

This will go out and fetch the current user's **User** object and set it as the Camel message body.

352.9. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 353. ZENDESK COMPONENT

Available as of Camel version 2.19

The Zendesk component provides access to all of the zendesk.com APIs accessible using zendesk-java-client. It allows producing messages to manage Zendesk ticket, user, organization, etc.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-zendesk</artifactId>
  <version>${camel-version}</version>
</dependency>
```

353.1. ZENDESK OPTIONS

The Zendesk component supports 3 options which are listed below.

Name	Description	Default	Type
configuration (common)	To use the shared configuration		ZendeskConfiguration
zendesk (advanced)	To use a shared Zendesk instance.		Zendesk
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The Zendesk endpoint is configured using URI syntax:

```
zendesk:methodName
```

with the following path and query parameters:

353.1.1. Path Parameters (1 parameters):

Name	Description	Default	Type
methodName	**Required** What operation to use		String

353.1.2. Query Parameters (10 parameters):

Name	Description	Default	Type
inBody (common)	Sets the name of a parameter to be passed in the exchange In Body		String
serverUrl (common)	The server URL to connect.		String
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean
oauthToken (security)	The OAuth token.		String
password (security)	The password.		String
token (security)	The security token.		String
username (security)	The user name.		String

353.2. URI FORMAT

```
zendesk://endpoint?[options]
```

353.3. PRODUCER ENDPOINTS:

Producer endpoints can use endpoint names and associated options described next.

353.4. CONSUMER ENDPOINTS:

Any of the producer endpoints can be used as a consumer endpoint. Consumer endpoints can use Scheduled Poll Consumer Options with a **consumer.** prefix to schedule endpoint invocation. Consumer endpoints that return an array or collection will generate one exchange per element, and their routes will be executed once for each exchange.

353.5. MESSAGE HEADER

Any of the options can be provided in a message header for producer endpoints with **CamelZendesk.** prefix. In principal, parameter names are same as the arugument name of each API methods on the original **org.zendesk.client.v2.Zendesk** class. However some of them are renamed to the other name to avoid confliction in the camel API component framework. To see actual parameter name, please check **org.apache.camel.component.zendesk.internal.ZendeskApiMethod**.

353.6. MESSAGE BODY

All result message bodies utilize objects provided by the Zendesk Java Client. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint parameter.

CHAPTER 354. ZIP DEFLATE COMPRESSION DATAFORMAT

Available as of Camel version 2.12

The Zip Data Format is a message compression and de-compression format. Messages marshalled using Zip compression can be unmarshalled using Zip decompression just prior to being consumed at the endpoint. The compression capability is quite useful when you deal with large XML and Text based payloads. It facilitates more optimal use of network bandwidth while incurring a small cost in order to compress and decompress payloads at the endpoint.

INFO:*About using with Files* The Zip data format, does not (yet) have special support for files. Which means that when using big files, the entire file content is loaded into memory. This is subject to change in the future, to allow a streaming based solution to have a low memory footprint.

354.1. OPTIONS

The Zip Deflate Compression dataformat supports 2 options which are listed below.

Name	Default	Java Type	Description
compressionLevel	**-1**	**Integer**	To specify a specific compression between 0-9. -1 is default compression, 0 is no compression, and 9 is best compression.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

354.2. MARSHAL

In this example we marshal a regular text/XML payload to a compressed payload employing zip compression **Deflater.BEST_COMPRESSION** and send it an ActiveMQ queue called MY_QUEUE.

```
from("direct:start").marshal().zip(Deflater.BEST_COMPRESSION).to("activemq:queue:MY_QUEUE")
;
```

Alternatively if you would like to use the default setting you could send it as

```
from("direct:start").marshal().zip().to("activemq:queue:MY_QUEUE");
```

354.3. UNMARSHAL

In this example we unmarshal a zipped payload from an ActiveMQ queue called MY_QUEUE to its original format, and forward it for processing to the UnZippedMessageProcessor. Note that the compression Level employed during the marshalling should be identical to the one employed during unmarshalling to avoid errors.

```
from("activemq:queue:MY_QUEUE").unmarshal().zip().process(new UnZippedMessageProcessor());
```

354.4. DEPENDENCIES

This data format is provided in **camel-core** so no additional dependencies are needed.

CHAPTER 355. ZIP FILE DATAFORMAT

Available as of Camel version 2.11

The Zip File Data Format is a message compression and de-compression format. Messages can be marshalled (compressed) to Zip files containing a single entry, and Zip files containing a single entry can be unmarshalled (decompressed) to the original file contents. This data format supports ZIP64, as long as Java 7 or later is being used].

355.1. ZIPFILE OPTIONS

The Zip File dataformat supports 4 options which are listed below.

Name	Default	Java Type	Description
usingIterator	**false**	**Boolean**	If the zip file has more then one entry, the setting this option to true, allows to work with the splitter EIP, to split the data using an iterator in a streaming mode.
allowEmptyDirectory	**false**	**Boolean**	If the zip file has more then one entry, setting this option to true, allows to get the iterator even if the directory is empty
preservePathElements	**false**	**Boolean**	If the file name contains path elements, setting this option to true, allows the path to be maintained in the zip file.
contentTypeHeader	**false**	**Boolean**	Whether the data format should set the Content-Type header with the type from the data format if the data format is capable of doing so. For example application/xml for data formats marshalling to XML, or application/json for data formats marshalling to JSon etc.

355.2. MARSHAL

In this example we marshal a regular text/XML payload to a compressed payload using Zip file compression, and send it to an ActiveMQ queue called MY_QUEUE.

```
from("direct:start")
    .marshal().zipFile()
    .to("activemq:queue:MY_QUEUE");
```

The name of the Zip entry inside the created Zip file is based on the incoming **CamelFileName** message header, which is the standard message header used by the file component. Additionally, the outgoing **CamelFileName** message header is automatically set to the value of the incoming **CamelFileName** message header, with the ".zip" suffix. So for example, if the following route finds a file named "test.txt" in the input directory, the output will be a Zip file named "test.txt.zip" containing a single Zip entry named "test.txt":

```
from("file:input/directory?antInclude=*/.txt")
    .marshal().zipFile()
    .to("file:output/directory");
```

If there is no incoming **CamelFileName** message header (for example, if the file component is not the consumer), then the message ID is used by default, and since the message ID is normally a unique generated ID, you will end up with filenames like **ID-MACHINENAME-2443-1211718892437-1-0.zip**. If you want to override this behavior, then you can set the value of the **CamelFileName** header explicitly in your route:

```
from("direct:start")
    .setHeader(Exchange.FILE_NAME, constant("report.txt"))
    .marshal().zipFile()
    .to("file:output/directory");
```

This route would result in a Zip file named "report.txt.zip" in the output directory, containing a single Zip entry named "report.txt".

355.3. UNMARSHAL

In this example we unmarshal a Zip file payload from an ActiveMQ queue called MY_QUEUE to its original format, and forward it for processing to the **UnZippedMessageProcessor**.

```
from("activemq:queue:MY_QUEUE")
    .unmarshal().zipFile()
    .process(new UnZippedMessageProcessor());
```

If the zip file has more then one entry, the usingIterator option of ZipFileDataFormat to be true, and you can use splitter to do the further work.

```
ZipFileDataFormat zipFile = new ZipFileDataFormat();
zipFile.setUsingIterator(true);

from("file:src/test/resources/org/apache/camel/dataformat/zipfile/?consumer.delay=1000&noop=true")
    .unmarshal(zipFile)
    .split(body(Iterator.class)).streaming()
        .process(new UnZippedMessageProcessor())
    .end();
```

Or you can use the ZipSplitter as an expression for splitter directly like this

```
from("file:src/test/resources/org/apache/camel/dataformat/zipfile?consumer.delay=1000&noop=true")
    .split(new ZipSplitter()).streaming()
        .process(new UnZippedMessageProcessor())
    .end();
```

355.4. AGGREGATE

NOTE

Please note that this aggregation strategy requires eager completion check to work properly.

In this example we aggregate all text files found in the input directory into a single Zip file that is stored in the output directory.

```
from("file:input/directory?antInclude=*/.txt")
    .aggregate(constant(true), new ZipAggregationStrategy())
        .completionFromBatchConsumer().eagerCheckCompletion()
        .to("file:output/directory");
```

The outgoing **CamelFileName** message header is created using java.io.File.createTempFile, with the ".zip" suffix. If you want to override this behavior, then you can set the value of the **CamelFileName** header explicitly in your route:

```
from("file:input/directory?antInclude=*/.txt")
    .aggregate(constant(true), new ZipAggregationStrategy())
        .completionFromBatchConsumer().eagerCheckCompletion()
        .setHeader(Exchange.FILE_NAME, constant("reports.zip"))
        .to("file:output/directory");
```

355.5. DEPENDENCIES

To use Zip files in your camel routes you need to add a dependency on **camel-zipfile** which implements this data format.

If you use Maven you can just add the following to your **pom.xml**, substituting the version number for the latest & greatest release (see the download page for the latest versions).

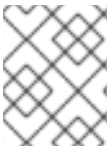
```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-zipfile</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

CHAPTER 356. ZIPKIN COMPONENT

Available as of Camel 2.18

The camel-zipkin component is used for tracing and timing incoming and outgoing Camel messages using zipkin.

Events (span) are captured for incoming and outgoing messages being sent to/from Camel.



NOTE

camel-zipkin is planned to be refactored in Camel 2.22.0 to not use zipkin-scribe but use the default http transport. This work may cause backwards incompatibility.

This means you need to configure which Camel endpoints map to zipkin service names.

The mapping can be configured using:

- route id - A Camel route id
- endpoint url - A Camel endpoint url

For both kinds you can match using wildcards and regular expressions, using the rules from Intercept.

To match all Camel messages you can use * in the pattern and configure that to the same service name.

If no mapping has been configured, Camel will fallback and use endpoint uri's as service names. However, it's recommended to configure service mappings so you can use human-readable names instead of Camel endpoint uris in the names.

Camel will auto-configure a span reporter one hasn't been explicitly configured, and if the hostname and port to a zipkin collector has been configured as environment variables

- ZIPKIN_COLLECTOR_HTTP_SERVICE_HOST - The http hostname
- ZIPKIN_COLLECTOR_HTTP_SERVICE_PORT - The port number

or

- ZIPKIN_COLLECTOR_THRIFT_SERVICE_HOST - The Scribe (Thrift RPC) hostname
- ZIPKIN_COLLECTOR_THRIFT_SERVICE_PORT - The port number

This makes it easy to use camel-zipkin in container platforms where the platform can run your application in a linux container where service configurations are provided as environment variables.

356.1. OPTIONS

Option	Default	Description
rate	1.0f	Configures a rate that decides how many events should be traced by zipkin. The rate is expressed as a percentage (1.0f = 100%, 0.5f is 50%, 0.1f is 10%).

Option	Default	Description
spanReporter		**Mandatory:** The reporter to use for sending zipkin span events to the zipkin server.
serviceName		To use a global service name that matches all Camel events
clientServiceMappings		Sets the **client** service mappings that matches Camel events to the given zipkin service name. The content is a Map<String, String> where the key is a pattern and the value is the service name. The pattern uses the rules from Intercept.
serverServiceMappings		Sets the **server** service mappings that matches Camel events to the given zipkin service name. The content is a Map<String, String> where the key is a pattern and the value is the service name. The pattern uses the rules from Intercept.
excludePatterns		Sets exclude pattern(s) that will disable tracing with zipkin for Camel messages that matches the pattern. The content is a Set<String> where the key is a pattern. The pattern uses the rules from Intercept.
includeMessageBody	false	Whether to include the Camel message body in the zipkin traces. This is not recommended for production usage, or when having big payloads. You can limit the size by configuring the max debug log size.
includeMessageBodyStreams	false	Whether to include message bodies that are stream based in the zipkin traces. This requires enabling streamcaching on the routes or globally on the CamelContext. This is not recommended for production usage, or when having big payloads. You can limit the size by configuring the max debug log size.

356.2. EXAMPLE

To enable camel-zipkin you need to configure first

```
ZipkinTracer zipkin = new ZipkinTracer();
// Configure a reporter, which controls how often spans are sent
//   (the dependency is io.zipkin.reporter2:zipkin-sender-okhttp3)
sender = OkHttpSender.create("http://127.0.0.1:9411/api/v2/spans");
zipkin.setSpanReporter(AsyncReporter.create(sender));
// and then add zipkin to the CamelContext
zipkin.init(camelContext);
```

The configuration above will trace all incoming and outgoing messages in Camel routes.

To use ZipkinTracer in XML, all you need to do is to define scribe and zipkin tracer beans. Camel will automatically discover and use them.

```
<!-- configure how to reporter spans to a Zipkin collector
     (the dependency is io.zipkin.reporter2:zipkin-reporter-spring-beans) -->
```

```
<bean id="http" class="zipkin2.reporter.beans.AsyncReporterFactoryBean">
  <property name="sender">
    <bean id="sender" class="zipkin2.reporter.beans.OkHttpSenderFactoryBean">
      <property name="endpoint" value="http://localhost:9411/api/v2/spans"/>
    </bean>
  </property>
  <!-- wait up to half a second for any in-flight spans on close -->
  <property name="closeTimeout" value="500"/>
</bean>

<!-- setup zipkin tracer -->
<bean id="zipkinTracer" class="org.apache.camel.zipkin.ZipkinTracer">
  <property name="serviceName" value="dude"/>
  <property name="spanReporter" ref="http"/>
</bean>
```

356.2.1. ServiceName

However, if you want to map Camel endpoints to human friendly logical names, you can add mappings

- ServiceName *

You can configure a global service name that all events will fallback and use, such as:

```
zipkin.setServiceName("invoices");
```

This will use the same service name for all incoming and outgoing zipkin traces. If your application uses different services, you should map them to more finely grained client / server service mappings

356.2.2. Client and Server Service Mappings

- ClientServiceMappings
- ServerServiceMappings

If your application hosts a service that others can call, you can map the Camel route endpoint to a server service mapping. For example, suppose your Camel application has the following route:

```
from("activemq:queue:inbox")
  .to("http:someserver/somepath");
```

And you want to make that as a server service, you can add the following mapping:

```
zipkin.addServerServiceMapping("activemq:queue:inbox", "orders");
```

Then when a message is consumed from that inbox queue, it becomes a zipkin server event with the service name 'orders'.

Now suppose that the call to http:someserver/somepath is also a service, which you want to map to a client service name, which can be done as:

```
zipkin.addClientServiceMapping("http:someserver/somepath", "audit");
```

Then in the same Camel application you have mapped incoming and outgoing endpoints to different zipkin service names.

You can use wildcards in the service mapping. To match all outgoing calls to the same HTTP server you can do:

```
zipkin.addClientServiceMapping("http:someserver*", "audit");
```

356.3. MAPPING RULES

The service name mapping for server occurs using the following rules

1. Is there an exclude pattern that matches the endpoint uri of the from endpoint? If yes then skip.
2. Is there a match in the serviceServiceMapping that matches the endpoint uri of the from endpoint? If yes, then use the found service name
3. Is there a match in the serviceServiceMapping that matches the route id of the current route? If yes, then use the found service name
4. Is there a match in the serviceServiceMapping that matches the original route id where the exchange started? If yes, then use the found service name
5. No service name was found, the exchange is not traced by zipkin

The service name mapping for client occurs using the following rules

1. Is there an exclude pattern that matches the endpoint uri of the from endpoint? If yes then skip.
2. Is there a match in the clientServiceMapping that matches the endpoint uri of endpoint where the message is being sent to? If yes, then use the found service name
3. Is there a match in the clientServiceMapping that matches the route id of the current route? If yes, then use the found service name
4. Is there a match in the clientServiceMapping that matches the original route id where the exchange started? If yes, then use the found service name
5. No service name was found, the exchange is not traced by zipkin

356.3.1. No client or server mappings

If there has been no configuration of client or server service mappings, CamelZipkin runs in a fallback mode, and uses endpoint uris as the service name.

In the example above, this would mean the service names would be defined as if you add the following code yourself:

```
zipkin.addServerServiceMapping("activemq:queue:inbox", "activemq:queue:inbox");
zipkin.addClientServiceMapping("http:someserver/somepath", "http:someserver/somepath");
```

This is not a recommended approach, but gets you up and running quickly without doing any service name mappings. However, when you have multiple systems across your infrastructure, then you should consider using human-readable service names, that you map to instead of using the camel endpoint uris.

356.4. CAMEL-ZIPIN-STARTER

If you are using Spring Boot then you can add the **camel-zipkin-starter** dependency, and turn on zipkin by annotating the main class with **@CamelZipkin**. You can then configure camel-zipkin in the **application.properties** file where you can configure the hostname and port number for the Zipkin Server, and all the other options as listed in the options table above.

You can find an example of this in the camel-example-zipkin

CHAPTER 357. ZOOKEEPER COMPONENT

Available as of Camel version 2.9

The ZooKeeper component allows interaction with a ZooKeeper cluster and exposes the following features to Camel:

1. Creation of nodes in any of the ZooKeeper create modes.
2. Get and Set the data contents of arbitrary cluster nodes (data being set must be convertible to **byte[]**).
3. Create and retrieve the list the child nodes attached to a particular node.
4. A Distributed **RoutePolicy** that leverages a Leader election coordinated by ZooKeeper to determine if exchanges should get processed.

Maven users will need to add the following dependency to their **pom.xml** for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-zookeeper</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

357.1. URI FORMAT

```
zookeeper://zookeeper-server[:port][/path][?options]
```

The path from the URI specifies the node in the ZooKeeper server (a.k.a. *znode*) that will be the target of the endpoint:

357.2. OPTIONS

The ZooKeeper component supports 2 options which are listed below.

Name	Description	Default	Type
configuration (advanced)	To use a shared ZooKeeperConfiguration		ZooKeeperConfiguration
resolveProperty Placeholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The ZooKeeper endpoint is configured using URI syntax:

```
zookeeper:serverUrls/path
```

with the following path and query parameters:

357.2.1. Path Parameters (2 parameters):

Name	Description	Default	Type
serverUrls	**Required** The zookeeper server hosts (multiple servers can be separated by comma)		String
path	**Required** The node in the ZooKeeper server (aka znode)		String

357.2.2. Query Parameters (12 parameters):

Name	Description	Default	Type
awaitExistence (common)	**Deprecated** Not in use	true	boolean
listChildren (common)	Whether the children of the node should be listed	false	boolean
timeout (common)	The time interval to wait on connection before timing out.	5000	int
backoff (consumer)	The time interval to backoff for after an error before retrying.	5000	long
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean
repeat (consumer)	Should changes to the znode be 'watched' and repeatedly processed.	false	boolean
sendEmptyMessageOnDelete (consumer)	Upon the delete of a znode, should an empty message be send to the consumer	true	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
create (producer)	Should the endpoint create the node if it does not currently exist.	false	boolean
createMode (producer)	The create mode that should be used for the newly created node	EPHEMERAL	String
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

357.3. USE CASES

357.3.1. Reading from a *znode*

The following snippet will read the data from the *znode*/**somepath**/**somenode**/ provided that it already exists. The data retrieved will be placed into an exchange and passed onto the rest of the route:

```
from("zookeeper://localhost:39913/somepath/somenode").to("mock:result");
```

If the node does not yet exist then a flag can be supplied to have the endpoint await its creation:

```
from("zookeeper://localhost:39913/somepath/somenode?awaitCreation=true").to("mock:result");
```

357.3.2. Reading from a *znode* (additional Camel 2.10 onwards)

When data is read due to a **WatchedEvent** received from the ZooKeeper ensemble, the **CamelZookeeperEventType** header holds ZooKeeper's **EventType** value from that **WatchedEvent**. If the data is read initially (not triggered by a **WatchedEvent**) the **CamelZookeeperEventType** header will not be set.

357.3.3. Writing to a *znode*

The following snippet will write the payload of the exchange into the znode at /**somepath**/**somenode**/ provided that it already exists:

```
from("direct:write-to-znode")
    .to("zookeeper://localhost:39913/somepath/somenode");
```

For flexibility, the endpoint allows the target *znode* to be specified dynamically as a message header. If a header keyed by the string **CamelZooKeeperNode** is present then the value of the header will be used as the path to the *znode* on the server. For instance using the same route definition above, the following code snippet will write the data not to **/somepath/somenode** but to the path from the header **/somepath/someothernode**.

WARNING

the **testPayload** must be convertible to **byte[]** as the data stored in ZooKeeper is byte based.

```
Object testPayload = ...
template.sendBodyAndHeader("direct:write-to-znode", testPayload, "CamelZooKeeperNode",
"/somepath/someothernode");
```

To also create the node if it does not exist the **create** option should be used.

```
from("direct:create-and-write-to-znode")
    .to("zookeeper://localhost:39913/somepath/somenode?create=true");
```

Starting **version 2.11** it is also possible to **delete** a node using the header **CamelZookeeperOperation** by setting it to **DELETE**:

```
from("direct:delete-znode")
    .setHeader(ZooKeeperMessage.ZOOKEEPER_OPERATION, constant("DELETE"))
    .to("zookeeper://localhost:39913/somepath/somenode");
```

or equivalently:

```
<route>
  <from uri="direct:delete-znode" />
  <setHeader headerName="CamelZookeeperOperation">
    <constant>DELETE</constant>
  </setHeader>
  <to uri="zookeeper://localhost:39913/somepath/somenode" />
</route>
```

ZooKeeper nodes can have different types; they can be 'Ephemeral' or 'Persistent' and 'Sequenced' or 'Unsequenced'. For further information of each type you can check here. By default endpoints will create unsequenced, ephemeral nodes, but the type can be easily manipulated via a uri config parameter or via a special message header. The values expected for the create mode are simply the names from the **CreateMode** enumeration:

- **PERSISTENT**
- **PERSISTENT_SEQUENTIAL**

- **EPHEMERAL**
- **EPHEMERAL_SEQUENTIAL**

For example to create a persistent *znode* via the URI config:

```
from("direct:create-and-write-to-persistent-znode")
    .to("zookeeper://localhost:39913/somepath/somenode?create=true&createMode=PERSISTENT");
```

or using the header **CamelZookeeperCreateMode**.

WARNING

the **testPayload** must be convertible to **byte[]** as the data stored in ZooKeeper is byte based.

```
Object testPayload = ...
template.sendBodyAndHeader("direct:create-and-write-to-persistent-znode", testPayload,
"CamelZooKeeperCreateMode", "PERSISTENT");
```

357.4. ZOOKEEPER ENABLED ROUTE POLICIES

ZooKeeper allows for very simple and effective leader election out of the box. This component exploits this election capability in a **RoutePolicy** to control when and how routes are enabled. This policy would typically be used in fail-over scenarios, to control identical instances of a route across a cluster of Camel based servers. A very common scenario is a simple 'Master-Slave' setup where there are multiple instances of a route distributed across a cluster but only one of them, that of the master, should be running at a time. If the master fails, a new master should be elected from the available slaves and the route in this new master should be started.

The policy uses a common *znode* path across all instances of the **RoutePolicy** that will be involved in the election. Each policy writes its id into this node and Zookeeper will order the writes in the order it received them. The policy then reads the listing of the node to see what position of its id; this position is used to determine if the route should be started or not. The policy is configured at startup with the number of route instances that should be started across the cluster and if its position in the list is less than this value then its route will be started. For a Master-slave scenario, the route is configured with 1 route instance and only the first entry in the listing will start its route. All policies watch for updates to the listing and if the listing changes they recalculate if their route should be started. For more info on Zookeeper's leader election capability see this page.

The following example uses the node **/someapplication/somepolicy** for the election and is set up to start only the top '1' entries in the node listing i.e. elect a master:

```
ZooKeeperRoutePolicy policy = new
ZooKeeperRoutePolicy("zookeeper:localhost:39913/someapp/somepolicy", 1);
from("direct:policy-controlled")
    .routePolicy(policy)
    .to("mock:controlled");
```

There are currently 3 policies defined in the component, with different SLAs:

- **ZooKeeperRoutePolicy**
- **CuratorLeaderRoutePolicy** (since 2.19)
- **MultiMasterCuratorLeaderRoutePolicy** (since 2.19)

ZooKeeperRoutePolicy supports multiple active nodes, but it's activation kicks in only after a Camel component and its correspondent Consumer have already been started, this introduces, depending on your routes definition, the risk that you component can already start consuming events and producing `Exchange`s, before the policy could estabilish that the node should not be activated.

CuratorLeaderRoutePolicy supports only a single active node, but it's bound to a different **CamelContext** lifecycle method; this Policy kicks in before any route or consumer is started thus you can be sure that no even is processed before the Policy takes its decision.

MultiMasterCuratorLeaderRoutePolicy support multiple active nodes, and it's bound to the same lifecycle method as **CuratorLeaderRoutePolicy**; this Policy kicks in before any route or consumer is started thus you can be sure that no even is processed before the Policy takes its decision.

357.5. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started

CHAPTER 358. ZOOKEEPER MASTER COMPONENT

Available as of Camel version 2.19

The **zookeeper-master:** endpoint provides a way to ensure only a single consumer in a cluster consumes from a given endpoint; with automatic failover if that JVM dies.

This can be very useful if you need to consume from some legacy back end which either doesn't support concurrent consumption or due to commercial or stability reasons you can only have a single connection at any point in time.

358.1. USING THE MASTER ENDPOINT

Just prefix any camel endpoint with **zookeeper-master:someName:** where *someName* is a logical name and is used to acquire the master lock. e.g.

```
from("zookeeper-master:cheese:jms:foo").to("activemq:wine");
```

The above simulates the [Exclusive Consumers](http://activemq.apache.org/exclusive-consumer.html) type feature in ActiveMQ; but on any third party JMS provider which maybe doesn't support exclusive consumers.

358.2. URI FORMAT

```
zookeeper-master:name:endpoint[?options]
```

Where endpoint is any Camel endpoint you want to run in master/slave mode.

358.3. OPTIONS

The ZooKeeper Master component supports 7 options which are listed below.

Name	Description	Default	Type
containerIdFactory (consumer)	To use a custom ContainerIdFactory for creating container ids.		ContainerIdFactory
zkRoot (consumer)	The root path to use in zookeeper where information is stored which nodes are master/slave etc. Will by default use: /camel/zookeepermaster/clusters/master	/camel/zookeepermaster/clusters/master	String
curator (advanced)	To use a custom configured CuratorFramework as connection to zookeeper ensemble.		CuratorFramework

Name	Description	Default	Type
maximumConnectionTimeout (consumer)	Timeout in millis to use when connecting to the zookeeper ensemble	10000	int
zooKeeperUrl (consumer)	The url for the zookeeper ensemble	localhost:2181	String
zooKeeperPassword (security)	The password to use when connecting to the zookeeper ensemble		String
resolvePropertyPlaceholders (advanced)	Whether the component should resolve property placeholders on itself when starting. Only properties which are of String type can use property placeholders.	true	boolean

The ZooKeeper Master endpoint is configured using URI syntax:

```
zookeeper-master:groupName:consumerEndpointUri
```

with the following path and query parameters:

358.3.1. Path Parameters (2 parameters):

Name	Description	Default	Type
groupName	**Required** The name of the cluster group to use		String
consumerEndpointUri	**Required** The consumer endpoint to use in master/slave mode		String

358.3.2. Query Parameters (4 parameters):

Name	Description	Default	Type
bridgeErrorHandler (consumer)	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while the consumer is trying to pickup incoming messages, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that will be logged at WARN or ERROR level and ignored.	false	boolean

Name	Description	Default	Type
exceptionHandler (consumer)	To let the consumer use a custom ExceptionHandler. Notice if the option bridgeErrorHandler is enabled then this options is not in use. By default the consumer will deal with exceptions, that will be logged at WARN or ERROR level and ignored.		ExceptionHandler
exchangePattern (consumer)	Sets the exchange pattern when the consumer creates an exchange.		ExchangePattern
synchronous (advanced)	Sets whether synchronous processing should be strictly used, or Camel is allowed to use asynchronous processing (if supported).	false	boolean

358.4. EXAMPLE

You can protect a clustered Camel application to only consume files from one active node.

```
// the file endpoint we want to consume from
String url = "file:target/inbox?delete=true";

// use the zookeeper master component in the clustered group named myGroup
// to run a master/slave mode in the following Camel url
from("zookeeper-master:myGroup:" + url)
    .log(name + " - Received file: ${file:name}")
    .delay(delay)
    .log(name + " - Done file:     ${file:name}")
    .to("file:target/outbox");
```

ZooKeeper will by default connect to **localhost:2181**, but you can configure this on the component level.

```
MasterComponent master = new MasterComponent();
master.setZooKeeperUrl("myzookeeper:2181");
```

However you can also configure the url of the ZooKeeper ensemble using environment variables.

```
export ZOOKEEPER_URL = "myzookeeper:2181"
```

CHAPTER 359. MASTER ROUTEPOLICY

You can also use a **RoutePolicy** to control routes in master/slave mode.

When doing so you must configure the route policy with

- url to zookeeper ensemble
- name of cluster group
- **important** and set the route to not auto startup

A little example

```
MasterRoutePolicy master = new MasterRoutePolicy();
master.setZooKeeperUrl("localhost:2181");
master.setGroupName("myGroup");

// its import to set the route to not auto startup
// as we let the route policy start/stop the routes when it becomes a master/slave etc
from("file:target/inbox?delete=true").noAutoStartup()
    // use the zookeeper master route policy in the clustered group
    // to run this route in master/slave mode
    .routePolicy(master)
    .log(name + " - Received file: ${file:name}")
    .delay(delay)
    .log(name + " - Done file:     ${file:name}")
    .to("file:target/outbox");
```

359.1. SEE ALSO

- Configuring Camel
- Component
- Endpoint
- Getting Started