



Red Hat JBoss Fuse 6.1

Apache Camel Component Reference

Quickly find the settings for all of the components

Red Hat JBoss Fuse 6.1 Apache Camel Component Reference

Quickly find the settings for all of the components

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Copyright © 2013 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution-Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

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 | 33 |
| 1.1. LIST OF COMPONENTS | 33 |
| CHAPTER 2. ACTIVEMQ | 53 |
| ACTIVEMQ COMPONENT | 53 |
| URI FORMAT | 53 |
| OPTIONS | 53 |
| CONFIGURING THE CONNECTION FACTORY | 53 |
| CONFIGURING THE CONNECTION FACTORY USING SPRING XML | 54 |
| USING CONNECTION POOLING | 54 |
| INVOKING MESSAGELISTENER POJOS IN A ROUTE | 55 |
| USING ACTIVEMQ DESTINATION OPTIONS | 55 |
| CONSUMING ADVISORY MESSAGES | 56 |
| GETTING COMPONENT JAR | 57 |
| CHAPTER 3. AHC | 58 |
| ASYNC HTTP CLIENT (AHC) COMPONENT | 58 |
| URI FORMAT | 58 |
| AHCENDPOINT OPTIONS | 58 |
| AHCCOMPONENT OPTIONS | 60 |
| MESSAGE HEADERS | 61 |
| MESSAGE BODY | 62 |
| RESPONSE CODE | 62 |
| AHCOPERATIONFAILEDEXCEPTION | 62 |
| CALLING USING GET OR POST | 62 |
| CONFIGURING URI TO CALL | 63 |
| CONFIGURING URI PARAMETERS | 63 |
| HOW TO SET THE HTTP METHOD (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) TO THE HTTP PRODUCER | 63 |
| CONFIGURING CHARSET | 64 |
| URI PARAMETERS FROM THE ENDPOINT URI | 64 |
| URI PARAMETERS FROM THE MESSAGE | 64 |
| GETTING THE RESPONSE CODE | 64 |
| CONFIGURING ASYNCHTTPCLIENT | 65 |
| SSL SUPPORT (HTTPS) | 65 |
| USING THE JSSE CONFIGURATION UTILITY | 65 |
| PROGRAMMATIC CONFIGURATION OF THE COMPONENT | 66 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 66 |
| CHAPTER 4. AMQP | 67 |
| AMQP | 67 |
| URI FORMAT | 67 |
| CHAPTER 5. APNS | 68 |
| APNS COMPONENT | 68 |
| URI FORMAT | 68 |
| OPTIONS | 68 |
| PRODUCER | 68 |
| CONSUMER | 69 |
| COMPONENT | 69 |
| EXCHANGE DATA FORMAT | 69 |
| MESSAGE HEADERS | 69 |

| | |
|--|------------|
| SAMPLES | 70 |
| CAMEL XML ROUTE | 70 |
| CAMEL JAVA ROUTE | 71 |
| CREATE CAMEL CONTEXT AND DECLARE APNS COMPONENT PROGRAMMATICALLY | 71 |
| APNSPRODUCER - IOS TARGET DEVICE DYNAMICALLY CONFIGURED VIA HEADER: "CAMELAPNSTOKENS" | 71 |
| APNSPRODUCER - IOS TARGET DEVICE STATICALLY CONFIGURED VIA URI | 72 |
| APNSCONSUMER | 72 |
| SEE ALSO | 72 |
| CHAPTER 6. ATOM | 73 |
| ATOM COMPONENT | 73 |
| URI FORMAT | 73 |
| OPTIONS | 73 |
| EXCHANGE DATA FORMAT | 75 |
| MESSAGE HEADERS | 75 |
| SAMPLES | 75 |
| CHAPTER 7. AVRO | 78 |
| AVRO COMPONENT | 78 |
| APACHE AVRO OVERVIEW | 78 |
| USING THE AVRO DATA FORMAT | 79 |
| USING AVRO RPC IN CAMEL | 80 |
| AVRO RPC URI OPTIONS | 80 |
| AVRO RPC HEADERS | 81 |
| EXAMPLES | 81 |
| CHAPTER 8. AWS | 83 |
| 8.1. INTRODUCTION TO THE AWS COMPONENTS | 83 |
| 8.2. AWS-CW | 83 |
| 8.3. AWS-DDB | 86 |
| 8.4. AWS-S3 | 93 |
| 8.5. AWS-SDB | 99 |
| 8.6. AWS-SES | 104 |
| 8.7. AWS-SNS | 107 |
| 8.8. AWS-SQS | 109 |
| CHAPTER 9. BEAN | 115 |
| BEAN COMPONENT | 115 |
| URI FORMAT | 115 |
| OPTIONS | 115 |
| USING | 115 |
| BEAN AS ENDPOINT | 116 |
| JAVA DSL BEAN SYNTAX | 117 |
| BEAN BINDING | 117 |
| CHAPTER 10. BEAN VALIDATION | 118 |
| BEAN VALIDATION COMPONENT | 118 |
| URI FORMAT | 118 |
| URI OPTIONS | 118 |
| EXAMPLE | 119 |
| CHAPTER 11. BROWSE | 122 |
| BROWSE COMPONENT | 122 |
| URI FORMAT | 122 |

| | |
|--|------------|
| SAMPLE | 122 |
| CHAPTER 12. CACHE | 123 |
| 12.1. CACHE COMPONENT | 123 |
| 12.2. CACHEREPLICATIONJMSEXAMPLE | 131 |
| CHAPTER 13. CDI | 136 |
| CAMEL CDI | 136 |
| DEPENDENCY INJECTING CAMEL WITH CDI | 136 |
| BOOTSTRAPPING CAMEL WITH CDI CONTAINER | 136 |
| SEE ALSO | 137 |
| CHAPTER 14. CLASS | 138 |
| CLASS COMPONENT | 138 |
| URI FORMAT | 138 |
| OPTIONS | 138 |
| USING | 138 |
| SETTING PROPERTIES ON THE CREATED INSTANCE | 139 |
| CHAPTER 15. CMIS | 140 |
| CMIS COMPONENT | 140 |
| URI FORMAT | 140 |
| URI OPTIONS | 140 |
| USAGE | 141 |
| MESSAGE HEADERS EVALUATED BY THE PRODUCER | 141 |
| MESSAGE HEADERS SET DURING QUERYING PRODUCER OPERATION | 141 |
| CHAPTER 16. COMETD | 143 |
| COMETD COMPONENT | 143 |
| URI FORMAT | 143 |
| EXAMPLES | 143 |
| OPTIONS | 143 |
| AUTHENTICATION | 145 |
| SETTING UP SSL FOR COMETD COMPONENT | 145 |
| USING THE JSSE CONFIGURATION UTILITY | 145 |
| PROGRAMMATIC CONFIGURATION OF THE COMPONENT | 145 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 146 |
| CHAPTER 17. CONTEXT | 147 |
| CONTEXT COMPONENT | 147 |
| URI FORMAT | 147 |
| EXAMPLE | 147 |
| DEFINING THE CONTEXT COMPONENT | 148 |
| USING THE CONTEXT COMPONENT | 148 |
| NAMING ENDPOINTS | 149 |
| CHAPTER 18. CONTROLBUS COMPONENT | 150 |
| CONTROLBUS COMPONENT | 150 |
| COMMANDS | 150 |
| OPTIONS | 150 |
| SAMPLES | 151 |
| USING ROUTE COMMAND | 151 |
| GETTING PERFORMANCE STATISTICS | 152 |
| USING SIMPLE LANGUAGE | 152 |

| | |
|--|------------|
| CHAPTER 19. COUCHDB | 154 |
| CAMEL COUCHDB COMPONENT | 154 |
| URI FORMAT | 154 |
| OPTIONS | 154 |
| HEADERS | 155 |
| MESSAGE BODY | 155 |
| SAMPLES | 155 |
| CHAPTER 20. CRYPTO (DIGITAL SIGNATURES) | 157 |
| CRYPTO COMPONENT FOR DIGITAL SIGNATURES | 157 |
| INTRODUCTION | 157 |
| URI FORMAT | 157 |
| OPTIONS | 158 |
| 1) RAW KEYS | 159 |
| 2) KEYSTORES AND ALIASES. | 159 |
| 3) CHANGING JCE PROVIDER AND ALGORITHM | 160 |
| 4) CHANGING THE SIGNATURE MESASGE HEADER | 161 |
| 5) CHANGING THE BUFFERSIZE | 161 |
| 6) SUPPLYING KEYS DYNAMICALLY. | 161 |
| CHAPTER 21. CXF | 164 |
| CXF COMPONENT | 164 |
| URI FORMAT | 164 |
| OPTIONS | 165 |
| THE DESCRIPTIONS OF THE DATAFORMATS | 168 |
| CONFIGURING THE CXF ENDPOINTS WITH APACHE ARIES BLUEPRINT. | 169 |
| HOW TO ENABLE CXF'S LOGGINGOUTINTERCEPTOR IN MESSAGE MODE | 171 |
| DESCRIPTION OF RELAYHEADERS OPTION | 171 |
| AVAILABLE ONLY IN POJO MODE | 171 |
| CHANGES SINCE RELEASE 2.0 | 172 |
| CONFIGURE THE CXF ENDPOINTS WITH SPRING | 174 |
| HOW TO MAKE THE CAMEL-CXF COMPONENT USE LOG4J INSTEAD OF JAVA.UTIL.LOGGING | 176 |
| HOW TO LET CAMEL-CXF RESPONSE MESSAGE WITH XML START DOCUMENT | 176 |
| HOW TO CONSUME A MESSAGE FROM A CAMEL-CXF ENDPOINT IN POJO DATA FORMAT | 177 |
| HOW TO PREPARE THE MESSAGE FOR THE CAMEL-CXF ENDPOINT IN POJO DATA FORMAT | 179 |
| HOW TO DEAL WITH THE MESSAGE FOR A CAMEL-CXF ENDPOINT IN PAYLOAD DATA FORMAT | 179 |
| HOW TO GET AND SET SOAP HEADERS IN POJO MODE | 180 |
| HOW TO GET AND SET SOAP HEADERS IN PAYLOAD MODE | 182 |
| SOAP HEADERS ARE NOT AVAILABLE IN MESSAGE MODE | 182 |
| HOW TO THROW A SOAP FAULT FROM APACHE CAMEL | 182 |
| HOW TO PROPAGATE A CXF ENDPOINT'S REQUEST AND RESPONSE CONTEXT | 183 |
| ATTACHMENT SUPPORT | 184 |
| HOW TO PROPAGATE STACK TRACE INFORMATION | 187 |
| STREAMING SUPPORT IN PAYLOAD MODE | 188 |
| CHAPTER 22. CXF BEAN COMPONENT | 189 |
| CXF BEAN COMPONENT (2.0 OR LATER) | 189 |
| URI FORMAT | 189 |
| OPTIONS | 189 |
| HEADERS | 190 |
| A WORKING SAMPLE | 192 |
| CHAPTER 23. CXFRS | 194 |
| CXFRS COMPONENT | 194 |

| | |
|--|------------|
| URI FORMAT | 194 |
| OPTIONS | 194 |
| HOW TO CONFIGURE THE REST ENDPOINT IN APACHE CAMEL | 197 |
| CONSUMING A REST REQUEST - SIMPLE BINDING STYLE | 199 |
| ENABLING THE SIMPLE BINDING STYLE | 199 |
| EXAMPLES OF REQUEST BINDING WITH DIFFERENT METHOD SIGNATURES | 199 |
| MORE EXAMPLES OF THE SIMPLE BINDING STYLE | 200 |
| CONSUMING A REST REQUEST - DEFAULT BINDING STYLE | 201 |
| HOW TO INVOKE THE REST SERVICE THROUGH CAMEL-CXF RS PRODUCER ? | 203 |
| CHAPTER 24. DATAFORMAT COMPONENT | 206 |
| DATA FORMAT COMPONENT | 206 |
| URI FORMAT | 206 |
| SAMPLES | 206 |
| CHAPTER 25. DATASET | 207 |
| DATASET COMPONENT | 207 |
| URI FORMAT | 207 |
| OPTIONS | 207 |
| CONFIGURING DATASET | 208 |
| EXAMPLE | 208 |
| PROPERTIES ON SIMPLEDATASET | 208 |
| CHAPTER 26. DIRECT | 210 |
| DIRECT COMPONENT | 210 |
| URI FORMAT | 210 |
| OPTIONS | 210 |
| SAMPLES | 211 |
| CHAPTER 27. DIRECT-VM | 212 |
| DIRECT VM COMPONENT | 212 |
| URI FORMAT | 212 |
| OPTIONS | 212 |
| SAMPLES | 213 |
| CHAPTER 28. DISRUPTOR | 214 |
| DISRUPTOR COMPONENT | 214 |
| URI FORMAT | 215 |
| OPTIONS | 215 |
| WAIT STRATEGIES | 217 |
| USE OF REQUEST REPLY | 218 |
| CONCURRENT CONSUMERS | 218 |
| THREAD POOLS | 219 |
| SAMPLE | 219 |
| USING MULTIPLECONSUMERS | 219 |
| EXTRACTING DISRUPTOR INFORMATION | 220 |
| CHAPTER 29. DNS | 221 |
| DNS | 221 |
| URI FORMAT | 221 |
| OPTIONS | 221 |
| HEADERS | 221 |
| EXAMPLES | 222 |
| IP LOOKUP | 222 |
| DNS LOOKUP | 222 |

| | |
|---|------------|
| DNS DIG | 223 |
| CHAPTER 30. EJB | 224 |
| EJB COMPONENT | 224 |
| URI FORMAT | 224 |
| OPTIONS | 224 |
| BEAN BINDING | 224 |
| EXAMPLES | 225 |
| USING JAVA DSL | 225 |
| USING SPRING XML | 226 |
| CHAPTER 31. ELASTICSEARCH | 227 |
| ELASTICSEARCH COMPONENT | 227 |
| URI FORMAT | 227 |
| ENDPOINT OPTIONS | 227 |
| MESSAGE OPERATIONS | 227 |
| INDEX EXAMPLE | 228 |
| FOR MORE INFORMATION, SEE THESE RESOURCES | 228 |
| CHAPTER 32. EVENTADMIN | 229 |
| EVENTADMIN COMPONENT | 229 |
| DEPENDENCIES | 229 |
| URI FORMAT | 229 |
| URI OPTIONS | 229 |
| MESSAGE HEADERS | 229 |
| MESSAGE BODY | 229 |
| EXAMPLE USAGE | 229 |
| CHAPTER 33. EXEC | 231 |
| EXEC COMPONENT | 231 |
| DEPENDENCIES | 231 |
| URI FORMAT | 231 |
| URI OPTIONS | 231 |
| MESSAGE HEADERS | 232 |
| MESSAGE BODY | 234 |
| EXECUTING WORD COUNT (LINUX) | 234 |
| EXECUTING JAVA | 235 |
| EXECUTING ANT SCRIPTS | 235 |
| EXECUTING ECHO (WINDOWS) | 236 |
| CHAPTER 34. FABRIC COMPONENT | 237 |
| DEPENDENCIES | 237 |
| URI FORMAT | 237 |
| URI OPTIONS | 238 |
| USE CASES FOR FABRIC ENDPOINTS | 238 |
| LOCATION DISCOVERY | 238 |
| LOAD-BALANCING CLUSTER | 239 |
| AUTO-RECONNECT FEATURE | 240 |
| PUBLISHING AN ENDPOINT URI | 240 |
| LOOKING UP AN ENDPOINT URI | 242 |
| LOAD-BALANCING EXAMPLE | 243 |
| OSGI BUNDLE PLUG-IN CONFIGURATION | 244 |
| CHAPTER 35. FACEBOOK | 246 |
| FACEBOOK COMPONENT | 246 |

| | |
|--|------------|
| URI FORMAT | 246 |
| FACEBOOKCOMPONENT | 246 |
| PRODUCER ENDPOINTS: | 248 |
| CONSUMER ENDPOINTS: | 260 |
| READING OPTIONS | 264 |
| MESSAGE HEADER | 265 |
| MESSAGE BODY | 265 |
| USE CASES | 265 |
| CHAPTER 36. FILE2 | 267 |
| FILE COMPONENT - APACHE CAMEL 2.0 ONWARDS | 267 |
| URI FORMAT | 267 |
| URI OPTIONS | 267 |
| CONSUMER ONLY | 269 |
| DEFAULT BEHAVIOR FOR FILE CONSUMER | 277 |
| PRODUCER ONLY | 277 |
| DEFAULT BEHAVIOR FOR FILE PRODUCER | 280 |
| MOVE AND DELETE OPERATIONS | 280 |
| FINE GRAINED CONTROL OVER MOVE AND PREMOVE OPTION | 281 |
| ABOUT MOVEFAILED | 282 |
| MESSAGE HEADERS | 282 |
| FILE PRODUCER ONLY | 282 |
| FILE CONSUMER ONLY | 282 |
| BATCH CONSUMER | 283 |
| EXCHANGE PROPERTIES, FILE CONSUMER ONLY | 283 |
| COMMON GOTCHAS WITH FOLDER AND FILENAMES | 284 |
| FILENAME EXPRESSION | 284 |
| CONSUMING FILES FROM FOLDERS WHERE OTHERS DROP FILES DIRECTLY | 284 |
| USING DONE FILES | 284 |
| WRITING DONE FILES | 285 |
| READ FROM A DIRECTORY AND WRITE TO ANOTHER DIRECTORY | 286 |
| READ FROM A DIRECTORY AND WRITE TO ANOTHER DIRECTORY USING A OVERRULE DYNAMIC NAME | 286 |
| READING RECURSIVELY FROM A DIRECTORY AND WRITING TO ANOTHER | 286 |
| USING FLATTEN | 286 |
| READING FROM A DIRECTORY AND THE DEFAULT MOVE OPERATION | 287 |
| READ FROM A DIRECTORY AND PROCESS THE MESSAGE IN JAVA | 287 |
| READ FILES FROM A DIRECTORY AND SEND THE CONTENT TO A JMS QUEUE | 287 |
| WRITING TO FILES | 288 |
| WRITE TO SUBDIRECTORY USING EXCHANGE.FILE_NAME | 289 |
| USING EXPRESSION FOR FILENAMES | 289 |
| AVOIDING READING THE SAME FILE MORE THAN ONCE (IDEMPOTENT CONSUMER) | 289 |
| USING A FILE BASED IDEMPOTENT REPOSITORY | 290 |
| USING A JPA BASED IDEMPOTENT REPOSITORY | 291 |
| FILTER USING ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER | 292 |
| FILTERING USING ANT PATH MATCHER | 292 |
| SORTING USING COMPARATOR | 293 |
| SORTING USING SORTBY | 294 |
| USING GENERICFILEPROCESSSTRATEGY | 295 |
| DEBUG LOGGING | 295 |
| CHAPTER 37. FLATPACK | 296 |
| FLATPACK COMPONENT | 296 |

| | |
|--|------------|
| URI FORMAT | 296 |
| URI OPTIONS | 296 |
| EXAMPLES | 297 |
| MESSAGE HEADERS | 297 |
| MESSAGE BODY | 297 |
| HEADER AND TRAILER RECORDS | 298 |
| USING THE ENDPOINT | 298 |
| CHAPTER 38. FOP | 299 |
| FOP COMPONENT | 299 |
| URI FORMAT | 299 |
| OUTPUT FORMATS | 299 |
| ENDPOINT OPTIONS | 300 |
| MESSAGE OPERATIONS | 300 |
| EXAMPLE | 301 |
| CHAPTER 39. FREEMARKER | 302 |
| FREEMARKER | 302 |
| URI FORMAT | 302 |
| OPTIONS | 302 |
| FREEMARKER CONTEXT | 302 |
| HOT RELOADING | 303 |
| DYNAMIC TEMPLATES | 303 |
| SAMPLES | 304 |
| THE EMAIL SAMPLE | 304 |
| CHAPTER 40. FTP2 | 306 |
| FTP/SFTP COMPONENT | 306 |
| URI FORMAT | 306 |
| URI OPTIONS | 306 |
| MORE URI OPTIONS | 315 |
| EXAMPLES | 315 |
| DEFAULT WHEN CONSUMING FILES | 316 |
| LIMITATIONS | 316 |
| MESSAGE HEADERS | 316 |
| ABOUT TIMEOUTS | 317 |
| USING LOCAL WORK DIRECTORY | 317 |
| STEPWISE CHANGING DIRECTORIES | 318 |
| USING STEPWISE=TRUE (DEFAULT MODE) | 318 |
| USING STEPWISE=FALSE | 320 |
| SAMPLES | 321 |
| CONSUMING A REMOTE FTP SERVER TRIGGERED BY A ROUTE | 322 |
| CONSUMING A REMOTE FTPS SERVER (IMPLICIT SSL) AND CLIENT AUTHENTICATION | 322 |
| CONSUMING A REMOTE FTPS SERVER (EXPLICIT TLS) AND A CUSTOM TRUST STORE CONFIGURATION | 322 |
| FILTER USING ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER | 322 |
| FILTERING USING ANT PATH MATCHER | 323 |
| USING A PROXY WITH SFTP | 324 |
| CONSUMING A SINGLE FILE USING A FIXED NAME | 324 |
| DEBUG LOGGING | 325 |
| CHAPTER 41. GAE | 326 |
| 41.1. INTRODUCTION TO THE GAE COMPONENTS | 326 |
| 41.2. GAUTH | 330 |

| | |
|--|------------|
| 41.3. GHTTP | 338 |
| 41.4. GLOGIN | 343 |
| 41.5. GMAIL | 347 |
| 41.6. GSEC | 349 |
| 41.7. GTASK | 350 |
| CHAPTER 42. GEOCODER | 354 |
| GEOCODER COMPONENT | 354 |
| URI FORMAT | 354 |
| OPTIONS | 354 |
| EXCHANGE DATA FORMAT | 354 |
| MESSAGE HEADERS | 355 |
| SAMPLES | 355 |
| CHAPTER 43. GUAVA EVENTBUS | 357 |
| GUAVA EVENTBUS COMPONENT | 357 |
| URI FORMAT | 357 |
| OPTIONS | 357 |
| USAGE | 358 |
| DEADEVENT CONSIDERATIONS | 358 |
| CONSUMING MULTIPLE TYPE OF EVENTS | 359 |
| CHAPTER 44. HAWTDB | 361 |
| HAWTDB | 361 |
| USING HAWTDBAGGREGATIONREPOSITORY | 361 |
| WHAT IS PRESERVED WHEN PERSISTING | 363 |
| RECOVERY | 363 |
| USING HAWTDBAGGREGATIONREPOSITORY IN JAVA DSL | 364 |
| USING HAWTDBAGGREGATIONREPOSITORY IN SPRING XML | 364 |
| DEPENDENCIES | 365 |
| CHAPTER 45. HAZELCAST COMPONENT | 366 |
| HAZELCAST COMPONENT | 366 |
| URI FORMAT | 366 |
| SECTIONS | 366 |
| USAGE OF MAP | 367 |
| MAP CACHE PRODUCER - TO("HAZELCAST:MAP:FOO") | 367 |
| SAMPLE FOR PUT: | 368 |
| SAMPLE FOR GET: | 368 |
| SAMPLE FOR UPDATE: | 368 |
| SAMPLE FOR DELETE: | 369 |
| SAMPLE FOR QUERY | 369 |
| MAP CACHE CONSUMER - FROM("HAZELCAST:MAP:FOO") | 370 |
| USAGE OF MULTI MAP | 371 |
| MULTIMAP CACHE PRODUCER - TO("HAZELCAST:MULTIMAP:FOO") | 372 |
| SAMPLE FOR PUT: | 372 |
| SAMPLE FOR GET: | 373 |
| SAMPLE FOR UPDATE: | 373 |
| SAMPLE FOR DELETE: | 374 |
| SAMPLE FOR QUERY | 374 |
| MAP CACHE CONSUMER - FROM("HAZELCAST:MAP:FOO") | 375 |
| USAGE OF MULTI MAP | 376 |
| MULTIMAP CACHE PRODUCER - TO("HAZELCAST:MULTIMAP:FOO") | 376 |
| SAMPLE FOR PUT: | 377 |

| | |
|---|------------|
| SAMPLE FOR REMOVEVALUE: | 378 |
| SAMPLE FOR GET: | 378 |
| SAMPLE FOR DELETE: | 379 |
| MULTIMAP CACHE CONSUMER - FROM("HAZELCAST:MULTIMAP:FOO") | 379 |
| USAGE OF QUEUE | 381 |
| QUEUE PRODUCER - TO("HAZELCAST:QUEUE:FOO") | 381 |
| SAMPLE FOR ADD: | 381 |
| SAMPLE FOR PUT: | 381 |
| SAMPLE FOR POLL: | 381 |
| SAMPLE FOR PEEK: | 382 |
| SAMPLE FOR OFFER: | 382 |
| SAMPLE FOR REMOVEVALUE: | 382 |
| QUEUE CONSUMER - FROM("HAZELCAST:QUEUE:FOO") | 382 |
| USAGE OF LIST | 383 |
| LIST PRODUCER - TO("HAZELCAST:LIST:FOO") | 383 |
| SAMPLE FOR ADD: | 383 |
| SAMPLE FOR GET: | 383 |
| SAMPLE FOR SETVALUE: | 383 |
| SAMPLE FOR REMOVEVALUE: | 383 |
| LIST CONSUMER - FROM("HAZELCAST:LIST:FOO") | 384 |
| USAGE OF SEDA | 384 |
| SEDA PRODUCER - TO("HAZELCAST:SEDA:FOO") | 384 |
| SEDA CONSUMER - FROM("HAZELCAST:SEDA:FOO") | 385 |
| USAGE OF ATOMIC NUMBER | 386 |
| ATOMIC NUMBER PRODUCER - TO("HAZELCAST:ATOMICNUMBER:FOO") | 386 |
| SAMPLE FOR SET: | 387 |
| SAMPLE FOR GET: | 388 |
| SAMPLE FOR INCREMENT: | 388 |
| SAMPLE FOR DECREMENT: | 389 |
| SAMPLE FOR DESTROY | 389 |
| CLUSTER SUPPORT | 390 |
| INSTANCE CONSUMER - FROM("HAZELCAST:INSTANCE:FOO") | 390 |
| CHAPTER 46. HBASE | 393 |
| HBASE COMPONENT | 393 |
| APACHE HBASE OVERVIEW | 393 |
| CAMEL AND HBASE | 393 |
| CONFIGURING THE COMPONENT | 394 |
| HBASE PRODUCER | 394 |
| SUPPORTED URI OPTIONS ON PRODUCER | 395 |
| PUT OPERATIONS. | 396 |
| GET OPERATIONS. | 398 |
| DELETE OPERATIONS. | 398 |
| SCAN OPERATIONS. | 398 |
| HBASE CONSUMER | 400 |
| SUPPORTED URI OPTIONS ON CONSUMER | 401 |
| HBASE IDEMPOTENT REPOSITORY | 402 |
| HBASE MAPPING | 403 |
| HBASE HEADER MAPPING EXAMPLES | 403 |
| BODY MAPPING EXAMPLES | 404 |
| SEE ALSO | 405 |
| CHAPTER 47. HDFS | 406 |

| | |
|---|------------|
| HDFS COMPONENT | 406 |
| URI FORMAT | 406 |
| OPTIONS | 406 |
| KEYTYPE AND VALUETYPE | 408 |
| SPLITTING STRATEGY | 409 |
| CONTROLLING TO CLOSE FILE STREAM | 410 |
| USING THIS COMPONENT IN OSGI | 410 |
| CHAPTER 48. HL7 | 411 |
| HL7 COMPONENT | 411 |
| HL7 MLLP PROTOCOL | 411 |
| EXPOSING A HL7 LISTENER | 412 |
| HL7 MODEL USING JAVA.LANG.STRING | 414 |
| HL7V2 MODEL USING HAPI | 414 |
| MESSAGE HEADERS | 415 |
| OPTIONS | 416 |
| DEPENDENCIES | 416 |
| TERSER LANGUAGE (CAMEL 2.11) | 418 |
| HL7 VALIDATION PREDICATE (CAMEL 2.11) | 418 |
| HL7 ACKNOWLEDGEMENT EXPRESSION (CAMEL 2.11) | 419 |
| MORE SAMPLES | 419 |
| SAMPLE USING PLAIN STRING OBJECTS | 421 |
| CHAPTER 49. HTTP | 423 |
| HTTP COMPONENT | 423 |
| URI FORMAT | 423 |
| EXAMPLES | 423 |
| HTTPEndpoint OPTIONS | 425 |
| AUTHENTICATION AND PROXY | 427 |
| HTTPComponent OPTIONS | 428 |
| MESSAGE HEADERS | 429 |
| MESSAGE BODY | 430 |
| RESPONSE CODE | 430 |
| HTTPOperationFailedException | 431 |
| CALLING USING GET OR POST | 431 |
| HOW TO GET ACCESS TO HTTPServletRequest AND HTTPServletResponse | 431 |
| USING CLIENT TIMEOUT - SO_TIMEOUT | 432 |
| CONFIGURING A PROXY | 432 |
| USING PROXY SETTINGS OUTSIDE OF URI | 432 |
| CONFIGURING CHARSET | 432 |
| SAMPLE WITH SCHEDULED POLL | 433 |
| GETTING THE RESPONSE CODE | 433 |
| USING THROWExceptionOnFailure=FALSE TO GET ANY RESPONSE BACK | 433 |
| DISABLING COOKIES | 434 |
| ADVANCED USAGE | 434 |
| SETTING MAXConnectionsPerHost | 434 |
| USING PREEMPTIVE AUTHENTICATION | 435 |
| ACCEPTING SELF SIGNED CERTIFICATES FROM REMOTE SERVER | 435 |
| USING THE JSSE CONFIGURATION UTILITY | 435 |
| CONFIGURING APACHE HTTP CLIENT DIRECTLY | 436 |
| CHAPTER 50. HTTP4 | 438 |
| HTTP4 COMPONENT | 438 |
| URI FORMAT | 438 |

| | |
|--|------------|
| HTTPCOMPONENT OPTIONS | 439 |
| HTTPEndpoint OPTIONS | 440 |
| SETTING BASIC AUTHENTICATION AND PROXY | 445 |
| MESSAGE HEADERS | 446 |
| MESSAGE BODY | 447 |
| RESPONSE CODE | 447 |
| HTTPOPERATIONFAILEDEXCEPTION | 448 |
| CALLING USING GET OR POST | 448 |
| HOW TO GET ACCESS TO HTTPServletREQUEST AND HTTPServletRESPONSE | 448 |
| CONFIGURING URI TO CALL | 449 |
| CONFIGURING URI PARAMETERS | 449 |
| HOW TO SET THE HTTP METHOD (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) TO THE HTTP PRODUCER | 450 |
| USING CLIENT TIMEOUT - SO_TIMEOUT | 450 |
| CONFIGURING A PROXY | 450 |
| USING PROXY SETTINGS OUTSIDE OF URI | 451 |
| CONFIGURING CHARSET | 451 |
| SAMPLE WITH SCHEDULED POLL | 451 |
| URI PARAMETERS FROM THE ENDPOINT URI | 452 |
| URI PARAMETERS FROM THE MESSAGE | 452 |
| GETTING THE RESPONSE CODE | 452 |
| DISABLING COOKIES | 453 |
| ADVANCED USAGE | 453 |
| USING THE JSSE CONFIGURATION UTILITY | 453 |
| PROGRAMMATIC CONFIGURATION OF THE COMPONENT | 453 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 453 |
| CONFIGURING APACHE HTTP CLIENT DIRECTLY | 454 |
| USING HTTPS TO AUTHENTICATE GOTCHAS | 455 |
| USING DIFFERENT SSLCONTEXTPARAMETERS | 455 |
| CHAPTER 51. IBATIS | 457 |
| IBATIS | 457 |
| URI FORMAT | 457 |
| OPTIONS | 458 |
| MESSAGE HEADERS | 459 |
| MESSAGE BODY | 460 |
| SAMPLES | 460 |
| USING STATEMENTTYPE FOR BETTER CONTROL OF IBATIS | 461 |
| SCHEDULED POLLING EXAMPLE | 461 |
| USING ONCONSUME | 462 |
| CHAPTER 52. IRC | 463 |
| IRC COMPONENT | 463 |
| URI FORMAT | 463 |
| OPTIONS | 463 |
| USING THE JSSE CONFIGURATION UTILITY | 465 |
| PROGRAMMATIC CONFIGURATION OF THE ENDPOINT | 465 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 466 |
| USING THE LEGACY BASIC CONFIGURATION OPTIONS | 466 |
| USING KEYS | 467 |
| CHAPTER 53. JASYPT | 468 |
| JASYPT COMPONENT | 468 |
| TOOLING | 468 |

| | |
|--|------------|
| TOOLING DEPENDENCIES FOR CAMEL 2.5 AND 2.6 | 469 |
| TOOLING DEPENDENCIES FOR CAMEL 2.7 OR BETTER | 470 |
| URI OPTIONS | 470 |
| PROTECTING THE MASTER PASSWORD | 470 |
| EXAMPLE WITH JAVA DSL | 471 |
| EXAMPLE WITH SPRING XML | 472 |
| SEE ALSO | 473 |
| CHAPTER 54. JAVASPACE | 474 |
| JAVASPACE COMPONENT | 474 |
| URI FORMAT | 474 |
| OPTIONS | 474 |
| SENDING AND RECEIVING ENTRIES | 475 |
| SENDING AND RECEIVING SERIALIZABLE OBJECTS | 475 |
| USING JAVASPACE AS A REMOTE INVOCATION TRANSPORT | 475 |
| CHAPTER 55. JBI | 476 |
| 55.1. JBI COMPONENT | 476 |
| 55.2. JBI SERVICE UNIT ARCHETYPE | 481 |
| CHAPTER 56. JCLOUDS | 483 |
| JCLOUDS COMPONENT | 483 |
| CONFIGURING THE COMPONENT | 483 |
| URI FORMAT | 484 |
| BLOBSTORE URI OPTIONS | 485 |
| MESSAGE HEADERS FOR BLOBSTORE | 486 |
| BLOBSTORE USAGE SAMPLES | 486 |
| EXAMPLE 1: PUTTING TO THE BLOB | 486 |
| EXAMPLE 2: GETTING/READING FROM A BLOB | 487 |
| EXAMPLE 3: CONSUMING A BLOB | 487 |
| COMPUTE SERVICE URI OPTIONS | 488 |
| COMPUTE USAGE SAMPLES | 489 |
| EXAMPLE 1: LISTING THE AVAILABLE IMAGES. | 489 |
| EXAMPLE 2: CREATE A NEW NODE. | 489 |
| EXAMPLE 3: RUN A SHELL SCRIPT ON RUNNING NODE. | 490 |
| SEE ALSO | 491 |
| CHAPTER 57. JCR | 492 |
| JCR COMPONENT | 492 |
| URI FORMAT | 492 |
| USAGE | 492 |
| PRODUCER | 492 |
| CONSUMER | 493 |
| EXAMPLE | 494 |
| CHAPTER 58. JDBC | 495 |
| JDBC COMPONENT | 495 |
| URI FORMAT | 495 |
| OPTIONS | 495 |
| RESULT | 498 |
| MESSAGE HEADERS | 498 |
| GENERATED KEYS | 499 |
| USING NAMED PARAMETERS | 499 |
| SAMPLES | 500 |

| | |
|---|------------|
| SAMPLE - POLLING THE DATABASE EVERY MINUTE | 501 |
| CHAPTER 59. JETTY | 503 |
| JETTY COMPONENT | 503 |
| URI FORMAT | 503 |
| OPTIONS | 503 |
| MESSAGE HEADERS | 509 |
| USAGE | 509 |
| COMPONENT OPTIONS | 509 |
| PRODUCER EXAMPLE | 512 |
| CONSUMER EXAMPLE | 512 |
| SESSION SUPPORT | 513 |
| USING THE JSSE CONFIGURATION UTILITY | 514 |
| PROGRAMMATIC CONFIGURATION OF THE COMPONENT | 514 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 515 |
| CONFIGURING JETTY DIRECTLY | 515 |
| CONFIGURING GENERAL SSL PROPERTIES | 517 |
| HOW TO OBTAIN REFERENCE TO THE X509CERTIFICATE | 518 |
| CONFIGURING GENERAL HTTP PROPERTIES | 518 |
| DEFAULT BEHAVIOR FOR RETURNING HTTP STATUS CODES | 519 |
| CUSTOMIZING HTTPBINDING | 519 |
| JETTY HANDLERS AND SECURITY CONFIGURATION | 520 |
| HOW TO RETURN A CUSTOM HTTP 500 REPLY MESSAGE | 521 |
| MULTI-PART FORM SUPPORT | 522 |
| JETTY JMX SUPPORT | 522 |
| CHAPTER 60. JING | 524 |
| JING COMPONENT | 524 |
| URI FORMAT | 524 |
| OPTIONS | 524 |
| EXAMPLE | 525 |
| CHAPTER 61. JMS | 526 |
| JMS COMPONENT | 526 |
| URI FORMAT | 526 |
| USING ACTIVEMQ | 527 |
| TRANSACTIONS AND CACHE LEVELS | 527 |
| DURABLE SUBSCRIPTIONS | 528 |
| MESSAGE HEADER MAPPING | 528 |
| OPTIONS | 529 |
| MOST COMMONLY USED OPTIONS | 529 |
| ALL THE OTHER OPTIONS | 533 |
| MESSAGE MAPPING BETWEEN JMS AND CAMEL | 547 |
| DISABLING AUTO-MAPPING OF JMS MESSAGES | 548 |
| USING A CUSTOM MESSAGECONVERTER | 548 |
| CONTROLLING THE MAPPING STRATEGY SELECTED | 548 |
| MESSAGE FORMAT WHEN SENDING | 549 |
| MESSAGE FORMAT WHEN RECEIVING | 550 |
| ABOUT USING CAMEL TO SEND AND RECEIVE MESSAGES AND JMSREPLYTO | 551 |
| JMSPRODUCER | 551 |
| JMSCONSUMER | 552 |
| REUSE ENDPOINT AND SEND TO DIFFERENT DESTINATIONS COMPUTED AT RUNTIME | 553 |
| CONFIGURING DIFFERENT JMS PROVIDERS | 554 |
| USING JNDI TO FIND THE CONNECTIONFACTORY | 555 |

| | |
|---|------------|
| CONCURRENT CONSUMING | 555 |
| REQUEST-REPLY OVER JMS | 556 |
| REQUEST-REPLY OVER JMS AND USING A SHARED FIXED REPLY QUEUE | 560 |
| REQUEST-REPLY OVER JMS AND USING AN EXCLUSIVE FIXED REPLY QUEUE | 561 |
| SYNCHRONIZING CLOCKS BETWEEN SENDERS AND RECEIVERS | 561 |
| ABOUT TIME TO LIVE | 562 |
| ENABLING TRANSACTED CONSUMPTION | 562 |
| USING JMSREPLYTO FOR LATE REPLIES | 564 |
| USING A REQUEST TIMEOUT | 565 |
| SAMPLES | 565 |
| RECEIVING FROM JMS | 565 |
| SENDING TO A JMS | 565 |
| USING ANNOTATIONS | 566 |
| SPRING DSL SAMPLE | 566 |
| OTHER SAMPLES | 566 |
| USING JMS AS A DEAD LETTER QUEUE STORING EXCHANGE | 566 |
| USING JMS AS A DEAD LETTER CHANNEL STORING ERROR ONLY | 567 |
| SENDING AN INONLY MESSAGE AND KEEPING THE JMSREPLYTO HEADER | 567 |
| SETTING JMS PROVIDER OPTIONS ON THE DESTINATION | 568 |
| CHAPTER 62. JMX | 570 |
| JMX COMPONENT | 570 |
| URI FORMAT | 570 |
| URI OPTIONS | 570 |
| OBJECTNAME CONSTRUCTION | 572 |
| DOMAIN WITH NAME PROPERTY | 572 |
| DOMAIN WITH HASHTABLE | 572 |
| EXAMPLE | 573 |
| FULL EXAMPLE | 573 |
| MONITOR TYPE CONSUMER | 573 |
| EXAMPLE | 573 |
| URI OPTIONS FOR MONITOR TYPE | 574 |
| CHAPTER 63. JPA | 576 |
| JPA COMPONENT | 576 |
| SENDING TO THE ENDPOINT | 576 |
| CONSUMING FROM THE ENDPOINT | 576 |
| URI FORMAT | 576 |
| OPTIONS | 577 |
| MESSAGE HEADERS | 580 |
| CONFIGURING ENTITYMANAGERFACTORY | 580 |
| CONFIGURING TRANSACTIONMANAGER | 580 |
| USING A CONSUMER WITH A NAMED QUERY | 581 |
| USING A CONSUMER WITH A QUERY | 581 |
| USING A CONSUMER WITH A NATIVE QUERY | 581 |
| EXAMPLE | 582 |
| USING THE JPA BASED IDEMPOTENT REPOSITORY | 582 |
| CHAPTER 64. JSCH | 584 |
| JSCH | 584 |
| URI FORMAT | 584 |
| OPTIONS | 584 |
| LIMITATIONS | 585 |

| | |
|--|------------|
| CHAPTER 65. JT400 | 586 |
| JT/400 COMPONENT | 586 |
| URI FORMAT | 586 |
| URI OPTIONS | 586 |
| USAGE | 588 |
| CONNECTION POOL | 588 |
| REMOTE PROGRAM CALL (CAMEL 2.7) | 588 |
| EXAMPLE | 589 |
| REMOTE PROGRAM CALL EXAMPLE (CAMEL 2.7) | 589 |
| WRITING TO KEYED DATA QUEUES | 589 |
| READING FROM KEYED DATA QUEUES | 590 |
| CHAPTER 66. KESTREL | 591 |
| KESTREL COMPONENT | 591 |
| URI FORMAT | 591 |
| OPTIONS | 592 |
| CONFIGURING THE KESTREL COMPONENT USING SPRING XML | 592 |
| USAGE EXAMPLES | 593 |
| EXAMPLE 1: CONSUMING | 594 |
| EXAMPLE 2: PRODUCING | 594 |
| EXAMPLE 3: SPRING XML CONFIGURATION | 594 |
| DEPENDENCIES | 595 |
| SPYMEMCACHED | 595 |
| CHAPTER 67. KRATI | 597 |
| KRATI COMPONENT | 597 |
| URI FORMAT | 597 |
| KRATI URI OPTIONS | 598 |
| MESSAGE HEADERS FOR DATASTORE | 599 |
| USAGE SAMPLES | 599 |
| EXAMPLE 1: PUTTING TO THE DATASTORE. | 599 |
| EXAMPLE 2: GETTING/READING FROM A DATASTORE | 600 |
| EXAMPLE 3: CONSUMING FROM A DATASTORE | 600 |
| IDEMPOTENT REPOSITORY | 600 |
| SEE ALSO | 601 |
| CHAPTER 68. LANGUAGE | 602 |
| LANGUAGE | 602 |
| URI FORMAT | 602 |
| URI OPTIONS | 602 |
| MESSAGE HEADERS | 603 |
| EXAMPLES | 603 |
| LOADING SCRIPTS FROM RESOURCES | 604 |
| CHAPTER 69. LDAP | 606 |
| LDAP COMPONENT | 606 |
| URI FORMAT | 606 |
| OPTIONS | 606 |
| RESULT | 607 |
| DIRCONTEXT | 607 |
| SAMPLES | 607 |
| BINDING USING CREDENTIALS | 608 |
| CHAPTER 70. LEVELDB | 610 |

| | |
|---|------------|
| LEVELDB | 610 |
| USING LEVELDBAGGREGATIONREPOSITORY | 610 |
| WHAT IS PRESERVED WHEN PERSISTING | 612 |
| RECOVERY | 612 |
| USING LEVELDBAGGREGATIONREPOSITORY IN JAVA DSL | 613 |
| USING LEVELDBAGGREGATIONREPOSITORY IN SPRING XML | 614 |
| DEPENDENCIES | 614 |
| CHAPTER 71. LIST | 616 |
| LIST COMPONENT | 616 |
| URI FORMAT | 616 |
| SAMPLE | 616 |
| CHAPTER 72. LOG | 618 |
| LOG COMPONENT | 618 |
| URI FORMAT | 618 |
| OPTIONS | 618 |
| FORMATTING | 619 |
| REGULAR LOGGER SAMPLE | 621 |
| REGULAR LOGGER WITH FORMATTER SAMPLE | 622 |
| THROUGHPUT LOGGER WITH GROUPSIZE SAMPLE | 622 |
| THROUGHPUT LOGGER WITH GROUPINTERVAL SAMPLE | 622 |
| FULL CUSTOMIZATION OF THE LOGGING OUTPUT | 622 |
| CHAPTER 73. LUCENE | 626 |
| LUCENE (INDEXER AND SEARCH) COMPONENT | 626 |
| URI FORMAT | 626 |
| INSERT OPTIONS | 626 |
| QUERY OPTIONS | 627 |
| MESSAGE HEADERS | 628 |
| LUCENE PRODUCERS | 628 |
| LUCENE PROCESSOR | 628 |
| EXAMPLE 1: CREATING A LUCENE INDEX | 628 |
| EXAMPLE 2: LOADING PROPERTIES INTO THE JNDI REGISTRY IN THE CAMEL CONTEXT | 628 |
| EXAMPLE 2: PERFORMING SEARCHES USING A QUERY PRODUCER | 629 |
| EXAMPLE 3: PERFORMING SEARCHES USING A QUERY PROCESSOR | 629 |
| CHAPTER 74. MAIL | 631 |
| MAIL COMPONENT | 631 |
| URI FORMAT | 632 |
| SAMPLE ENDPOINTS | 632 |
| DEFAULT PORTS | 633 |
| OPTIONS | 633 |
| SSL SUPPORT | 639 |
| USING THE JSSE CONFIGURATION UTILITY | 639 |
| PROGRAMMATIC CONFIGURATION OF THE ENDPOINT | 639 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 639 |
| CONFIGURING JAVAMAIL DIRECTLY | 640 |
| MAIL MESSAGE CONTENT | 640 |
| HEADERS TAKE PRECEDENCE OVER PRE-CONFIGURED RECIPIENTS | 641 |
| MULTIPLE RECIPIENTS FOR EASIER CONFIGURATION | 641 |
| SETTING SENDER NAME AND EMAIL | 641 |
| SUN JAVAMAIL | 642 |
| SAMPLES | 642 |

| | |
|---|------------|
| SENDING MAIL WITH ATTACHMENT SAMPLE | 643 |
| SSL SAMPLE | 644 |
| CONSUMING MAILS WITH ATTACHMENT SAMPLE | 644 |
| HOW TO SPLIT A MAIL MESSAGE WITH ATTACHMENTS | 645 |
| USING CUSTOM SEARCHTERM | 646 |
| CHAPTER 75. MASTER COMPONENT | 650 |
| DEPENDENCIES | 650 |
| URI FORMAT | 650 |
| URI OPTIONS | 650 |
| HOW TO USE THE MASTER COMPONENT | 651 |
| EXAMPLE OF A MASTER-SLAVE CLUSTER POLLING A JMS ACTIVEMQ BROKER | 652 |
| STEPS TO CREATE A CLUSTER THAT POLLS MESSAGES FROM AN ACTIVEMQ BROKER | 652 |
| OSGI BUNDLE PLUG-IN CONFIGURATION | 656 |
| CHAPTER 76. MINA [DEPRECATED] | 658 |
| MINA COMPONENT | 658 |
| URI FORMAT | 658 |
| OPTIONS | 659 |
| USING A CUSTOM CODEC | 662 |
| SAMPLE WITH SYNC=FALSE | 662 |
| SAMPLE WITH SYNC=TRUE | 662 |
| SAMPLE WITH SPRING DSL | 663 |
| CONFIGURING MINA ENDPOINTS USING SPRING BEAN STYLE | 663 |
| CLOSING SESSION WHEN COMPLETE | 664 |
| GET THE IOSESSION FOR MESSAGE | 665 |
| CONFIGURING MINA FILTERS | 665 |
| CHAPTER 77. MINA2 [DEPRECATED] | 668 |
| MINA 2 COMPONENT | 668 |
| URI FORMAT | 668 |
| OPTIONS | 669 |
| USING A CUSTOM CODEC | 673 |
| SAMPLE WITH SYNC=FALSE | 673 |
| SAMPLE WITH SYNC=TRUE | 673 |
| SAMPLE WITH SPRING DSL | 674 |
| CLOSING SESSION WHEN COMPLETE | 674 |
| GET THE IOSESSION FOR MESSAGE | 675 |
| CONFIGURING MINA FILTERS | 675 |
| CHAPTER 78. MOCK | 676 |
| MOCK COMPONENT | 676 |
| URI FORMAT | 677 |
| OPTIONS | 677 |
| SIMPLE EXAMPLE | 677 |
| USING ASSERTPERIOD | 678 |
| SETTING EXPECTATIONS | 678 |
| ADDING EXPECTATIONS TO SPECIFIC MESSAGES | 679 |
| MOCKING EXISTING ENDPOINTS | 680 |
| MOCKING EXISTING ENDPOINTS USING THE CAMEL-TEST COMPONENT | 682 |
| MOCKING EXISTING ENDPOINTS WITH XML DSL | 683 |
| MOCKING ENDPOINTS AND SKIP SENDING TO ORIGINAL ENDPOINT | 685 |
| LIMITING THE NUMBER OF MESSAGES TO KEEP | 686 |
| TESTING WITH ARRIVAL TIMES | 687 |

| | |
|---|------------|
| CHAPTER 79. MONGODB | 689 |
| CAMEL MONGODB COMPONENT | 689 |
| URI FORMAT | 689 |
| ENDPOINT OPTIONS | 690 |
| MONGODB OPERATIONS - PRODUCER ENDPOINTS | 698 |
| QUERY OPERATIONS | 698 |
| FINDBYID | 698 |
| FINDONEBYQUERY | 699 |
| FINDALL | 699 |
| SPECIFYING A FIELDS FILTER | 702 |
| CREATE/UPDATE OPERATIONS | 703 |
| INSERT | 703 |
| SAVE | 704 |
| UPDATE | 705 |
| DELETE OPERATIONS | 706 |
| REMOVE | 706 |
| OTHER OPERATIONS | 707 |
| COUNT | 707 |
| GETDBSTATS | 707 |
| GETCOLSTATS | 708 |
| DYNAMIC OPERATIONS | 709 |
| TAILABLE CURSOR CONSUMER | 709 |
| HOW THE TAILABLE CURSOR CONSUMER WORKS | 710 |
| PERSISTENT TAIL TRACKING | 711 |
| ENABLING PERSISTENT TAIL TRACKING | 712 |
| TYPE CONVERSIONS | 713 |
| SEE ALSO | 714 |
| CHAPTER 80. MQTT | 715 |
| MQTT COMPONENT | 715 |
| URI FORMAT | 715 |
| OPTIONS | 715 |
| SAMPLES | 718 |
| CHAPTER 81. MSV | 719 |
| MSV COMPONENT | 719 |
| URI FORMAT | 719 |
| OPTIONS | 719 |
| EXAMPLE | 719 |
| CHAPTER 82. MUSTACHE | 721 |
| MUSTACHE | 721 |
| URI FORMAT | 721 |
| OPTIONS | 721 |
| MUSTACHE CONTEXT | 722 |
| DYNAMIC TEMPLATES | 722 |
| SAMPLES | 723 |
| THE EMAIL SAMPLE | 724 |
| CHAPTER 83. MVEL COMPONENT | 725 |
| MVEL COMPONENT | 725 |
| URI FORMAT | 725 |
| OPTIONS | 725 |
| MESSAGE HEADERS | 726 |

| | |
|---|------------|
| MVEL CONTEXT | 726 |
| HOT RELOADING | 727 |
| DYNAMIC TEMPLATES | 727 |
| SAMPLES | 728 |
| CHAPTER 84. MYBATIS | 729 |
| MYBATIS | 729 |
| URI FORMAT | 729 |
| OPTIONS | 729 |
| MESSAGE HEADERS | 731 |
| MESSAGE BODY | 732 |
| SAMPLES | 732 |
| USING STATEMENTTYPE FOR BETTER CONTROL OF MYBATIS | 732 |
| USING INSERTLIST STATEMENTTYPE | 733 |
| USING UPDATELIST STATEMENTTYPE | 734 |
| USING DELETEDLIST STATEMENTTYPE | 734 |
| NOTICE ON INSERTLIST, UPDATELIST AND DELETEDLIST STATEMENTTYPES | 735 |
| SCHEDULED POLLING EXAMPLE | 735 |
| USING ONCONSUME | 736 |
| PARTICIPATING IN TRANSACTIONS | 736 |
| CHAPTER 85. NAGIOS | 739 |
| NAGIOS | 739 |
| URI FORMAT | 739 |
| OPTIONS | 739 |
| HEADERS | 740 |
| SENDING MESSAGE EXAMPLES | 740 |
| USING NAGIOSEVENTNOTIFER | 741 |
| CHAPTER 86. NETTY | 742 |
| NETTY COMPONENT | 742 |
| URI FORMAT | 742 |
| OPTIONS | 743 |
| REGISTRY BASED OPTIONS | 749 |
| USING NON SHAREABLE ENCODERS OR DECODERS | 750 |
| SENDING MESSAGES TO/FROM A NETTY ENDPOINT | 751 |
| NETTY PRODUCER | 751 |
| NETTY CONSUMER | 751 |
| A UDP NETTY ENDPOINT USING REQUEST-REPLY AND SERIALIZED OBJECT PAYLOAD | 752 |
| A TCP BASED NETTY CONSUMER ENDPOINT USING ONE-WAY COMMUNICATION | 752 |
| AN SSL/TCP BASED NETTY CONSUMER ENDPOINT USING REQUEST-REPLY COMMUNICATION | 752 |
| USING THE JSSE CONFIGURATION UTILITY | 752 |
| PROGRAMMATIC CONFIGURATION OF THE COMPONENT | 753 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 753 |
| USING BASIC SSL/TLS CONFIGURATION ON THE JETTY COMPONENT | 753 |
| GETTING ACCESS TO SSLSESSION AND THE CLIENT CERTIFICATE | 754 |
| USING MULTIPLE CODECS | 755 |
| CLOSING CHANNEL WHEN COMPLETE | 757 |
| ADDING CUSTOM CHANNEL PIPELINE FACTORIES TO GAIN COMPLETE CONTROL OVER A CREATED PIPELINE | 757 |
| REUSING NETTY BOSS AND WORKER THREAD POOLS | 759 |
| SEE ALSO | 761 |
| CHAPTER 87. NETTY HTTP | 762 |

| | |
|---|------------|
| NETTY HTTP COMPONENT | 762 |
| URI FORMAT | 763 |
| HTTP OPTIONS | 763 |
| MESSAGE HEADERS | 768 |
| ACCESS TO NETTY TYPES | 770 |
| EXAMPLES | 770 |
| HOW DO I LET NETTY MATCH WILDCARDS | 770 |
| USING MULTIPLE ROUTES WITH SAME PORT | 771 |
| REUSING SAME SERVER BOOTSTRAP CONFIGURATION WITH MULTIPLE ROUTES | 772 |
| REUSING SAME SERVER BOOTSTRAP CONFIGURATION WITH MULTIPLE ROUTES ACROSS MULTIPLE BUNDLES IN OSGI CONTAINER | 773 |
| USING HTTP BASIC AUTHENTICATION | 773 |
| SPECIFYING ACL ON WEB RESOURCES | 774 |
| CHAPTER 88. NMR | 777 |
| NMR COMPONENT | 777 |
| INSTALLING IN APACHE SERVICEMIX | 777 |
| INSTALLING IN PLAIN APACHE KARAF | 777 |
| CONFIGURATION | 777 |
| NMR CONSUMER AND PRODUCER ENDPOINTS | 778 |
| URI FORMAT | 778 |
| URI OPTIONS | 779 |
| EXAMPLES | 779 |
| USING STREAM BODIES | 780 |
| TESTING | 780 |
| CHAPTER 89. PAX-LOGGING | 782 |
| PAXLOGGING COMPONENT | 782 |
| DEPENDENCIES | 782 |
| URI FORMAT | 782 |
| URI OPTIONS | 783 |
| MESSAGE HEADERS | 783 |
| MESSAGE BODY | 783 |
| EXAMPLE USAGE | 783 |
| CHAPTER 90. PRINTER | 785 |
| PRINTER COMPONENT | 785 |
| URI FORMAT | 785 |
| OPTIONS | 786 |
| PRINTER PRODUCER | 786 |
| EXAMPLE 1: PRINTING TEXT BASED PAYLOADS ON A DEFAULT PRINTER USING LETTER STATIONARY AND ONE-SIDED MODE | 787 |
| EXAMPLE 2: PRINTING GIF BASED PAYLOADS ON A REMOTE PRINTER USING A4 STATIONARY AND ONE-SIDED MODE | 787 |
| EXAMPLE 3: PRINTING JPEG BASED PAYLOADS ON A REMOTE PRINTER USING JAPANESE POSTCARD STATIONARY AND ONE-SIDED MODE | 787 |
| CHAPTER 91. PROPERTIES | 788 |
| PROPERTIES COMPONENT | 788 |
| URI FORMAT | 788 |
| OPTIONS | 788 |
| SEE ALSO | 789 |
| CHAPTER 92. QUARTZ | 790 |
| QUARTZ COMPONENT | 790 |

| | |
|--|------------|
| URI FORMAT | 790 |
| OPTIONS | 790 |
| CONFIGURING QUARTZ.PROPERTIES FILE | 792 |
| STARTING THE QUARTZ SCHEDULER | 793 |
| CLUSTERING | 793 |
| MESSAGE HEADERS | 794 |
| USING CRON TRIGGERS | 794 |
| SPECIFYING TIME ZONE | 794 |
| CHAPTER 93. QUARTZ2 | 796 |
| QUARTZ2 COMPONENT | 796 |
| URI FORMAT | 796 |
| OPTIONS | 797 |
| CONFIGURING QUARTZ.PROPERTIES FILE | 798 |
| STARTING THE QUARTZ SCHEDULER | 799 |
| CLUSTERING | 800 |
| MESSAGE HEADERS | 800 |
| USING CRON TRIGGERS | 800 |
| SPECIFYING TIME ZONE | 801 |
| USING QUARTZSCHEDULEDPOLLCONSUMERSCHEDULER | 802 |
| CHAPTER 94. QUICKFIX | 804 |
| QUICKFIX/J COMPONENT | 804 |
| URI FORMAT | 804 |
| ENDPOINTS | 805 |
| EXCHANGE FORMAT | 805 |
| QUICKFIX/J CONFIGURATION EXTENSIONS | 806 |
| COMMUNICATION CONNECTORS | 806 |
| LOGGING | 807 |
| MESSAGE STORE | 807 |
| MESSAGE FACTORY | 807 |
| JMX | 808 |
| OTHER DEFAULTS | 808 |
| MINIMAL INITIATOR CONFIGURATION EXAMPLE | 808 |
| USING THE INOUT MESSAGE EXCHANGE PATTERN | 808 |
| IMPLEMENTING INOUT EXCHANGES FOR CONSUMERS | 808 |
| IMPLEMENTING INOUT EXCHANGES FOR PRODUCERS | 809 |
| EXAMPLE | 810 |
| SPRING CONFIGURATION | 810 |
| EXCEPTION HANDLING | 812 |
| FIX SEQUENCE NUMBER MANAGEMENT | 813 |
| ROUTE EXAMPLES | 813 |
| QUICKFIX/J COMPONENT PRIOR TO CAMEL 2.5 | 814 |
| URI FORMAT | 814 |
| EXCHANGE DATA FORMAT | 815 |
| SAMPLES | 815 |
| CHAPTER 95. RABBITMQ | 817 |
| RABBITMQ COMPONENT | 817 |
| URI FORMAT | 817 |
| OPTIONS | 817 |
| HEADERS | 818 |
| MESSAGE BODY | 820 |
| SAMPLES | 820 |

| | |
|--|------------|
| CHAPTER 96. REF | 822 |
| REF COMPONENT | 822 |
| URI FORMAT | 822 |
| RUNTIME LOOKUP | 822 |
| SAMPLE | 822 |
| CHAPTER 97. RESTLET | 824 |
| RESTLET COMPONENT | 824 |
| URI FORMAT | 824 |
| OPTIONS | 824 |
| COMPONENT OPTIONS | 825 |
| MESSAGE HEADERS | 827 |
| MESSAGE BODY | 829 |
| RESTLET ENDPOINT WITH AUTHENTICATION | 829 |
| SINGLE RESTLET ENDPOINT TO SERVICE MULTIPLE METHODS AND URI TEMPLATES (2.0 OR LATER) | 831 |
| USING RESTLET API TO POPULATE RESPONSE | 832 |
| CONFIGURING MAX THREADS ON COMPONENT | 832 |
| USING THE RESTLET SERVLET WITHIN A WEBAPP | 833 |
| CHAPTER 98. RMI | 835 |
| RMI COMPONENT | 835 |
| URI FORMAT | 835 |
| OPTIONS | 835 |
| USING | 836 |
| CHAPTER 99. ROUTEBOX | 837 |
| ROUTEBOX COMPONENT | 837 |
| THE NEED FOR A CAMEL ROUTEBOX ENDPOINT | 838 |
| URI FORMAT | 839 |
| OPTIONS | 839 |
| SENDING/RECEIVING MESSAGES TO/FROM THE ROUTEBOX | 841 |
| STEP 1: LOADING INNER ROUTE DETAILS INTO THE REGISTRY | 841 |
| STEP 2: OPTIONALLY USING A DISPATCH STRATEGY INSTEAD OF A DISPATCH MAP | 842 |
| STEP 2: LAUNCHING A ROUTEBOX CONSUMER | 843 |
| STEP 3: USING A ROUTEBOX PRODUCER | 843 |
| CHAPTER 100. RSS | 845 |
| RSS COMPONENT | 845 |
| URI FORMAT | 845 |
| OPTIONS | 845 |
| EXCHANGE DATA TYPES | 847 |
| MESSAGE HEADERS | 847 |
| RSS DATAFORMAT | 847 |
| FILTERING ENTRIES | 848 |
| SEE ALSO | 849 |
| CHAPTER 101. SALESFORCE | 850 |
| SALESFORCE COMPONENT | 850 |
| URI FORMAT | 850 |
| SUPPORTED SALESFORCE APIS | 850 |
| REST API | 851 |
| REST BULK API | 853 |
| REST STREAMING API | 855 |

| | |
|--|------------|
| CAMEL SALESFORCE MAVEN PLUGIN | 856 |
| USAGE | 856 |
| CHAPTER 102. SAP COMPONENT | 858 |
| 102.1. OVERVIEW | 858 |
| 102.2. MESSAGE BODY | 863 |
| 102.3. CONFIGURATION OVERVIEW | 870 |
| 102.4. DESTINATION CONFIGURATION | 871 |
| 102.5. SERVER CONFIGURATION | 875 |
| 102.6. REPOSITORY CONFIGURATION | 878 |
| 102.7. TRANSACTION SUPPORT | 885 |
| 102.8. XML SERIALIZATION | 886 |
| 102.9. EXAMPLE 1: READING DATA FROM SAP | 889 |
| 102.10. EXAMPLE 2: WRITING DATA TO SAP | 892 |
| 102.11. EXAMPLE 3: HANDLING REQUESTS FROM SAP | 893 |
| CHAPTER 103. SAP NETWEAVER | 898 |
| SAP NETWEAVER GATEWAY COMPONENT | 898 |
| URI FORMAT | 898 |
| PREREQUISITES | 898 |
| COMPONENT AND ENDPOINT OPTIONS | 899 |
| MESSAGE HEADERS | 899 |
| EXAMPLES | 900 |
| CHAPTER 104. SEDA | 903 |
| SEDA COMPONENT | 903 |
| URI FORMAT | 903 |
| OPTIONS | 904 |
| CHOOSING BLOCKINGQUEUE IMPLEMENTATION | 907 |
| USE OF REQUEST REPLY | 908 |
| CONCURRENT CONSUMERS | 908 |
| DIFFERENCE BETWEEN THREAD POOLS AND CONCURRENT CONSUMERS | 909 |
| THREAD POOLS | 909 |
| SAMPLE | 909 |
| USING MULTIPLECONSUMERS | 910 |
| EXTRACTING QUEUE INFORMATION. | 911 |
| CHAPTER 105. SERVLET | 912 |
| SERVLET COMPONENT | 912 |
| URI FORMAT | 912 |
| OPTIONS | 912 |
| MESSAGE HEADERS | 913 |
| USAGE | 913 |
| PUTTING CAMEL JARS IN THE APP SERVER BOOT CLASSPATH | 913 |
| SAMPLE | 914 |
| SAMPLE WHEN USING SPRING 3.X | 916 |
| SAMPLE WHEN USING SPRING 2.X | 917 |
| SAMPLE WHEN USING OSGI | 917 |
| CHAPTER 106. SERVLETLISTENER COMPONENT | 921 |
| SERVLETLISTENER COMPONENT | 921 |
| USING | 921 |
| OPTIONS | 923 |
| EXAMPLES | 925 |

| | |
|---|------------|
| CONFIGURING ROUTES | 925 |
| USING A ROUTEBUILDER CLASS | 925 |
| USING PACKAGE SCANNING | 926 |
| USING A XML FILE | 926 |
| CONFIGURING PROPERT PLACEHOLDERS | 927 |
| CONFIGURING JMX | 927 |
| JNDI OR SIMPLE AS CAMEL REGISTRY | 928 |
| USING CUSTOM CAMELCONTEXTLIFECYCLE | 928 |
| SEE ALSO | 930 |
| CHAPTER 107. SHIRO SECURITY | 931 |
| SHIRO SECURITY COMPONENT | 931 |
| SHIRO SECURITY BASICS | 931 |
| INSTANTIATING A SHIROSECURITYPOLICY OBJECT | 932 |
| SHIROSECURITYPOLICY OPTIONS | 932 |
| APPLYING SHIRO AUTHENTICATION ON A CAMEL ROUTE | 934 |
| APPLYING SHIRO AUTHORIZATION ON A CAMEL ROUTE | 935 |
| CREATING A SHIROSECURITYTOKEN AND INJECTING IT INTO A MESSAGE EXCHANGE | 935 |
| SENDING MESSAGES TO ROUTES SECURED BY A SHIROSECURITYPOLICY | 936 |
| SENDING MESSAGES TO ROUTES SECURED BY A SHIROSECURITYPOLICY (MUCH EASIER FROM CAMEL 2.12 ONWARDS) | 936 |
| USING SHIROSECURITYTOKEN | 937 |
| CHAPTER 108. SIP | 938 |
| SIP COMPONENT | 938 |
| URI FORMAT | 939 |
| OPTIONS | 939 |
| REGISTRY BASED OPTIONS | 942 |
| SENDING MESSAGES TO/FROM A SIP ENDPOINT | 943 |
| CREATING A CAMEL SIP PUBLISHER | 943 |
| CREATING A CAMEL SIP SUBSCRIBER | 944 |
| CHAPTER 109. SJMS | 946 |
| SJMS COMPONENT | 946 |
| URI FORMAT | 948 |
| COMPONENT OPTIONS AND CONFIGURATIONS | 949 |
| PRODUCER CONFIGURATION OPTIONS | 950 |
| PRODUCER USAGE | 951 |
| INONLY PRODUCER - (DEFAULT) | 951 |
| INOUT PRODUCER | 952 |
| CONSUMERS CONFIGURATION OPTIONS | 952 |
| CONSUMER USAGE | 953 |
| INONLY CONSUMER - (DEFAULT) | 953 |
| INOUT CONSUMER | 954 |
| ADVANCED USAGE NOTES | 954 |
| PLUGABLE CONNECTION RESOURCE MANAGEMENT | 954 |
| SESSION, CONSUMER, & PRODUCER POOLING & CACHING MANAGEMENT | 956 |
| BATCH MESSAGE SUPPORT | 956 |
| CUSTOMIZABLE TRANSACTION COMMIT STRATEGIES (LOCAL JMS TRANSACTIONS ONLY) | 957 |
| TRANSACTIONED BATCH CONSUMERS & PRODUCERS | 957 |
| ADDITIONAL NOTES | 958 |
| MESSAGE HEADER FORMAT | 958 |
| MESSAGE CONTENT | 959 |
| CLUSTERING | 959 |

| | |
|---|-------------|
| TRANSACTION SUPPORT | 960 |
| DOES SPRINGLESS MEAN I CAN'T USE SPRING? | 960 |
| CHAPTER 110. SMPP | 962 |
| SMPP COMPONENT | 962 |
| URI FORMAT | 962 |
| URI OPTIONS | 962 |
| PRODUCER MESSAGE HEADERS | 969 |
| CONSUMER MESSAGE HEADERS | 976 |
| EXCEPTION HANDLING | 981 |
| SAMPLES | 981 |
| DEBUG LOGGING | 982 |
| CHAPTER 111. SNMP | 983 |
| SNMP COMPONENT | 983 |
| URI FORMAT | 983 |
| OPTIONS | 983 |
| THE RESULT OF A POLL | 984 |
| EXAMPLES | 985 |
| CHAPTER 112. SOLR | 986 |
| SOLR COMPONENT | 986 |
| URI FORMAT | 986 |
| ENDPOINT OPTIONS | 986 |
| MESSAGE OPERATIONS | 987 |
| EXAMPLE | 988 |
| QUERYING SOLR | 989 |
| CHAPTER 113. SPRINGBATCH | 991 |
| SPRING BATCH COMPONENT | 991 |
| URI FORMAT | 991 |
| OPTIONS | 991 |
| USAGE | 992 |
| EXAMPLES | 992 |
| SUPPORT CLASSES | 993 |
| CAMELITEMREADER | 993 |
| CAMELITEMWRITER | 994 |
| CAMELITEMPROCESSOR | 994 |
| CAMELJOBEXECUTIONLISTENER | 995 |
| CHAPTER 114. SPRINGINTEGRATION | 997 |
| SPRING INTEGRATION COMPONENT | 997 |
| URI FORMAT | 997 |
| OPTIONS | 997 |
| USAGE | 998 |
| USING THE SPRING INTEGRATION ENDPOINT | 999 |
| THE SOURCE AND TARGET ADAPTER | 1000 |
| CHAPTER 115. SPRING EVENT | 1003 |
| SPRING EVENT COMPONENT | 1003 |
| URI FORMAT | 1003 |
| CHAPTER 116. SPRING LDAP | 1004 |
| SPRING LDAP COMPONENT | 1004 |
| URI FORMAT | 1004 |

| | |
|---|-------------|
| OPTIONS | 1004 |
| USAGE | 1005 |
| SEARCH | 1005 |
| BIND | 1005 |
| UNBIND | 1006 |
| CHAPTER 117. SPRING REDIS | 1007 |
| SPRING REDIS COMPONENT | 1007 |
| URI FORMAT | 1007 |
| URI OPTIONS | 1007 |
| USAGE | 1008 |
| MESSAGE HEADERS EVALUATED BY THE REDIS PRODUCER | 1008 |
| REDIS CONSUMER | 1019 |
| DEPENDENCIES | 1019 |
| CHAPTER 118. SPRING SECURITY | 1021 |
| SPRING SECURITY | 1021 |
| CREATING AUTHORIZATION POLICIES | 1021 |
| CONTROLLING ACCESS TO CAMEL ROUTES | 1022 |
| AUTHENTICATION | 1024 |
| HANDLING AUTHENTICATION AND AUTHORIZATION ERRORS | 1025 |
| DEPENDENCIES | 1026 |
| CHAPTER 119. SPRING WEB SERVICES | 1027 |
| SPRING WEB SERVICES COMPONENT | 1027 |
| URI FORMAT | 1027 |
| OPTIONS | 1028 |
| REGISTRY BASED OPTIONS | 1030 |
| MESSAGE HEADERS | 1031 |
| ACCESSING WEB SERVICES | 1032 |
| SENDING SOAP AND WS-ADDRESSING ACTION HEADERS | 1032 |
| USING SOAP HEADERS | 1033 |
| THE HEADER AND ATTACHMENT PROPAGATION | 1034 |
| HOW TO USE MTOM ATTACHMENTS | 1034 |
| THE CUSTOM HEADER AND ATTACHMENT FILTERING | 1035 |
| USING A CUSTOM MESSAGESENDER AND MESSAGEFACTORY | 1036 |
| EXPOSING WEB SERVICES | 1037 |
| ENDPOINT MAPPING IN ROUTES | 1038 |
| ALTERNATIVE CONFIGURATION, USING EXISTING ENDPOINT MAPPINGS | 1038 |
| POJO (UN)MARSHALLING | 1039 |
| CHAPTER 120. SQL COMPONENT | 1041 |
| SQL COMPONENT | 1041 |
| URI FORMAT | 1041 |
| OPTIONS | 1042 |
| TREATMENT OF THE MESSAGE BODY | 1048 |
| RESULT OF THE QUERY | 1048 |
| HEADER VALUES | 1049 |
| CONFIGURATION | 1049 |
| SAMPLE | 1049 |
| USING NAMED PARAMETERS | 1051 |
| USING THE JDBC BASED IDEMPOTENT REPOSITORY | 1051 |
| CUSTOMIZE THE JDBCMESSAGEIDREPOSITORY | 1053 |
| USING THE JDBC BASED AGGREGATION REPOSITORY | 1054 |

| | |
|---|-------------|
| WHAT IS PRESERVED WHEN PERSISTING | 1057 |
| RECOVERY | 1057 |
| DATABASE | 1058 |
| STORING BODY AND HEADERS AS TEXT | 1059 |
| CODEC (SERIALIZATION) | 1060 |
| TRANSACTION | 1060 |
| SERVICE (START/STOP) | 1060 |
| AGGREGATOR CONFIGURATION | 1060 |
| OPTIMISTICK LOCKING | 1061 |
| CHAPTER 121. SSH | 1064 |
| SSH | 1064 |
| URI FORMAT | 1064 |
| OPTIONS | 1064 |
| CONSUMER ONLY OPTIONS | 1065 |
| USAGE AS A PRODUCER ENDPOINT | 1066 |
| AUTHENTICATION | 1066 |
| CERTIFICATE DEPENDENCIES | 1068 |
| EXAMPLE | 1068 |
| CHAPTER 122. STAX | 1069 |
| STAX COMPONENT | 1069 |
| URI FORMAT | 1069 |
| USAGE OF A CONTENT HANDLER AS STAX PARSER | 1069 |
| ITERATE OVER A COLLECTION USING JAXB AND STAX | 1070 |
| THE PREVIOUS EXAMPLE WITH XML DSL | 1072 |
| CHAPTER 123. STOMP | 1073 |
| STOMP COMPONENT | 1073 |
| URI FORMAT | 1073 |
| OPTIONS | 1073 |
| SAMPLES | 1074 |
| CHAPTER 124. STREAM | 1075 |
| STREAM COMPONENT | 1075 |
| URI FORMAT | 1075 |
| OPTIONS | 1075 |
| MESSAGE CONTENT | 1077 |
| SAMPLES | 1077 |
| CHAPTER 125. STRINGTEMPLATE | 1080 |
| STRING TEMPLATE | 1080 |
| URI FORMAT | 1080 |
| OPTIONS | 1080 |
| HEADERS | 1081 |
| HOT RELOADING | 1081 |
| STRINGTEMPLATE ATTRIBUTES | 1081 |
| SAMPLES | 1082 |
| THE EMAIL SAMPLE | 1082 |
| CHAPTER 126. STUB | 1084 |
| STUB COMPONENT | 1084 |
| URI FORMAT | 1084 |
| EXAMPLES | 1084 |

| | |
|--|-------------|
| CHAPTER 127. TEST | 1086 |
| TEST COMPONENT | 1086 |
| URI FORMAT | 1086 |
| URI OPTIONS | 1086 |
| EXAMPLE | 1087 |
| CHAPTER 128. TIMER | 1088 |
| TIMER COMPONENT | 1088 |
| URI FORMAT | 1088 |
| OPTIONS | 1088 |
| EXCHANGE PROPERTIES | 1089 |
| MESSAGE HEADERS | 1090 |
| SAMPLE | 1090 |
| FIRING ONLY ONCE | 1091 |
| CHAPTER 129. TWITTER | 1092 |
| TWITTER | 1092 |
| URI FORMAT | 1092 |
| TWITTERCOMPONENT: | 1092 |
| CONSUMER ENDPOINTS: | 1093 |
| PRODUCER ENDPOINTS: | 1094 |
| URI OPTIONS | 1094 |
| MESSAGE HEADER | 1096 |
| MESSAGE BODY | 1096 |
| TO CREATE A STATUS UPDATE WITHIN YOUR TWITTER PROFILE, SEND THIS PRODUCER A STRING BODY. | 1096 |
| TO POLL, EVERY 5 SEC., ALL STATUSES ON YOUR HOME TIMELINE: | 1096 |
| TO SEARCH FOR ALL STATUSES WITH THE KEYWORD 'CAMEL': | 1097 |
| SEARCHING USING A PRODUCER WITH STATIC KEYWORDS | 1097 |
| SEARCHING USING A PRODUCER WITH DYNAMIC KEYWORDS FROM HEADER | 1097 |
| EXAMPLE | 1097 |
| CHAPTER 130. URLREWRITE | 1098 |
| URLREWRITE COMPONENT | 1098 |
| OPTIONS | 1098 |
| USAGE | 1099 |
| USING LOAD BALANCING EIP | 1101 |
| USING APACHE MOD_REWRITE STYLE CONFIGURATION | 1101 |
| WRITING CUSTOM URL REWRITERS | 1102 |
| A SIMPLE URL REWRITE FILTER | 1103 |
| CHAPTER 131. VALIDATION | 1105 |
| VALIDATION COMPONENT | 1105 |
| URI FORMAT | 1105 |
| OPTIONS | 1106 |
| EXAMPLE | 1106 |
| CHAPTER 132. VELOCITY | 1108 |
| VELOCITY | 1108 |
| URI FORMAT | 1108 |
| OPTIONS | 1108 |
| VELOCITY CONTEXT | 1109 |
| HOT RELOADING | 1109 |
| DYNAMIC TEMPLATES | 1109 |

| | |
|--|-------------|
| SAMPLES | 1110 |
| THE EMAIL SAMPLE | 1111 |
| CHAPTER 133. VERTX | 1113 |
| VERTX COMPONENT | 1113 |
| URI FORMAT | 1113 |
| CHAPTER 134. VM | 1114 |
| VM COMPONENT | 1114 |
| URI FORMAT | 1114 |
| OPTIONS | 1115 |
| SAMPLES | 1115 |
| CHAPTER 135. WEATHER | 1117 |
| WEATHER COMPONENT | 1117 |
| URI FORMAT | 1117 |
| OPTIONS | 1117 |
| EXCHANGE DATA FORMAT | 1119 |
| MESSAGE HEADERS | 1119 |
| SAMPLES | 1120 |
| CHAPTER 136. WEBSOCKET | 1121 |
| WEBSOCKET COMPONENT | 1121 |
| URI FORMAT | 1121 |
| COMPONENT OPTIONS | 1121 |
| ENDPOINT OPTIONS | 1123 |
| MESSAGE HEADERS | 1124 |
| USAGE | 1124 |
| SETTING UP SSL FOR WEBSOCKET COMPONENT | 1125 |
| USING THE JSSE CONFIGURATION UTILITY | 1125 |
| PROGRAMMATIC CONFIGURATION OF THE COMPONENT | 1125 |
| SPRING DSL BASED CONFIGURATION OF ENDPOINT | 1126 |
| JAVA DSL BASED CONFIGURATION OF ENDPOINT | 1126 |
| CHAPTER 137. XMLRPC | 1128 |
| XMLRPC COMPONENT | 1128 |
| XMLRPC OVERVIEW | 1128 |
| URI FORMAT | 1129 |
| OPTIONS | 1129 |
| MESSAGE HEADERS | 1131 |
| USING THE XMLRPC DATA FORMAT | 1131 |
| INVOKE XMLRPC SERVICE FROM CLIENT | 1132 |
| HOW TO CONFIGURE THE XMLRPCCLIENT WITH JAVA CODE | 1132 |
| CHAPTER 138. XML SECURITY COMPONENT | 1134 |
| XML SECURITY COMPONENT | 1134 |
| XML SIGNATURE WRAPPING MODES | 1134 |
| URI FORMAT | 1136 |
| BASIC EXAMPLE | 1137 |
| COMMON SIGNING AND VERIFYING OPTIONS | 1138 |
| SIGNING OPTIONS | 1140 |
| VERIFYING OPTIONS | 1144 |
| OUTPUT NODE DETERMINATION IN ENVELOPING XML SIGNATURE CASE | 1149 |
| SEE ALSO | 1151 |

| | |
|--|-------------|
| CHAPTER 139. XMPP | 1152 |
| XMPP COMPONENT | 1152 |
| URI FORMAT | 1152 |
| OPTIONS | 1152 |
| HEADERS AND SETTING SUBJECT OR LANGUAGE | 1153 |
| EXAMPLES | 1153 |
| CHAPTER 140. XQUERY ENDPOINT | 1155 |
| XQUERY | 1155 |
| URI FORMAT | 1155 |
| CHAPTER 141. XSLT | 1156 |
| XSLT | 1156 |
| URI FORMAT | 1156 |
| OPTIONS | 1156 |
| USING XSLT ENDPOINTS | 1159 |
| GETTING PARAMETERS INTO THE XSLT TO WORK WITH | 1159 |
| SPRING XML VERSIONS | 1160 |
| USING XSL:INCLUDE | 1160 |
| USING XSL:INCLUDE AND DEFAULT PREFIX | 1161 |
| DYNAMIC STYLESHEETS | 1162 |
| CHAPTER 142. YAMMER | 1163 |
| YAMMER | 1163 |
| URI FORMAT | 1163 |
| YAMMERCOMPONENT | 1163 |
| CONSUMING MESSAGES | 1164 |
| URI OPTIONS FOR CONSUMING MESSAGES | 1165 |
| MESSAGE FORMAT | 1166 |
| CREATING MESSAGES | 1168 |
| RETRIEVING USER RELATIONSHIPS | 1169 |
| URI OPTIONS FOR RETRIEVING RELATIONSHIPS | 1170 |
| RETRIEVING USERS | 1170 |
| URI OPTIONS FOR RETRIEVING USERS | 1171 |
| USING AN ENRICHER | 1172 |
| CHAPTER 143. ZOOKEEPER | 1173 |
| ZOOKEEPER | 1173 |
| URI FORMAT | 1173 |
| OPTIONS | 1174 |
| USE CASES | 1174 |
| READING FROM A ZNODE. | 1175 |
| READING FROM A ZNODE - (ADDITIONAL CAMEL 2.10 ONWARDS) | 1175 |
| WRITING TO A ZNODE. | 1175 |
| ZOOKEEPER ENABLED ROUTE POLICY. | 1177 |

CHAPTER 1. COMPONENTS OVERVIEW

Abstract

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

1.1. LIST OF COMPONENTS

Table of components

The following components are available for use with Apache Camel.

Table 1.1. Apache Camel Components

| Component | Endpoint URI | Artifact ID | Description |
|-----------|--|----------------------------|--|
| ActiveMQ | <code>activemq:[queue: topic:]DestinationName</code> | <code>activemq-core</code> | For JMS Messaging with Apache ActiveMQ. |
| AHC | <code>ahc:http[s]://Hostname[:Port]/ResourceUri</code> | <code>camel-ahc</code> | To call external HTTP services using Async Http Client |
| AMQP | <code>amqp:[queue: topic:]DestinationName[?Options]</code> | <code>camel-amqp</code> | For messaging with the AMQP protocol. |
| APNS | <code>apns:notify[?Options]</code> <code>apns:consumer[?Options]</code> | <code>camel-apns</code> | For sending notifications to Apple iOS devices. |
| Atom | <code>atom://AtomUri[?Options]</code> | <code>camel-atom</code> | Working with Apache Abdera for atom integration, such as consuming an atom feed. |
| Avro | <code>avro:http://HostName[:Port][?Options]</code> | <code>camel-avro</code> | Working with Apache Avro for data serialization. |
| AWS-CW | <code>aws-cw://Namespace[?Options]</code> | <code>camel-aws</code> | For sending metrics to Amazon CloudWatch . |

| Component | Endpoint URI | Artifact ID | Description |
|---------------------------------|---|-----------------------------------|--|
| AWS-DDB | <code>aws-ddb://TableName[?Options]</code> | <code>camel-aws</code> | For working with Amazon's DynamoDB (DDB) . |
| AWS-SDB | <code>aws-sdb://DomainName[?Options]</code> | <code>camel-aws</code> | For working with Amazon's SimpleDB (SDB) . |
| AWS-SES | <code>aws-ses://From[?Options]</code> | <code>camel-aws</code> | For working with Amazon's Simple Email Service (SES) . |
| AWS-S3 | <code>aws-s3://BucketName[?Options]</code> | <code>camel-aws</code> | For working with Amazon's Simple Storage Service (S3) . |
| AWS-SNS | <code>aws-sns://TopicName[?Options]</code> | <code>camel-aws</code> | For Messaging with Amazon's Simple Notification Service (SNS) . |
| AWS-SQS | <code>aws-sqs://QueueName[?Options]</code> | <code>camel-aws</code> | For Messaging with Amazon's Simple Queue Service (SQS) . |
| Bean | <code>bean:BeanID[?methodName=Method]</code> | <code>camel-core</code> | Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects). |
| Bean Validation | <code>bean-validator:Something[?Options]</code> | <code>camel-bean-validator</code> | Validates the payload of a message using the Java Validation API (JSR 303 and JAXP Validation) and its reference implementation Hibernate Validator . |

| Component | Endpoint URI | Artifact ID | Description |
|-----------|--|---------------|---|
| Browse | <code>browse:Name</code> | camel-core | 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. |
| Cache | <code>cache://CacheName[?Options]</code> | camel-cache | The cache component enables you to perform caching operations using EHCACHE as the Cache Implementation. |
| Class | <code>class:ClassName[?method=MethodName]</code> | camel-core | Uses the Bean binding to bind message exchanges to beans in the registry. Is also used for exposing and invoking POJOs (Plain Old Java Objects). |
| Cometd | <code>cometd://HostName[:Port]/ChannelName[?Options]</code> | camel-cometd | A transport for working with the jetty implementation of the cometd/bayeux protocol. |
| Context | <code>context:CamelContextId:LocalEndpointName</code> | camel-context | Refers to an endpoint in a different CamelContext . |
| Crypto | <code>crypto:sign:Name[?Options]</code> <code>crypto:verify:Name[?Options]</code> | camel-crypto | Sign and verify exchanges using the Signature Service of the Java Cryptographic Extension. |
| CXF | <code>cxf://Address[?Options]</code> | camel-cxf | Working with Apache CXF for web services integration. |

| Component | Endpoint URI | Artifact ID | Description |
|------------|--|-------------|--|
| CXF Bean | <code>cxf:BeanName</code> | camel-cxf | Process the exchange using a JAX WS or JAX RS annotated bean from the registry. |
| CXFRS | <code>cxfrs:bean:RsEndpoint[?Options]</code> | camel-cxf | Provides integration with Apache CXF for connecting to JAX-RS services hosted in CXF. |
| DataFormat | <code>dataformat:Name:(marshal unmarshal)[?Options]</code> | camel-core | Enables you to marshal or unmarshal a message in one of the standard Camel data formats, by sending it to an endpoint. |
| DataSet | <code>dataset:Name[?Options]</code> | camel-core | For load & soak testing the DataSet provides a way to create huge numbers of messages for sending to Components or asserting that they are consumed correctly. |
| Direct | <code>direct:EndpointID[?Options]</code> | camel-core | Synchronous call (single-threaded) to another endpoint from <i>same</i> CamelContext. |
| Direct-VM | <code>direct-vm:EndpointID[?Options]</code> | camel-core | Synchronous call (single-threaded) to another endpoint in another CamelContext running in the same JVM. |

| Component | Endpoint URI | Artifact ID | Description |
|------------|---|-------------------------------|---|
| Disruptor | <code>disruptor:Name[?Options]</code> <code>disruptor-vm:Name[?Options]</code> | <code>camel-disruptor</code> | Similar to a SEDA endpoint, but uses a Disruptor instead of a blocking queue. |
| DNS | <code>dns:Operation</code> | <code>camel-dns</code> | Look up domain information and run DNS queries using DNSJava |
| EJB | <code>ejb:EjbName[?method=MethodName]</code> | <code>camel-ejb</code> | Uses the Bean binding to bind message exchanges to EJBs. It works like the Bean component, but just for accessing EJBs. Supports EJB 3.0 onwards. |
| EventAdmin | <code>eventadmin:topic</code> | <code>camel-eventadmin</code> | |
| Exec | <code>exec://Executable[?Options]</code> | <code>camel-exec</code> | Execute system command. |
| Fabric | <code>fabric:ClusterID[:PublishedURI][?Options]</code> | <code>fabric-camel</code> | Look up or publish a fabric endpoint. |
| Facebook | <code>facebook://[Endpoint][?Options]</code> | <code>camel-facebook</code> | Provides access to all of the Facebook APIs accessible using Facebook4J . |
| File2 | <code>file://Directory Name[?Options]</code> | <code>camel-core</code> | Sending messages to a file or polling a file or directory. |
| Flatpack | <code>flatpack:[fixed delim]: ConfigFile</code> | <code>camel-flatpack</code> | Processing fixed width or delimited files or messages using the FlatPack library |
| FOP | <code>fop:OutputFormat</code> | <code>camel-fop</code> | Renders the message into different output formats using Apache FOP . |

| Component | Endpoint URI | Artifact ID | Description |
|------------|---|-------------------------|--|
| Freemarker | freemarker: <i>TemplateResource</i> | camel-freemarker | Generates a response using a Freemarker template. |
| FTP2 | ftp:// [Username@]Hostname[:Port]/Directoryname[?Options] | camel-ftp | Sending and receiving files over FTP. |
| GAuth | gauth://Name [?Options] | camel-gae | Used by web applications to implement a Google-specific OAuth consumer |
| GHTTP | ghttp:// Path[?Options] ghttp:// Hostname[:Port]/Path[?Options] ghttps:// Hostname[:Port]/Path[?Options] | camel-gae | Provides connectivity to the GAE URL fetch service and can also be used to receive messages from servlets. |
| GLogin | glogin:// Hostname[:Port][?Options] | camel-gae | Used by Camel applications outside Google App Engine (GAE) for programmatic login to GAE applications. |
| GMail | gmail:// Username@gmail.com[?Options] gmail:// Username@googlemail.com[?Options] | camel-gae | Supports sending of emails via the GAE mail service. |
| GTask | gtask:// QueueName | camel-gae | Supports asynchronous message processing on GAE using the task queueing service as a message queue. |

| Component | Endpoint URI | Artifact ID | Description |
|----------------|---|----------------------|--|
| Geocoder | <pre>geocoder:Address:Name[?Options] geocoder:latlng:Latitude,Longitude[?Options]</pre> | camel-geocoder | Looks up geocodes (latitude and longitude) for a given address, or performs reverse look-up. |
| Guava EventBus | <pre>guava-eventbus:BusName[?EventClassName]</pre> | camel-guava-eventbus | 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). This component provides integration bridge between Camel and Google Guava EventBus infrastructure. |
| Hazelcast | <pre>hazelcast://StoreType:CacheName[?Options]</pre> | camel-hazelcast | Hazelcast is a data grid entirely implemented in Java (single JAR). This component supports map, multimap, seda, queue, set, atomic number and simple cluster. |
| HBase | <pre>hbase://Table[?Options]</pre> | camel-hbase | For reading/writing from/to an HBase store (Hadoop database). |
| HDFS | <pre>hdfs://Path[?Options]</pre> | camel-hdfs | For reading/writing from/to an HDFS filesystem. |

| Component | Endpoint URI | Artifact ID | Description |
|-----------|--|------------------------------|---|
| HL7 | <code>mina:tcp://Host[:Port]</code> | <code>camel-hl7</code> | For working with the HL7 MLLP protocol and the HL7 model using the HAPI library . |
| HTTP | <code>http://Hostname[:Port][/ResourceUri]</code> | <code>camel-http</code> | For calling out to external HTTP servers, using Apache HTTP Client 3.x. |
| HTTP4 | <code>http://Hostname[:Port][/ResourceUri]</code> | <code>camel-http4</code> | For calling out to external HTTP servers, using Apache HTTP Client 4.x. |
| iBATIS | <code>ibatis:OperationName[?Options]</code> | <code>camel-ibatis</code> | Performs a query, poll, insert, update or delete in a relational database using Apache iBATIS. |
| IMap | <code>imap://[UserName@]Host[:Port][?Options]</code> | <code>camel-mail</code> | Receiving email using IMap. |
| IRC | <code>irc:Host[:Port]/#Room</code> | <code>camel-irc</code> | For IRC communication. |
| JavaSpace | <code>javaspace:jini://Host[?Options]</code> | <code>camel-javaspace</code> | Sending and receiving messages through JavaSpace. |
| JBIP | <pre> jbi:service:serviceNamespace[sep]serviceName jbi:endpoint:serviceNamespace[sep]serviceName[sep]endpointName jbi:name:endpointName </pre> | <code>camel-jbi</code> | For JBI integration such as working with Apache ServiceMix. |

| Component | Endpoint URI | Artifact ID | Description |
|-----------|---|----------------------------|---|
| JClouds | <code>jclouds:[Blobstore ComputeService]:Provider</code> | <code>camel-jclouds</code> | For interacting with cloud compute & blobstore service via JClouds . |
| JCR | <code>jcr://UserName:Password@Repository/path/to/node</code> | <code>camel-jcr</code> | Storing a message in a JCR (JSR-170) compliant repository like Apache Jackrabbit. |
| JDBC | <code>jdbc:DataSourceName[?Options]</code> | <code>camel-jdbc</code> | For performing JDBC queries and operations. |
| Jetty | <code>jetty:http://Host[:Port]/ResourceUri</code> | <code>camel-jetty</code> | For exposing services over HTTP. |
| Jing | <code>ring:LocalOrRemoteResource</code> <code>rnc:LocalOrRemoteResource</code> | <code>camel-jing</code> | Validates the payload of a message using RelaxNG or RelaxNG compact syntax. |
| JMS | <code>jms:[temp:][queue: topic:]DestinationName[?Options]</code> | <code>camel-jms</code> | Working with JMS providers. |
| JMX | <code>jmx://Platform[?Options]</code> | <code>camel-jmx</code> | For working with JMX notification listeners. |
| JPA | <code>jpa:[EntityClassName][?Options]</code> | <code>camel-jpa</code> | For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink. |
| Jsch | <code>scp://Hostname/Destination</code> | <code>camel-jsch</code> | Support for the <code>scp</code> protocol. |
| JT400 | <code>jt400://User:Pwd@System/PathToDTAQ</code> | <code>camel-jt400</code> | For integrating with data queues on an AS/400 (aka System i, IBM i, i5, ...) system. |

| Component | Endpoint URI | Artifact ID | Description |
|--------------------------|---|----------------------------|---|
| Kestrel | <code>kestrel://[AddressList/]QueueName[?Options]</code> | <code>camel-kestrel</code> | For producing to or consuming from Kestrel queues. |
| Krati | <code>krati://[PathToDataStore/][?Options]</code> | <code>camel-krati</code> | For producing to or consuming to Krati datastores. |
| Language | <code>language://LanguageName[:Script][?Options]</code> | <code>camel-core</code> | Executes language scripts. |
| LDAP | <code>ldap:Host[:Port]?base=... [&scope=Scope]</code> | <code>camel-ldap</code> | Performing searches on LDAP servers (<i>Scope</i> must be one of object onelevel subtree). |
| LevelDB | N/A | <code>camel-leveldb</code> | A very lightweight and embeddable key-value database. |
| List | <code>list:ListID</code> | <code>camel-core</code> | Provides a simple <code>BrowsableEndpoint</code> which can be useful for testing, visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed. |
| Log | <code>log:LoggingCategory[?level=LoggingLevel]</code> | <code>camel-core</code> | Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j. |
| Lucene | <pre> lucene:SearcherName:insert[? analyzer=Analyzer] lucene:SearcherName:query[? analyzer=Analyzer] </pre> | <code>camel-lucene</code> | Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities. |

| Component | Endpoint URI | Artifact ID | Description |
|-----------|---|--------------------|---|
| Master | REVISIT | | |
| MINA | <pre>mina:tcp://Host name[:Port] [?Options] mina:udp://Host name[:Port] [?Options] mina:multicast ://Hostname[:P ort][?Options] mina:vm://Host name[:Port] [?Options]</pre> | camel-mina | Working with Apache MINA. |
| MINA2 | <pre>mina2:tcp://Ho stname[:Port] [?Options] mina2:udp://Ho stname[:Port] [?Options] mina2:vm://Hos tname[:Port] [?Options]</pre> | camel-mina2 | Working with Apache MINA 2.x . |
| Mock | mock:EndpointID | camel-core | For testing routes and mediation rules using mocks. |

| Component | Endpoint URI | Artifact ID | Description |
|--------------------------|---|-----------------------------|---|
| MongoDB | <code>mongodb:Connecti on[?Options]</code> | <code>camel-mongodb</code> | Interacts with MongoDB databases and collections. Offers producer endpoints to perform CRUD-style operations and more against databases and collections, as well as consumer endpoints to listen on collections and dispatch objects to Camel routes. |
| MQTT | <code>mqtt:Name</code> | <code>camel-mqtt</code> | Component for communicating with MQTT M2M message brokers |
| MSV | <code>msv:LocalOrRemot eResource</code> | <code>camel-msv</code> | Validates the payload of a message using the MSV Library. |
| Mustache | <code>mustache:Templat eName[?Options]</code> | <code>camel-mustache</code> | Enables you to process a message using a Mustache template. |
| MVEL | <code>mvel:TemplateNam e[?Options]</code> | <code>camel-mvel</code> | Enables you to process a message using an MVEL template. |
| MyBatis | <code>mybatis:Statemen tName</code> | <code>camel-mybatis</code> | Performs a query, poll, insert, update or delete in a relational database using MyBatis. |
| Nagios | <code>nagios://Host[:P ort][?Options]</code> | <code>camel-nagios</code> | Sending passive checks to Nagios using JSendNSCA . |

| Component | Endpoint URI | Artifact ID | Description |
|-------------|---|------------------|--|
| Netty | <pre>netty:tcp://localhost:99999[?Options] netty:udp://RemoteHost:99999/[?Options]</pre> | camel-netty | Working with TCP and UDP protocols using Java NIO based capabilities offered by the JBoss Netty community project. |
| Netty HTTP | <pre>netty-http:http://HostName[:Port][?Options]</pre> | camel-netty-http | An extension to the Netty component, facilitating the HTTP transport. |
| NMR | nmr:serviceMixURI | servicemix-camel | For OSGi integration when working with Red Hat JBoss Fuse. Enables you to specify the URI of a ServiceMix endpoint . |
| Pax-Logging | paxlogging:Appender | camel-paxlogging | |
| POP | <pre>pop3://[UserName@]Host[:Port][?Options]</pre> | camel-mail | Receiving email using POP3 and JavaMail. |
| Printer | <pre>lpr://localhost[:Port]/default[?Options] lpr://RemoteHost[:Port]/path/to/printer[?Options]</pre> | camel-printer | Provides a way to direct payloads on a route to a printer. |
| Properties | properties://Key[?Options] | camel-properties | Facilitates using property placeholders directly in endpoint URI definitions. |

| Component | Endpoint URI | Artifact ID | Description |
|-----------|---|-----------------------|--|
| Quartz | <pre>quartz://[GroupName/]TimerName[?Options] quartz://GroupName/TimerName/CronExpression</pre> | camel-quartz | Provides a scheduled delivery of messages using the Quartz scheduler. |
| Quartz2 | <pre>quartz2://[GroupName/]TimerName[?Options] quartz2://GroupName/TimerName/CronExpression</pre> | camel-quartz2 | Provides a scheduled delivery of messages using the Quartz Scheduler 2.x . |
| Quickfix | <pre>quickfix-server:ConfigFile quickfix-client:ConfigFile</pre> | camel-quickfix | Implementation of the QuickFix for Java engine which allow to send/receive FIX messages. |
| RabbitMQ | rabbitmq://HostName[:Port]/ExchangeName[?Options] | camel-rabbitmq | Enables you to produce and consume messages from a RabbitMQ instance. |
| Ref | ref:EndpointID | camel-core | Component for lookup of existing endpoints bound in the Registry. |
| Restlet | restlet:RestletURL[?Options] | camel-restlet | Component for consuming and producing Restful resources using Restlet . |
| RMI | rmi://RmiRegistryHost:RmiRegistryPort/RegistryPath | camel-rmi | Working with RMI. |

| Component | Endpoint URI | Artifact ID | Description |
|--|---|---|--|
| Routebox | <code>routebox:routeboxName[?Options]</code> | <code>camel-routebox</code> | |
| RSS | <code>rss:Uri</code> | <code>camel-rss</code> | Working with ROME for RSS integration, such as consuming an RSS feed. |
| RNC | <code>rnc:LocalOrRemoteResource</code> | <code>camel-jing</code> | Validates the payload of a message using RelaxNG Compact Syntax. |
| RNG | <code>rng:LocalOrRemoteResource</code> | <code>camel-jing</code> | Validates the payload of a message using RelaxNG. |
| Salesforce | <code>salesforce:Topic[?Options]</code> | <code>camel-salesforce</code> | Enables producer and consumer endpoints to communicate with Salesforce using Java DTOs. |
| SAP | <code>sap:[destination:DestinationName server:ServerName]rfcName[?Options]</code> | <code>camel-sap</code> | Enables outbound and inbound communication to and from SAP systems using synchronous remote function calls, sRFC. |
| Chapter 103, SAP NetWeaver | <code>sap-netweaver:https://Hostname[:Port]/Path[?Options]</code> | <code>camel-sap-netweaver</code> | Integrates with the SAP NetWeaver Gateway using HTTP transports. |
| Scalate | <code>scalate:TemplateName[?Options]</code> | <code>org.fusesource.scalate/scalate-camel</code> | Uses the given Scalate template to transform the message. |
| SEDA | <code>seda:EndpointID</code> | <code>camel-core</code> | Used to deliver messages to a <code>java.util.concurrent.BlockingQueue</code> , useful when creating SEDA style processing pipelines within the same CamelContext. |

| Component | Endpoint URI | Artifact ID | Description |
|---------------------------------|---|------------------------------------|--|
| SERVLET | <code>servlet://RelativePath[?Options]</code> | <code>camel-servlet</code> | Provides HTTP based endpoints for consuming HTTP requests that arrive at a HTTP endpoint and this endpoint is bound to a published Servlet. |
| ServletListener | N/A | <code>camel-servletlistener</code> | Used for bootstrapping Camel applications in Web applications. |
| SFTP | <code>sftp://[Username@]Hostname[:Port]/DirectoryName[?Options]</code> | <code>camel-ftp</code> | Sending and receiving files over SFTP. |
| Sip | <pre> sip://User@Hostname[:Port][?Options] sips://User@Hostname[:Port][?Options] </pre> | <code>camel-sip</code> | Publish/subscribe communication capability using the telecom SIP protocol. RFC3903 - Session Initiation Protocol (SIP) Extension for Event |
| SJMS | <code>sjms:[queue: topic:]destinationName[?Options]</code> | <code>camel-sjms</code> | A JMS client for Camel that employs best practices for JMS client creation and configuration. |
| SMPP | <code>smpp://UserInfo@Host[:Port][?Options]</code> | <code>camel-smpp</code> | To send and receive SMS using Short Messaging Service Center using the JSMPP library . |
| SMTP | <code>smtp://[UserName@]Host[:Port][?Options]</code> | <code>camel-mail</code> | Sending email using SMTP and JavaMail. |
| SNMP | <code>snmp://Hostname[:Port][?Options]</code> | <code>camel-snmp</code> | Gives you the ability to poll SNMP capable devices or receive traps. |

| Component | Endpoint URI | Artifact ID | Description |
|---------------------|--|---------------------------------------|--|
| Solr | <code>solr://Hostname[:Port]/Solr[?Options]</code> | <code>camel-solr</code> | Uses the Solrj client API to interface with an Apache Lucene Solr server. |
| Spring Batch | <code>spring-batch:Job[?Options]</code> | <code>camel-spring-batch</code> | To bridge Camel and Spring Batch . |
| Spring Event | <code>spring-event://dummy</code> | <code>camel-spring</code> | Working with Spring ApplicationEvents. |
| Spring Integration | <code>spring-integration:DefaultChannelName[?Options]</code> | <code>camel-spring-integration</code> | The bridge component of Camel and Spring Integration. |
| Spring LDAP | <code>spring-ldap:SpringLdapTemplate[?Options]</code> | <code>camel-spring-ldap</code> | Provides a Camel wrapper for Spring LDAP . |
| Spring Redis | <code>spring-redis://Hostname[:Port][?Options]</code> | <code>camel-spring-redis</code> | Enables sending and receiving messages from Redis , which is an advanced key-value store, where keys can contain strings, hashes, lists, sets and sorted sets. |
| Spring Web Services | <code>spring-ws:[MappingType:]Address[?Options]</code> | <code>camel-spring-ws</code> | Client-side support for accessing web services, and server-side support for creating your own contract-first web services using Spring Web Services . |
| SQL | <code>sql:SqlQueryString[?Options]</code> | <code>camel-sql</code> | Performing SQL queries using JDBC. |
| SSH | <code>ssh:[Username[:Password]@]Host[:Port][?Options]</code> | <code>camel-ssh</code> | For sending commands to a SSH server. |

| Component | Endpoint URI | Artifact ID | Description |
|-----------------|--|----------------------|--|
| StAX | <code>stax:ContentHandlerClassName</code> | camel-stax | Process messages through a SAX ContentHandler . |
| Stomp | <code>stomp:queue:Destination[?Options]</code> | camel-stomp | For sending messages to or receiving messages from a Stomp compliant broker, such as Apache ActiveMQ . |
| Stream | <code>stream:[in out err header][?Options]</code> | camel-stream | Read or write to an input/output/error/file stream rather like Unix pipes. |
| String Template | <code>string-template:TemplateURI[?Options]</code> | camel-stringtemplate | Generates a response using a String Template. |
| Stub | <code>stub:SomeOtherCamelUri</code> | camel-core | Allows you to stub out some physical middleware endpoint for easier testing or debugging. |
| Test | <code>test:RouterEndpointUri</code> | camel-spring | Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint. |
| Timer | <code>timer:EndpointID[?Options]</code> | camel-core | A timer endpoint. |
| Twitter | <code>twitter://[Endpoint][?Options]</code> | camel-twitter | A Twitter endpoint. |
| UrlRewrite | N/A | camel-urlrewrite | Enables you to plug URL rewrite functionality into the HTTP, HTTP4, Jetty, or AHC components. |

| Component | Endpoint URI | Artifact ID | Description |
|--------------|---|--------------------------------|---|
| Validation | <code>validator:LocalRemoteResource</code> | <code>camel-spring</code> | Validates the payload of a message using XML Schema and JAXP Validation. |
| Velocity | <code>velocity:TemplateURI[?Options]</code> | <code>camel-velocity</code> | Generates a response using an Apache Velocity template. |
| Vertx | <code>vertx:ChannelName[?Options]</code> | <code>camel-vertx</code> | For working with the Vertx Event Bus. |
| VM | <code>vm:EndpointID</code> | <code>camel-core</code> | Used to deliver messages to a <code>java.util.concurrent.BlockingQueue</code> , useful when creating SEDA style processing pipelines within the same JVM. |
| Weather | <code>weather://DummyName[?Options]</code> | <code>camel-weather</code> | Polls weather information from Open Weather Map : a site that provides free global weather and forecast information. |
| WebSocket | <code>websocket://HostName[:Port]/Path</code> | <code>camel-websocket</code> | Communicating with WebSocket clients. |
| XML RPC | <code>xmlrpc://ServerURI[?Options]</code> | <code>camel-xmlrpc</code> | Provides a data format for XML, which allows serialization and deserialization of request messages and response message using Apache XmlRpc's binary data format. |
| XML Security | N/A | <code>camel-xmlsecurity</code> | Generate and validate XML signatures as described in the W3C standard XML Signature Syntax and Processing . |

| Component | Endpoint URI | Artifact ID | Description |
|---------------------------|---|------------------------------|--|
| XMPP | <code>xmpp:Hostname[:Port]/Room</code> | <code>camel-xmpp</code> | Working with XMPP and Jabber. |
| XQuery | <code>xquery:TemplateURI</code> | <code>camel-saxon</code> | Generates a response using an XQuery template. |
| XSLT | <code>xslt:TemplateURI[?Options]</code> | <code>camel-spring</code> | Enables you to process a message using an XSLT template. |
| Yammer | <code>yammer:[function][?Options]</code> | <code>camel-yammer</code> | Enables you to interact with the Yammer enterprise social network. |
| Zookeeper | <code>zookeeper://HostName[:Port]/Path</code> | <code>camel-zookeeper</code> | Working with ZooKeeper cluster(s). |

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 [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 [JMS](#) 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`.



TRANSACTIONED 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 [JMS](#) component as all these options also apply for this component.

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" />
```

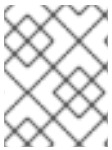
```

</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 [Bean](#) 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&estination.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>${in.body}&#13;</simple>
  </transform>
  <to uri="file://data/activemq/?
fileExist=Append&fileName=advisoryConnection-${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 [JMS](#) 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

ASYNCH HTTP CLIENT (AHC) COMPONENT

Available as of Camel 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>
```

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&...**

AHCENDPOINT OPTIONS

| Name | Default Value | Description |
|--------------------------------|---------------|---|
| throwExceptionOnFailure | true | 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. |
| bridgeEndpoint | false | 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. |

| | | |
|--------------------------|--------------|--|
| transferException | false | 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. |
| client | null | To use a custom com.ning.http.client.AsynchHttpClient . |
| clientConfig | null | To configure the AsynchHttpClient to use a custom com.ning.http.client.AsynchHttpClientConfig . |
| clientConfig.x | null | To configure additional properties of the com.ning.http.client.AsynchHttpClientConfig instance used by the endpoint. Note that configuration options set using this parameter will be merged with those set using the clientConfig parameter or the instance set at the component level with properties set using this parameter taking priority. |

| | | |
|-----------------------------------|-------------------|--|
| <code>clientConfig.realm.x</code> | <code>null</code> | Camel 2.11: To configure realm properties of the <code>com.ning.http.client.AsyncHttpClientConfig</code> . The options which can be used are the options from <code>com.ning.http.client.Realm.RealmBuilder</code> . eg to set scheme, you can configure <code>"clientConfig.realm.scheme=DIGEST"</code> |
| <code>binding</code> | <code>null</code> | To use a custom <code>org.apache.camel.component.ahc.AhcBinding</code> . |
| <code>sslContextParameters</code> | <code>null</code> | Camel 2.9: Reference to a <code>org.apache.camel.util.jse.SSLContextParameters</code> in the CAMEL:Registry. This reference overrides any configured <code>SSLContextParameters</code> 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 <code>clientConfig</code> option at the endpoint or component level. |
| <code>bufferSize</code> | <code>4096</code> | Camel 2.10.3: The initial in-memory buffer size used when transferring data between Camel and AHC Client. |

AHCCOMPONENT OPTIONS

| Name | Default Value | Description |
|---------------------------|-------------------|---|
| <code>client</code> | <code>null</code> | To use a custom <code>com.ning.http.client.AsyncHttpClient</code> . |
| <code>clientConfig</code> | <code>null</code> | To configure the <code>AsyncHttpClients</code> use a custom <code>com.ning.http.client.AsyncHttpClientConfig</code> . |

| | | |
|-----------------------------|-------------|--|
| binding | null | To use a custom <code>org.apache.camel.component.ahc.AhcBinding</code> . |
| sslContextParameters | null | Camel 2.9: To configure custom SSL/TLS configuration options at the component level. See Using the JSSE Configuration Utility for more details. Note that configuring this option will override any SSL/TLS configuration options provided through the <code>clientConfig</code> option at the endpoint or component level. |

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**.

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 <code>HTTP_URI</code> . If the path is start with <code>"/"</code> , http producer will try to find the relative path based on the <code>Exchange.HTTP_BASE_URI</code> header or the <code>exchange.getFromEndpoint().getEndpointUri()</code> ; |
| 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. |

| | | |
|--|---------------------|---|
| <code>Exchange.CONTENT_TYPE</code> | <code>String</code> | The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as <code>text/html</code> . |
| <code>Exchange.CONTENT_ENCODING</code> | <code>String</code> | The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as <code>gzip</code> . |

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.

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. 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.

AHCOOPERATIONFAILEDEXCEPTION

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

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.

CONFIGURING URI TO CALL

You can set the HTTP producer's URI directly from 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");
```

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");
```

HOW TO SET THE HTTP METHOD (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) 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>

```

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");

```

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);

```

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.

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);

```

CONFIGURING ASYNCHTTPCLIENT

The `AsyncHttpClient` client uses a `AsyncHttpClientConfig` to configure the client. See the documentation at [Async Http Client](#) for more details.

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

```

// create a client config builder
AsyncHttpClientConfig.Builder builder = new
AsyncHttpClientConfig.Builder();
// use the builder to set the options we want, in this case we want to
follow redirects and try
// at most 3 retries to send a request to the host
AsyncHttpClientConfig config =
builder.setFollowRedirects(true).setMaxRequestRetry(3).build();

// lookup AhcComponent
AhcComponent component = context.getComponent("ahc", AhcComponent.class);
// and set our custom client config to be used
component.setClientConfig(config);

```

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")

```

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"/>
...

```

- [Jetty](#)
- [HTTP](#)
- [HTTP4](#)

CHAPTER 4. AMQP

AMQP

The AMQP component supports the [AMQP protocol](#) via the [Qpid](#) project.

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

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

URI FORMAT

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

You can specify all of the various configuration options of the [JMS](#) component after the destination name.

CHAPTER 5. APNS

APNS COMPONENT

Available as of Camel 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>
```

URI FORMAT

To send notifications:

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

To consume feedback:

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

OPTIONS

PRODUCER

| Property | Default | Description |
|----------|---------|-------------|
|----------|---------|-------------|

| | | |
|---------------------|--|---|
| <code>tokens</code> | | Empty by default. Configure this property in case you want to statically declare tokens related to devices you want to notify. Tokens are separated by comma. |
|---------------------|--|---|

CONSUMER

| Property | Default | Description |
|-----------------------------|----------------|--|
| <code>delay</code> | 3600 | Delay in seconds between each poll. |
| <code>initialDelay</code> | 10 | Seconds before polling starts. |
| <code>timeUnit</code> | SECONDS | Time Unit for polling. |
| <code>userFixedDelay</code> | true | If true , use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |

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

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](#).

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.

MESSAGE HEADERS

Camel Apns uses these headers.

| Property | Default | Description |
|------------------------------|---------|-------------------|
| <code>CamelApnsTokens</code> | | Empty by default. |

| | | |
|-----------------------------|------------------------|---|
| CamelApnsMessageType | STRING, PAYLOAD | 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 |
|-----------------------------|------------------------|---|

SAMPLES

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&elay=3600&imeUnit=SECONDS" />
    <to uri="log:org.apache.camel.component.apns?
showAll=true&ultiline=true" />
    <to uri="mock:result" />
  </route>
</camelContext>

</beans>

```

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;
}

```

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");
```

SEE ALSO

- [Component](#)
- [Endpoint](#)
- [Blog about using APNS \(in french\)](#)

CHAPTER 6. ATOM

ATOM COMPONENT

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

Apache 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>
```

URI FORMAT

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

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

OPTIONS

| Property | Default | Description |
|---------------------|-------------|---|
| splitEntries | true | If true Apache Camel will poll the feed and for the subsequent polls return each entry poll by poll. If the feed contains 7 entries then Apache Camel will return the first entry on the first poll, the 2nd entry on the next poll, until no more entries where as Apache Camel will do a new update on the feed. If false then Apache Camel will poll a fresh feed on every invocation. |

| | | |
|--------------------------------------|--------------------|---|
| <code>filter</code> | <code>true</code> | Is only used by the split entries to filter the entries to return. Apache Camel will default use the UpdateDateFilter that only return new entries from the feed. So the client consuming from the feed never receives the same entry more than once. The filter will return the entries ordered by the newest last. |
| <code>lastUpdate</code> | <code>null</code> | Is only used by the filter, as the starting timestamp for selection never entries (uses the entry.updated timestamp). Syntax format is: yyyy-MM-ddTHH:MM:ss . Example: 2007-12-24T17:45:59 . |
| <code>throttleEntries</code> | <code>true</code> | Camel 2.5: 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 is set to true . |
| <code>feedHeader</code> | <code>true</code> | Sets whether to add the Abdera Feed object as a header. |
| <code>sortEntries</code> | <code>false</code> | If splitEntries is true , this sets whether to sort those entries by updated date. |
| <code>consumer.delay</code> | <code>60000</code> | Delay in millis between each poll. |
| <code>consumer.initialDelay</code> | <code>1000</code> | Millis before polling starts. |
| <code>consumer.userFixedDelay</code> | <code>false</code> | If true , use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |

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

EXCHANGE DATA FORMAT

Apache Camel will set the In body on the returned **Exchange** with the entries. Depending on the **splitEntries** flag Apache 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>) |

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

MESSAGE HEADERS

Apache Camel atom uses these headers.

| Header | Description |
|----------------------|--|
| CamelAtomFeed | Apache Camel 2.0: When consuming the org.apache.abdera.model.Feed object is set to this header. |

SAMPLES

In the following 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 set up Apache Camel standalone, not running in any container or using Spring.

```
@Override
protected CamelContext createCamelContext() throws Exception {
    // First we register a blog service in our bean registry
    SimpleRegistry registry = new SimpleRegistry();
    registry.put("blogService", new BlogService());

    // Then we create the camel context with our bean registry
    context = new DefaultCamelContext(registry);
}
```

```

    // Then we add all the routes we need using the route builder DSL
syntax
    context.addRoutes(createMyRoutes());

    // And finally we must start Camel to let the magic routing begins
    context.start();

    return context;
}

/**
 * This is the route builder where we create our routes using the Camel
DSL syntax
 */
protected RouteBuilder createMyRoutes() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // We pool the atom feeds from the source for further
processing in the seda queue
            // we set the delay to 1 second for each pool as this is a
unit test also and we can
            // not wait the default poll interval of 60 seconds.
            // Using splitEntries=true will during polling only fetch one
Atom Entry at any given time.
            // As the feed.atom file contains 7 entries, using this will
require 7 polls to fetch the entire
            // content. When Camel have reach the end of entries it will
refresh the atom feed from URI source
            // and restart - but as Camel by default uses the
UpdatedDateFilter it will only deliver new
            // blog entries to "seda:feeds". So only when James Straham
updates his blog with a new entry
            // Camel will create an exchange for the seda:feeds.
            from("atom:file:src/test/data/feed.atom?
splitEntries=true&consumer.delay=1000").to("seda:feeds");

            // From the feeds we filter each blot entry by using our blog
service class
            from("seda:feeds").filter().method("blogService",
"isGoodBlog").to("seda:goodBlogs");

            // And the good blogs is moved to a mock queue as this sample
is also used for unit testing
            // this is one of the strengths in Camel that you can also use
the mock endpoint for your
            // unit tests
            from("seda:goodBlogs").to("mock:result");
        }
    };
}

/**
 * This is the actual junit test method that does the assertion that our
routes is working
 * as expected
 */

```



```
@Test
public void testFiltering() throws Exception {
    // create and start Camel
    context = createCamelContext();
    context.start();

    // Get the mock endpoint
    MockEndpoint mock = context.getEndpoint("mock:result",
MockEndpoint.class);

    // There should be at least two good blog entries from the feed
    mock.expectedMinimumMessageCount(2);

    // Asserts that the above expectations is true, will throw assertions
exception if it failed
    // Camel will default wait max 20 seconds for the assertions to be
true, if the conditions
    // is true sooner Camel will continue
    mock.assertIsSatisfied();

    // stop Camel after use
    context.stop();
}

/**
 * Services for blogs
 */
public class BlogService {

    /**
     * Tests the blogs if its a good blog entry or not
     */
    public boolean isGoodBlog(Exchange exchange) {
        Entry entry = exchange.getIn().getBody(Entry.class);
        String title = entry.getTitle();

        // We like blogs about Camel
        boolean good = title.toLowerCase().contains("camel");
        return good;
    }
}
}
```

CHAPTER 7. AVRO

AVRO COMPONENT

Available as of Camel 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>
```

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.

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.

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 directly in the "in" message body without array wrapping.

AVRO RPC URI OPTIONS

| Name | Version | Description |
|--------------------------|---------|--|
| protocolClassName | | The class name of the avro protocol. |
| singleParameter | 2.12 | If true, consumer parameter won't be wrapped into array. Will fail if protocol specifies more then 1 parameter for the message |

| | | |
|---------------------------|------|--|
| protocol | | Avro procol object. Can be used instead of protocolClassName when complex protocol needs to be created. One cane used #name notation to refer beans from the Registry |
| reflectionProtocol | 2.12 | If protocol object provided is reflection protocol. Should be used only with protocol parameter because for protocolClassName protocol type will be autodetected |

AVRO RPC HEADERS

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

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&singleP
arameter=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 8. AWS

8.1. INTRODUCTION TO THE AWS COMPONENTS

Camel Components for Amazon Web Services

The Camel Components for [Amazon Web Services](#) provide connectivity to AWS services from Camel.

| AWS service | Camel component | Camel Version | Component description |
|---|-------------------------|---------------|---|
| Simple Queue Service (SQS) | AWS-SQS | 2.6 | Supports sending and receiving messages using SQS |
| Simple Notification Service (SNS) | AWS-SNS | 2.8 | Supports sending messages using SNS |
| Simple Storage Service (S3) | AWS-S3 | 2.8 | Supports storing and retrieving of objects using S3 |
| Simple Email Service (SES) | AWS-SES | 2.8.4 | Supports sending emails using SES |
| SimpleDB | AWS-SDB | 2.8.4 | Supports storing retrieving data to/from SDB |
| DynamoDB | AWS-DDB | 2.10.0 | Supports storing retrieving data to/from DDB |
| CloudWatch | AWS-CW | 2.10.3 | Supports sending metrics to CloudWatch |

8.2. AWS-CW

CW Component

*Available as of Camel 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](#).

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&...`

URI Options

| Name | Default Value | Context | Description |
|----------------|---------------|----------|--|
| amazonCwClient | null | Producer | Reference to a <code>com.amazonaws.services.cloudwatch.AmazonCloudWatch</code> in the Registry . |
| accessKey | null | Producer | Amazon AWS Access Key |
| secretKey | null | Producer | Amazon AWS Secret Key |
| name | null | Producer | The metric name which is used if the message header 'CamelAwsCwMetricName' is not present. |
| value | 1.0 | Producer | The metric value which is used if the message header 'CamelAwsCwMetricValue' is not present. |
| unit | Count | Producer | The metric unit which is used if the message header 'CamelAwsCwMetricUnit' is not present. |

| | | | |
|------------------|-------------|----------|--|
| namespace | null | Producer | The metric namespace which is used if the message header 'CamelAwsCwMetricNamespace' is not present. |
| timestamp | null | Producer | The metric timestamp which is used if the message header 'CamelAwsCwMetricTimestamp' is not present. |
| amazonCwEndpoint | null | Producer | The region with which the AWS-CW client wants to work with. |



REQUIRED CW COMPONENT OPTIONS

You have to provide the `amazonCwClient` in the [Registry](#) or your `accessKey` and `secretKey` to access the [Amazon's CloudWatch](#).

Usage

Message headers evaluated by the CW producer

| Header | Type | Description |
|---|---------------|--|
| <code>CamelAwsCwMetricName</code> | String | The Amazon CW metric name. |
| <code>CamelAwsCwMetricValue</code> | Double | The Amazon CW metric value. |
| <code>CamelAwsCwMetricUnit</code> | String | The Amazon CW metric unit. |
| <code>CamelAwsCwMetricNamespace</code> | String | The Amazon CW metric namespace. |
| <code>CamelAwsCwMetricTimestamp</code> | Date | The Amazon CW metric timestamp. |
| <code>CamelAwsCwMetricDimensionName</code> | String | Camel 2.12: The Amazon CW metric dimension name. |
| <code>CamelAwsCwMetricDimensionValue</code> | String | Camel 2.12: The Amazon CW metric dimension value. |

| | | |
|---|--|---|
| <code>CamelAwsCwMetricDimensions</code> | <code>Map<String, String></code> | Camel 2.12: A map of dimension names and dimension values. |
|---|--|---|

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://namespace?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);
```

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).

- [AWS Component](#)

8.3. AWS-DDB

DDB Component

Available as of Camel 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](#).

URI Format

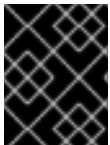
```
aws-ddb://domainName[?options]
```

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

URI Options

| Name | Default Value | Context | Description |
|-------------------|---------------|----------|---|
| amazonDDBClient | null | Producer | Reference to a com.amazonaws.services.dynamodb . AmazonDynamoDB in the Registry . |
| accessKey | null | Producer | Amazon AWS Access Key |
| secretKey | null | Producer | Amazon AWS Secret Key |
| amazonDdbEndpoint | null | Producer | The region with which the AWS-DDB client wants to work with. |
| tableName | null | Producer | The name of the table currently worked with. |
| readCapacity | 0 | Producer | The provisioned throughput to reserve for reading resources from your table |
| writeCapacity | 0 | Producer | The provisioned throughput to reserved for writing resources to your table |

| | | | |
|----------------|----------------------|----------|---|
| consistentRead | false | Producer | Determines whether or not strong consistency should be enforced when data is read. |
| operation | PutAttributes | Producer | Valid values are BatchGetItems, DeleteItem, DeleteTable, DescribeTable, GetItem, PutItem, Query, Scan, UpdateItem, UpdateTable. |



REQUIRED DDB COMPONENT OPTIONS

You have to provide the `amazonDDBClient` in the [Registry](#) or your `accessKey` and `secretKey` to access the [Amazon's DynamoDB](#).

Usage

Message headers evaluated by the DDB producer

| Header | Type | Description |
|---|--|---|
| <code>CamelAwsDdbBatchItems</code> | <code>Map<String, KeysAndAttributes></code> | A map of the table name and corresponding items to get by primary key. |
| <code>CamelAwsDdbTableName</code> | <code>String</code> | Table Name for this operation. |
| <code>CamelAwsDdbKey</code> | <code>Key</code> | The primary key that uniquely identifies each item in a table. |
| <code>CamelAwsDdbReturnValues</code> | <code>String</code> | 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). |
| <code>CamelAwsDdbUpdateCondition</code> | <code>Map<String, ExpectedAttributeValue></code> | Designates an attribute for a conditional modification. |
| <code>CamelAwsDdbAttributeNames</code> | <code>Collection<String></code> | 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. |
| 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. |
| 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. |
| 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. |
| 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. |

Message headers set during BatchGetItems operation

| Header | Type | Description |
|--------|------|-------------|
|--------|------|-------------|

| | | |
|---|---|---|
| <code>CamelAwsDdbBatchResponse</code> | <code>Map<String, BatchResponse></code> | Table names and the respective item attributes from the tables. |
| <code>CamelAwsDdbUnprocessedKeys</code> | <code>Map<String, KeysAndAttributes></code> | Contains a map of tables and their respective keys that were not processed with the current response. |

Message headers set during DeleteItem operation

| Header | Type | Description |
|------------------------------------|--|---|
| <code>CamelAwsDdbAttributes</code> | <code>Map<String, AttributeValue></code> | The list of attributes returned by the operation. |

Message headers set during DeleteTable operation

| Header | Type | Description |
|---|---|--|
| <code>CamelAwsDdbProvisionedThroughput</code> | <code>ProvisionedThroughputDescription</code> | The value of the ProvisionedThroughput property for this table |
| <code>CamelAwsDdbCreationDate</code> | <code>Date</code> | Creation DateTime of this table. |
| <code>CamelAwsDdbTableItemCount</code> | <code>Long</code> | Item count for this table. |
| <code>CamelAwsDdbKeySchema</code> | <code>KeySchema</code> | The KeySchema that identifies the primary key for this table. |
| <code>CamelAwsDdbTableName</code> | <code>String</code> | The table name. |
| <code>CamelAwsDdbTableSize</code> | <code>Long</code> | The table size in bytes. |
| <code>CamelAwsDdbTableStatus</code> | <code>String</code> | The status of the table: CREATING, UPDATING, DELETING, ACTIVE |

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. |
| CamelAwsDdbTableItemCount | Long | Item count for this table. |
| CamelAwsDdbKeySchema | {{KeySchema | The KeySchema that identifies the primary key for this table. |
| 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. |

Message headers set during GetItem operation

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

Message headers set during PutItem operation

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

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. |
| CamelAwsDdbCount | Integer | Number of items in the response. |

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. |

Message headers set during UpdateItem operation

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

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);
```

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).

- [AWS Component](#)

8.4. AWS-S3

S3 Component

Available as of Camel 2.8

The S3 component supports storing and retrieving objects 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](#).

URI Format

```
aws-s3://bucket-name[?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&...

URI Options

| Name | Default Value | Context | Description |
|------------------|---------------|----------|---|
| amazonS3Client | null | Shared | Reference to a com.amazonaws.services.sqs.AmazonS3 in the Registry . |
| accessKey | null | Shared | Amazon AWS Access Key |
| secretKey | null | Shared | Amazon AWS Secret Key |
| amazonS3Endpoint | null | Shared | The region with which the AWS-S3 client wants to work with. |
| region | null | Producer | The region who the bucket is located. This option is used in the com.amazonaws.services.s3.model.CreateBucketRequest . |
| deleteAfterRead | true | Consumer | Delete objects from S3 after it has been retrieved. |
| deleteAfterWrite | false | Producer | Camel 2.11.0 Delete file object after the S3 file has been uploaded |

| | | | |
|--------------------|-------------|----------|---|
| maxMessagesPerPoll | 10 | Consumer | The maximum number of objects which can be retrieved in one poll. Used in in the com.amazonaws.services.s3.model.ListObjectsRequest . |
| policy | null | Shared | *Camel 2.8.4*: The policy for this queue to set in the com.amazonaws.services.s3.AmazonS3#setBucketPolicy() method. |
| storageClass | null | Producer | *Camel 2.8.4*: The storage class to set in the com.amazonaws.services.s3.model.PutObjectRequest request. |
| prefix | null | Consumer | *Camel 2.10.1*: The prefix which is used in the com.amazonaws.services.s3.model.ListObjectsRequest to only consume objects we are interested in. |



REQUIRED S3 COMPONENT OPTIONS

You have to provide the `amazonS3Client` in the [Registry](#) or your `accessKey` and `secretKey` to access the [Amazon's S3](#).

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.

Usage

Message headers evaluated by the S3 producer

| Header | Type | Description |
|---|--|--|
| <code>CamelAwsS3Key</code> | <code>String</code> | The key under which this object will be stored. |
| <code>CamelAwsS3ContentLength</code> | <code>Long</code> | The content length of this object. |
| <code>CamelAwsS3ContentType</code> | <code>String</code> | The content type of this object. |
| <code>CamelAwsS3ContentControl</code> | <code>String</code> | The content control of this object. |
| <code>CamelAwsS3ContentDisposition</code> | <code>String</code> | The content disposition of this object. |
| <code>CamelAwsS3ContentEncoding</code> | <code>String</code> | The content encoding of this object. |
| <code>CamelAwsS3ContentMD5</code> | <code>String</code> | The md5 checksum of this object. |
| <code>CamelAwsS3LastModified</code> | <code>java.util.Date</code> | The last modified timestamp of this object. |
| <code>CamelAwsS3StorageClass</code> | <code>String</code> | *Camel 2.8.4:* The storage class of this object. |
| <code>CamelAwsS3CannedAcl</code> | <code>String</code> | Camel 2.11.0: The canned acl that will be applied to the object. see <code>com.amazonaws.services.s3.model.CannedAccessControlList</code> for allowed values. |
| <code>CamelAwsS3Acl</code> | <code>com.amazonaws.services.s3.model.AccessControlList</code> | Camel 2.11.0: a well constructed Amazon S3 Access Control List object. see <code>com.amazonaws.services.s3.model.AccessControlList</code> for more details |

Message headers set by the S3 producer

| Header | Type | Description |
|--------|------|-------------|
|--------|------|-------------|

| | | |
|----------------------------------|---------------|--|
| <code>CamelAwsS3ETag</code> | String | The ETag value for the newly uploaded object. |
| <code>CamelAwsS3VersionId</code> | String | The optional version ID of the newly uploaded object. |

Message headers set by the S3 consumer

| Header | Type | Description |
|-------------------------------------|---------------|--|
| <code>CamelAwsS3Key</code> | String | The key under which this object is stored. |
| <code>CamelAwsS3BucketName</code> | String | The name of the bucket in which this object is contained. |
| <code>CamelAwsS3ETag</code> | 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. |
| <code>CamelAwsS3LastModified</code> | 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. |
| <code>CamelAwsS3VersionId</code> | 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. |
| <code>CamelAwsS3ContentType</code> | 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. |

| | | |
|---|---------------------|---|
| <code>CamelAwsS3ContentMD5</code> | <code>String</code> | 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. |
| <code>CamelAwsS3ContentLength</code> | <code>Long</code> | The Content-Length HTTP header indicating the size of the associated object in bytes. |
| <code>CamelAwsS3ContentEncoding</code> | <code>String</code> | 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. |
| <code>CamelAwsS3ContentDisposition</code> | <code>String</code> | The optional Content-Disposition HTTP header, which specifies presentational information such as the recommended filename for the object to be saved as. |
| <code>CamelAwsS3ContentControl</code> | <code>String</code> | The optional Cache-Control HTTP header which allows the user to specify caching behavior along the HTTP request/reply chain. |

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");
```

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).

- [AWS Component](#)

8.5. AWS-SDB

SDB Component

Available as of Camel 2.8.4

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](#).

URI Format

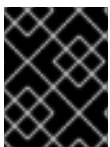
```
aws-sdb://domainName[?options]
```

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

URI Options

| Name | Default Value | Context | Description |
|------|---------------|---------|-------------|
|------|---------------|---------|-------------|

| | | | |
|--------------------|----------------------|----------|--|
| amazonSDBClient | null | Producer | Reference to a <code>com.amazonaws.services.simpledb</code> . <code>AmazonSimpleDB</code> in the Registry . |
| accessKey | null | Producer | Amazon AWS Access Key |
| secretKey | null | Producer | Amazon AWS Secret Key |
| amazonSdbEndpoint | null | Producer | The region with which the AWS-SDB client wants to work with. |
| domainName | null | Producer | The name of the domain currently worked with. |
| maxNumberOfDomains | 100 | Producer | The maximum number of domain names you want returned. The range is 1 * to 100. |
| consistentRead | false | Producer | Determines whether or not strong consistency should be enforced when data is read. |
| operation | PutAttributes | Producer | Valid values are BatchDeleteAttributes, BatchPutAttributes, DeleteAttributes, DeleteDomain, DomainMetadata, GetAttributes, ListDomains, PutAttributes, Select. |



REQUIRED SDB COMPONENT OPTIONS

You have to provide the `amazonSDBClient` in the [Registry](#) or your `accessKey` and `secretKey` to access the [Amazon's SDB](#).

Usage

Message headers evaluated by the SDB producer

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

Message headers set during DomainMetadata operation

| Header | Type | Description |
|---|---------|--|
| <code>CamelAwsSdbTimestamp</code> | Integer | The data and time when metadata was calculated, in Epoch (UNIX) seconds. |
| <code>CamelAwsSdbItemCount</code> | Integer | The number of all items in the domain. |
| <code>CamelAwsSdbAttributeNameCount</code> | Integer | The number of unique attribute names in the domain. |
| <code>CamelAwsSdbAttributeValueCount</code> | Integer | The number of all attribute name/value pairs in the domain. |
| <code>CamelAwsSdbAttributeNameSize</code> | Long | The total size of all unique attribute names in the domain, in bytes. |
| <code>CamelAwsSdbAttributeValueSize</code> | Long | The total size of all attribute values in the domain, in bytes. |
| <code>CamelAwsSdbItemNameSize</code> | Long | The total size of all item names in the domain, in bytes. |

Message headers set during GetAttributes operation

| Header | Type | Description |
|------------------------------------|-----------------|---|
| <code>CamelAwsSdbAttributes</code> | List<Attribute> | The list of attributes returned by the operation. |

Message headers set during ListDomains operation

| Header | Type | Description |
|-------------------------------------|--------------|---|
| <code>CamelAwsSdbDomainNames</code> | 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 <code>MaxNumberOfDomains</code> still available. |
|-----------------------------|---------------|--|

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 <code>MaxNumberOfItems</code> were matched, the response size exceeded 1 megabyte, or the execution time exceeded 5 seconds. |

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);
```

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).

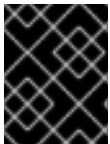
- [AWS Component](#)

8.6. AWS-SES

SES Component

Available as of Camel 2.8.4

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](#).

URI Format

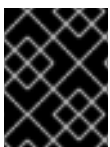
```
aws-ses://from[?options]
```

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

URI Options

| Name | Default Value | Context | Description |
|-----------------|---------------|----------|---|
| amazonSESClient | null | Producer | Reference to a <code>com.amazonaws.services.simpleemail.AmazonSimpleEmailService</code> in the Registry . |
| accessKey | null | Producer | Amazon AWS Access Key |
| secretKey | null | Producer | Amazon AWS Secret Key |

| | | | |
|-------------------|-------------|----------|---|
| amazonSESEndpoint | null | Producer | The region with which the AWS-SES client wants to work with. |
| subject | null | Producer | The subject which is used if the message header 'CamelAwsSesSubject' is not present. |
| to | null | Producer | List of destination email address. Can be overridden with 'CamelAwsSesTo' header. |
| returnPath | null | Producer | The email address to which bounce notifications are to be forwarded, override it using 'CamelAwsSesReturn Path' header. |
| replyToAddresses | null | Producer | List of reply-to email address(es) for the message, override it using 'CamelAwsSesReplyToAddresses' header. |



REQUIRED SES COMPONENT OPTIONS

You have to provide the `amazonSESClient` in the [Registry](#) or your `accessKey` and `secretKey` to access the [Amazon's SES](#).

Usage

Message headers evaluated by the SES producer

| Header | Type | Description |
|---------------------------------|---------------------------------|------------------------------------|
| <code>CamelAwsSesFrom</code> | <code>String</code> | The sender's email address. |
| <code>CamelAwsSesTo</code> | <code>List<String></code> | The destination(s) for this email. |
| <code>CamelAwsSesSubject</code> | <code>String</code> | The subject of the message. |

| | | |
|--|---------------------------------|--|
| <code>CamelAwsSesReplyToAddresses</code> | <code>List<String></code> | The reply-to email address(es) for the message. |
| <code>CamelAwsSesReturnPath</code> | <code>String</code> | The email address to which bounce notifications are to be forwarded. |

Message headers set by the SES producer

| Header | Type | Description |
|-----------------------------------|---------------------|----------------------------|
| <code>CamelAwsSesMessageId</code> | <code>String</code> | The Amazon SES message ID. |

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);
```

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).

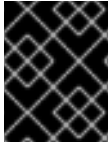
- [AWS Component](#)

8.7. AWS-SNS

SNS Component

Available as of Camel 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](#).

URI Format

```
aws-sns://topicName[?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&...**

URI Options

| Name | Default Value | Context | Description |
|-------------------|---------------|----------|--|
| amazonSNSClient | null | Producer | Reference to a com.amazonaws.services.sns.AmazonSNS in the Registry . |
| accessKey | null | Producer | Amazon AWS Access Key |
| secretKey | null | Producer | Amazon AWS Secret Key |
| subject | null | Producer | The subject which is used if the message header 'CamelAwsSnsSubject' is not present. |
| amazonSNSEndpoint | null | Producer | The region with which the AWS-SNS client wants to work with. |

| | | | |
|--------|------|----------|---|
| policy | null | Producer | Camel 2.8.4: The policy for this queue to set in the <code>com.amazonaws.services.sns.model.SetTopicAttributesRequest</code> . |
|--------|------|----------|---|



REQUIRED SNS COMPONENT OPTIONS

You have to provide the `amazonSNSClient` in the [Registry](#) or your `accessKey` and `secretKey` to access the [Amazon's SNS](#).

Usage

Message headers evaluated by the SNS producer

| Header | Type | Description |
|---------------------------------|--------|---|
| <code>CamelAwsSnsSubject</code> | String | The Amazon SNS message subject. If not set, the subject from the <code>SnsConfiguration</code> is used. |

Message headers set by the SNS producer

| Header | Type | Description |
|-----------------------------------|--------|----------------------------|
| <code>CamelAwsSnsMessageId</code> | String | The Amazon SNS message ID. |

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);
```

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).

- [AWS Component](#)

8.8. AWS-SQS

SQS Component

Available as of Camel 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](#).

URI Format

```
aws-sqs://queue-name[?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&...`

URI Options

| Name | Default Value | Context | Description |
|-----------------|---------------|---------|--|
| amazonSQSClient | null | Shared | Reference to a <code>com.amazonaws.services.sqs.AmazonSQS</code> in the Registry . |

| | | | |
|--------------------------|-------------|----------|--|
| accessKey | null | Shared | Amazon AWS Access Key |
| secretKey | null | Shared | Amazon AWS Secret Key |
| amazonSQSEndpoint | null | Shared | The region with which the AWS-SQS client wants to work with. |
| attributeNames | null | Consumer | A list of attributes to set in the <code>com.amazonaws.services.sqs.model.ReceiveMessageRequest</code> . |
| defaultVisibilityTimeout | null | Shared | The visibility timeout (in seconds) to set in the <code>com.amazonaws.services.sqs.model.CreateQueueRequest</code> . |
| deleteAfterRead | true | Consumer | Delete message from SQS after it has been read |
| maxMessagesPerPoll | null | Consumer | The maximum number of messages which can be received in one poll to set in the <code>com.amazonaws.services.sqs.model.ReceiveMessageRequest</code> . |

| | | | |
|--------------------------|--------------|----------|--|
| visibilityTimeout | null | Shared | The duration (in seconds) that the received messages are hidden from subsequent retrieve requests after being retrieved by a <code>ReceiveMessage</code> request to set in the <code>com.amazonaws.services.sqs.model.SetQueueAttributesRequest</code> . This only make sense if its different from <code>defaultVisibilityTimeout</code> . It changes the queue visibility timeout attribute permanently. |
| messageVisibilityTimeout | null | Consumer | Camel 2.8: The duration (in seconds) that the received messages are hidden from subsequent retrieve requests after being retrieved by a <code>ReceiveMessage</code> request to set in the <code>com.amazonaws.services.sqs.model.ReceiveMessageRequest</code> . It does NOT change the queue visibility timeout attribute permanently. |
| extendMessageVisibility | false | Consumer | Camel 2.10: If enabled then a scheduled background task will keep extending the message visibility on SQS. This is needed if it taks a long time to process the message. If set to true <code>defaultVisibilityTimeout</code> must be set. See details at Amazon docs . |

| | | | |
|-------------------------------|-------------|----------|---|
| maximumMessageSize | null | Shared | Camel 2.8: The maximumMessageSize (in bytes) an SQS message can contain for this queue, to set in the com.amazonaws.services.sqs.model.SetQueueAttributesRequest . |
| messageRetentionPeriod | null | Shared | Camel 2.8: The messageRetentionPeriod (in seconds) a message will be retained by SQS for this queue, to set in the com.amazonaws.services.sqs.model.SetQueueAttributesRequest . |
| policy | null | Shared | Camel 2.8: The policy for this queue to set in the com.amazonaws.services.sqs.model.SetQueueAttributesRequest . |
| delaySeconds | null | Producer | Camel 2.9.3: Delay sending messages for a number of seconds. |
| waitTimeSeconds | 0 | Producer | Camel 2.11: 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. |
| receiveMessageWaitTimeSeconds | 0 | Shared | Camel 2.11: If you do not specify WaitTimeSeconds in the request, the queue attribute ReceiveMessageWaitTimeSeconds is used to determine how long to wait. |

| | | |
|------------------------|------|--|
| queueOwnerAWSAccountId | null | Camel 2.12: Specify the queue owner aws account id when you need to connect the queue with different account owner. |
|------------------------|------|--|



REQUIRED SQS COMPONENT OPTIONS

You have to provide the `amazonSQSClient` in the [Registry](#) or your `accessKey` and `secretKey` to access the [Amazon's SQS](#).

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.

Usage

Message headers set by the SQS producer

| Header | Type | Description |
|--------------------------------------|---------|---|
| <code>CamelAwsSqsMD5ofBody</code> | String | The MD5 checksum of the Amazon SQS message. |
| <code>CamelAwsSqsMessageId</code> | String | The Amazon SQS message ID. |
| <code>CamelAwsSqsDelaySeconds</code> | Integer | Since Camel 2.11 , the delay seconds that the Amazon SQS message can be see by others. |

Message headers set by the SQS consumer

| Header | Type | Description |
|---------------------------------------|--------|---|
| <code>CamelAwsSqsMD5ofBody</code> | String | The MD5 checksum of the Amazon SQS message. |
| <code>CamelAwsSqsMessageId</code> | String | The Amazon SQS message ID. |
| <code>CamelAwsSqsReceiptHandle</code> | String | The Amazon SQS message receipt handle. |

| | | |
|------------------------------------|--|------------------------------------|
| <code>CamelAwsSqsAttributes</code> | <code>Map<String, String></code> | The Amazon SQS message attributes. |
|------------------------------------|--|------------------------------------|

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");
```

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).

- [AWS Component](#)

CHAPTER 9. BEAN

BEAN COMPONENT

The **bean:** component binds beans to Apache Camel message exchanges.

URI FORMAT

```
bean:beanID[?options]
```

Where **beanID** can be any string which is used to lookup look up the bean in the [Registry](#)

OPTIONS

| Name | Type | Default | Description |
|--------|---------|---------|--|
| method | String | null | The method name from the bean that will be invoked. If not provided, Camel will try to determine the method itself. In case of ambiguity an exception will be thrown. See Bean Binding for more details. |
| cache | boolean | false | If enabled, Apache 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. |

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

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, put the bean in JNDI.

```
// lets populate the context with the services we need
// note that we could just use a spring.xml file to avoid this step
JndiContext context = new JndiContext();
```

```
context.bind("bye", new SayService("Good Bye!"));

CamelContext camelContext = new DefaultCamelContext(context);
```

Once an endpoint has been registered, you can build routes that use it to process exchanges.

```
// lets add simple route
camelContext.addRoutes(new RouteBuilder() {
    public void configure() {
        from("direct:hello").to("bean:bye");
    }
});
```

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:

```
Endpoint endpoint = camelContext.getEndpoint("direct:hello");
ISay proxy = ProxyHelper.createProxy(endpoint, ISay.class);
String rc = proxy.say();
assertEquals("Good Bye!", rc);
```

And the same route using Spring DSL:

```
<route>
  <from uri="direct:hello">
    <to uri="bean:bye"/>
  </route>
```

BEAN AS ENDPOINT

Apache Camel also supports invoking [Bean](#) as an Endpoint. In the route below:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="myBean"/>
    <to uri="mock:results"/>
  </route>
</camelContext>

<bean id="myBean" class="org.apache.camel.spring.bind.ExampleBean"/>
```

What happens is that when the exchange is routed to the **myBean**, Apache Camel will use the [Bean Binding](#) to invoke the bean. The source for the bean is just a plain POJO:

```
public class ExampleBean {

    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}
```



```
}
}
```

Apache 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.

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);
```

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 Apache Camel.

- [Class](#) component
- [Bean Binding](#)
- [Bean Integration](#)

CHAPTER 10. BEAN VALIDATION

BEAN VALIDATION COMPONENT

Available as of Apache Camel 2.3

The Validation 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.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI FORMAT

```
bean-validator:something[?options]
```

or

```
bean-validator://something[?options]
```

Where **something** must be present to provide a valid URL. You can append query options to the URI in the following format, `?option=value&option=value&...`

URI OPTIONS

The following URI options are supported:

| Option | Default | Description |
|----------------------------------|---|---|
| <code>group</code> | <code>javax.validation.groups.Default</code> | The custom validation group to use. |
| <code>messageInterpolator</code> | <code>org.hibernate.validator.engine.ResourceBundleMessageInterpolator</code> | Reference to a custom <code>javax.validation.MessageInterpolator</code> in the Registry . |
| <code>traversableResolver</code> | <code>org.hibernate.validator.engine.resolver.DefaultTraversableResolver</code> | Reference to a custom <code>javax.validation.TraversableResolver</code> in the Registry . |

| | | |
|-----------------------------------|--|--|
| constraintValidatorFactory | org.hibernate.validator.engine.ConstraintValidatorFactoryImpl | Reference to a custom javax.validation.ConstraintValidatorFactory in the Registry . |
|-----------------------------------|--|--|

EXAMPLE

Assumed we have a java bean with the following annotations



CAR.JAVA

```
// 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 Apache Camel route, only the **@NotNull** constraints on the attributes `manufacturer` and `licensePlate` will be validated (Apache 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 look 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&traversableReso
lver=#myTraversableResolver&constraintValidatorFactory=#myConstraintValida
torFactory")
.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 look 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/c
onfiguration">
  <default-
provider>org.hibernate.validator.HibernateValidator</default-
provider>
  <message-
interpolator>org.hibernate.validator.engine.ResourceBundleMessa
geInterpolator</message-interpolator>
  <traversable-
resolver>org.hibernate.validator.engine.resolver.DefaultTravers
ableResolver</traversable-resolver>
  <constraint-validator-
factory>org.hibernate.validator.engine.ConstraintValidatorFacto
ryImpl</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/m
  apping 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>
```

CHAPTER 11. BROWSE

BROWSE COMPONENT

Available as of Apache Camel 2.0

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.

URI FORMAT

```
browse:someName
```

Where **someName** can be any string to uniquely identify the endpoint.

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();
        ...
    }
}
```

CHAPTER 12. CACHE

12.1. CACHE COMPONENT

Available as of Camel 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>
```

URI format

```
cache://cacheName[?options]
```

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

Options

The Cache component supports the following options:

| Name | Default Value | Description |
|----------------------------|---------------|--|
| maxElementsInMemory | 1000 | The number of elements that may be stored in the defined cache |

| | | |
|--|--|--|
| <code>memoryStoreEvictionPolicy</code> | <code>MemoryStoreEvictionPolicy.LFU</code> | <p>The number of elements that may be stored in the defined cache. Options include</p> <ul style="list-style-type: none"> • <code>MemoryStoreEvictionPolicy.LFU</code> - Least frequently used • <code>MemoryStoreEvictionPolicy.LRU</code> - Least recently used • <code>MemoryStoreEvictionPolicy.FIFO</code> - first in first out, the oldest element by creation time |
| <code>overflowToDisk</code> | <code>true</code> | Specifies whether cache may overflow to disk |
| <code>eternal</code> | <code>false</code> | Sets whether elements are eternal. If eternal, timeouts are ignored and the element never expires. |
| <code>timeToLiveSeconds</code> | <code>300</code> | The maximum time between creation time and when an element expires. Is used only if the element is not eternal |
| <code>timeToIdleSeconds</code> | <code>300</code> | The maximum amount of time between accesses before an element expires |
| <code>diskPersistent</code> | <code>false</code> | Whether the disk store persists between restarts of the Virtual Machine. |
| <code>diskExpiryThreadIntervalSeconds</code> | <code>120</code> | The number of seconds between runs of the disk expiry thread. |
| <code>cacheManagerFactory</code> | <code>null</code> | <p>*Camel 2.8:* If you want to use a custom factory which instantiates and creates the <code>EHCache</code></p> <p><code>net.sf.ehcache.CacheManager</code>. <i>Type:</i> <code>abstract org.apache.camel.component.cache.CacheManagerFactory</code></p> |

| | | |
|------------------------------|-------------|--|
| eventListenerRegistry | null | *Camel 2.8:* Sets a list of EHCACHE net.sf.ehcache.event.CacheEventListener for all new caches\ - no need to define it per cache in EHCACHE xml config anymore. <i>Type:</i> org.apache.camel.component.cache.CacheEventListenerRegistry |
| cacheLoaderRegistry | null | *Camel 2.8:* Sets a list of org.apache.camel.component.cache.CacheLoaderWrapper that extends EHCACHE net.sf.ehcache.loader.CacheLoader for all new caches\ - no need to define it per cache in EHCACHE xml config anymore. <i>Type:</i> org.apache.camel.component.cache.CacheLoaderRegistry |
| key | null | *Camel 2.10:* To configure using a cache key by default. If a key is provided in the message header, then the key from the header takes precedence. |
| operation | null | *Camel 2.10:* To configure using an cache operation by default. If an operation in the message header, then the operation from the header takes precedence. |

Message Headers Camel 2.8\+

| Header | Description |
|--------|-------------|
|--------|-------------|

| | |
|----------------------------|--|
| CamelCacheOperation | The operation to be performed on the cache. The valid options are <ul style="list-style-type: none"> • 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 |



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.

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. |

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

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.

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

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")
```

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")
    }
};
```

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")
    }
};

```

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")
    }
};

```

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");
    }
};

```

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
                }
            })
    }
};

```

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");
    }
};

```

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");

```

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();
```

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.

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.

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](#).

12.2. CACHEREPLICATIONJMSEXAMPLE

Example: JMS cache replication



NOTE

Please note, that this example is not finished yet. It is based on OSGi iTest instead of real life example. But no matter to that it is very good starting point for all Camel Cache Riders!

JMS replication is the most powerful and secured way. Used altogether with Camel Cache replication options is also the most easy way. This basic example is divided to few important steps that have to be made to get the cache replication to work.

The first step is to write your own implementation of `CacheManagerFactory`.

```
public class TestingCacheManagerFactory extends CacheManagerFactory {
```

```

[...]

    //This constructor is very useful when using Camel with
    Spring/Blueprint
    public TestingCacheManagerFactory(String xmlName,
        TopicConnection replicationTopicConnection, Topic
    replicationTopic,
        QueueConnection getQueueConnection, Queue getQueue) {
        this.xmlName = xmlName;
        this.replicationTopicConnection = replicationTopicConnection;
        this.replicationTopic = replicationTopic;
        this.getQueue = getQueue;
        this.getQueueConnection = getQueueConnection;
    }

    @Override
    protected synchronized CacheManager createCacheManagerInstance() {
        //Singleton
        if (cacheManager == null) {
            cacheManager = new
    WrappedCacheManager(getClass().getResourceAsStream(xmlName));
        }

        return cacheManager;
    }

    //Wrapping Ehcache's CacheManager to be able to add
    JMSCacheManagerPeerProvider
    public class WrappedCacheManager extends CacheManager {
        public WrappedCacheManager(InputStream xmlConfig) {
            super(xmlConfig);
            JMSCacheManagerPeerProvider jmsCMPP = new
    JMSCacheManagerPeerProvider(this,
                replicationTopicConnection,
                replicationTopic,
                getQueueConnection,
                getQueue,
                AcknowledgementMode.AUTO_ACKNOWLEDGE,
                true);
            cacheManagerPeerProviders.put(jmsCMPP.getScheme(), jmsCMPP);
            jmsCMPP.init();
        }
    }
}

```

Next step is to write your own implementation of **CacheLoaderWrapper**, the easiest one is:

```

public class WrappedJMSCacheLoader implements CacheLoaderWrapper {

    [...]

    //This constructor is very useful when using Camel with
    Spring/Blueprint
    public WrappedJMSCacheLoader(QueueConnection getQueueConnection,
        Queue getQueue, AcknowledgementMode acknowledgementMode,

```



```

        int timeoutMillis) {
            this.getQueueConnection = getQueueConnection;
            this.getQueue = getQueue;
            this.acknowledgementMode = acknowledgementMode;
            this.timeoutMillis = timeoutMillis;
        }

        @Override
        public void init(Ehcache cache) {
            jmsCacheLoader = new JMSCacheLoader(cache, defaultLoaderArgument,
                getQueueConnection, getQueue, acknowledgementMode,
                timeoutMillis);
        }

        @Override
        public CacheLoader clone(Ehcache arg0) throws
CloneNotSupportedException {
            return jmsCacheLoader.clone(arg0);
        }

        @Override
        public void dispose() throws CacheException {
            jmsCacheLoader.dispose();
        }

        [...]
    }

```

At the third step you can take care about Camel Cache options (prepare their values):

- cacheManagerFactory
- eventListenerRegistry
- cacheLoaderRegistry

```

<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">

    <bean id="queueConnection1" factory-bean="amqCF" factory-
method="createQueueConnection" class="javax.jms.QueueConnection" />
    <bean id="topicConnection1" factory-bean="amqCF" factory-
method="createTopicConnection" class="javax.jms.TopicConnection" />
    <bean id="queue1" class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg ref="getQueue" />
    </bean>
    <bean id="topic1" class="org.apache.activemq.command.ActiveMQTopic">
        <constructor-arg ref="getTopic" />
    </bean>

```

```

    <bean id="jmsListener1"
class="net.sf.ehcache.distribution.jms.JMSCacheReplicator">
    <constructor-arg index="0" value="true" />
    <constructor-arg index="1" value="true" />
    <constructor-arg index="2" value="true" />
    <constructor-arg index="3" value="true" />
    <constructor-arg index="4" value="false" />
    <constructor-arg index="5" value="0" />
</bean>

    <bean id="jmsLoader1"
class="my.cache.replication.WrappedJMSCacheLoader">
    <constructor-arg index="0" ref="queueConnection1" />
    <constructor-arg index="1" ref="queue1" />
    <constructor-arg index="2" value="AUTO_ACKNOWLEDGE" />
    <constructor-arg index="3" value="30000" />
</bean>

    <bean id="cacheManagerFactory1"
class="my.cache.replication.TestingCacheManagerFactory">
    <constructor-arg index="0" value="ehcache_jms_test.xml" />
    <constructor-arg index="1" ref="topicConnection1" />
    <constructor-arg index="2" ref="topic1" />
    <constructor-arg index="3" ref="queueConnection1" />
    <constructor-arg index="4" ref="queue1" />
</bean>

    <bean id="eventListenerRegistry1"
class="org.apache.camel.component.cache.CacheEventListenerRegistry">
    <constructor-arg>
    <list>
    <ref bean="jmsListener1" />
    </list>
    </constructor-arg>
</bean>

    <bean id="cacheLoaderRegistry1"
class="org.apache.camel.component.cache.CacheLoaderRegistry">
    <constructor-arg>
    <list>
    <ref bean="jmsLoader1"/>
    </list>
    </constructor-arg>
</bean>
</beans>

```

The final step is to define some routes using Cache component

```

<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">

    <bean id="getQueue" class="java.lang.String">
        <constructor-arg value="replicationGetQueue" />
    </bean>

    <bean id="getTopic" class="java.lang.String">
        <constructor-arg value="replicationTopic" />
    </bean>

    <!-- Import the xml file explained at step three -->
    <import resource="JMSReplicationCache1.xml"/>

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <camel:endpoint id="fooCache1" uri="cache:foo?
cacheManagerFactory=#cacheManagerFactory1&ventListenerRegistry=#eventListe
nerRegistry1&acheLoaderRegistry=#cacheLoaderRegistry1"/>

        <camel:route>
            <camel:from uri="direct:addRoute"/>
            <camel:setHeader headerName="CamelCacheOperation">
                <camel:constant>CamelCacheAdd</camel:constant>
            </camel:setHeader>
            <camel:setHeader headerName="CamelCacheKey">
                <camel:constant>foo</camel:constant>
            </camel:setHeader>
            <camel:to ref="fooCache1"/>
        </camel:route>

    </camelContext>

    <bean id="amqCF"
class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="vm://localhost?
broker.persistent=false"/>
    </bean>

    <bean id="activemq"
class="org.apache.camel.component.jms.JmsComponent">
        <property name="connectionFactory">
            <ref bean="amqCF"/>
        </property>
    </bean>

</beans>

```

CHAPTER 13. CDI

CAMEL CDI

As of 2.10 we now have support [Contexts and Dependency Injection - JSR299](#) and [Dependency Injection for Java - JSR330](#) as a dependency injection framework. This offers new opportunities to develop and deploy Apache Camel projects in [Java EE 6 containers](#) but also in standalone Java SE or [CDI container](#)

The current project is under active development and does not provide all the features that we have with injection frameworks like Spring or Blueprint

DEPENDENCY INJECTING CAMEL WITH CDI

Basically, two things should be done to use Apache Camel in a CDI environment. First, we just need to create a [BootStrap](#) class which will be use by the Java EE 6 container or Java SE to start the Camel Context. The [CdiCamelContext](#) when instantiated will add a [CDIBean Registry](#). That will allow Camel to perform lookup of beans injected and registered in CDI container. Next, we must add CDI annotated beans ([@inject](#), [@named](#), ...) to use them from the Apache Camel routes.

BOOTSTRAPPING CAMEL WITH CDI CONTAINER

The following example shows how we can bootstrap an Apache Camel Context in a Boot Strap class. This class contains important annotations like the [javax.ejb.Singleton](#). This annotation will tell the container to create a Singleton instance of the BootStrapClass class. This mechanism is similar to Bean instance creation that we have with Spring framework. By combining this annotation with [javax.ejb.Startup](#), the container will start the camel context at the startup of the CDI container.

```
@Singleton
@Startup
public class BootStrap {
    ...
}
```

When the [@PreConstruct](#) annotation is called, then we inject a [CdiCamelContext](#) objet, register a [SimpleCamelRoute](#) using [@Inject](#) annotation and starts the Camel Route.

```
@PostConstruct
public void init() throws Exception {
    logger.info(">> Create CamelContext and register Camel
Route.");

    // Define Timer URI
    simpleRoute.setTimerUri("timer://simple?
fixedRate=true&period=10s");

    // Add Camel Route
    camelCtx.addRoutes(simpleRoute);

    // Start Camel Context
    camelCtx.start();
}
```

When you look to the following Camel Route code, you can see that we do a lookup to find a bean "helloWorld" which has been injected. This is possible because the CdiCamelContext registers a Camel Registry containing a reference to a CDI BeanManager.

```
@Override
public void configure() throws Exception {

    from(timerUri)
        .setBody()
            .simple("Bean Injected")

        // Lookup for bean injected by CDI container
        // The HelloWorld class is annotated using @Named
        .beanRef("helloWorld", "sayHello")

        .log(">> Response : ${body}");

}
```

Here is the code of the HelloWorld Bean

```
@Named
public class HelloWorld {

    public String sayHello(@Body String message) {
        return ">> Hello " + message + " user.";
    }

}
```

This project is started using the [GlassFish maven plugin](#) but alternatively, you can deploy the war file produced in any Java EE 6 servers : Glassfish, JBoss AS 7, OpenEJB, Apache TomEE or Apache KarafEE or using a [Java SE](#).

SEE ALSO

- Simple [Camel CDI BootStrap project](#)
- [JSR299](#) and [JSR330](#) reference documentations
- [Apache DeltaSpike project](#) - CDI extensions and JavaSE BootStrap
- CDI revealed by Antonio Goncalves - [part 1](#), [part 2](#), [part 3](#) and OpenEJB team - see [examples](#)
- Apache implementation of the specs JSR299, 330 - [OpenWebbeans](#) and Apache [OpenEJB](#) which provide the container to deploy CDI projects
- Apache Karaf featured with OpenEJB and CDI - [Apache KarafEE](#)

CHAPTER 14. CLASS

CLASS COMPONENT

Available as of Apache Camel 2.4

The **class**: component binds beans to 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.

URI FORMAT

```
class:className[?options]
```

Where **className** is the fully qualified class name to create and use as bean.

OPTIONS

| Name | Type | Default | Description |
|----------------------------|----------------|--------------|--|
| method | String | null | The method name that bean will be invoked. If not provided, Apache Camel will try to pick the method itself. In case of ambiguity an exception is thrown. See Bean Binding for more details. |
| multiParameterArray | boolean | false | How to treat the parameters which are passed from the message body; if it is true , the In message body should be an array of parameters. |

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

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");
```

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:

```
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](#).

```
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.



NOTE

See more details at the Bean component as the **class** component works in much the same way.

- [Bean](#)
- [Bean Binding](#)
- [Bean Integration](#)

CHAPTER 15. CMIS

CMIS COMPONENT

Available as of Camel 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.

URI FORMAT

```
cmis://cmisServerUrl[?options]
```

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

URI OPTIONS

| Name | Default Value | Context | Description |
|--------------|---------------|----------|---|
| queryMode | false | Producer | If true, will execute the cmis query from the message body and return result, otherwise will create a node in the cmis repository |
| query | String | 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 |
| username | null | Both | Username for the cmis repository |
| password | null | Both | Password for the cmis repository |
| repositoryId | null | Both | The Id of the repository to use. If not specified the first available repository is used |

| | | | |
|-------------|--------------|------|--|
| pageSize | 100 | Both | Number of nodes to retrieve per page |
| readCount | 0 | Both | Max number of nodes to read |
| readContent | false | Both | If set to true, the content of document node will be retrieved in addition to the properties |

USAGE

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:objectId | null | The type of the node |
| cmis:contentStreamMimeType | null | The mimetype to set for a document |

MESSAGE HEADERS SET DURING QUERYING PRODUCER OPERATION

| Header | Type | Description |
|----------------------|---------|--|
| CamelCMISResultCount | Integer | Number of nodes returned from the query. |



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).

CHAPTER 16. COMETD

COMETD COMPONENT

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 Apache Camel messages directly into the browser using an AJAX based mechanism.

URI FORMAT

```
cometd://host:port/channelName[?options]
```

The **channelName** represents a topic that can be subscribed to by the Apache Camel endpoints.

EXAMPLES

```
cometd://localhost:8080/service/mychannel
cometds://localhost:8443/service/mychannel
```

where **cometds**: represents an SSL configured endpoint.

OPTIONS

| Name | Default Value | Description |
|---------------------|---------------|--|
| resourceBase | | 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 |
| baseResource | | Camel 2.7: 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 |

| | | |
|----------------------------|---------------|--|
| timeout | 240000 | The server side poll timeout in milliseconds. This is how long the server will hold a reconnect request before responding. |
| interval | 0 | The client side poll timeout in milliseconds. How long a client will wait between reconnects |
| maxInterval | 30000 | The max client side poll timeout in milliseconds. A client will be removed if a connection is not received in this time. |
| multiFrameInterval | 1500 | The client side poll timeout, if multiple connections are detected from the same browser. |
| jsonCommented | true | 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. |
| logLevel | 1 | 0 =none, 1 =info, 2 =debug. |
| sslContextParameters | | Camel 2.9: 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. |
| crossOriginFilterOn | false | Camel 2.10: If true , the server will support for cross-domain filtering |
| allowedOrigins | * | Camel 2.10: The origins domain that support to cross, if the crosssOriginFilterOn is true |

| | | |
|-------------------------------|-------------|---|
| filterPath | | Camel 2.10: The filterPath will be used by the CrossOriginFilter, if the crosssOriginFilterOn is true |
| disconnectLocalSession | true | Camel 2.10.5/2.11.1: (Producer only): 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. |

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

Here is some examples of how to pass the parameters.

For file (when the Webapp resources are located in the Web Application directory) **cometd://localhost:8080?resourceBase=file:./webapp**. For classpath (when the web resources are packaged inside the Webapp folder) **cometd://localhost:8080?resourceBase=classpath:webapp**.

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](#)

SETTING UP SSL FOR COMETD COMPONENT

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.

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 cometdComponent = getContext().getComponent("cometds",
CometdComponent.class);
cometdComponent.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>...
...
<to uri="cometds://127.0.0.1:443/service/test?
baseResource=file:./target/test-
classes/webapp&timeout=240000&interval=0&maxInterval=30000&multiFrameInter
val=1500&jsonCommented=true&logLevel=2&sslContextParameters=#sslContextPar
ameters"/>...
```

CHAPTER 17. CONTEXT

CONTEXT COMPONENT

Available as of Camel 2.7

The **context** component allows you to create new Camel Components from a CamelContext with a number of routes which is then treated as a black box, allowing you to refer to the local endpoints within the component from other CamelContexts.

It is similar to the [Routebox](#) component in idea, though the Context component tries to be really simple for end users; just a simple convention over configuration approach to refer to local endpoints inside the CamelContext 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-context</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI FORMAT

```
context:camelContextId:localEndpointName[?options]
```

Or you can omit the "context:" prefix.

```
camelContextId:localEndpointName[?options]
```

- **camelContextId** is the ID you used to register the CamelContext into the [Registry](#).
- **localEndpointName** can be a valid Camel URI evaluated within the black box CamelContext. Or it can be a logical name which is mapped to any local endpoints. For example if you locally have endpoints like **direct:invoices** and **seda:purchaseOrders** inside a CamelContext of id **supplyChain**, then you can just use the URIs **supplyChain:invoices** or **supplyChain:purchaseOrders** to omit the physical endpoint kind and use pure logical URIs.

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

EXAMPLE

In this example we'll create a black box context, then we'll use it from another CamelContext.

DEFINING THE CONTEXT COMPONENT

First you need to create a CamelContext, add some routes in it, start it and then register the CamelContext into the [Registry](#) (JNDI, Spring, Guice or OSGi etc).

This can be done in the usual Camel way from this [test case](#) (see the createRegistry() method); this example shows Java and JNDI being used...

```
// lets create our black box as a camel context and a set of routes
DefaultCamelContext blackBox = new DefaultCamelContext(registry);
blackBox.setName("blackBox");
blackBox.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        // receive purchase orders, lets process it in some way then send
an invoice
        // to our invoice endpoint
        from("direct:purchaseOrder").
            setHeader("received").constant("true").
            to("direct:invoice");
    }
});
blackBox.start();

registry.bind("accounts", blackBox);
```

Notice in the above route we are using pure local endpoints (**direct** and **seda**). Also note we expose this CamelContext using the **accounts** ID. We can do the same thing in Spring via

```
<camelContext id="accounts" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:purchaseOrder"/>
    ...
    <to uri="direct:invoice"/>
  </route>
</camelContext>
```

USING THE CONTEXT COMPONENT

Then in another CamelContext we can then refer to this "accounts black box" by just sending to **accounts:purchaseOrder** and consuming from **accounts:invoice**.

If you prefer to be more verbose and explicit you could use **context:accounts:purchaseOrder** or even **context:accounts:direct://purchaseOrder** if you prefer. But using logical endpoint URIs is preferred as it hides the implementation detail and provides a simple logical naming scheme.

For example if we wish to then expose this accounts black box on some middleware (outside of the black box) we can do things like...

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



```
<route>
  <!-- consume from an ActiveMQ into the black box -->
  <from uri="activemq:Accounts.PurchaseOrders"/>
  <to uri="accounts:purchaseOrders"/>
</route>
<route>
  <!-- lets send invoices from the black box to a different ActiveMQ
Queue -->
  <from uri="accounts:invoice"/>
  <to uri="activemq:UK.Accounts.Invoices"/>
</route>
</camelContext>
```

NAMING ENDPOINTS

A context component instance can have many public input and output endpoints that can be accessed from outside it's CamelContext. When there are many it is recommended that you use logical names for them to hide the middleware as shown above.

However when there is only one input, output or error/dead letter endpoint in a component we recommend using the common posix shell names **in**, **out** and **err**

CHAPTER 18. CONTROLBUS COMPONENT

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.

COMMANDS

| Command | Description |
|----------|--|
| route | To control routes using the routeId and action parameter. |
| 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. |

OPTIONS

| Name | Default Value | Description |
|---------|---------------|---------------------------------------|
| routeId | null | To specify a route by its id . |

| | | |
|---------------------|--------------|--|
| action | null | 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 statistics 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 . |
| async | false | 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. |
| loggingLevel | INFO | Logging level used for logging when task is done, or if any exceptions occurred during processing the task. |

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

SAMPLES

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);
```

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);
```

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:

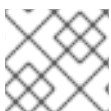
```
String status = template.requestBody("controlbus:language:simple",
"${camelContext.getRouteStatus('myRoute')}");
```

Notice: its 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()}");
```

Notice 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.



NOTE

You can also use other languages such as [Groovy](#), etc.

- [ControlBus](#) EIP

- [JMX](#) Component
- Using [JMX](#) with Camel

CHAPTER 19. COUCHDB

CAMEL COUCHDB COMPONENT

Available as of Camel 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 or update 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 sequenceld, 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>
```

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.

OPTIONS

| Property | Default | Description |
|----------------|-------------|--|
| deletes | true | document deletes are published as events |
| updates | true | document inserts/updates are published as events |

| | | |
|-----------------------|--------------|---|
| heartbeat | 30000 | how often to send an empty message to keep socket alive in millis |
| createDatabase | true | create the database if it does not already exist |
| username | null | username in case of authenticated databases |
| password | null | password for authenticated databases |

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.

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.

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://lo  
calhost:9999")
```


CHAPTER 20. CRYPTO (DIGITAL SIGNATURES)

CRYPTO COMPONENT FOR DIGITAL SIGNATURES

Available as of Apache Camel 2.3

Using Apache Camel cryptographic endpoints and Java's Cryptographic extension it is easy to create Digital Signatures for [Exchanges](#). Apache 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>
```

INTRODUCTION

Digital signatures make use 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 your messages. Messages are signed by encrypting a digest of the message with the private key. 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 digest match the verifier knows only the holder of the private key could have created the signature.

Apache 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 sources 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](#)

URI FORMAT

As mentioned Apache 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 **Exchange.SIGNATURE**, i.e. "**CamelDigitalSignature**".
- **crypto:verify** will read in the contents of this header and do the verification calculation.

In order to correctly function, sign and verify need to share a pair of keys, sign requiring a **PrivateKey** and verify a **PublicKey** (or a **Certificate** containing one). Using the JCE 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 sign and verify should be configured identically.

OPTIONS

| Name | Type | Default | Description |
|--------------------|--------------------|-------------|--|
| algorithm | String | DSA | The name of the JCE Signature algorithm that will be used. |
| alias | String | null | An alias name that will be used to select a key from the keystore. |
| bufferSize | Integer | 2048 | the size of the buffer used in the signature process. |
| certificate | Certificate | null | A Certificate used to verify the signature of the exchange's payload. Either this or a Public Key is required. |
| keystore | KeyStore | null | A reference to a JCE Keystore that stores keys and certificates used to sign and verify. |
| provider | String | null | The name of the JCE Security Provider that should be used. |

| | | | |
|---------------------|---------------------|-------------|--|
| privateKey | PrivatKey | null | The private key used to sign the exchange's payload. |
| publicKey | PublicKey | null | The public key used to verify the signature of the exchange's payload. |
| secureRandom | secureRandom | null | A reference to a SecureRandom object that will be used to initialize the Signature service. |
| password | char[] | null | The password for the keystore. |
| clearHeaders | String | true | Remove camel crypto headers from Message after a verify operation (value can be "true"/ {{"false"}}). |

1) RAW KEYS

The most basic way to way to sign and verify an exchange is with a KeyPair as follows.

```
from("direct:keypair").to("crypto:sign://basic?privateKey=#myPrivateKey",
"crypto:verify://basic?publicKey=#myPublicKey", "mock:result");
```

The same can be achieved with the [Spring XML Extensions](#) using references to keys

```
<route>
  <from uri="direct:keypair"/>
  <to uri="crypto:sign://basic?privateKey=#myPrivateKey" />
  <to uri="crypto:verify://basic?publicKey=#myPublicKey" />
  <to uri="mock:result"/>
</route>
```

2) KEYSTORES AND ALIASES.

The JCE provides a very versatile KeyStore for housing pairs of PrivateKeys and Certificates keeping them encrypted and password protected. They can be retrieved from it 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.

```
from("direct:keystore").to("crypto:sign://keystore?
keystore=#keystore&alias=bob&password=letmein", "crypto:verify://keystore?
keystore=#keystore&alias=bob", "mock:result");
```

Again in Spring a ref is used to lookup an actual keystore instance.

```
<route>
  <from uri="direct:keystore"/>
  <to uri="crypto:sign://keystore?
keystore=#keystore&alias=bob&password=letmein" />
  <to uri="crypto:verify://keystore?keystore=#keystore&alias=bob" />
  <to uri="mock:result"/>
</route>
```

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.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(512, new SecureRandom());
keyPair = keyGen.generateKeyPair();
PrivateKey privateKey = keyPair.getPrivate();
PublicKey publicKey = keyPair.getPublic();

// we can set the keys explicitly on the endpoint instances.
context.getEndpoint("crypto:sign://rsa?algorithm=MD5withRSA",
DigitalSignatureEndpoint.class).setPrivateKey(privateKey);
context.getEndpoint("crypto:verify://rsa?algorithm=MD5withRSA",
DigitalSignatureEndpoint.class).setPublicKey(publicKey);
from("direct:algorithm").to("crypto:sign://rsa?algorithm=MD5withRSA",
"crypto:verify://rsa?algorithm=MD5withRSA", "mock:result");
```

```
from("direct:provider").to("crypto:sign://provider?
privateKey=#myPrivateKey&provider=SUN", "crypto:verify://provider?
publicKey=#myPublicKey&provider=SUN", "mock:result");
```

or

```
<route>
  <from uri="direct:algorithm"/>
  <to uri="crypto:sign://rsa?
algorithm=MD5withRSA&privateKey=#rsaPrivateKey" />
  <to uri="crypto:verify://rsa?
algorithm=MD5withRSA&publicKey=#rsaPublicKey" />
  <to uri="mock:result"/>
</route>
```

```
<route>
```

```

    <from uri="direct:provider"/>
    <to uri="crypto:sign://provider?privateKey=#myPrivateKey&rovider=SUN"
  />
    <to uri="crypto:verify://provider?publicKey=#myPublicKey&rovider=SUN"
  />
    <to uri="mock:result"/>
</route>

```

4) CHANGING THE SIGNATURE MESASGE 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

```

from("direct:signature-header").to("crypto:sign://another?
privateKey=#myPrivateKey&signatureHeader=AnotherDigitalSignature",
                                "crypto:verify://another?
publicKey=#myPublicKey&signatureHeader=AnotherDigitalSignature",
                                "mock:result");

```

or

```

<route>
  <from uri="direct:signature-header"/>
  <to uri="crypto:sign://another?
privateKey=#myPrivateKey&signatureHeader=AnotherDigitalSignature" />
  <to uri="crypto:verify://another?
publicKey=#myPublicKey&signatureHeader=AnotherDigitalSignature" />
  <to uri="mock:result"/>
</route>

```

5) CHANGING THE BUFFERSIZE

In case you need to update the size of the buffer...

```

from("direct:bufferize").to("crypto:sign://buffer?
privateKey=#myPrivateKey&bufferize=1024", "crypto:verify://buffer?
publicKey=#myPublicKey&bufferize=1024", "mock:result");

```

or

```

<route>
  <from uri="direct:bufferize" />
  <to uri="crypto:sign://buffer?privateKey=#myPrivateKey&uffersize=1024"
  />
  <to uri="crypto:verify://buffer?publicKey=#myPublicKey&uffersize=1024"
  />
  <to uri="mock:result"/>
</route>

```

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 the 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"`

```
from("direct:headerkey-sign").to("crypto:sign://alias");
from("direct:headerkey-verify").to("crypto:verify://alias",
"mock:result");
```

or

```
<route>
  <from uri="direct:headerkey-sign"/>
  <to uri="crypto:sign://headerkey" />
</route>
<route>
  <from uri="direct:headerkey-verify"/>
  <to uri="crypto:verify://headerkey" />
  <to uri="mock:result"/>
</route>
```

Better again would be to dynamically supply a keystore alias. Again the alias can be supplied in a message header

- `Exchange.KEYSTORE_ALIAS`, `"CamelSignatureKeyStoreAlias"`

```
from("direct:alias-sign").to("crypto:sign://alias?keystore=#keystore");
from("direct:alias-verify").to("crypto:verify://alias?keystore=#keystore",
"mock:result");
```

or

```
<route>
  <from uri="direct:alias-sign"/>
  <to uri="crypto:sign://alias?keystore=#keystore" />
</route>
<route>
  <from uri="direct:alias-verify"/>
  <to uri="crypto:verify://alias?keystore=#keystore" />
  <to uri="mock:result"/>
</route>
```

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);
```

See also:

- [Crypto](#) Crypto is also available as a [Data Format](#)

CHAPTER 21. 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 as a consumer, the CAMEL: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.



NOTE

When using CXF in streaming modes (see [DataFormat](#) option), then also read about [Stream caching](#).

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 |
|---------------------|--|--|
| wsdURL | No | <p>The location of the WSDL. WSDL is obtained from endpoint address by default. For example:</p> <p>file://local/wsd/hello.wsd or wsd/hello.wsd</p> |
| serviceClass | Yes | <p>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 wsdURL option is provided, serviceClass is not required for PAYLOAD and MESSAGE mode. When wsdURL option is used without serviceClass, the serviceName and portName (endpointName for Spring configuration) options MUST be provided.</p> <p>Since 2.0, it is possible to use # notation to reference a serviceClass object instance from the registry..</p> <p>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.</p> <p>Since 2.8, it is possible to omit both wsdURL 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.</p> <p>For example: org.apache.camel.Hello</p> |
| serviceName | Only if more than one serviceName present in WSDL | <p>The service name this service is implementing, it maps to the wsd:service@name. For example:</p> <p>{http://org.apache.camel}ServiceName</p> |

| | | |
|----------------------|--|---|
| portName | 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.came1}PortName |
| dataFormat | No | Which message data format the CXF endpoint supports. Possible values are: POJO (<i>default</i>), 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 <i>Default: true</i> <i>Example: true, false</i> |
| wrapped | No | Which kind of operation the CXF endpoint producer will invoke. Possible values are: true, false (<i>default</i>). |
| 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 | Specifies whether or not to use the default CXF bus for this endpoint. Possible values are: true, false (<i>default</i>). |
| 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 (<i>default</i>). |
| 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. |

| | | |
|-----------------------------------|----|--|
| <code>publishedEndpointUrl</code> | 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 <code>?wsdl</code> . For example: <code>publishedEndpointUrl=http://example.com/service</code> |
| <code>properties.propName</code> | No | Camel 2.8: Allows you to set custom CXF properties in the endpoint URI. For example, setting <code>properties.mtom-enabled=true</code> to enable MTOM. |
| <code>allowStreaming</code> | 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 <code>javax.xml.transform.Source</code> object that would allow streaming in some cases. |
| <code>skipFaultLogging</code> | No | New in 2.11. This option controls whether the <code>PhaseInterceptorChain</code> skips logging the Fault that it catches. |

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 interceptor 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
http://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
  http://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.ReportIncidentEndpoi
nt">
  </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:

<https://svn.apache.org/repos/asf/camel/branches/camel-1.x/components/camel-cxf/src/test/java/org/apache/camel/component/cxf/soap/headers/CxfMessageHeadersRelayTes>

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.CxfHeaderFilterS
trategy">

  <!-- 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.CxfHeaderFilterS
trategy">
    <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.SoopMessageHeade
rFilter"/>

            <!-- Add custom filter here -->
            <bean
class="org.apache.camel.component.cxf.soap.headers.CustomHeaderFilte
r"/>
        </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) |

| | | | | |
|----------------------------------|---|----------------|----|-------------------------------|
| allowFilterNamespaceClass | 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** endpoint

```
<cxf:cxfEndpoint id="routerEndpoint"
address="http://localhost:${CXFTestSupport.port2}/CXFGreeterRouterTest/Cam
elContext/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.toSt
ring());
        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 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 *alist 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());

```

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).h
andled(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 operation 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 themselves. Attachments are propagated to Camel message's attachments since 2.1. So, it is possible to retrieve 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}/CxfMtomRouterPayloadMode
Test/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 format 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<Source> elements = new ArrayList<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<Source> elements = new ArrayList<Source>();
        elements.add(new DOMSource(DOMUtils.readXml(new
StringReader(MtomTestHelper.RESP_MESSAGE)).getDocumentElement()));
        CxfPayload<SoapHeader> sbody = new CxfPayload<SoapHeader>
(new ArrayList<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.

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 feature, 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 it off. That sets the global default, but setting the endpoint property above will override this value for that endpoint.

CHAPTER 22. CXF BEAN COMPONENT

CXF BEAN COMPONENT (2.0 OR LATER)

The **cxfbean:** component allows other Camel endpoints to send exchange and invoke Web service bean objects. (**Currently, it only supports JAXRS, JAXWS(new to camel2.1) annotated service bean.**)



IMPORTANT

CxfBeanEndpoint is a **ProcessorEndpoint** so it has no consumers. It works similarly to a Bean component.

URI FORMAT

`cxfbean:serviceBeanRef`

Where **serviceBeanRef** is a registry key to look up the service bean object. If **serviceBeanRef** references a **List** object, elements of the **List** are the service bean objects accepted by the endpoint.

OPTIONS

| Name | Description | Example | Required? | Default Value |
|-----------------------|---|--------------------------------|-----------|--|
| bus | CXF bus reference specified by the <code>\#</code> notation. The referenced object must be an instance of org.apache.cxf.Bus . | bus=#busName | No | Default bus created by CXF Bus Factory |
| cxfBeanBinding | CXF bean binding specified by the <code>\#</code> notation. The referenced object must be an instance of org.apache.camel.component.cxf.cxfbean.CxfBeanBinding . | cxfBinding=#bindingName | No | DefaultCxfBeanBinding |

| | | | | |
|-----------------------------|--|---|----|--------------------------------|
| headerFilterStrategy | Header filter strategy specified by the \# notation. The referenced object must be an instance of org.apache.camel.spi.HeaderFilterStrategy . | headerFilterStrategy=#strategyName | No | CxfHeaderFilterStrategy |
| populateFromClass | Since 2.3, the wsdlLocation annotated in the POJO is ignored (by default) unless this option is set to false . Prior to 2.3, the wsdlLocation annotated in the POJO is always honored and it is not possible to ignore. | true, false | No | true |
| providers | Since 2.5, setting the providers for the CXFRS endpoint. | providers=#providerRef1, #providerRef2 | No | null |
| setDefaultBuses | Will set the default bus when CXF endpoint create a bus by itself. | true, false | No | false |

HEADERS

| Name | Description | Type | Required? | Default Value | In/Out | Examples |
|------|-------------|------|-----------|---------------|--------|----------|
|------|-------------|------|-----------|---------------|--------|----------|

| | | | | | | |
|--|---|---------------|-----|---|----|---|
| CamelHttpCharacterEncoding (before 2.0-m2: CamelCxfBeanCharacterEncoding) | Character encoding | String | No | None | In | ISO-8859-1 |
| CamelContentType (before 2.0-m2: CamelCxfBeanContentType) | Content type | String | No | **/** | In | text/xml |
| CamelHttpBaseUri (2.0-m3 and before: CamelCxfBeanRequestBasePath) | The value of this header will be set in the CXF message as the Message.BASE_PATH property. It is needed by CXF JAX-RS processing. Basically, it is the scheme, host and port portion of the request URI. | String | Yes | The Endpoint URI of the source endpoint in the Camel exchange | In | http://localhost:9000 |
| CamelHttpPath (before 2.0-m2: CamelCxfBeanRequestPath) | Request URI's path | String | Yes | None | In | consumer/123 |

| | | | | | | |
|---|----------------------|----------------|-----|------|-----|-------------------------------|
| CamelHttpMethod (before 2.0-m2: CamelCxfBeanVerb) | RESTful request verb | String | Yes | None | In | GET, PUT, POST, DELETE |
| CamelHttpResponseCode | HTTP response code | Integer | No | None | Out | 200 |

**NOTE**

Currently, CXF Bean component has (only) been tested with Jetty HTTP component it can understand headers from Jetty HTTP component without requiring conversion.

A WORKING SAMPLE

This sample shows how to create a route that starts a Jetty HTTP server. The route sends requests to a CXF Bean and invokes a JAXRS annotated service.

First, create a route as follows. The **from** endpoint is a Jetty HTTP endpoint that is listening on port 9000. Notice that the **matchOnUriPrefix** option must be set to **true** because RESTful request URI will not match the endpoint's URI `http://localhost:9000` exactly.

```
<route>
  <from uri="jetty:http://localhost:9000?matchOnUriPrefix=true" />
  <to uri="cxfbean:customerServiceBean" />
  <to uri="mock:endpointA" />
</route>
```

The **to** endpoint is a CXF Bean with bean name **customerServiceBean**. The name will be looked up from the registry. Next, we make sure our service bean is available in Spring registry. We create a bean definition in the Spring configuration. In this example, we create a List of service beans (of one element). We could have created just a single bean without a List.

```
<util:list id="customerServiceBean">
  <bean
class="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService" />
</util:list>

<bean class="org.apache.camel.wsdl_first.PersonImpl" id="jaxwsBean" />
```

That's it. Once the route is started, the web service is ready for business. A HTTP client can make a request and receive response.

```
url = new
URL("http://localhost:9000/customerservice/orders/223/products/323");
in = url.openStream();
assertEquals("{\"Product\":{\"description\":\"product 323\",\"id\":323}}",
```

```
CxfUtils.getStringFromInputStream(in);
```

CHAPTER 23. CXFRS

CXFRS COMPONENT

The **cxfrs:** component provides integration with [Apache CXF](#) for connecting to JAX-RS 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

When using CXF as a consumer, the CAMEL:CFX 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.

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 represents 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
```

OPTIONS

| Name | Description | Example | Required? | default value |
|------|-------------|---------|-----------|---------------|
|------|-------------|---------|-----------|---------------|

| | | | | |
|--------------------------------|---|---|----|--------------|
| resourceClasses | The resource classes which you want to export as REST service. Multiple classes can be separated by a comma. | resourceClasses=org.apache.camel.rs.Example1,org.apache.camel.rs.Exchange2 | No | <i>None</i> |
| httpClientAPI | New to Apache Camel 2.1 If true, the CxfRsProducer will use the HttpClientAPI to invoke the service | httpClientAPI=true | No | true |
| synchronous | New in 2.5, this option will let CxfRsConsumer decide to use sync or async API to do the underlying work. The default value is false which means it will try to use async API by default. | synchronous=true | No | false |
| throwExceptionOnFailure | New in 2.6, this option tells the CxfRsProducer to inspect return codes and will generate an Exception if the return code is larger than 207. | throwExceptionOnFailure=true | No | true |

| | | | | |
|---------------------------|---|-----------------------------|----|--------------|
| maxClientCacheSize | New in 2.6, you can set the <i>In</i> message header, CamelDestinationOverrideUrl , to dynamically override the target destination Web Service or REST Service defined in your routes. The implementation caches CXF clients or ClientFactoryBean in CxfProvider and CxfRsProvider . This option allows you to configure the maximum size of the cache. | maxClientCacheSize=5 | No | 10 |
| setDefaultBuses | New in 2.9.0. Will set the default bus when CXF endpoint create a bus by itself | setDefaultBuses=true | No | false |
| bus | New in 2.9.0. A default bus created by CXF Bus Factory. Use <i>\#</i> notation to reference a bus object from the registry. The referenced object must be an instance of org.apache.cxf.Bus . | bus=#busName | No | <i>None</i> |

| | | | | |
|---------------------|---|------------------------------------|----|----------------|
| bindingStyle | <p>As of 2.11. Sets how requests and responses will be mapped to/from Camel. Two values are possible:</p> <ul style="list-style-type: none"> • Simple Consumer => see the Consuming a REST Request with the Simple Binding Style below. • Default => the default style. For consumers this passes on a MessageContentsList to the route, requiring low-level processing in the route. | bindingStyle=SimpleConsumer | No | <i>Default</i> |
|---------------------|---|------------------------------------|----|----------------|

You can also configure the CXF REST endpoint through the Spring configuration. Since there are lots of differences between the CXF REST client and CXF REST Server, we provide different configurations for them. Please check out the [schema file](#) and [CXF REST user guide](#) for more information.

HOW TO CONFIGURE THE REST ENDPOINT IN APACHE 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 Apache Camel REST service route configuration example here.

```

<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"
  xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/cxf
    http://camel.apache.org/schema/cxf/camel-cxf.xsd
    http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
    http://camel.apache.org/schema/spring
    http://camel.apache.org/schema/spring/camel-spring.xsd
  ">
  <!-- Defined the real JAXRS back end service -->
  <jaxrs:server id="restService"
    address="http://localhost:9002/rest"
    staticSubresourceResolution="true">
    <jaxrs:serviceBeans>
      <ref bean="customerService"/>
    </jaxrs:serviceBeans>
  </jaxrs:server>

  <!-- bean id="jsonProvider"
  class="org.apache.cxf.jaxrs.provider.JSONProvider"/-->

  <bean id="customerService"
  class="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService" />

  <!-- Defined the server endpoint to create the cxf-rs consumer -->
  <cxf:rsServer id="rsServer" address="http://localhost:9000/route"

  serviceClass="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService"
  loggingFeatureEnabled="true" loggingSizeLimit="20"
  skipFaultLogging="true"/>

  <!-- Defined the client endpoint to create the cxf-rs consumer -->
  <cxf:rsClient id="rsClient" address="http://localhost:9002/rest"

  serviceClass="org.apache.camel.component.cxf.jaxrs.testbean.CustomerService"
  loggingFeatureEnabled="true" skipFaultLogging="true"/>

  <!-- The camel route context -->
  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="cxf:rs://bean://rsServer"/>
      <!-- We can remove this configure as the CXFRS producer is using
  the HttpAPI by default -->
      <setHeader headerName="CamelCxfRsUsingHttpAPI">
        <constant>True</constant>
      </setHeader>
      <to uri="cxf:rs://bean://rsClient"/>
    </route>
  </camelContext>

```

```
</beans>
```

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.

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");
```

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.

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;
}
```

Served 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](#).

CONSUMING A REST REQUEST - DEFAULT BINDING STYLE

CXF JAXRS front end implements the JAXRS(JSR311) 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`, you don't need to specify the URI template within your restlet endpoint, CXF take care of the REST request URI to resource class method mapping according to the JSR311 specification. All you need to do in Apache 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.CustomerServ
iceResource";

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() {
            errorHandler(new NoErrorHandlerBuilder());
            from(CXF_RS_ENDPOINT_URI).process(new Processor() {

                public void process(Exchange exchange) throws Exception {
                    Message inMessage = exchange.getIn();
                    // Get the operation name from in message
                    String operationName =
inMessage.getHeader(CxfConstants.OPERATION_NAME, String.class);
                    if ("getCustomer".equals(operationName)) {
                        String httpMethod =
inMessage.getHeader(Exchange.HTTP_METHOD, String.class);
                        assertEquals("Get a wrong http method", "GET",
httpMethod);

                            String path =
inMessage.getHeader(Exchange.HTTP_PATH, String.class);
                            // The parameter of the invocation is stored in
the body of in message
                            String id = inMessage.getBody(String.class);
                            if
("/customerservice/customers/126".equals(path)) {
                                Customer customer = new Customer();
                                customer.setId(Long.parseLong(id));
                                customer.setName("Willem");
                                // We just put the response Object into the
out message body
                                exchange.getOut().setBody(customer);
                            } else {
                                if
("/customerservice/customers/400".equals(path)) {
```

```

        // We return the remote client IP
address this time
        org.apache.cxf.message.Message
cxfMessage = inMessage.getHeader(CxfConstants.CAMEL_CXF_MESSAGE,
org.apache.cxf.message.Message.class);
        ServletRequest request =
(ServletRequest) cxfMessage.get("HTTP.REQUEST");
        String remoteAddress =
request.getRemoteAddr();
        Response r =
Response.status(200).entity("The remoteAddress is " +
remoteAddress).build();
        exchange.getOut().setBody(r);
        return;
    }
    if
("/customerservice/customers/123".equals(path)) {
        // send a customer response back
        Response r =
Response.status(200).entity("customer response back!").build();
        exchange.getOut().setBody(r);
        return;
    }
    if
("/customerservice/customers/456".equals(path)) {
        Response r =
Response.status(404).entity("Can't found the customer with uri " +
path).build();
        throw new WebApplicationException(r);
    } else {
        throw new RuntimeException("Can't
found the customer with uri " + path);
    }
}
    if ("updateCustomer".equals(operationName)) {
        assertEquals("Get a wrong customer message
header", "header1;header2", inMessage.getHeader("test"));
        String httpMethod =
inMessage.getHeader(Exchange.HTTP_METHOD, String.class);
        assertEquals("Get a wrong http method", "PUT",
httpMethod);
        Customer customer =
inMessage.getBody(Customer.class);
        assertNotNull("The customer should not be null.",
customer);
        // Now you can do what you want on the customer
object
        assertEquals("Get a wrong customer name.",
"Mary", customer.getName());
        // set the response back
        exchange.getOut().setBody(Response.ok().build());
    }
}
});

```

```

    }
};
}

```

The corresponding resource class used to configure the endpoint is defined as an interface:

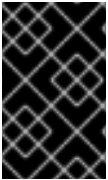
```

@Path("/customerservice/")
public interface CustomerServiceResource {

    @GET
    @Path("/customers/{id}/")
    Customer getCustomer(@PathParam("id") String id);

    @PUT
    @Path("/customers/")
    Response updateCustomer(Customer customer);
}

```



IMPORTANT

The resource class is used to configure the JAXRS properties *only*. The methods will *not* be executed during the routing of messages to the endpoint, the route itself is responsible for all processing instead.

HOW TO INVOKE THE REST SERVICE THROUGH CAMEL-CXFRS PRODUCER ?

[CXF JAXRS front end](#) implements a [proxy based client API](#), with this API you can invoke the remote REST service through a proxy. `camel-cxf-rs` producer is based on this [proxy API](#). So, you just need to specify the operation name in the message header and prepare the parameter in the message body, `camel-cxf-rs` 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();
        setupDestinationURL(inMessage);
        // 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");
        // 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 ", String.valueOf(response.getId()),
"123");
assertEquals("Get a wrong customer name", response.getName(), "John");
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"));
```

[CXF JAXRS front end](#) also provides a [http centric client API](#), You can also invoke this API from `camel-cxf-rs` producer. You need to specify the `HTTP_PATH` and `Http` method and let the the producer know to use the HTTP centric client by using the URI option `httpClientAPI` or set the message header with `CxfConstants.CAMEL_CXF_RS_USING_HTTP_API`. You can turn the response object to the type class that you specify with `CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS`.

```
Exchange exchange = template.send("direct://http", new Processor() {

    public void process(Exchange exchange) throws Exception {
        exchange.setPattern(ExchangePattern.InOut);
        Message inMessage = exchange.getIn();
        setupDestinationURL(inMessage);
        // using the http central client API
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_USING_HTTP_API,
Boolean.TRUE);
        // set the Http method
        inMessage.setHeader(Exchange.HTTP_METHOD, "GET");
        // set the relative path
        inMessage.setHeader(Exchange.HTTP_PATH,
"/customerservice/customers/123");
        // Specify the response class , cxf-rs will use InputStream as the
response object type
        inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_RESPONSE_CLASS,
Customer.class);
        // set a customer header
        inMessage.setHeader("key", "value");
        // since we use the Get method, so we don't need to set the
message body
        inMessage.setBody(null);
    }

});

// get the response message
Customer response = (Customer) exchange.getOut().getBody();

assertNotNull("The response should not be null ", response);
assertEquals("Get a wrong customer id ", String.valueOf(response.getId()),
"123");
assertEquals("Get a wrong customer name", response.getName(), "John");
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"));
```

From Apache Camel 2.1, we also support to specify the query parameters from CXFRS URI for the CXFRS HTTP centric client.

```
Exchange exchange = template.send("cxfrs://http://localhost:" + getPort2()
+ "/" + getClass().getSimpleName() + "/testQuery?
httpClientAPI=true&q1=12&q2=13"
```

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.

```
Map<String, String> queryMap = new LinkedHashMap<String, String>();
queryMap.put("q1", "new");
queryMap.put("q2", "world");
inMessage.setHeader(CxfConstants.CAMEL_CXF_RS_QUERY_MAP, queryMap);
```

CHAPTER 24. DATAFORMAT COMPONENT

DATA FORMAT COMPONENT

Available as of Camel 2.12

The **dataformat:** component allows to use [Data Format](#) as a Camel [Component](#).

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.

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>
```

- [Data Format](#)

CHAPTER 25. DATASET

DATASET COMPONENT

The DataSet component (available since 1.3.0) 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.

Apache Camel will use the [throughput logger](#) when sending dataset's.

URI FORMAT

```
dataset:name[?options]
```

Where **name** is used to find the [DataSet instance](#) in the [Registry](#)

Apache 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. Apache Camel also ships with a default implementation, the `org.apache.camel.component.dataset.SimpleDataSet` that can be used for testing.

OPTIONS

| Option | Default | Description |
|---------------------------|---------|--|
| <code>produceDelay</code> | 3 | Allows a delay in ms to be specified, which causes producers to pause in order to simulate slow producers. Uses a minimum of 3 ms delay unless you set this option to -1 to force no delay at all. |
| <code>consumeDelay</code> | 0 | Allows a delay in ms to be specified, which causes consumers to pause in order to simulate slow consumers. |
| <code>preloadSize</code> | 0 | Sets how many messages should be preloaded (sent) before the route completes its initialization. |
| <code>initialDelay</code> | 1000 | Camel 2.1: Time period in millis to wait before starting sending messages. |

| | | |
|----------------|---|--|
| minRate | 0 | Wait until the DataSet contains at least this number of messages |
|----------------|---|--|

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

CONFIGURING DATASET

Apache 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>
```

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.

PROPERTIES ON SIMPLEDATASET

| Property | Type | Default | Description |
|--------------------|---------------|---|--|
| defaultBody | Object | <code><hello>world! </hello></code> | 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 . |

| | | | |
|--------------------|-------------|-----------|--|
| 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. |
| size | long | 10 | Specifies how many messages to send/consume. |

- [Spring Testing](#)

CHAPTER 26. DIRECT

DIRECT COMPONENT

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.



NOTE

The Seda component provides asynchronous invocation of any consumers when a producer sends a message exchange.



NOTE

The VM component provides connections between Camel contexts as long they run in the same **JVM**.

URI FORMAT

```
direct:someName[?options]
```

Where **someName** can be any string to uniquely identify the endpoint

OPTIONS

| Name | Default Value | Description |
|-------------------------------------|--------------------|--|
| <code>allowMultipleConsumers</code> | <code>true</code> | @deprecated If set to false , then when a second consumer is started on the endpoint, an IllegalStateException is thrown. Will be removed in Camel 2.1: Direct endpoint does not support multiple consumers. |
| <code>block</code> | <code>false</code> | Camel 2.11.1: 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. |
| <code>timeout</code> | <code>30000</code> | Camel 2.11.1: The timeout value to use if block is enabled. |

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

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.

- [SEDA](#)
- [VM](#)

CHAPTER 27. DIRECT-VM

DIRECT VM COMPONENT

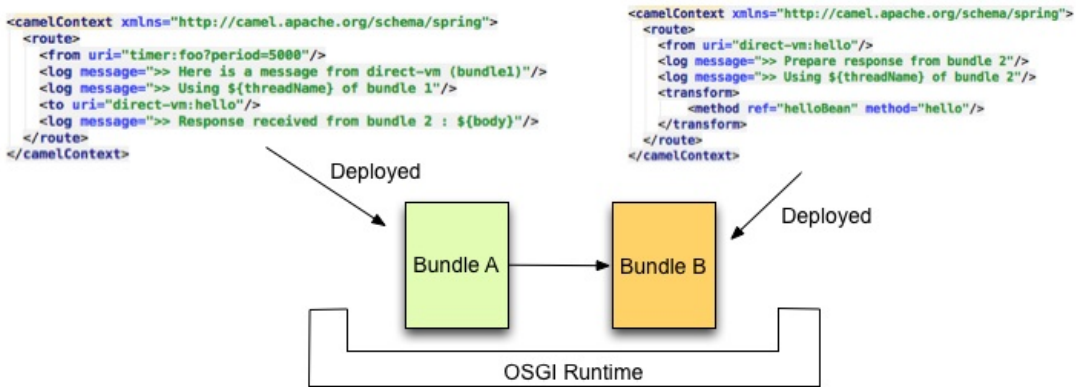
Available as of Camel 2.10

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.



```

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
    
```

URI FORMAT

direct-vm:someName

Where **someName** can be any string to uniquely identify the endpoint

OPTIONS

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|----------------|--------------|---|
| block | false | Camel 2.11.1: 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. |
| timeout | 30000 | Camel 2.11.1: The timeout value to use if block is enabled. |

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>
```

- [Direct](#)
- [SEDA](#)
- [VM](#)

CHAPTER 28. DISRUPTOR

DISRUPTOR COMPONENT

Available as of Camel 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>

```

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&...
```

OPTIONS

All the following options are valid for both the ***disruptor:*** and ***disruptor-vm:*** components.

| Name | Default | Description |
|------------|---------|--|
| size | 1024 | 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. |
| bufferSize | | Component only: The maximum default size (capacity of the number of messages it can hold) of the Disruptors ringbuffer. This option is used if size is not in use. |

| | | |
|--------------------------|-----------------|---|
| queueSize | | Component only: Additional option to specify the <code>bufferSize</code> to maintain maximum compatibility with the SEDA Component. |
| concurrentConsumers | 1 | Number of concurrent threads processing exchanges. |
| waitForTaskToComplete | IfReplyExpected | Option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: <i>Always</i> , <i>Never</i> or <i>IfReplyExpected</i> . The first two values are self-explanatory. The last value, <i>IfReplyExpected</i> , will only wait if the message is Request Reply based. See more information about Async messaging. |
| timeout | 30000 | Timeout (in milliseconds) before a producer will stop waiting for an asynchronous task to complete. See <i>waitForTaskToComplete</i> and Async for more details. You can disable timeout by using 0 or a negative value. |
| defaultMultipleConsumers | | Component only: Allows to set the default allowance of multiple consumers for endpoints created by this component used when <i>multipleConsumers</i> is not provided. |
| limitConcurrentConsumers | true | Whether to limit the number of concurrentConsumers to the maximum of 500. By default, an exception will be thrown if a Disruptor endpoint is configured with a greater number. You can disable that check by turning this option off. |

| | | |
|----------------------|----------|---|
| blockWhenFull | true | 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. |
| defaultBlockWhenFull | | Component only: Allows to set the default producer behaviour when the ringbuffer is full for endpoints created by this component used when <i>blockWhenFull</i> is not provided. |
| waitStrategy | Blocking | Defines the strategy used by consumer threads to wait on new exchanges to be published. The options allowed are: <i>Blocking</i> , <i>Sleeping</i> , <i>BusySpin</i> and <i>Yielding</i> . Refer to the section below for more information on this subject |
| defaultWaitStrategy | | Component only: Allows to set the default wait strategy for endpoints created by this component used when <i>waitStrategy</i> is not provided. |
| producerType | Multi | Defines the producers allowed on the Disruptor. The options allowed are: <i>Multi</i> to allow multiple producers and <i>Single</i> to enable certain optimizations only allowed when one concurrent producer (on one thread or otherwise synchronized) is active. |

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. |
| Sleeping | Sleeping strategy that initially spins, then uses a <code>Thread.yield()</code> , 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 <code>Thread.yield()</code> 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. |

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.

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.

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 advised to directly configure number of threads that process messages on a Disruptor endpoint using the `concurrentConsumers` option.

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.

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);
    }
}
```

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 29. DNS

DNS

Available as of Camel 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 address.

lookup

To look up information about the domain.

dig

To run DNS queries.



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>
```

URI FORMAT

The URI scheme for a DNS component is as follows

```
dns://operation
```

This component only supports producers.

OPTIONS

None.

HEADERS

| Header | Type | Operations | Description |
|-------------------------|---------------------|--------------------------|---|
| <code>dns.domain</code> | <code>String</code> | <code>ip</code> | The domain name. Mandatory. |
| <code>dns.name</code> | <code>String</code> | <code>lookup</code> | The name to lookup. Mandatory. |
| <code>dns.type</code> | - | <code>lookup, dig</code> | The type of the lookup. Should match the values of <code>org.xbill.dns.Type</code> . Optional. |
| <code>dns.class</code> | - | <code>lookup, dig</code> | The DNS class of the lookup. Should match the values of <code>org.xbill.dns.DClass</code> . Optional. |
| <code>dns.query</code> | <code>String</code> | <code>dig</code> | The query itself. Mandatory. |
| <code>dns.server</code> | <code>String</code> | <code>dig</code> | The server in particular for the query. If none is given, the default one specified by the OS will be used. Optional. |

EXAMPLES

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"`.

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**".

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**".

CHAPTER 30. EJB

EJB COMPONENT

Available as of Apache Camel 2.4

The **ejb:** component binds EJBs to message exchanges.

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](#)

OPTIONS

| Name | Type | Default | Description |
|---------------------|---------|---------|--|
| method | String | null | The method name that bean will be invoked. If not provided, Apache Camel will try to pick the method itself. In case of ambiguity an exception is thrown. See Bean Binding for more details. |
| multiParameterArray | boolean | false | How to treat the parameters which are passed from the message body; if it is true , the In message body should be an array of parameters. |

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

The [EJB](#) component extends the [Bean](#) component in which most of the details from the [Bean](#) component applies to this component as well.

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 Apache Camel.

EXAMPLES

In the following examples we use the Greater EJB which is defined as follows:

```
public interface GreaterLocal {
    String hello(String name);
    String bye(String name);
}
```

And the implementation

```
@Stateless
public class GreaterImpl implements GreaterLocal {

    public String hello(String name) {
        return "Hello " + name;
    }

    public String bye(String name) {
        return "Bye " + name;
    }
}
```

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 Apache 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");

```



NOTE

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.

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.LocalInitialCo
ntextFactory</prop>
</p:properties>

```

Before we are ready to use **EJB** in the Apache 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>

```

- [Bean](#)
- [Bean Binding](#)
- [Bean Integration](#)

CHAPTER 31. ELASTICSEARCH

ELASTICSEARCH COMPONENT

Available as of Camel 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>
```

URI FORMAT

```
elasticsearch://clusterName?[options]
```

TIP

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.

ENDPOINT OPTIONS

The following options may be configured on the ElasticSearch endpoint. All are required to be set as either an endpoint URI parameter or as a header (headers override endpoint properties)

| name | description |
|-----------|--|
| operation | required, indicates the operation to perform |
| indexName | the name of the index to act against |
| ip | the TransportClient remote host ip to use Camel 2.12 |

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 |
| 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 |

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);
```

FOR MORE INFORMATION, SEE THESE RESOURCES

[ElasticSearch Main Site](#)

[ElasticSearch Java API](#)

CHAPTER 32. EVENTADMIN

EVENTADMIN COMPONENT

Available in Camel 2.6

The `eventadmin` component can be used in an OSGi environment to receive OSGi EventAdmin events and process them.

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).

URI FORMAT

```
eventadmin:topic[?options]
```

where `topic` is the name of the topic to listen too.

URI OPTIONS

| Name | Default value | Description |
|-------------------|--------------------|--|
| <code>send</code> | <code>false</code> | Whether to use 'send' or 'synchronous' deliver. Default false (async delivery) |

MESSAGE HEADERS

| Name | Type | Message | Description |
|------|------|---------|-------------|
|------|------|---------|-------------|

MESSAGE BODY

The `in` message body will be set to the received Event.

EXAMPLE USAGE

```
<route>
```

```
<from uri="eventadmin:*" />  
  <to uri="stream:out" />  
</route>
```

CHAPTER 33. EXEC

EXEC COMPONENT

Available in Apache Camel 2.3

The `exec` component can be used to execute system commands.

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 Apache Camel (2.3.0 or higher).

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.

URI OPTIONS

| Name | Default value | Description |
|-------------------------|-------------------|---|
| <code>args</code> | <code>null</code> | The arguments of the executable. The arguments may be one or many whitespace-separated tokens, that can be quoted with <code>"</code> , e.g. <code>args="arg 1" arg2</code> will use two arguments <code>arg 1</code> and <code>arg2</code> . To include the quotes use <code>""</code> , e.g. <code>args=""arg 1"" arg2</code> will use the arguments <code>"arg 1"</code> and <code>arg2</code> . |
| <code>workingDir</code> | <code>null</code> | The directory in which the command should be executed. If <code>null</code> , the working directory of the current process will be used. |

| | | |
|-------------------------------|--|---|
| timeout | Long.MAX_VALUE | The timeout, in milliseconds, after which the executable should be terminated. If execution has not finished within the timeout, the component will send a termination request. |
| outFile | null | 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 considered as output. |
| binding | a DefaultExecBinding instance | A reference to a org.apache.commons.exec.ExecBinding in the Registry . |
| commandExecutor | a DefaultCommandExecutor instance | 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 . It adds a shutdown hook for every executed command. |
| useStderrOnEmptyStdout | false | 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. |

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 the executable in the URI. |

| | | | |
|---|---|-----------------|---|
| <code>ExecBinding.EXEC_COMMAND_ARGS</code> | <code>java.util.List<String></code> | <code>in</code> | The arguments of the executable. The arguments are used literally, no quoting is applied. Overrides existing args in the URI. |
| <code>ExecBinding.EXEC_COMMAND_ARGS</code> | <code>String</code> | <code>in</code> | 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 existing args in the URI. |
| <code>ExecBinding.EXEC_COMMAND_OUT_FILE</code> | <code>String</code> | <code>in</code> | The name of a file, created by the executable, that should be considered as output of the executable. Overrides existing outFile in the URI. |
| <code>ExecBinding.EXEC_COMMAND_TIMEOUT</code> | <code>long</code> | <code>in</code> | The timeout, in milliseconds, after which the executable should be terminated. Overrides any existing timeout in the URI. |
| <code>ExecBinding.EXEC_COMMAND_WORKING_DIR</code> | <code>String</code> | <code>in</code> | The directory in which the command should be executed. Overrides any existing workingDir in the URI. |

| | | | |
|--|----------------------------------|------------------|--|
| <code>ExecBinding.EXEC_EXIT_VALUE</code> | <code>int</code> | <code>out</code> | The value of this header is the <i>exit value</i> of the executable. Non-zero exit values typically indicate abnormal termination. Note that the exit value is OS-dependent. |
| <code>ExecBinding.EXEC_STDERR</code> | <code>java.io.InputStream</code> | <code>out</code> | The value of this header points to the standard error stream (<code>stderr</code>) of the executable. If no <code>stderr</code> is written, the value is null . |
| <code>ExecBinding.EXEC_USE_STDERR_ON_EMPTY_STDOUT</code> | <code>boolean</code> | <code>in</code> | Indicates that when stdout is empty, this component will populate the Camel Message Body with stderr . This behavior is disabled (false) by default. |

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 |
|-------------------------|-----------------------------------|
| <code>ExecResult</code> | <code>java.io.InputStream</code> |
| <code>ExecResult</code> | <code>String</code> |
| <code>ExecResult</code> | <code>byte []</code> |
| <code>ExecResult</code> | <code>org.w3c.dom.Document</code> |

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 in 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
        assertInstanceOf(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
    }
});
```

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 system property `user.name`.

```
from("direct:exec")
.to("exec:c:/program files/jdk/bin/java?args=-server -version -
Duser.name=Camel&workingDir=c:/temp")
```

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);
```

```
        assertInstanceOf(InputStream.class, outFile);  
        // do something with the out file here  
    }  
});
```

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")
```


CHAPTER 34. FABRIC COMPONENT

Abstract

The Fabric component implements a location discovery mechanism for Apache Camel endpoints. This mechanism can also be used to provide load-balancing over a cluster of endpoints. On the client side (producer endpoints), endpoints are represented by an abstract ID and at run time, the ID is resolved to a specific endpoint URI. Because the URI is stored in a distributed registry (provided by Fuse Fabric), this enables you to create flexible applications whose topology can be specified at deploy time and updated dynamically.

DEPENDENCIES

The Fabric component can only be used in the context of a fabric-enabled Red Hat JBoss Fuse container. You must ensure that the **fabric-camel** feature is installed. If necessary, you can install it using the following console command:

```
karaf@root> features:install fabric-camel
```

Alternatively, if you decide to use a custom feature to deploy your application, you can ensure that the **fabric-camel** feature is installed by including it in your feature definition. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<features>
  <feature name="fabric-component-example">
    <feature>fabric-camel</feature>
    <bundle>URIforMyBundle</bundle>
    <!-- Specify any other required bundles or features -->
    ...
  </feature>
</features>
```

For more details about features, see [Deploying Features](#).

URI FORMAT

A fabric endpoint has the following URI format:

```
fabric:ClusterID[:PublishedURI[?Options]]
```

The format of the URI depends on whether it is used to specify a consumer endpoint or a producer endpoint.

For a Fabric *producer endpoint*, the URI format is:

```
fabric:ClusterID:PublishedURI[?Options]
```

Where the specified URI, **PublishedURI**, is published in the fabric registry and associated with the **ClusterId** cluster. The options, **Options**, are used when creating the producer endpoint instance, but the options are *not* published with the **PublishedURI** in the fabric registry.

For a Fabric *consumer endpoint*, the URI format is:

```
fabric:ClusterID
```

Where the client looks up the ID, *ClusterId*, in the fabric registry to discover the URI to connect to.

URI OPTIONS

The Fabric component itself does *not* support any URI options. It is possible, however, to specify options for the published URI. These options are stored in the fabric registry as part of the URI and are used as follows:

- *Server-only options*—options that are applicable only to the server are applied to the server endpoint (consumer endpoint) at run time.
- *Client-only options*—options that are applicable only to the client are applied to the client endpoint (producer endpoint) at run time.
- *Common options*—options common to the client and the server are applied to both.

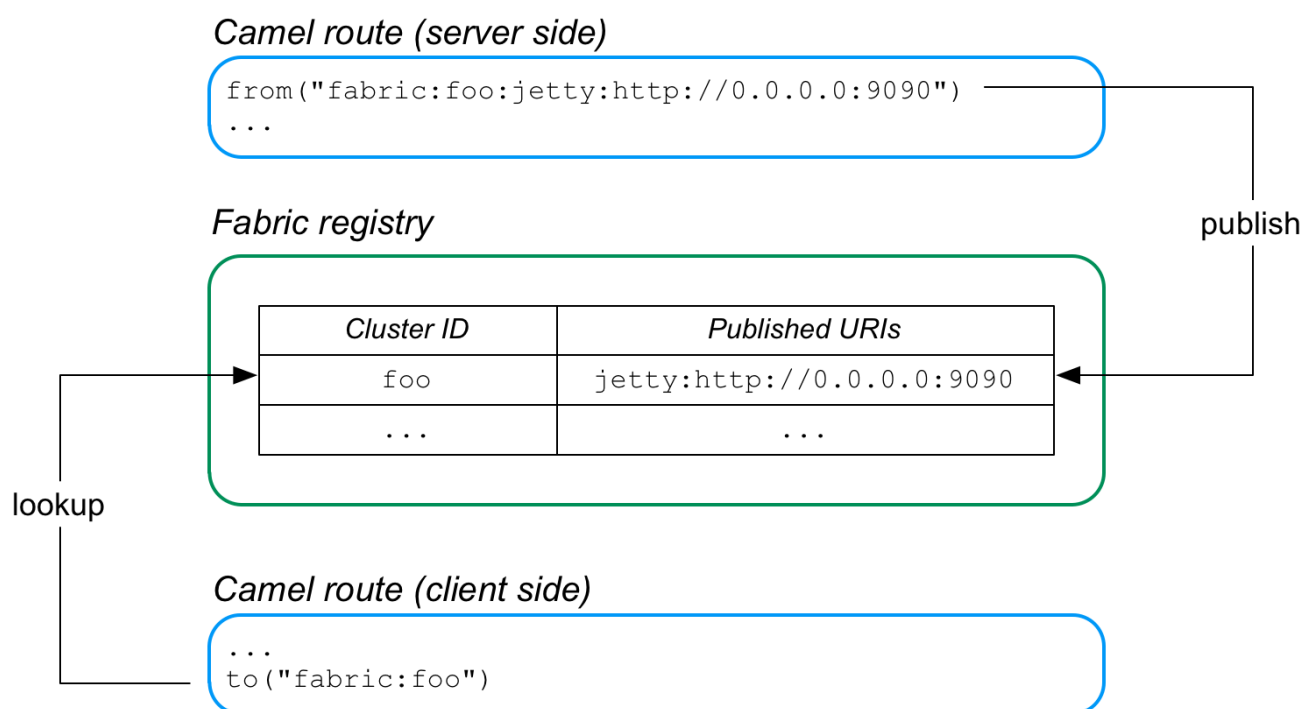
USE CASES FOR FABRIC ENDPOINTS

Fabric endpoints essentially provide a discovery mechanism for Apache Camel endpoints. For example, they support the following basic use cases:

- [the section called “Location discovery”](#).
- [the section called “Load-balancing cluster”](#).

LOCATION DISCOVERY

[Figure 34.1, “Location Discovery through Fabric”](#) gives an overview of how Fabric endpoints enable location discovery at run time.

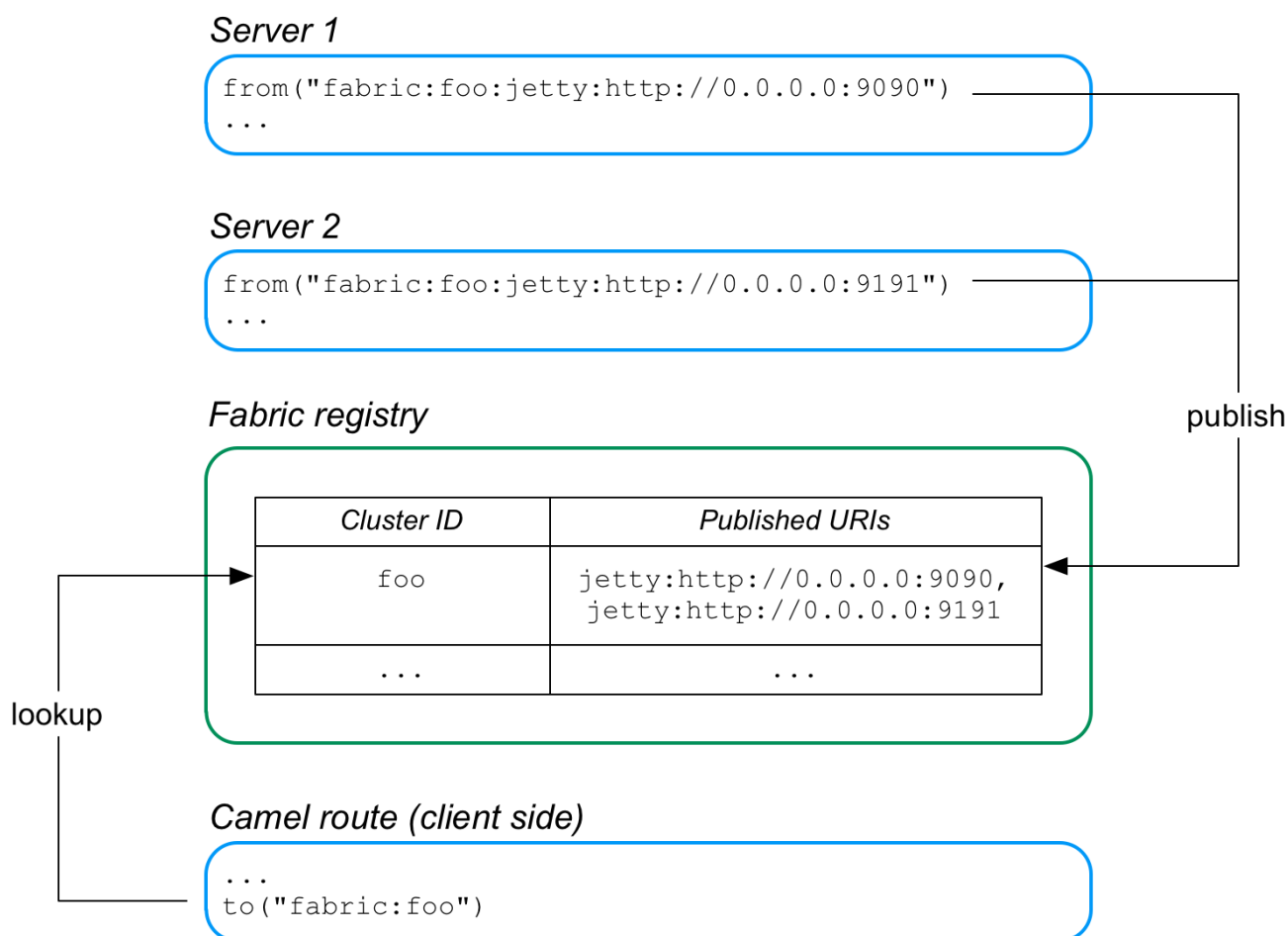
Figure 34.1. Location Discovery through Fabric

The server side of this application is defined by a route that starts with a Fabric endpoint, where the Fabric endpoint publishes the URI, **jetty:http://0.0.0.0:9090**. When this route is started, it automatically registers the Jetty URI in the fabric registry, under the cluster ID, **foo**.

The client side of the application is defined by a route that ends with the Fabric endpoint, **fabric:foo**. Now, when the client route starts, it automatically looks up the ID **foo**, in the fabric registry and retrieves the associated Jetty endpoint URI. The client then creates a producer endpoint using the discovered Jetty URI and connects to the corresponding server port.

LOAD-BALANCING CLUSTER

Figure 34.2, “Load Balancing through Fabric” gives an overview of how Fabric endpoints enable you to create a load-balancing cluster.

Figure 34.2. Load Balancing through Fabric

In this case, two Jetty servers are created, with the URIs, **jetty:http://0.0.0.0:9090** and **jetty:http://0.0.0.0:9191**. Because these published URIs are both prefixed by **fabric:foo:**, both of the Jetty URIs are registered under the same cluster ID, **foo**, in the fabric registry.

Now, when the client routes starts, it automatically looks up the ID, **foo**, in the fabric registry. Because the **foo** ID is associated with multiple endpoint URIs, fabric implements a random load balancing algorithm to choose one of the available URIs. The client then creates a producer endpoint, using the chosen URI.

AUTO-RECONNECT FEATURE

Fabric endpoints support auto-reconnection. So, if a client endpoint (producer endpoint) loses its connection to a server endpoint, it will automatically go back to the fabric registry, ask for another URI, and then connect to the new URI.

PUBLISHING AN ENDPOINT URI

To publish an endpoint URI, **PublishedURI**, in the fabric registry, define a fabric endpoint with the publisher syntax, **FabricScheme:ClusterID:PublishedURI**. Note that this syntax can only be used in a consumer endpoint (that is, an endpoint that appears in a **from** DSL command).

[Example 34.1, "Publishing a URI"](#) shows a route that implements a Jetty HTTP server, where the Jetty URI is published to the fabric registry under the ID, **cluster**. The route is a simple HTTP server that returns the constant message, **Response from Zookeeper agent**,

in the body of the HTTP response.

Example 34.1. Publishing a URI

```
<?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">

  <reference id="curator"
    interface="org.apache.curator.framework.CuratorFramework"/>

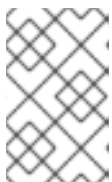
  <bean id="fabric-camel" class="io.fabric8.camel.FabricComponent">
    <property name="curator" ref="curator"/>
  </bean>

  <camelContext id="camel" trace="false"
    xmlns="http://camel.apache.org/schema/blueprint">
    <route id="fabric-server">
      <from uri="fabric-
        camel:cluster:jetty:http://0.0.0.0:9090/fabric"/>
      <log message="Request received : ${body}"/>
      <setHeader headerName="karaf.name">
        <simple>${sys.karaf.name}</simple>
      </setHeader>
      <transform>
        <simple>Response from Zookeeper agent</simple>
      </transform>
    </route>
  </camelContext>

</blueprint>
```

Note the following points about the preceding sample:

- A reference to the `org.apache.curator.framework.CuratorFramework` OSGi service is created using the `reference` element. [Apache Curator](#) is a ZooKeeper client component that manages connections to the ZooKeeper server, providing a high quality of service (including automatic reconnection).



NOTE

The `CuratorFramework` OSGi service is provided by the fabric agent. Hence, a fabric agent must already be installed and running in the container where the route is deployed.

- The Fabric component uses the `CuratorFramework` object to connect to the ZooKeeper server (Fabric registry).
- The `from` DSL command defines the fabric URI, `fabric-camel:cluster:jetty:http://0.0.0.0:9090/fabric`. At run time, this causes two things to happen:

- The specified `jetty` URI is published to the fabric registry under the cluster ID, `cluster`.
- The Jetty endpoint is activated and used as the consumer endpoint of the route (just as if it had been specified without the `fabric-camel:cluster:` prefix).

Because the route is implemented in blueprint XML, you would normally add the file containing this code to the `src/main/resources/OSGI-INF/blueprint` directory of a Maven project.

LOOKING UP AN ENDPOINT URI

To look up a URI in the fabric registry, simply specify the fabric endpoint URI with an ID, in the format, `FabricScheme:ClusterID`. This syntax is used in a producer endpoint (for example, an endpoint that appears in a `to` DSL command).

[Example 34.2, “Looking up a URI”](#) shows a route that implements a HTTP client, where the HTTP endpoint is discovered dynamically at run time, by looking up the specified ID, `cluster`, in the fabric registry.

Example 34.2. Looking up a URI

```
<?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">

  <reference id="curator"
    interface="org.apache.curator.framework.CuratorFramework"/>

  <bean id="fabric-camel" class="io.fabric8.camel.FabricComponent">
    <property name="curator" ref="curator"/>
  </bean>

  <camelContext id="camel" trace="false"
    xmlns="http://camel.apache.org/schema/blueprint">

    <route id="fabric-client">
      <from uri="timer://foo?fixedRate=true&period=10000"/>
      <setBody>
        <simple>Hello from Zookeeper server</simple>
      </setBody>
      <to uri="fabric-camel:cluster"/>
      <log message=">>> ${body} : ${header.karaf.name}"/>
    </route>

  </camelContext>

  <reference interface="org.apache.camel.spi.ComponentResolver"
    filter="(component=jetty)"/>

</blueprint>
```

The client route also needs a reference to the `org.apache.curator.framework.CuratorFramework` OSGi service, which the Fabric component uses to connect to the underlying fabric.

Because the route is implemented in blueprint XML, you would normally add the file containing this code to the `src/main/resources/OSGI-INF/blueprint` directory of a Maven project.

LOAD-BALANCING EXAMPLE

In principle, implementing load balancing is easy using fabric endpoints. All that you have to do is to publish more than one endpoint URI under the *same* cluster ID. Now, when a client looks up that cluster ID, it gets a random selection out of the list of available endpoint URIs.

The servers in the load-balancing cluster have almost the same configuration. Essentially, the only difference between them is that they publish an endpoint URI with a different hostname and/or IP port. Instead of creating a separate OSGi bundle for every single server in the load-balancing cluster, however, it is better to define a template that enables you to specify the host or port using a configuration variable.

[Example 34.3, “Server Template for a Load-Balancing Cluster”](#) illustrates the template approach to defining servers in a load-balancing cluster.

Example 34.3. Server Template for a Load-Balancing Cluster

```
<?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"
  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">

  <!-- osgi blueprint property placeholder -->
  <cm:property-placeholder
    id="myConfig"
    persistent-
id="io.fabric8.examples.camel.loadbalancing.server"/>

  <reference id="curator"
interface="org.apache.curator.framework.CuratorFramework"/>

  <bean id="fabric-camel" class="io.fabric8.camel.FabricComponent">
    <property name="curator" ref="curator"/>
  </bean>

  <camelContext id="camel" trace="false"
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:myConfig"
      prefixToken="[[" suffixToken="]]"/>
```

```

        <route id="fabric-server">
            <from uri="fabric-camel:cluster:jetty:http://0.0.0.0:
[[portNumber]]/fabric"/>
            <log message="Request received : ${body}"/>
            <setHeader headerName="karaf.name">
                <simple>${sys.karaf.name}</simple>
            </setHeader>
            <transform>
                <simple>Response from Zookeeper agent</simple>
            </transform>
        </route>
    </camelContext>

</blueprint>

```

First of all, you need to initialize the OSGi blueprint property placeholder. The property placeholder mechanism enables you to read property settings from the OSGi Config Admin service and substitute the properties in the blueprint configuration file. In this example, the property placeholder accesses properties from the `io.fabric8.examples.camel.loadbalancing.server` persistent ID. A persistent ID in the OSGi Config Admin service identifies a collection of related property settings. After initializing the property placeholder, you can access any property values from the persistent ID using the syntax, `[[PropName]]`.

The Fabric endpoint URI exploits the property placeholder mechanism to substitute the value of the Jetty port, `[[portNumber]]`, at run time. At deploy time, you can specify the value of the `portName` property. For example, if using a custom feature, you could specify the property in the feature definition (see [Add OSGi configurations to the feature](#)). Alternatively, you can specify configuration properties when defining deployment profiles in the *Fuse Management Console*.

OSGI BUNDLE PLUG-IN CONFIGURATION

When defining an OSGi bundle that uses Fabric endpoints, the **Import -Package** bundle header must be configured to import the following Java packages:

```

io.fabric8.zookeeper.spring
io.fabric8.zookeeper

```

For example, assuming that you use Maven to build your application, [Example 34.4, “Maven Bundle Plug-In Configuration”](#) shows how you can configure the Maven bundle plug-in to import the required packages.

Example 34.4. Maven Bundle Plug-In Configuration

```

<project ... >
    ...
    <build>
        <defaultGoal>install</defaultGoal>
        <plugins>
            ...
            <plugin>
                <groupId>org.apache.felix</groupId>

```



```
<artifactId>maven-bundle-plugin</artifactId>
<extensions>>true</extensions>
<configuration>
  <instructions>
    <Bundle-
SymbolicName>${project.groupId}.${project.artifactId}</Bundle-
SymbolicName>
      <Import-Package>
        io.fabric8.zookeeper.spring,
        io.fabric8.zookeeper,
        *
      </Import-Package>
    </instructions>
  </configuration>
</plugin>
</plugins>
</build>
...
</project>
```

CHAPTER 35. FACEBOOK

FACEBOOK COMPONENT

Available as of Camel 2.12

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>
```

URI FORMAT

```
facebook://[endpoint]?[options]
```

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 omitted but that will only allow access to application APIs.

You can also configure these options directly in an endpoint URI.

| Option | Description |
|------------------|------------------------|
| oAuthAppId | The application Id |
| oAuthAppSecret | The application Secret |
| oAuthAccessToken | The user access token |

In addition to the above settings, non-mandatory options below can be used to configure the underlying Facebook4J runtime through either the component's **configuration** property or in an endpoint URI.

| Option | Description | Default Value |
|--------------------------|---|---|
| oAuthAuthorizationURL | OAuth authorization URL | https://www.facebook.com/dialog/oauth |
| oAuthPermissions | Default OAuth permissions. Comma separated permission names. See https://developers.facebook.com/docs/reference/login/#permissions for the detail | null |
| oAuthAccessTokenURL | OAuth access token URL | https://graph.facebook.com/oauth/access_token |
| debugEnabled | Enables debug output. Effective only with the embedded logger | false |
| gzipEnabled | Use Facebook GZIP encoding | true |
| httpConnectionTimeout | Http connection timeout in milliseconds | 20000 |
| httpDefaultMaxPerRoute | HTTP maximum connections per route | 2 |
| httpMaxTotalConnections | HTTP maximum total connections | 20 |
| httpProxyHost | HTTP proxy server host name | null |
| httpProxyPassword | HTTP proxy server password | null |
| httpProxyPort | HTTP proxy server port | null |
| httpProxyUser | HTTP proxy server user name | null |
| httpReadTimeout | Http read timeout in milliseconds | 120000 |
| httpRetryCount | Number of HTTP retries | 0 |
| httpRetryIntervalSeconds | HTTP retry interval in seconds | 5 |

| | | |
|--------------------------|--|---|
| httpStreamingReadTimeout | HTTP streaming read timeout in milliseconds | 40000 |
| jsonStoreEnabled | If set to true, raw JSON forms will be stored in DataObjectFactory | false |
| mbeanEnabled | If set to true, Facebook4J mbean will be registered | false |
| prettyDebugEnabled | prettify JSON debug output if set to true | false |
| restBaseURL | API base URL | https://graph.facebook.com/ |
| useSSL | Use SSL | true |
| videoBaseURL | Video API base URL | https://graph-video.facebook.com/ |
| clientURL | Facebook4J API client URL | http://facebook4j.org/en/facebook4j-<ersion>xml |
| clientVersion | Facebook4J client API version | 1.1.12 |

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.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 `RuntimeException` with a `facebook4j.FacebookException` cause.

| Endpoint | Short Name | Options | Body Type |
|--------------------|-----------------|--------------------------|---|
| Accounts | | | |
| getAccounts | accounts | [reading],[userId] | facebook4j.ResponseList<acebook4j.Account> |
| Activities | | | |
| getActivities | activities | [reading],[userId] | facebook4j.ResponseList<acebook4j.Activity> |
| Albums | | | |
| addAlbumPhoto | addAlbumPhoto | albumId,source,[message] | String |
| commentAlbum | commentAlbum | albumId,message | String |
| createAlbum | createAlbum | albumCreate,[userId] | String |
| getAlbum | album | albumId,[reading] | facebook.Album |
| getAlbumComments | albumComments | albumId,[reading] | facebook4j.ResponseList<acebook4j.Comment> |
| getAlbumCoverPhoto | albumCoverPhoto | albumId | java.net.URL |
| getAlbumLikes | albumLikes | albumId,[reading] | facebook4j.ResponseList<acebook4j.Like> |
| getAlbumPhotos | albumPhotos | albumId,[reading] | facebook4j.ResponseList<acebook4j.Photos> |
| getAlbums | albums | [reading],[userId] | facebook4j.ResponseList<acebook4j.Album> |
| likeAlbum | likeAlbum | albumId | boolean |
| unlikeAlbum | unlikeAlbum | albumId | boolean |
| Checkins | | | |

| | | | |
|--------------------|-----------------|----------------------------|---|
| checkin | checkin | checkinCreate, [userId] | String |
| commentCheckin | commentCheckin | checkinId,message | String |
| getCheckin | checkin | checkinId,[reading] | facebook4j.Checkin |
| getCheckinComments | checkinComments | checkinId,[reading] | facebook4j.ResponseList<facebook4j.Comment> |
| getCheckinLikes | checkinLikes | checkinId,[reading] | facebook4j.ResponseList<facebook4j.Like> |
| getCheckins | checkins | [reading],[userId] | facebook4j.ResponseList<facebook4j.Checkin> |
| likeCheckin | likeCheckin | checkinId | boolean |
| unlikeCheckin | unlikeCheckin | checkinId | boolean |
| Comments | | | |
| deleteComment | deleteComment | commentId | boolean |
| getComment | comment | commentId | facebook4j.Comment |
| getCommentLikes | commentLikes | commentId,[reading] | facebook4j.ResponseList<facebook4j.Like> |
| likeComment | likeComment | commentId | boolean |
| unlikeComment | unlikeComment | commentId | boolean |
| Domains | | | |
| getDomain | domain | domainId | facebook4j.Domain |
| getDomainByName | domainByName | domainName | facebook4j.Domain |
| getDomainsByName | domainsByName | domainNames | java.util.List<facebook4j.Domain> |
| Events | | | |
| createEvent | createEvent | eventUpdate,[userId] | String |

| | | | |
|--------------------------|-----------------------|----------------------------|--|
| deleteEvent | deleteEvent | eventId | boolean |
| deleteEventPicture | deleteEventPicture | eventId | boolean |
| editEvent | editEvent | eventId,eventUpdate | boolean |
| getEvent | event | eventId,[reading] | facebook4j.Event |
| getEventFeed | eventFeed | eventId,[reading] | facebook4j.ResponseList<facebook4j.Post> |
| getEventPhotos | eventPhotos | eventId,[reading] | facebook4j.ResponseList<facebook4j.Photo> |
| getEventPictureURL | eventPictureURL | eventId,[size] | java.net.URL |
| getEvents | events | [reading],[userId] | facebook4j.ResponseList<facebook4j.Event> |
| getEventVideos | eventVideos | eventId,[reading] | facebook4j.ResponseList<facebook4j.Video> |
| getRSVPStatusAsInvited | rsvpStatusAsInvited | eventId,[userId] | facebook4j.ResponseList<facebook4j.RSVPStatus> |
| getRSVPStatusAsNoreply | rsvpStatusAsNoreply | eventId,[userId] | facebook4j.ResponseList<facebook4j.RSVPStatus> |
| getRSVPStatusInAttending | rsvpStatusInAttending | eventId,[userId] | facebook4j.ResponseList<facebook4j.RSVPStatus> |
| getRSVPStatusInDeclined | rsvpStatusInDeclined | eventId,[userId] | facebook4j.ResponseList<facebook4j.RSVPStatus> |
| getRSVPStatusInMaybe | rsvpStatusInMaybe | eventId,[userId] | facebook4j.ResponseList<facebook4j.RSVPStatus> |
| inviteToEvent | inviteToEvent | eventId,[userId],[userIds] | boolean |
| postEventFeed | postEventFeed | eventId,postUpdate | String |
| postEventLink | postEventLink | eventId,link,[message] | String |

| | | | |
|------------------------|------------------------|--|--|
| postEventPhoto | postEventPhoto | eventId,source, [message] | String |
| postEventStatusMessage | postEventStatusMessage | eventId,message | String |
| postEventVideo | postEventVideo | eventId,source, [title,description] | String |
| rsvpEventAsAttending | rsvpEventAsAttending | eventId | boolean |
| rsvpEventAsDeclined | rsvpEventAsDeclined | eventId | boolean |
| rsvpEventAsMaybe | rsvpEventAsMaybe | eventId | boolean |
| uninviteFromEvent | uninviteFromEvent | eventId,userId | boolean |
| updateEventPicture | updateEventPicture | eventId,source | boolean |
| Family | | | |
| getFamily | family | [reading],[userId] | facebook4j.ResponseList<facebook4j.Family> |
| Favorites | | | |
| getBooks | books | [reading],[userId] | facebook4j.ResponseList<facebook4j.Book> |
| getGames | games | [reading],[userId] | facebook4j.ResponseList<facebook4j.Game> |
| getInterests | interests | [reading],[userId] | facebook4j.ResponseList<facebook4j.Interest> |
| getMovies | movies | [reading],[userId] | facebook4j.ResponseList<facebook4j.Movie> |
| getMusic | music | [reading],[userId] | facebook4j.ResponseList<facebook4j.Music> |
| getTelevision | television | [reading],[userId] | facebook4j.ResponseList<facebook4j.Television> |

| Facebook Query Language (FQL) | | | |
|--------------------------------------|------------------------|--|---|
| executeFQL | executeFQL | query,[locale] | facebook4j.internal.org.json.JSONArray |
| executeMultiFQL | executeMultiFQL | queries,[locale] | java.util.Map<tring,facebook4j.internal.org.json.JSONArray> |
| Friends | | | |
| addFriendlistMember | addFriendlistMember | friendlistId,userId | boolean |
| createFriendlist | createFriendlist | friendlistName,[userId] | String |
| deleteFriendlist | deleteFriendlist | friendlistId | boolean |
| getBelongsFriend | belongsFriend | friendId,[reading],[userId] | facebook4j.ResponseList<acebook4j.Friend> |
| getFriendlist | friendlist | friendlistId,[reading] | facebook4j.FriendList |
| getFriendlistMembers | friendlistMembers | friendlistId | facebook4j.ResponseList<acebook4j.Friend> |
| getFriendlists | friendlists | [reading],[userId] | facebook4j.ResponseList<acebook4j.FriendList> |
| getFriendRequests | friendRequests | [reading],[userId] | facebook4j.ResponseList<acebook4j.FriendRequest> |
| getFriends | friends | [reading],[userId] | facebook4j.ResponseList<acebook4j.Friend> |
| getMutualFriends | mutualFriends | [friendUserId],[reading],[userId1,userId2] | facebook4j.ResponseList<acebook4j.Friend> |
| removeFriendlistMember | removeFriendlistMember | friendlistId,userId | boolean |
| Games | | | |

| | | | |
|------------------------|------------------------|-----------------------------|---|
| deleteAchievement | deleteAchievement | achievementURL, [userId] | boolean |
| deleteScore | deleteScore | [userId] | boolean |
| getAchievements | achievements | [reading],[userId] | facebook4j.ResponseList<facebook4j.Achievement> |
| getScores | scores | [reading],[userId] | facebook4j.ResponseList<facebook4j.Score> |
| postAchievement | postAchievement | achievementURL, [userId] | String |
| postScore | postScore | scoreValue,[userId] | String |
| Groups | | | |
| getGroup | group | groupId,[reading] | facebook4j.Group |
| getGroupDocs | groupDocs | groupId,[reading] | facebook4j.ResponseList<facebook4j.GroupDoc> |
| getGroupFeed | groupFeed | groupId,[reading] | facebook4j.ResponseList<facebook4j.Post> |
| getGroupMembers | groupMembers | groupId,[reading] | facebook4j.ResponseList<facebook4j.GroupMember> |
| getGroupPictureURL | groupPictureURL | groupId | java.net.URL |
| getGroups | groups | [reading],[userId] | facebook4j.ResponseList<facebook4j.Group> |
| postGroupFeed | postGroupFeed | groupId,postUpdate | String |
| postGroupLink | postGroupLink | groupId,link, [message] | String |
| postGroupStatusMessage | postGroupStatusMessage | groupId,message | String |
| Insights | | | |

| | | | |
|------------------|--------------|-------------------------------|--|
| getInsights | insights | objectId,metric, [reading] | facebook4j.ResponseList<facebook4j.Insight> |
| Likes | | | |
| getUserLikes | userLikes | [reading],[userId] | facebook4j.ResponseList<facebook4j.Like> |
| Links | | | |
| commentLink | commentLink | linkId,message | String |
| getLink | link | linkId,[reading] | facebook4j.Link |
| getLinkComments | linkComments | linkId,[reading] | facebook4j.ResponseList<facebook4j.Comment> |
| getLinkLikes | linkLikes | linkId,[reading] | facebook4j.ResponseList<facebook4j.Like> |
| likeLink | likeLink | linkId | boolean |
| unlikeLink | unlikeLink | linkId | boolean |
| Locations | | | |
| getLocations | locations | [reading],[userId] | facebook4j.ResponseList<facebook4j.Location> |
| Messages | | | |
| getInbox | inbox | [reading],[userId] | facebook4j.InboxResponseList<facebook4j.Inbox> |
| getMessage | message | messageId,[reading] | facebook4j.Message |
| getOutbox | outbox | [reading],[userId] | facebook4j.ResponseList<facebook4j.Message> |
| getUpdates | updates | [reading],[userId] | facebook4j.ResponseList<facebook4j.Message> |
| Notes | | | |

| | | | |
|------------------------|------------------------|--|--|
| commentNote | commentNote | noteId,message | String |
| createNote | createNote | subject,message, [userId] | String |
| getNote | note | noteId,[reading] | facebook4j.Note |
| getNoteComments | noteComments | noteId,[reading] | facebook4j.ResponseList<facebook4j.Comment> |
| getNoteLikes | noteLikes | noteId,[reading] | facebook4j.ResponseList<facebook4j.Like> |
| getNotes | notes | [reading],[userId] | facebook4j.ResponseList<facebook4j.Note> |
| likeNote | likeNote | noteId | boolean |
| unlikeNote | unlikeNote | noteId | boolean |
| Notifications | | | |
| getNotifications | notifications | [includeRead], [reading],[userId] | facebook4j.ResponseList<facebook4j.Notification> |
| markNotificationAsRead | markNotificationAsRead | notificationId | boolean |
| Permissions | | | |
| getPermissions | permissions | [userId] | java.util.List<facebook4j.Permission> |
| revokePermission | revokePermission | permissionName, [userId] | boolean |
| Photos | | | |
| addTagToPhoto | addTagToPhoto | photoId,[toUserId], [toUserIds], [tagUpdate] | boolean |
| commentPhoto | commentPhoto | photoId,message | String |
| deletePhoto | deletePhoto | photoId | boolean |

| | | | |
|------------------|------------------|---|---|
| getPhoto | photo | photoId,[reading] | facebook4j.Photo |
| getPhotoComments | photoComments | photoId,[reading] | facebook4j.ResponseList<facebook4j.Comment> |
| getPhotoLikes | photoLikes | photoId,[reading] | facebook4j.ResponseList<facebook4j.Like> |
| getPhotos | photos | [reading],[userId] | facebook4j.ResponseList<facebook4j.Photo> |
| getPhotoURL | photoURL | photoId | java.net.URL |
| getTagsOnPhoto | tagsOnPhoto | photoId,[reading] | facebook4j.ResponseList<facebook4j.Tag> |
| likePhoto | likePhoto | photoId | boolean |
| postPhoto | postPhoto | source,[message],[place],[noStory],[userId] | String |
| unlikePhoto | unlikePhoto | photoId | boolean |
| updateTagOnPhoto | updateTagOnPhoto | photoId,[toUserId],[tagUpdate] | boolean |
| Pokes | | | |
| getPokes | pokes | [reading],[userId] | facebook4j.ResponseList<facebook4j.Poke> |
| Posts | | | |
| commentPost | commentPost | postId,message | String |
| deletePost | deletePost | postId | boolean |
| getFeed | feed | [reading],[userId] | facebook4j.ResponseList<facebook4j.Post> |
| getHome | home | [reading] | facebook4j.ResponseList<facebook4j.Post> |
| getLinks | links | [reading],[userId] | facebook4j.ResponseList<facebook4j.Link> |

| | | | |
|------------------------|---------------------|---|---|
| getPost | post | postId,[reading] | facebook4j.Post |
| getPostComments | postComments | postId,[reading] | facebook4j.ResponseList<facebook4j.Comment> |
| getPostLikes | postLikes | postId,[reading] | facebook4j.ResponseList<facebook4j.Like> |
| getPosts | posts | [reading],[userId] | facebook4j.ResponseList<facebook4j.Post> |
| getStatuses | statuses | [reading],[userId] | facebook4j.ResponseList<facebook4j.Post> |
| getTagged | tagged | [reading],[userId] | facebook4j.ResponseList<facebook4j.Post> |
| likePost | likePost | postId | boolean |
| postFeed | postFeed | postUpdate,[userId] | String |
| postLink | postLink | link,[message],[userId] | String |
| postStatusMessage | postStatusMessage | message,[userId] | String |
| unlikePost | unlikePost | postId | boolean |
| Questions | | | |
| addQuestionOption | addQuestionOption | questionId,optionDescription | String |
| createQuestion | createQuestion | question,[options],[allowNewOptions],[userId] | String |
| deleteQuestion | deleteQuestion | questionId | boolean |
| getQuestion | question | questionId,[reading] | facebook4j.Question |
| getQuestionOptions | questionOptions | questionId,[reading] | facebook4j.ResponseList<facebook4j.Question.Option> |
| getQuestionOptionVotes | questionOptionVotes | questionId | facebook4j.ResponseList<facebook4j.QuestionVotes> |

| | | | |
|--------------------|--------------------|---|--|
| getQuestions | questions | [reading],[userId] | facebook4j.ResponseList<facebook4j.Question> |
| getSubscribedto | subscribedto | [reading],[userId] | facebook4j.ResponseList<facebook4j.Subscribedto> |
| getSubscribers | subscribers | [reading],[userId] | facebook4j.ResponseList<facebook4j.Subscriber> |
| Test Users | | | |
| createTestUser | createTestUser | appId,[name],[userLocale],[permissions] | facebook4j.TestUser |
| deleteTestUser | deleteTestUser | testUserId | boolean |
| getTestUsers | testUsers | appId | java.util.List<facebook4j.TestUser> |
| makeFriendTestUser | makeFriendTestUser | testUser1,testUser2 | boolean |
| Users | | | |
| getMe | me | [reading] | facebook4j.User |
| getPictureURL | pictureURL | [size],[userId] | java.net.URL |
| getUser | user | userId,[reading] | facebook4j.User |
| getUsers | users | ids | java.util.List<facebook4j.User> |
| Videos | | | |
| commentVideo | commentVideo | videoId,message | String |
| getVideo | video | videoId,[reading] | facebook4j.Video |
| getVideoComments | videoComments | videoId,[reading] | facebook4j.ResponseList<facebook4j.Comment> |
| getVideoCover | videoCover | videoId | java.net.URL |

| | | | |
|-----------------|-------------|---|--|
| getVideoLikes | videoLikes | videoid,[reading] | facebook4j.ResponseList<facebook4j.Like> |
| getVideos | videos | [reading],[userId] | facebook4j.ResponseList<facebook4j.Video> |
| likeVideo | likeVideo | videoid | boolean |
| postVideo | postVideo | source, [title,description], [userId] | String |
| unlikeVideo | unlikeVideo | videoid | boolean |
| Search | | | |
| search | search | query,[reading] | facebook4j.ResponseList<facebook4j.internal.org.json.JSONObject> |
| searchCheckins | checkins | [reading] | facebook4j.ResponseList<facebook4j.Checkin> |
| searchEvents | events | query,[reading] | facebook4j.ResponseList<facebook4j.Event> |
| searchGroups | groups | query,[reading] | facebook4j.ResponseList<facebook4j.Group> |
| searchLocations | locations | [center,distance], [reading],[placeld] | facebook4j.ResponseList<facebook4j.Location> |
| searchPlaces | places | query,[reading], [center,distance] | facebook4j.ResponseList<facebook4j.Place> |
| searchPosts | posts | query,[reading] | facebook4j.ResponseList<facebook4j.Post> |
| searchUsers | users | query,[reading] | facebook4j.ResponseList<facebook4j.User> |

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).

1. URI Options

| Name | Type | Description |
|-----------------|--------------------------|--|
| achievementURL | java.net.URL | The unique URL of the achievement |
| albumCreate | facebook4j.AlbumCreate | The facebook Album to be created |
| albumId | String | The album ID |
| allowNewOptions | boolean | True if allows other users to add new options |
| appId | String | The ID of the Facebook Application |
| center | facebook4j.GeoLocation | Location latitude and longitude |
| checkinCreate | facebook4j.CheckinCreate | The checkin to be created. Deprecated , instead create a Post with an attached location |
| checkinId | String | The checkin ID |
| commentId | String | The comment ID |
| description | String | The description text |
| distance | int | Distance in meters |
| domainId | String | The domain ID |
| domainName | String | The domain name |
| domainNames | String[] | The domain names |
| eventId | String | The event ID |

| | | |
|----------------|------------------------|---|
| eventUpdate | facebook4j.EventUpdate | The event to be created or updated |
| friendId | String | The friend ID |
| friendUserId | String | The friend user ID |
| friendlistId | String | The friend list ID |
| friendlistName | String | The friend list Name |
| groupId | String | The group ID |
| ids | String[] | The ids of users |
| includeRead | boolean | Enables notifications that the user has already read in addition to unread ones |
| link | java.net.URL | Link URL |
| linkId | String | The link ID |
| locale | java.util.Locale | Desired FQL locale |
| message | String | The message text |
| messageId | String | The message ID |
| metric | String | The metric name |
| name | String | Test user name, must be of the form 'first last' |
| noStory | boolean | If set to true, optionally suppresses the feed story that is automatically generated on a user's profile when they upload a photo using your application. |
| noteId | String | The note ID |
| notificationId | String | The notification ID |
| objectId | String | The insight object ID |

| | | |
|-------------------|------------------------|--|
| optionDescription | String | The question's answer option description |
| options | java.util.List<tring> | The question's answer options |
| permissionName | String | The permission name |
| permissions | String | Test user permissions in the format perm1,perm2,... |
| photoId | String | The photo ID |
| place | String | The Facebook ID of the place associated with the Photo |
| placeId | String | The place ID |
| postId | String | The post ID |
| postUpdate | facebook4j.PostUpdate | The post to create or update |
| queries | java.util.Map<tring> | FQL queries |
| query | String | FQL query or search terms for search* endpoints |
| question | String | The question text |
| questionId | String | The question id |
| reading | facebook4j.Reading | Optional reading parameters. See Reading Options(#reading) |
| scoreValue | int | The numeric score with value |
| size | facebook4j.PictureSize | The picture size, one of large, normal, small or square |
| source | facebook4j.Media | The media content from either a java.io.File or java.io.InputStream |
| subject | String | The note of the subject |
| tagUpdate | facebook4j.TagUpdate | Photo tag information |
| testUser1 | facebook4j.TestUser | Test user |

| | | |
|------------|-----------------------|-------------------------------------|
| testUser2 | facebook4j.TestUser | Test user |
| testUserId | String | The ID of the test user |
| title | String | The title text |
| toUserId | String | The ID of the user to tag |
| toUserIds | java.util.List<tring> | The IDs of the users to tag |
| userId | String | The Facebook user ID |
| userId1 | String | The ID of a user |
| userId2 | String | The ID of a user |
| userIds | String[] | The IDs of users to invite to event |
| userLocale | String | The test user locale |
| videoid | String | The video ID |

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.

| Option | Description |
|----------------|---|
| reading.fields | Field names to retrieve, in the format field1,field2,... |
| reading.limit | Limit for number of items to return for list results, e.g. a limit of 10 returns items 1 through 10 |
| reading.offset | Starting offset for list results, e.g. a limit of 10, and offset of 10 returns items 11 through 20 |

| | |
|------------------|---|
| reading.until | A Unix timestamp or strtotime data value that points to the end of the range of time-based data |
| reading.since | A Unix timestamp or strtotime data value that points to the start of the range of time-based data |
| reading.locale | Retrieve localized content in a particular locale, specified as a String with the format language[,country][,variant] |
| reading.with | Retrieve information about objects that have location information attached, set it to true |
| reading.metadata | Use Facebook Graph API Introspection to retrieve object metadata, set it to true |
| reading.filter | User's stream filter key. See Facebook stream_filter |

MESSAGE HEADER

Any of the [URI options#urioptions](#) can be provided in a message header for producer endpoints with **CamelFacebook.** prefix.

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.

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., all statuses on your home feed:

```
from("facebook://home?delay=5")
  .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 36. FILE2

FILE COMPONENT - APACHE CAMEL 2.0 ONWARDS

The File component provides access to file systems, allowing files to be processed by any other Apache Camel [Components](#) or messages from other components to be saved to disk.

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&...**



NOTE

Apache Camel only supports 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.



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 Apache Camel thinks the file is not locked by another process and start consuming it. Therefore you have to do your own investigation as to what suits your environment. To help with this, Apache Camel provides different **readLock** options and the **doneFileOption** option that you can use. See also the section [the section called “Consuming files from folders where others drop files directly”](#).

URI OPTIONS

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|-------------------|-------------|---|
| autoCreate | true | 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 where the files should be written. |
| bufferSize | 128kb | Write buffer sized in bytes. |
| fileName | null | 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 . From Camel 2.11 onwards the producers support the CamelOverrideFileName header which takes precedence over any existing CamelFileName header; the CamelOverrideFileName 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. |

| | | |
|----------------------------------|--------------|--|
| flatten | false | 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 recived in CamelFileName header will be stripped for any leading paths. |
| charset | null | Camel 2.5: this option is used to specify the encoding of the file, and camel will set the Exchange property with Exchange.CHARSET_NAME with the value of this option. |
| copyAndDeleteOnRenameFail | true | Camel 2.9: 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. |

CONSUMER ONLY

| Name | Default Value | Description |
|------------------------|----------------------|---|
| initialDelay | 1000 | Milliseconds before polling the file/directory starts. |
| delay | 500 | Milliseconds before the next poll of the file/directory. |
| useFixedDelay | true | Set to true to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |
| runLoggingLevel | TRACE | Camel 2.8: The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that. |

| | | |
|-------------------|---------------|--|
| recursive | false | If a directory, will look for files in all the sub-directories as well. |
| delete | false | If true , the file will be deleted after it is processed |
| noop | false | 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 , Apache Camel will set idempotent=true as well, to avoid consuming the same files over and over again. |
| preMove | null | Use Expression such as File Language 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 . |
| move | .camel | Use Expression such as File Language to dynamically set the filename when moving it after processing. To move files into a .done subdirectory just enter .done . |
| moveFailed | null | Use Expression such as File Language to dynamically set the filename when moving failed files after processing. To move files into a error subdirectory just enter error . Note: When moving the files to another location it can/will handle the error when you move it to another location so Apache Camel cannot pick up the file again. |
| include | null | Is used to include files, if filename matches the regex pattern. |
| exclude | null | Is used to exclude files, if filename matches the regex pattern. |

| | | |
|-------------------------------|-------------------|---|
| antInclude | null | Camel 2.10: Ant style filter inclusion, for example antInclude=*{*}*/*{ }.txt . Multiple inclusions may be specified in comma-delimited format. See below for more details about ant path filters. |
| antExclude | null | Camel 2.10: 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. See below for more details about ant path filters. |
| antFilterCaseSensitive | true | Camel 2.11: Ant style filter which is case sensitive or not. |
| idempotent | false | Option to use the Idempotent Consumer EIP pattern to let Apache 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. |
| idempotentKey | Expression | Camel 2.11: 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=\$-\$. |
| idempotentRepository | null | Pluggable repository as a org.apache.camel.processor.idempotent.MessageIdRepository class. Will by default use MemoryMessageIdRepository if none is specified and idempotent is true . |

| | | |
|-----------------------------|-------------------|--|
| inProgressRepository | memory | Pluggable in-progress repository as a org.apache.camel.processor.idempotent.MessageIdRepository class. The in-progress repository is used to account the current in progress files being consumed. By default a memory based repository is used. |
| filter | null | Pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. Will skip files if filter returns false in its accept() method. Apache Camel also ships with an ANT path matcher filter in the camel-spring component. More details in section below. |
| sorter | null | Pluggable sorter as a java.util.Comparator<org.apache.camel.component.file.GenericFile> class. |
| sortBy | null | Built-in sort 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. See sorting section below for details. |
| readLock | markerFile | 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). Apache Camel will wait until the file lock is granted. The readLock option supports the following built-in strategies: <ul style="list-style-type: none"> • changed uses a length/modification timestamp to detect whether the file is currently being copied or not. Will wait at least 1 second 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.

- **rename** attempts to rename the file, in order to test whether we can get an exclusive read-lock.
- **none** is for no read locks at all.



WARNI NG

None of the read lock strategies are suitable for use in clustered mode. That is, you cannot have multiple consumers attempting to read the same file in the same directory. In this case, the read locks will not function reliably.

| | | |
|------------------------------|-------------------------------------|--|
| readLockTimeout | 0 (for FTP, 2000) | Optional timeout in milliseconds for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout triggered, then Apache Camel will skip the file. At next poll Apache Camel, will try the file again, and this time maybe the read-lock could be granted. Currently fileLock , changed and rename support the timeout. |
| readLockCheckInterval | 1000 (for FTP, 5000) | Camel 2.6: 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 <i>slow writes</i> . The default of 1 sec. may be <i>too fast</i> if the producer is very slow writing the file. |
| readLockMinLength | 1 | Camel 2.10.1: 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. |
| readLockLoggingLevel | WARN | Camel 2.12: 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, rename. |

| | | |
|----------------------------------|--------------------------|---|
| exclusiveReadLockStrategy | null | Pluggable read-lock as a org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy implementation. |
| minDepth | 0 | Camel 2.8: 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. This option is not supported by FTP consumer. |
| maxDepth | Integer.MAX_VALUE | Camel 2.8: The maximum depth to traverse when recursively processing a directory. This option is not supported by FTP consumer. |
| doneFileName | null | Camel 2.6: If provided, Camel will only consume files if a <i>done</i> file exists. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The <i>done</i> file is always expected in the same folder as the original file. See <i>using done file</i> and <i>writing done file</i> sections for examples. |
| processStrategy | null | 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 <i>ready</i> file exists. If this option is set then the readLock option does not apply. |

| | | |
|-----------------------------------|--------------|--|
| maxMessagesPerPoll | 0 | An integer that defines 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 having the server read thousands of files as it starts up. Set a value of 0 or negative to disabled it. |
| startingDirectoryMustExist | false | 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 throw an exception, if the directory doesn't exist. |
| directoryMustExist | false | Similar to startingDirectoryMustExist but this applies during polling recursive sub-directories. |
| scheduler | null | Camel 2.12: To use a custom scheduler to trigger the consumer to run. See more details at Polling Consumer , for example there is a Quartz2 , and Spring based scheduler that supports CRON expressions. |

| | | |
|------------------------------|----------|---|
| backoffMultiplier | 0 | Camel 2.12: 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. See more details at Polling Consumer . |
| backoffIdleThreshold | 0 | Camel 2.12: The number of subsequent idle polls that should happen before the backoffMultiplier should kick-in. |
| backoffErrorThreshold | 0 | Camel 2.12: The number of subsequent error polls (failed due some error) that should happen before the backoffMultiplier should kick-in. |

DEFAULT BEHAVIOR FOR FILE CONSUMER

- By default the file is locked for the duration of the processing.
- After the route has completed, files are moved into the `.camel` subdirectory, so that they appear to be deleted.
- The File Consumer will always skip any file whose name starts with a dot, such as `.`, `.camel`, `.m2` or `.groovy`.
- Only files (not directories) are matched for valid filename, if options such as: `includeNamePrefix`, `includeNamePostfix`, `excludeNamePrefix`, `excludeNamePostfix`, `regexPattern` are used.

PRODUCER ONLY

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|-------------------|-----------------|---|
| fileExist | Override | <p>What to do if a file already exists with the same name. The following values can be specified: Override, Append, Fail, Ignore, Move, and TryRename (Camel 2.11.1). 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. The Move option requires Camel 2.10.1 onwards, and the corresponding 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. TryRenameCamel 2.11.1 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.</p> |
| tempPrefix | null | <p>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.</p> |

| | | |
|------------------------------|--------------|--|
| tempFileName | null | Camel 2.1: 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 . |
| keepLastModified | false | Camel 2.2: 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 <i>cannot</i> use this option with any of the ftp producers. |
| eagerDeleteTargetFile | true | Camel 2.3: 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. |

| | | |
|----------------------|--------------|---|
| doneFileName | null | Camel 2.6: If provided, then Camel will write a 2nd <i>done</i> file when the original file has been written. The <i>done</i> 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 <i>done</i> file will always be written in the same folder as the original file. See <i>writing done file</i> section for examples. |
| allowNullBody | false | Camel 2.10.1: 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 <code>GenericFileWriteException</code> of 'Cannot write null body to file.' will be thrown. If the <code>`fileExist`</code> option is set to 'Override', then the file will be truncated, and if set to <code>`append`</code> the file will remain unchanged. |
| forceWrites | true | Camel 2.10.5/2.11: 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. |

DEFAULT BEHAVIOR FOR FILE PRODUCER

- By default it will override any existing file, if one exist with the same name. In Apache Camel 1.x the **Append** is the default for the file producer. We have changed this to **Override** in Apache Camel 2.0 as this is also the default file operation using `java.io.File`. And also the default for the FTP library we use in the [camel-ftp](#) component.

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 **preMove** options is 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, Apache 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://inbox?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.

FINE GRAINED CONTROL OVER MOVE AND PREMOVE OPTION

The **move** and **preMove** option is [Expression](#)-based, so we have the full power of the [File Language](#) to do advanced configuration of the directory and name pattern. Apache Camel will, in fact, internally convert the directory name you enter into a [File Language](#) expression. So when we enter **move=.done** Apache Camel will convert this into: **\${file:parent}/.done/\${file:onlyname}**. This is only done if Apache Camel detects that you have not provided a **\${ }** in the option value yourself. So when you enter an expression containing **\${ }**, the expression is interpreted as a File Language expression.

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}
```

ABOUT MOVEFAILED

The `moveFailed` option allows you to move files that **could not** be processed successfully 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:name.ext}`.

See more examples at [File Language](#).

MESSAGE HEADERS

The following headers are supported by this component:

FILE PRODUCER ONLY

| Header | Description |
|------------------------------------|---|
| <code>CamelFileName</code> | Specifies the name of the file to write (relative to the endpoint directory). The name can be a String ; a String with a File Language or Simple expression; or an Expression object. If it's null then Apache Camel will auto-generate a filename based on the message unique ID. |
| <code>CamelFileNameProduced</code> | 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. |
| <code>CamelOverruleFileName</code> | Camel 2.11: Is used for overruling <code>CamelFileName</code> 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 <code>fileName</code> has been configured, then this is still being evaluated. |

FILE CONSUMER ONLY

| Header | Description |
|----------------------------|---|
| <code>CamelFileName</code> | 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. |
| CamelFileParent | The parent path. |
| CamelFileLength | A long value containing the file size. |
| CamelFileLastModified | A Date value containing the last modified timestamp of the file. |

BATCH CONSUMER

This component implements the [Batch Consumer](#).

EXCHANGE PROPERTIES, FILE CONSUMER ONLY

As the file consumer is **BatchConsumer** it supports batching the files it polls. By batching it means that Apache Camel will add some properties to the [Exchange](#) so you know the number of files polled the current index in that order.

| 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 exists in this batch and for instance let the [Aggregator](#) aggregate this number of files.

COMMON GOTCHAS WITH FOLDER AND FILENAMES

When Apache Camel is producing files (writing files) there are a few gotchas affecting how to set a filename of your choice. By default, Apache 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");
```

Or 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");
```

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.

CONSUMING FILES FROM FOLDERS WHERE OTHERS DROP FILES DIRECTLY

Beware if you consume files from a folder where other applications write files 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 read lock 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`) to signal when a file is done and ready to be consumed.

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 file name done exists in the same directory as the target files. Camel will automatically delete the done file when it's done consuming the files.

However its 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

WRITING DONE FILES

Available as of Camel 2.6

After you have written af file you may want to write an additional *done* file as a kinda 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 its 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.

READ FROM A DIRECTORY AND WRITE TO ANOTHER DIRECTORY

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

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**.

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
```

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
```

READING FROM A DIRECTORY AND THE DEFAULT MOVE OPERATION

Apache 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
```

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.

READ FILES FROM A DIRECTORY AND SEND THE CONTENT TO A JMS QUEUE

```
from("file://inputdir/").convertBodyTo(String.class).to("jms:test.queue")
```

By default the file endpoint sends a **FileMessage** which contains a **File** object as the body. If you send this directly to the JMS component the JMS message will only contain the **File** object but not the content. By converting the **File** to a **String**, the message will contain the file contents, which is probably what you want.

The route above using Spring DSL:

```
<route>
  <from uri="file://inputdir/"/>
  <convertBodyTo type="java.lang.String"/>
  <to uri="jms:test.queue"/>
</route>
```

WRITING TO FILES

Apache 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 written to a directory.

```
public void testToFile() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedFileExists("target/test-reports/report.txt");

    template.sendBody("direct:reports", "This is a great report");

    assertMockEndpointsSatisfied();
}

protected JndiRegistry createRegistry() throws Exception {
    // bind our processor in the registry with the given id
    JndiRegistry reg = super.createRegistry();
    reg.bind("processReport", new ProcessReport());
    return reg;
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // the reports from the seda queue is processed by our
            processor
            // before they are written to files in the target/reports
            directory

            from("direct:reports").processRef("processReport").to("file://target/test-
            reports", "mock:result");
        }
    };
}

private static class ProcessReport implements Processor {

    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        // do some business logic here
    }
}
```

```

        // set the output to the file
        exchange.getOut().setBody(body);

        // set the output filename using java code logic, notice that this
is done by setting
        // a special header property of the out exchange
        exchange.getOut().setHeader(Exchange.FILE_NAME, "report.txt");
    }
}

```

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.

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.

AVOIDING READING THE SAME FILE MORE THAN ONCE (IDEMPOTENT CONSUMER)

Apache 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&dempotentKey=${file:name}-
  ${file-size}"/>
  <to uri="bean:processInbox"/>
</route>

```

By default Apache Camel uses an 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&dempotentRepository=#myStore"/>
  <to uri="bean:processInbox"/>
</route>

```

Apache 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

```

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 and when the file grows larger, Apache Camel will truncate the file store and rebuild 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:

```

<!-- this is our file based idempotent store configured to use the
.filestore.dat as file -->
<bean id="fileStore"
class="org.apache.camel.processor.idempotent.FileIdempotentRepository">
  <!-- the filename for the store -->
  <property name="fileStore"
value="target/fileidempotent/.filestore.dat"/>
  <!-- the max filesize in bytes for the file. Apache Camel will trunk
and flush the cache
  if the file gets bigger -->
  <property name="maxFileStoreSize" value="512000"/>
  <!-- the number of elements in our store -->
  <property name="cacheSize" value="250"/>

```

```

</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://target/fileidempotent/?
idempotent=true&dempotentRepository=#fileStore&ove=done/${file:name}"/>
    <to uri="mock:result"/>
  </route>
</camelContext>

```

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"/>
  </properties>
</persistence-unit>

```

Then we need to setup a Spring **jpaTemplate** in the spring XML file:

```

<!-- this is standard spring JPA configuration -->
<bean id="jpaTemplate" class="org.springframework.orm.jpa.JpaTemplate">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <!-- we use idempotentDB as the persitence unit name defined in the
persistence.xml file -->
  <property name="persistenceUnitName" value="idempotentDb"/>
</bean>

```

And finally 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 spring jpaTemplate -->

```

```

    <constructor-arg index="0" ref="jpaTemplate"/>
    <!-- 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 then we just need to reference the **jpaStore** bean in the file consumer endpoint, using the **idempotentRepository** option and the # syntax:

```

<route>
  <from uri="file://inbox?
idempotent=true&dempotentRepository=#jpaStore"/>
  <to uri="bean:processInbox"/>
</route>

```

FILTER USING ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER

Apache 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:

```

public class MyFileFilter implements GenericFileFilter {
    public boolean accept(GenericFile pathname) {
        // we dont accept any files starting with skip in the name
        return !pathname.getFileName().startsWith("skip");
    }
}

```

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.MyFileSorter"/>

<route>
  <from uri="file://inbox?filter=#myFilter"/>
  <to uri="bean:processInbox"/>
</route>

```

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

The sample below demonstrates how to use it:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <template id="camelTemplate"/>

  <!-- use myFilter as filter to allow setting ANT paths for which files
to scan for -->
  <endpoint id="myFileEndpoint" uri="file:///target/antpathmatcher?
recursive=true&ilter=#myAntFilter"/>

  <route>
    <from ref="myFileEndpoint"/>
    <to uri="mock:result"/>
  </route>
</camelContext>

<!-- we use the antpath file filter to use ant paths for includes and
exlucde -->
<bean id="myAntFilter"
class="org.apache.camel.component.file.AntPathMatcherGenericFileFilter">
  <!-- include and file in the subfolder that has day in the name -->
  <property name="includes" value="**/subfolder/**/*day"/>
  <!-- exclude all files with bad in name or .xml files. Use comma to
seperate multiple excludes -->
  <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>
```

SORTING USING COMPARATOR

Apache 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 Apache Camel sort the files before being processed.

In the sample we have built our own comparator that just sorts by file name:

```
public class MyFileSorter implements Comparator<GenericFile> {
  public int compare(GenericFile o1, GenericFile o2) {
    return o1.getFileName().compareToIgnoreCase(o2.getFileName());
  }
}
```

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>
```

URI OPTIONS CAN REFERENCE BEANS USING THE # SYNTAX

In the Spring DSL route about notice that we can reference beans in the [Registry](#) by prefixing the id with #. So writing `sorter=#mySorter`, will instruct Apache Camel to go look in the [Registry](#) for a bean with the ID, `mySorter`.

SORTING USING SORTBY

Apache 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 modification 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 the `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
```

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 have 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 `commit()` method we can move the actual file and also delete the *ready* file.



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.

DEBUG LOGGING

This component has log level `TRACE` that can be helpful if you have problems.

See also:

- [File Language](#)
- [FTP2](#)

CHAPTER 37. FLATPACK

FLATPACK COMPONENT

The Flatpack component supports fixed width and delimited file parsing using 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>
```

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&...`

URI OPTIONS

| Name | Default Value | Description |
|--------------------------------|-------------------|---|
| <code>delimiter</code> | <code>,</code> | The default character delimiter for delimited files. |
| <code>textQualifier</code> | <code>"</code> | The text qualifier for delimited files. |
| <code>ignoreFirstRecord</code> | <code>true</code> | Whether the first line is ignored for delimited files (for the column headers). |
| <code>splitRows</code> | <code>true</code> | As of Apache Camel 1.5, the component can either process each row one by one or the entire content at once. |

| | | |
|---------------------------------|--------------------|---|
| <code>allowShortLines</code> | <code>false</code> | *Camel 2.9.3:* Allows for lines to be shorter than expected and ignores the extra characters. |
| <code>ignoreExtraColumns</code> | <code>false</code> | *Camel 2.9.3:* Allows for lines to be longer than expected and ignores the extra characters. |

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.

MESSAGE HEADERS

Apache Camel will store the following headers on the IN message:

| Header | Description |
|-----------------------------------|--|
| <code>camelFlatpackCounter</code> | The current row index. For <code>splitRows=false</code> the counter is the total number of rows. |

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");
```

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>
```

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](#)

CHAPTER 38. FOP

FOP COMPONENT

Available as of Camel 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>
```

URI FORMAT

```
fop://outputFormat?[options]
```

OUTPUT FORMATS

The primary output format is PDF but other output [formats](#) are also supported:

| Name | Output Format | Description |
|------|---|--------------------------|
| PDF | <code>application/pdf</code> | Portable Document Format |
| PS | <code>application/postscript</code> | Adobe Postscript |
| PCL | <code>application/x-pcl</code> | Printer Control Language |
| PNG | <code>image/png</code> | PNG images |
| JPEG | <code>image/jpeg</code> | JPEG images |
| SVG | <code>image/svg+xml</code> | Scalable Vector Graphics |
| XML | <code>application/X-fop-areatree</code> | Area tree representation |
| MIF | <code>application/mif</code> | FrameMaker's MIF |
| RTF | <code>application/rtf</code> | Rich Text Format |
| TXT | <code>text/plain</code> | Text |

The complete list of valid output formats can be found [here](#)

ENDPOINT OPTIONS

| name | default value | description |
|----------------------------|---------------|--|
| <code>userConfigURL</code> | none | 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. |

MESSAGE OPERATIONS

| name | default value | description |
|--|---------------|---|
| <code>CamelFop.Output.Format</code> | | Overrides the output format for that message |
| <code>CamelFop.Encrypt.userPassword</code> | | PDF user password |
| <code>CamelFop.Encrypt.ownerPassword</code> | | PDF owner password |
| <code>CamelFop.Encrypt.allowPrint</code> | true | Allows printing the PDF |
| <code>CamelFop.Encrypt.allowCopyContent</code> | true | Allows copying content of the PDF |
| <code>CamelFop.Encrypt.allowEditContent</code> | true | Allows editing content of the PDF |
| <code>CamelFop.Encrypt.allowEditAnnotations</code> | true | Allows editing annotation of the PDF |
| <code>CamelFop.Render.producer</code> | Apache FOP | Metadata element for the system/software that produces the document |

| | | |
|---|--|---|
| <code>CamelFop.Render.creator</code> | | Metadata element for the user that created the document |
| <code>CamelFop.Render.creationDate</code> | | Creation Date |
| <code>CamelFop.Render.author</code> | | Author of the content of the document |
| <code>CamelFop.Render.title</code> | | Title of the document |
| <code>CamelFop.Render.subject</code> | | Subject of the document |
| <code>CamelFop.Render.keywords</code> | | Set of keywords applicable to this document |

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");
```

CHAPTER 39. FREEMARKER

FREEMARKER

The **freemarker**: component allows you to process 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>
```

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 (for example, `file://folder/myfile.ftl`).

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

OPTIONS

| Option | Default | Description |
|----------------------------------|-------------------|---|
| <code>contentCache</code> | <code>true</code> | Cache for the resource content when it's loaded. Note: as of Camel 2.9 cached resource content can be cleared via JMX using the endpoint's clearContentCache operation. |
| <code>encoding</code> | <code>null</code> | Character encoding of the resource content. |
| <code>templateUpdateDelay</code> | <code>5</code> | *Camel 2.9:* Number of seconds the loaded template resource will remain in the cache. |

FREEMARKER CONTEXT

Apache 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). |

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 Apache Camel will not cache the resource and hot reloading is thus enabled. This scenario can be used in development.

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 | Support Version |
|---|--------------------------------------|--|------------------|
| FreemarkerConstants.FREEMARKER_RESOURCE | org.springframework.core.io.Resource | The template resource | <= 1.6.2, <= 2.1 |
| 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 |

SAMPLES

For example, you can define a route like the following:

```
from("activemq:My.Queue").
  to("freemarker:com/acme/MyResponse.ftl");
```

To use a FreeMarker template to formulate a response to *InOut* message exchanges (where there is a **JMSReplyTo** header).

If you want to process *InOnly* exchanges, you could use a FreeMarker template to transform the message before sending it on to another endpoint:

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut, "freemarker:com/acme/MyResponse.ftl").
  to("activemq:Another.Queue");
```

And to disable the content cache (for example, for development usage where the **.ftl** template should be hot reloaded):

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut, "freemarker:com/acme/MyResponse.ftl?
contentCache=false").
  to("activemq:Another.Queue");
```

And for a file-based resource:

```
from("activemq:My.Queue").
  to(ExchangePattern.InOut, "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/m
y/template.ftl").
  to("freemarker:dummy");
```

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:

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testFreemarkerLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Dear Ibsen, Claus\n\nThanks for the order
of Camel in Action."
    + "\n\nRegards Camel Riders Bookstore\nPS: Next beer is on me,
James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:a")

                .to("freemarker:org/apache/camel/component/freemarker/letter.ftl")
                    .to("mock:result");
        }
    };
}
```

CHAPTER 40. FTP2

FTP/SFTP COMPONENT

This component provides access to remote file systems over the FTP and SFTP protocols.

CONSUMING FROM REMOTE FTP SERVER

Make sure you read the section titled *Default when consuming files* further below for details related to consuming files.

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. Can contain nested folders.

If no **username** is provided, then **anonymous** login is attempted using no password. If no **port** number is provided, Apache Camel will provide default values according to the protocol (ftp = 21, sftp = 22, ftps = 21).

This component uses two different libraries for the actual FTP work. FTP and FTPS use [Apache Commons Net](#) while SFTP uses [JCraft JSCH](#).

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

URI OPTIONS

The options below are exclusive to the FTP component:

| Name | Default Value | Description |
|-----------------|---------------|---|
| username | null | Specifies the username to use to log in to the remote file system. |
| password | null | Specifies the password to use to log in to the remote file system. |
| binary | false | Specifies the file transfer mode, BINARY or ASCII. Default is ASCII (false). |

| | | |
|---|--------------|---|
| disconnect | false | Camel 2.2: Whether or not to disconnect from remote FTP server right after use. Can be used for both consumer and producer. 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. |
| localWorkDirectory | null | 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. See below for more details. |
| passiveMode | false | FTP only: Specifies whether to use passive mode connections. Default is active mode {false} . |
| securityProtocol | TLS | FTPS only: Sets the underlying security protocol. The following values are defined: TLS : Transport Layer Security SSL : Secure Sockets Layer |
| disableSecureDataChannelDefaults | false | Camel 2.4: FTPS only: Whether or not to disable using default values for execPbsz and execProt when using secure data transfer. You can set this option to true if you want to be in absolute full control what the options execPbsz and execProt should be used. |

| | | |
|-----------------------|--------------|---|
| download | true | Camel 2.11: 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. |
| streamDownload | false | Camel 2.11: Whether the consumer should download the entire file up front, the default behavior, or if it should pass an InputStream read from the remote resource rather than an in-memory array as the in body of the camel Exchange . This option is ignored if download is false or localWorkDirectory is provided. This option is useful for working with large remote files. |
| execProt | null | Camel 2.4: FTPS only: Will by default use option P if secure data channel defaults hasn't been disabled. Possible values are: C : Clear S : Safe (SSL protocol only) E : Confidential (SSL protocol only) P : Private |
| execPbsz | null | Camel 2.4: FTPS only: This option specifies the buffer size of the secure data channel. If option useSecureDataChannel has been enabled and this option has not been explicitly set, then value 0 is used. |
| isImplicit | false | FTPS only: Sets the security mode(implicit/explicit). Default is explicit (false). |
| knownHostsFile | null | SFTP only: Sets the known_hosts file, so that the SFTP endpoint can do host key verification. |

| | | |
|---------------------------------------|-------------------|---|
| <code>knownHostsUri</code> | <code>null</code> | SFTP only:Camel 2.11.1: Sets the <code>known_hosts</code> file (loaded from classpath by default), so that the SFTP endpoint can do host key verification. |
| <code>keyPair</code> | <code>null</code> | SFTP only:Camel 2.12.0: Sets the Java KeyPair for SSH public key authentication, it supports DSA or RSA keys. |
| <code>privateKeyFile</code> | <code>null</code> | SFTP only: Set the private key file to that the SFTP endpoint can do private key verification. |
| <code>privateKeyUri</code> | <code>null</code> | SFTP only:Camel 2.11.1: Set the private key file (loaded from classpath by default) to that the SFTP endpoint can do private key verification. |
| <code>privateKey</code> | <code>null</code> | SFTP only:Camel 2.11.1: Set the private key as byte[] to that the SFTP endpoint can do private key verification. |
| <code>privateKeyFilePassphrase</code> | <code>null</code> | SFTP only: Set the private key file passphrase to that the SFTP endpoint can do private key verification. |
| <code>privateKeyPassphrase</code> | <code>null</code> | SFTP only:Camel 2.11.1: Set the private key file passphrase to that the SFTP endpoint can do private key verification. |
| <code>preferredAuthentications</code> | <code>null</code> | SFTP only:Camel 2.10.7, 2.11.2,2.12.0: 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. |

| | | |
|---------------------------------|--------------|--|
| ciphers | null | Camel 2.8.2, 2.9: SFTP only 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. |
| fastExistsCheck | false | Camel 2.8.2, 2.9: 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. Note from Camel 2.10.1 onwards 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. |
| strictHostKeyChecking | no | SFTP only:Camel 2.2: Sets whether to use strict host key checking. Possible values are: no , yes and ask . ask does not make sense to use as Camel cannot answer the question for you as its meant for human intervention. Note: The default in Camel 2.1 and below was ask . |
| maximumReconnectAttempts | 3 | Specifies the maximum reconnect attempts Apache Camel performs when it tries to connect to the remote FTP server. Use 0 to disable this behavior. |

| | | |
|---------------------------------------|-------|--|
| reconnectDelay | 1000 | Delay in millis Apache Camel will wait before performing a reconnect attempt. |
| connectTimeout | 10000 | Camel 2.4: Is the connect timeout in millis. This corresponds to using ftpClient.connectTimeout for the FTP/FTPS. For SFTP this option is also used when attempting to connect. |
| soTimeout | null | FTP and FTPS Only:Camel 2.4: Is the SocketOptions.SO_TIMEOUT value in millis. Note SFTP will automatic use the connectTimeout as the soTimeout . |
| timeout | 30000 | FTP and FTPS Only:Camel 2.4: Is the data timeout in millis. This corresponds to using ftpClient.dataTimeout for the FTP/FTPS. For SFTP there is no data timeout. |
| throwExceptionOnConnect Failed | false | Camel 2.5: Whether or not to thrown an exception if a successful connection and login could not be establish. This allows a custom pollStrategy to deal with the exception, for example to stop the consumer or the likes. |
| siteCommand | null | FTP and FTPS Only:Camel 2.5: To execute site commands after successful login. Multiple site commands can be separated using a new line character (\n). Use help site to see which site commands your FTP server supports. |

| | | |
|----------------------------|-------------|---|
| stepwise | true | When consuming directories, specifies whether or not to use stepwise mode for traversing the directory tree. Stepwise means that it will CD one directory at a time. For more details, see the section called "Stepwise changing directories" . |
| separator | Auto | Camel 2.6: Dictates what path separator char to use when uploading files. Auto means use the path provided without altering it. UNIX means use UNIX style path separators. Windows means use Windows style path separators. |
| chmod | null | *SFTP Producer Only:* Camel 2.9: Allows you to set chmod on the stored file. For example chmod=640 . |
| compression | 0 | *SFTP Only:* Camel 2.8.3/2.9: 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. |
| ftpClient | null | FTP and FTPS Only:Camel 2.1: Allows you to use a custom org.apache.commons.net.ftp.FTPClient instance. |
| ftpClientConfig | null | FTP and FTPS Only:Camel 2.1: Allows you to use a custom org.apache.commons.net.ftp.FTPClientConfig instance. |
| serverAliveInterval | 0 | SFTP Only:Camel 2.8 Allows you to set the serverAliveInterval of the sftp session |

| | | |
|---|----------------------|--|
| <code>serverAliveCountMax</code> | <code>1</code> | SFTP Only: Camel 2.8 Allows you to set the <code>serverAliveCountMax</code> of the <code>sftp</code> session |
| <code>ftpClient.trustStore.file</code> | <code>null</code> | FTPS Only: Sets the trust store file, so that the FTPS client can look up for trusted certificates. |
| <code>ftpClient.trustStore.type</code> | <code>JKS</code> | FTPS Only: Sets the trust store type. |
| <code>ftpClient.trustStore.algorithm</code> | <code>SunX509</code> | FTPS Only: Sets the trust store algorithm. |
| <code>ftpClient.trustStore.password</code> | <code>null</code> | FTPS Only: Sets the trust store password. |
| <code>ftpClient.keyStore.file</code> | <code>null</code> | FTPS Only: Sets the key store file, so that the FTPS client can look up for the private certificate. |
| <code>ftpClient.keyStore.type</code> | <code>JKS</code> | FTPS Only: Sets the key store type. |
| <code>ftpClient.keyStore.algorithm</code> | <code>SunX509</code> | FTPS Only: Sets the key store algorithm. |
| <code>ftpClient.keyStore.password</code> | <code>null</code> | FTPS Only: Sets the key store password. |
| <code>ftpClient.keyStore.keyPassword</code> | <code>null</code> | FTPS Only: Sets the private key password. |
| <code>sslContextParameters</code> | <code>null</code> | FTPS Only: Camel 2.9: Reference to a <code>org.apache.camel.util.jse.SSLContextParameters</code> in the Registry . This reference overrides any configured SSL related options on <code>ftpClient</code> as well as the <code>securityProtocol</code> (SSL, TLS, etc.) set on <code>FtpsConfiguration</code> . See Using the JSSE Configuration Utility . |

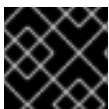
| | | |
|--|--------------------|--|
| <code>proxy</code> | <code>null</code> | SFTP Only:Camel 2.10.7, 2.11.1: Reference to a <code>com.jcraft.jsch.Proxy</code> in the Registry . This proxy is used to consume/send messages from the target SFTP host. |
| <code>useList</code> | <code>true</code> | FTP/FTPS Only:Camel 2.12.1: Whether the consumer should use FTP LIST command to retrieve directory listing to see which files exists. If this option is set to false , then stepwise=false must be configured, and also fileName must be configured to a fixed name, so the consumer knows the name of the file to retrieve. When doing this only that single file can be retrieved. See further below for more details. |
| <code>ignoreFileNotFoundOrPermissionError</code> | <code>false</code> | Camel 2.12.1: Whether the consumer should ignore when a file was attempted to be retrieved but did not exist (for some reason), or failure due insufficient file permission error. |



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.



MORE OPTIONS

See [File](#) for more options, as all the options from `File` are inherited by `FTP2`.

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.serverLanguage
Code=fr")
    .to("bean:foo");
```

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.

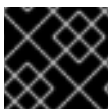
For example:

```
<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");
```

MORE URI OPTIONS



IMPORTANT

See [File2](#) as all the options there also applies for this component.

EXAMPLES

Here are some examples of FTP endpoint URIs:

```
ftp://someone@someftpserver.com/public/upload/images/holiday2008?
password=secret&binary=true
ftp://someoneelse@someotherftpserver.co.uk:12049/reports/2008/password=sec
ret&binary=false
ftp://publicftpserver.com/download
```



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.

MORE INFORMATION

This component is an extension of the [File2](#) component. So there are more samples and details on the [File2](#) component page.

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 explicit 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.

LIMITATIONS

The option `readLock` can be used to force Apache Camel **not** to consume files that are currently in the process of being written. However, this option is turned off by default, as it requires that the user has write access. 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.

The ftp producer does **not** support appending to existing files. Any existing files on the remote server will be deleted before the file is written.

MESSAGE HEADERS

The following message headers can be used to affect the behavior of the component

| Header | Description |
|----------------------------|---|
| <code>CamelFileName</code> | 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 absolute filepath (path + name) for the output file that was written. This header is set by Apache Camel and its purpose is providing end-users the name of the file that was written. |
| CamelFileBatchIndex | Current index out of total number of files being consumed in this batch. |
| CamelFileBatchSize | 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 |

ABOUT TIMEOUTS

The two sets of libraries (see above) have different APIs for setting the timeout. You can use the **connectTimeout** option for both of them to set a timeout in milliseconds 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 milliseconds.

USING LOCAL WORK DIRECTORY

Apache 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**.

Apache 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");
```

OPTIMIZATION BY RENAMING WORK FILE

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 Apache 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.

STEPWISE CHANGING DIRECTORIES

Camel `FTP` can operate in two modes in terms of traversing directories when consuming files (for example, downloading) or producing files (for example, 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.

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.

USING STEPWISE=TRUE (DEFAULT MODE)

The following log shows the conversation between the FTP endpoint and the remote FTP server when the FTP endpoint is operating in *stepwise* 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`.

USING `STEPWISE=FALSE`

The following log shows the conversation between the FTP endpoint and the remote FTP server when the FTP endpoint is operating in *non-stepwise* mode:

```
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.

SAMPLES

In the sample below we set up Apache 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.

```

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // we use a delay of 60 minutes (eg. once pr. hour we poll the
FTP server
            long delay = 60 * 60 * 1000L;

            // from the given FTP server we poll (= download) all the
files
            // from the public/reports folder as BINARY types and store
this as files
            // in a local directory. Apache Camel will use the filenames
from the FTPServer

            // notice that the FTPConsumer properties must be prefixed
with "consumer." in the URL
            // the delay parameter is from the FileConsumer component so
we should use consumer.delay as
            // the URI parameter name. The FTP Component is an extension
of the File Component.
            from("ftp://tiger:scott@localhost/public/reports?
binary=true&consumer.delay=" + delay).
                to("file://target/test-reports");
        }
    };
}

```

And the route using Spring DSL:

```

<route>
  <from uri="ftp://scott@localhost/public/reports?
password=tiger&inary=true&elay=60000"/>
  <to uri="file://target/test-reports"/>
</route>

```

CONSUMING A REMOTE FTP SERVER TRIGGERED BY A ROUTE

The FTP consumer is built as a scheduled consumer to be used in the **from** route. However, if you want to start consuming from an FTP server triggered within a route, use a route like the following:

```
from("seda:start")
    // define the file name so that only a single file is polled and
    // deleted once retrieved
    .pollEnrich("ftp://admin@localhost:21/getme?
password=admin&binary=false&fileName=myFile.txt&delete=true")
    .to("mock:result");
```

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=passw
ord")
    .to("bean:foo");
```

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&f
tpClient.trustStore.password=password")
    .to("bean:foo");
```

FILTER USING `ORG.APACHE.CAMEL.COMPONENT.FILE.GENERICFILEFILTER`

Apache Camel supports pluggable filtering strategies. You define a filter strategy by implementing the `org.apache.camel.component.file.GenericFileFilter` interface in Java. You can then configure the endpoint with the filter to skip certain files.

In the following sample we define a filter that only accepts files whose filename starts with **report**.

```
public class MyFileFilter<T> implements GenericFileFilter<T> {

    public boolean accept(GenericFile<T> file) {
        // we only want report files
        return file.getFileName().startsWith("report");
    }
}
```

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@someftpsrv.com?
password=secret&filter=#myFilter"/>
  <to uri="bean:processInbox"/>
</route>
```

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:

```
<bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigu
rer"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">

  <template id="camelTemplate"/>

  <!-- use myFilter as filter to allow setting ANT paths for which files
to scan for -->
  <endpoint id="myFTPEndpoint"
uri="ftp://admin@localhost:${SpringFileAntPathMatcherRemoteFileFilterTest.
ftpPort}/antpath?
password=admin&recursive=true&delay=10000&initialDelay=2000&filter=#myAntFilde
r"/>

  <route>
    <from ref="myFTPEndpoint"/>
    <to uri="mock:result"/>
  </route>
</camelContext>

<!-- we use the AntPathMatcherRemoteFileFilter to use ant paths for
includes and exclude -->
<bean id="myAntFilter"
class="org.apache.camel.component.file.AntPathMatcherGenericFileFilter">
  <!-- include any files in the sub-folder that have day in the name -->
  <property name="includes" value="**/subfolder/**/*day*"/>
```

```

    <!-- exclude all files with bad in name or .xml files. Use comma to
seperate multiple excludes -->
    <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>

```

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.

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&ignoreFileNotFoundOrPermission
Error=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&ignoreFileNotFoundOrPermission

```



```
Error=true&fileName=report.txt&delete=true", String.class);
```

DEBUG LOGGING

This component has log level **TRACE** that can be helpful if you have problems.

CHAPTER 41. GAE

41.1. INTRODUCTION TO THE GAE COMPONENTS

Apache Camel Components for Google App Engine

TUTORIALS

- A good starting point for using Apache Camel on GAE is the [Tutorial for Camel on Google App Engine](#)
- The [OAuth tutorial](#) demonstrates how to implement [OAuth](#) in web applications.

The Apache Camel components for [Google App Engine](#) (GAE) are part of the `camel-gae` project and provide connectivity to GAE's [cloud computing services](#). They make the GAE cloud computing environment accessible to applications via Apache Camel interfaces. Following this pattern for other cloud computing environments could make it easier to port Apache Camel applications from one cloud computing provider to another. The following table lists the cloud computing services provided by Google App Engine and the supporting Apache Camel components. The documentation of each component can be found by following the link in the *Camel Component* column.

| GAE service | Camel component | Component description |
|---------------------------------------|-----------------------|--|
| URL fetch service | ghttp | Provides connectivity to the GAE URL fetch service but can also be used to receive messages from servlets. |
| Task queueing service | gtask | Supports asynchronous message processing on GAE by using the task queueing service as message queue. |
| Mail service | gmail | Supports sending of emails via the GAE mail service. Receiving mails is not supported yet but will be added later. |
| Memcache service | | Not supported yet. |
| XMPP service | | Not supported yet. |
| Images service | | Not supported yet. |
| Datastore service | | Not supported yet. |

| | | |
|------------------|--------------|--|
| Accounts service | gauth glogin | <p>These components interact with the Google Accounts API for authentication and authorization. Google Accounts is not specific to Google App Engine but is often used by GAE applications for implementing security. The gauth component is used by web applications to implement a Google-specific OAuth consumer. This component can also be used to OAuth-enable non-GAE web applications. The glogin component is used by Java clients (outside GAE) for programmatic login to GAE applications. For instructions how to protect GAE applications against unauthorized access refer to the Security for page.</p> |
|------------------|--------------|--|

Camel context

Setting up a `SpringCamelContext` on Google App Engine differs between Camel 2.1 and higher versions. The problem is that usage of the Camel-specific Spring configuration XML schema from the <http://camel.apache.org/schema/spring> namespace requires JAXB and Camel 2.1 depends on a Google App Engine SDK version that doesn't support JAXB yet. This limitation has been removed since Camel 2.2.

JMX must be disabled in any case because the `javax.management` package isn't on the App Engine JRE whitelist.

Apache Camel 2.1

`camel-gae` 2.1 comes with the following `CamelContext` implementations.

- `org.apache.camel.component.gae.context.GaeDefaultCamelContext` (extends `org.apache.camel.impl.DefaultCamelContext`)
- `org.apache.camel.component.gae.context.GaeSpringCamelContext` (extends `org.apache.camel.spring.SpringCamelContext`)

Both disable JMX before startup. The `GaeSpringCamelContext` additionally provides setter methods adding route builders as shown in the next example.



APPCTX.XML

```
<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-
    2.5.xsd">

  <bean id="camelContext"

class="org.apache.camel.component.gae.context.GaeSpringCamelCon
text">
  <property name="routeBuilder" ref="myRouteBuilder" />
</bean>

  <bean id="myRouteBuilder"
    class="org.example.MyRouteBuilder">
</bean>

</beans>
```

Alternatively, use the `routeBuilders` property of the `GaeSpringCamelContext` for setting a list of route builders. Using this approach, a `SpringCamelContext` can be configured on GAE without the need for JAXB.

Apache Camel 2.2

With Camel 2.2 or higher, applications can use the <http://camel.apache.org/schema/spring> namespace for configuring a `SpringCamelContext` but still need to disable JMX. Here's an example.



APPCTX.XML

```
<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">

  <camel:camelContext id="camelContext">
    <camel:jmxAgent id="agent" disabled="true" />
    <camel:routeBuilder ref="myRouteBuilder"/>
  </camel:camelContext>

  <bean id="myRouteBuilder"
    class="org.example.MyRouteBuilder">
</bean>

</beans>
```

The web.xml

Running Apache Camel on GAE requires usage of the **CamelHttpTransportServlet** from **camel-servlet**. The following example shows how to configure this servlet together with a Spring application context XML file.

WEB.XML

```
<web-app
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
version="2.5">

    <servlet>
        <servlet-name>CamelServlet</servlet-name>
        <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServ
let</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>appctx.xml</param-value>
        </init-param>
    </servlet>

    <!--
        Mapping used for external requests
    -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

    <!--
        Mapping used for web hooks accessed by task queueing
        service.
    -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/worker/*</url-pattern>
    </servlet-mapping>

</web-app>
```

The location of the Spring application context XML file is given by the **contextConfigLocation** init parameter. The **appctx.xml** file must be on the classpath. The servlet mapping makes the Apache Camel application accessible under **http://<appname>..appspot.com/camel/...** when deployed to Google App Engine where **<appname>** must be replaced by a real GAE application name. The second servlet mapping is used internally by the task queueing service for background processing via [web hooks](#). This mapping is relevant for the [gtask](#) component and is explained there in more detail.

41.2. GAUTH

gauth Component

Available in Apache Camel 2.3

The **gauth** component is used by web applications to implement a [Google-specific OAuth](#) consumer. It will be later extended to support other [OAuth](#) providers as well. Although this component belongs to the [Camel Components for Google App Engine](#) (GAE), it can also be used to OAuth-enable non-GAE web applications. For a detailed description of Google's OAuth implementation refer to the Google [OAuth API reference](#).

URI format

```
gauth://name[?options]
```

The endpoint **name** can be either **authorize** or **upgrade**. An **authorize** endpoint is used to obtain an unauthorized request token from Google and to redirect the user to the authorization page. The **upgrade** endpoint is used to process OAuth callbacks from Google and to upgrade an authorized request token to a long-lived access token. Refer to the [usage section](#) for an example.

Options

| Name | Default Value | Required | Description |
|-----------------|---------------|--|---|
| callback | null | true (can alternatively be set via GAuthAuthorizeBinding.GAUTH_CALLBACK message header) | URL where to redirect the user after having granted or denied access. |
| scope | null | true (can alternatively be set via GAuthAuthorizeBinding.GAUTH_SCOPE message header) | URL identifying the service(s) to be accessed. Scopes are defined by each Google service; see the service's documentation for the correct value. To specify more than one scope, list each one separated with a comma. Example: http://www.google.com/calendar/feeds/ . |

| | | | |
|-----------------------|-------------|---|--|
| consumerKey | null | true (can alternatively be set on component-level). | Domain identifying the web application. This is the domain used when registering the application with Google. Example: camelcloud.appspot.com . For a non-registered application use anonymous . |
| consumerSecret | null | one of consumerSecret or keyLoaderRef is required (can alternatively be set on component-level). | Consumer secret of the web application. The consumer secret is generated when registering the application with Google. It is needed if the HMAC-SHA1 signature method shall be used. For a non-registered application use anonymous . |
| keyLoaderRef | null | one of consumerSecret or keyLoaderRef is required (can be alternatively set on component-level) | Reference to a private key loader in the registry. Part of camel-gae are two key loaders: GAuthPk8Loader for loading a private key from a PKCS#8 file and GAuthJksLoader to load a private key from a Java key store. It is needed if the RSA-SHA1 signature method shall be used. These classes are defined in the org.apache.camel.component.gae.auth package. |

| | | | |
|----------------------------|---|-------|---|
| authorizeBindingRef | Reference to GAuthAuthorizeBinding | false | Reference to a OutboundBinding<GAuthEndpoint, GoogleOAuthParameters, GoogleOAuthParameters> in the registry for customizing how an Exchange is bound to GoogleOAuthParameters . This binding is used for the authorization phase. Most applications won't change the default value. |
| upgradeBindingRef | Reference to GAuthAuthorizeBinding | false | Reference to a OutboundBinding<GAuthEndpoint, GoogleOAuthParameters, GoogleOAuthParameters> in the registry. for customizing how an Exchange is bound to GoogleOAuthParameters . This binding is used for the token upgrade phase. Most applications won't change the default value. |

Message headers

| Name | Type | Endpoint | Message | Description |
|---|--------|------------------------|---------|---------------------------------------|
| GAuthAuthorizeBinding.GAUTH_CALLBACK | String | gauth:authorize | in | Overrides the callback option. |
| GAuthAuthorizeBinding.GAUTH_SCOPE | String | gauth:authorize | in | Overrides the scope option. |

| | | | | |
|--|---------------|----------------------|-----|--|
| GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN | String | gauth:upgrade | out | Contains the long-lived access token. This token should be stored by the applications in context of a user. |
| GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_SECRET | String | gauth:upgrade | out | Contains the access token secret. This token secret should be stored by the applications in context of a user. |

Message body

The **gauth** component doesn't read or write message bodies.

Component configuration

Some endpoint options such as **consumerKey**, **consumerSecret** or **keyLoader** are usually set to the same values on **gauth:authorize** and **gauth:upgrade** endpoints. The **gauth** component allows to configure them on component-level. These settings are then inherited by **gauth** endpoints and need not be set redundantly in the endpoint URIs. Here are some configuration examples.



COMPONENT CONFIGURATION FOR A REGISTERED WEB APPLICATION USING THE HMAC-SHA1 SIGNATURE METHOD

```
<bean id="gauth"
class="org.apache.camel.component.gae.auth.GAuthComponent">
  <property name="consumerKey" value="example.appspot.com" />
  <property name="consumerSecret" value="QAtA...HfQ" />
</bean>
```



COMPONENT CONFIGURATION FOR AN UNREGISTERED WEB APPLICATION USING THE HMAC-SHA1 SIGNATURE METHOD

```
<bean id="gauth"
class="org.apache.camel.component.gae.auth.GAuthComponent">
  <!-- Google will display a warning message on the
authorization page -->
  <property name="consumerKey" value="anonymous" />
  <property name="consumerSecret" value="anonymous" />
</bean>
```



COMPONENT CONFIGURATION FOR A REGISTERED WEB APPLICATION USING THE RSA-SHA1 SIGNATURE METHOD

```
<bean id="gauth"
class="org.apache.camel.component.gae.auth.GAuthComponent">
  <property name="consumerKey" value="ipfcloud.appspot.com"
/>
  <property name="keyLoader" ref="jksLoader" />
  <!--<property name="keyLoader" ref="pk8Loader" />-->
</bean>

<!-- Loads the private key from a Java key store -->
<bean id="jksLoader"
class="org.apache.camel.component.gae.auth.GAuthJksLoader">
  <property name="keyStoreLocation" value="myKeytore.jks" />
  <property name="keyAlias" value="myKey" />
  <property name="keyPass" value="myKeyPassword" />
  <property name="storePass" value="myStorePassword" />
</bean>

<!-- Loads the private key from a PKCS#8 file -->
<bean id="pk8Loader"
class="org.apache.camel.component.gae.auth.GAuthPk8Loader">
  <property name="keyStoreLocation" value="myKeyfile.pk8" />
</bean>
```

Usage

Here's the minimum setup for adding OAuth to a (non-GAE) web application. In the following example, it is assumed that the web application is running on `gauth.example.org`.

GAUTHROUTEBUILDER.JAVA

```

import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class GAuthRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        // Calback URL to redirect user from Google
        Authorization back to the web application
        String encodedCallback =
        URLEncoder.encode("https://gauth.example.org:8443/handler",
        "UTF-8");
        // Application will request for authorization to access
        a user's Google Calendar
        String encodedScope =
        URLEncoder.encode("http://www.google.com/calendar/feeds/",
        "UTF-8");

        // Route 1: A GET request to
        http://gauth.example.org/authorize will trigger the the OAuth
        // sequence of interactions. The gauth:authorize
        endpoint obtains an unauthorized request
        // token from Google and then redirects the user
        (browser) to a Google authorization page.
        from("jetty:http://0.0.0.0:8080/authorize")
            .to("gauth:authorize?callback=" + encodedCallback +
            "&scope=" + encodedScope);

        // Route 2: Handle callback from Google. After the user
        granted access to Google Calendar
        // Google redirects the user to
        https://gauth.example.org:8443/handler (see callback) along
        // with an authorized request token. The gauth:access
        endpoint exchanges the authorized
        // request token against a long-lived access token.
        from("jetty:https://0.0.0.0:8443/handler")
            .to("gauth:upgrade")
            // The access token can be obtained from
            //
        exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TO
        KEN)
            // The access token secret can be obtained from
            //
        exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TO
        KEN_SECRET)
            .process(/* store the tokens in context of the
            current user ... */);
    }
}

```

The OAuth sequence is triggered by sending a GET request to <http://gauth.example.org/authorize> . The user is then redirected to a Google authorization page. After having granted access on this page, Google redirects the user to the web application which handles the callback and finally obtains a long-lived access token from Google.

These two routes can perfectly co-exist with any other web application framework. The framework provides the basis for web application-specific functionality whereas the OAuth service provider integration is done with Apache Camel. The OAuth integration part could even use resources from an existing servlet container by using the `servlet` component instead of the `jetty` component.

WHAT TO DO WITH THE OAUTH ACCESS TOKEN?

- Application should store the access token in context of the current user. If the user logs in next time, the access token can directly be loaded from the database, for example, without doing the *OAuth dance* again.
- The access token is then used to get access to Google services, such as a Google Calendar API, on behalf of the user. Java applications will most likely use the [GData Java library](#) for that. See below for an [example](#) how to use the access token with the GData Java library to read a user's calendar feed.
- The user can revoke the access token at any time from his [Google Accounts](#) page. In this case, access to the corresponding Google service will throw an authorization exception. The web application should remove the stored access token and redirect the user again to the Google authorization page for creating another one.

The above example relies on the following component configuration.

```
<bean id="gauth"
class="org.apache.camel.component.gae.auth.GAuthComponent">
  <property name="consumerKey" value="anonymous" />
  <property name="consumerSecret" value="anonymous" />
</bean>
```

If you don't want that Google displays a warning message on the authorization page, you'll need to [register](#) your web application and change the `consumerKey` and `consumerSecret` settings.

GAE example

To OAuth-enable a Google App Engine application, only some small changes in the route builder are required. Assuming the GAE application hostname is `camelcloud.appspot.com` a configuration might look as follows. Here, the `ghttp` component is used to handle HTTP(S) requests instead of the `jetty` component.

GAUTHROUTEBUILDER

```
import java.net.URLEncoder;
import org.apache.camel.builder.RouteBuilder;

public class TutorialRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        String encodedCallback =
URLEncoder.encode("https://camelcloud.appspot.com/handler",
"UTF-8");
        String encodedScope =
URLEncoder.encode("http://www.google.com/calendar/feeds/",
"UTF-8");

        from("ghttp:///authorize")
            .to("gauth:authorize?callback=" + encodedCallback +
"&scope=" + encodedScope);

        from("ghttp:///handler")
            .to("gauth:upgrade")
            .process(/* store the tokens in context of the
current user ... */);
    }
}
```

Access token usage

Here's an example how to use an access token to access a user's Google Calendar data with the [GData Java library](#). The example application writes the titles of the user's public and private calendars to **stdout**.

ACCESS TOKEN USAGE

```

import com.google.gdata.client.authn.oauth.OAuthHmacSha1Signer;
import com.google.gdata.client.authn.oauth.OAuthParameters;
import com.google.gdata.client.calendar.CalendarService;
import com.google.gdata.data.calendar.CalendarEntry;
import com.google.gdata.data.calendar.CalendarFeed;

import java.net.URL;

public class AccessExample {

    public static void main(String... args) throws Exception {
        String accessToken = ...
        String accessTokenSecret = ...

        CalendarService myService = new
CalendarService("exampleCo-exampleApp-1.0");
        OAuthParameters params = new OAuthParameters();
        params.setOAuthConsumerKey("anonymous");
        params.setOAuthConsumerSecret("anonymous");
        params.setOAuthToken(accessToken);
        params.setOAuthTokenSecret(accessTokenSecret);
        myService.setOAuthCredentials(params, new
OAuthHmacSha1Signer());

        URL feedUrl = new
URL("http://www.google.com/calendar/feeds/default/");
        CalendarFeed resultFeed = myService.getFeed(feedUrl,
CalendarFeed.class);

        System.out.println("Your calendars:");
        System.out.println();

        for (int i = 0; i < resultFeed.getEntries().size();
i++) {
            CalendarEntry entry =
resultFeed.getEntries().get(i);

            System.out.println(entry.getTitle().getPlainText());
        }
    }
}

```

41.3. GHTTP

ghttp Component

The **ghttp** component contributes to the [Camel Components for Google App Engine \(GAE\)](#). It provides connectivity to the GAE [URL fetch service](#) but can also be used to receive messages from servlets (the only way to receive HTTP requests on GAE). This is achieved by extending the [Servlet component](#). As a consequence, **ghttp** URI formats and options sets differ on the consumer-side (**from**) and producer-side (**to**).

URI format

| Format | Context | Comment |
|---|----------|--|
| <code>ghttp:///path[?options]</code> | Consumer | See also Servlet component |
| <code>ghttp://hostname[:port][/path][?options]</code> <code>ghttps://hostname[:port][/path][?options]</code> | Producer | See also Http component |

Options

| Name | Default Value | Context | Description |
|--------------------------------------|-------------------|----------|--|
| <code>bridgeEndpoint</code> | <code>true</code> | Producer | If set to <code>true</code> the <code>Exchange.HTTP_URI</code> header will be ignored. To override the default endpoint URI with the <code>Exchange.HTTP_URI</code> header set this option to <code>false</code> . |
| <code>throwExceptionOnFailure</code> | <code>true</code> | Producer | Throw a <code>org.apache.camel.component.gae.http</code> if the response code is ≥ 400 . To disable throwing an exception set this option to <code>false</code> . |

| | | | |
|---------------------------|----------------------------------|----------|---|
| inboundBindingRef | reference to GHttpBinding | Consumer | Reference to an InboundBinding<GHttpEndpoint, HttpServletRequest, HttpServletResponse> in the Registry for customizing the binding of an Exchange to the Servlet API. The referenced binding is used as post-processor to org.apache.camel.component.http.GHttpBinding . |
| outboundBindingRef | reference to GHttpBinding | Producer | Reference to an OutboundBinding<GHttpEndpoint, HTTPRequest, HTTPResponse> in the Registry for customizing the binding of an Exchange to the URLFetchService . |

On the consumer-side, all options of the [Servlet component](#) are supported.

Message headers

On the producer side, the following headers of the [Http component](#) are supported.

| Name | Type | Description |
|----------------------------------|---------------|--|
| 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_METHOD | String | The HTTP method to execute. One of GET , POST , PUT and DELETE . If not set, POST will be used if the message body is not null , GET otherwise. |

| | | |
|-------------------------------------|---------------------|--|
| <code>Exchange.HTTP_QUERY</code> | <code>String</code> | Overrides the query part of the endpoint URI or the query part of <code>Exchange.HTTP_URI</code> (if defined). The query string must be in decoded form. |
| <code>Exchange.HTTP_URI</code> | <code>String</code> | Overrides the default endpoint URI if the <code>bridgeEndpoint</code> option is set to <code>false</code> . The URI string must be in decoded form. |
| <code>Exchange.RESPONSE_CODE</code> | <code>int</code> | The HTTP response code from URL fetch service responses. |

On the consumer-side all headers of the [Servlet component](#) component are supported.

Message body

On the producer side the `in` message body is converted to `abyte[]`. The `out` message body is made available as `InputStream`. If the response size exceeds 1 megabyte a [ResponseTooLargeException](#) is thrown by the URL fetch service (see [quotas and limits](#)).

Receiving messages

For receiving messages via the `ghttp` component, a `CamelHttpTransportServlet` must be configured and mapped in the application's `web.xml` (see [the section called "The web.xml"](#)). For example, to handle requests targeted at `http://<appname>.appspot.com/camel/*` or `http://localhost/camel/*` (when using a local development server) the following servlet mapping must be defined:



WEB.XML

```

...
<servlet>
  <servlet-name>CamelServlet</servlet-name>
  <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServ
let</servlet-class>
  ...
</servlet>
...
<servlet-mapping>
  <servlet-name>CamelServlet</servlet-name>
  <url-pattern>/camel/*</url-pattern>
</servlet-mapping>
...

```

Endpoint URI path definitions are relative to this servlet mapping e.g. the route

```
from("ghttp:///greeting").transform().constant("Hello")
```

processes requests targeted at <http://<appname>.appspot.com/camel/greeting>. In this example, the request body is ignored and the response body is set to **Hello**. Requests targeted at http://<appname>.appspot.com/camel/greeting/* are not processed by default. This requires setting the option `matchOnUriPrefix` to `true`.

```
from("ghttp:///greeting?
matchOnUriPrefix=true").transform().constant("Hello")
```

Sending messages

For sending requests to external HTTP services the `ghttp` component uses the [URL fetch service](#). For example, the Apache Camel homepage can be retrieved with the following endpoint definition on the producer-side.

```
from(...)
...
.to("ghttp://camel.apache.org")
...
```

The HTTP method used depends on the `Exchange.HTTP_METHOD` message header or on the presence of an in-message body (`GET` if `null`, `POST` otherwise). Retrieving the Camel homepage via a GAE application is as simple as

```
from("ghttp:///home")
.to("ghttp://camel.apache.org")
```

Sending a `GET` request to <http://<appname>.appspot.com/camel/home> returns the Camel homepage. HTTPS-based communication with external services can be enabled with the `ghttps` scheme.

```
from(...)
...
```

```
.to("ghttps://svn.apache.org/repos/asf/camel/trunk/")
...
```

Dependencies

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



POM.XML

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

where `${camel-version}` must be replaced by the actual version of Apache Camel (2.1.0 or higher).

41.4. GLOGIN

glogin Component

Available in Apache Camel 2.3 (or latest [development snapshot](#)).

The **glogin** component is used by Apache Camel applications outside Google App Engine (GAE) for programmatic login to GAE applications. It is part of the [Chapter 41, GAE. Security-enabled GAE applications](#) normally redirect the user to a login page. After submitting username and password for authentication, the user is redirected back to the application. That works fine for applications where the client is a browser. For all other applications, the login process must be done programmatically. All the [necessary steps](#) for programmatic login are implemented by the **glogin** component. These are

1. Get an authentication token from [Google Accounts](#) via the [ClientLogin API](#).
2. Get an authorization cookie from Google App Engine's login API.

The authorization cookie must then be send with subsequent HTTP requests to the GAE application. It expires after 24 hours and must then be renewed.

URI format

```
glogin://hostname[:port][?options]
```

The **hostname** is either the internet hostname of a GAE application (e.g. `camelcloud.appspot.com`) or the name of the host where the [development server](#) is running (e.g. `localhost`). The **port** is only used when connecting to a development server (i.e. when `devMode=true`, see [options](#)) and defaults to **8080**.

Options

| Name | Default Value | Required | Description |
|------|---------------|----------|-------------|
|------|---------------|----------|-------------|

| | | | |
|-------------------|-------------------------|--|--|
| clientName | apache-camel-2.x | false | A client name with recommended (but not required) format <organization>\-<appname>\-<version> . |
| userName | null | true (can alternatively be set via GLoginBinding.GLOGIN_USER_NAME message header) | Login username (an email address). |
| password | null | true (can alternatively be set via GLoginBinding.GLOGIN_PASSWORD message header) | Login password. |
| devMode | false | false | If set to true a login to a development server is attempted. |
| devAdmin | false | false | If set to true a login to a development server in admin role is attempted. |

Message headers

| Name | Type | Message | Description |
|---------------------------------------|---------------|---------|---|
| GLoginBinding.GLOGIN_HOST_NAME | String | in | Overrides the hostname defined in the endpoint URI. |
| GLoginBinding.GLOGIN_USER_NAME | String | in | Overrides the userName option. |
| GLoginBinding.GLOGIN_PASSWORD | String | in | Overrides the password option. |

| | | | |
|------------------------------------|---------------|-----|---|
| GLoginBinding.GLOGIN_TOKEN | String | out | Contains the authentication token obtained from Google Accounts . Login to a development server does not set this header. |
| GLoginBinding.GLOGIN_COOKIE | String | out | Contains the application-specific authorization cookie obtained from Google App Engine (or a development server). |

Message body

The `glogin` component doesn't read or write message bodies.

Usage

The following JUnit test show an example how to login to a development server as well as to a deployed GAE application located at <http://camelcloud.appspot.com>.

GLOGINTEST.JAVA

```

import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.ProducerTemplate;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.junit4.SpringJUnit4ClassRunner
;

import static
org.apache.camel.component.gae.login.GLoginBinding.*;
import static org.junit.Assert.*;

public class GLoginTest {

    private ProducerTemplate template = ...

    @Test
    public void testDevLogin() {
        Exchange result =
template.request("glogin://localhost:8888?
userName=test@example.org&devMode=true", null);

assertNotNull(result.getOut().getHeader(GLOGIN_COOKIE));
    }

    @Test
    public void testRemoteLogin() {
        Exchange result =
template.request("glogin://camelcloud.appspot.com", new
Processor() {
            public void process(Exchange exchange) throws
Exception {
                exchange.getIn().setHeader(GLOGIN_USER_NAME,
"replaceme@gmail.com");
                exchange.getIn().setHeader(GLOGIN_PASSWORD,
"replaceme");
            }
        });

assertNotNull(result.getOut().getHeader(GLOGIN_COOKIE));
    }
}

```

The resulting authorization cookie from login to a development server looks like

```
ahlogincookie=test@example.org:false:11223191102230730701;Path=/
```

The resulting authorization cookie from login to a deployed GAE application looks (shortened) like

```
ACSID=AJKiYcE...XxhH9P_jR_V3; expires=Sun, 07-Feb-2010 15:14:51 GMT; path=/
```

41.5. GMAIL

gmail Component

The **gmail** component contributes to the [Camel Components for Google App Engine \(GAE\)](#). It supports sending of emails via the GAE [mail service](#). Receiving mails is not supported yet but will be added later. Currently, only Google accounts that are application administrators can send emails.

URI format

```
gmail://user@gmail.com[?options]
gmail://user@googlemail.com[?options]
```

Options

| Name | Default Value | Context | Description |
|----------------|---------------|----------|--|
| to | null | Producer | To-receiver of the email. This can be a single receiver or a comma-separated list of receivers. |
| cc | null | Producer | Cc-receiver of the email. This can be a single receiver or a comma-separated list of receivers. |
| bcc | null | Producer | Bcc-receiver of the email. This can be a single receiver or a comma-separated list of receivers. |
| subject | null | Producer | Subject of the email. |

| | | | |
|---------------------------------|----------------------------------|----------|---|
| outboundBindingReference | reference to GMailBinding | Producer | Reference to an OutboundBinding<GMailEndpoint, MailService.Message, void> in the Registry for customizing the binding of an Exchange to the mail service. |
|---------------------------------|----------------------------------|----------|---|

Message headers

| Name | Type | Context | Description |
|-----------------------------------|---------------|----------|---|
| GMailBinding.GMAIL_SUBJECT | String | Producer | Subject of the email. Overrides subject endpoint option. |
| GMailBinding.GMAIL_SENDER | String | Producer | Sender of the email. Overrides sender definition in endpoint URI. |
| GMailBinding.GMAIL_TO | String | Producer | To-receiver(s) of the email. Overrides to endpoint option. |
| GMailBinding.GMAIL_CC | String | Producer | Cc-receiver(s) of the email. Overrides cc endpoint option. |
| GMailBinding.GMAIL_BCC | String | Producer | Bcc-receiver(s) of the email. Overrides bcc endpoint option. |

Message body

On the producer side the **in** message body is converted to a **String**.

Usage

```
...
.setHeader(GMailBinding.GMAIL_SUBJECT, constant("Hello"))
.setHeader(GMailBinding.GMAIL_TO, constant("account2@somewhere.com"))
.to("gmail://account1@gmail.com");
```

Sends an email with subject **Hello** from **account1@gmail.com** to **account2@somewhere.com**. The mail message body is taken from the **in** message body. Please note that **account1@gmail.com** must be an administrator account for the current GAE application.

Dependencies

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



POM.XML

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

where `${camel-version}` must be replaced by the actual version of Apache Camel (2.1.0 or higher).

41.6. GSEC

Security for Apache Camel GAE Applications

Securing GAE applications from unauthorized access is described in the [Security and Authentication](#) section of the Google App Engine documentation. Authorization constraints are declared in the `web.xml` file (see [the section called "The web.xml"](#)). This applies to Apache Camel applications as well. In the following example, the application is configured to only allow authenticated users (in any role) to access the application. Additionally, access to `/worker/*` URLs may only be done by users in the admin role. By default, web hook URLs installed by the `gtask` component match the `/worker/*` pattern and should not be accessed by normal users. With this authorization constraint, only the task queuing service (which is always in the admin role) is allowed to access the web hooks. For implementing custom, non-declarative authorization logic, Apache Camel GAE applications should use the [Google Accounts Java API](#).

Example 41.1. web.xml with authorization constraint

```
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</serv
let-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>appctx.xml</param-value>
    </init-param>
  </servlet>

  <!--
```

```

        Mapping used for external requests
    -->
<servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/camel/*</url-pattern>
</servlet-mapping>

<!--
    Mapping used for web hooks accessed by task queueing service.
-->
<servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/worker/*</url-pattern>
</servlet-mapping>

<!--
    By default allow any user who is logged in to access the whole
    application.
-->
<security-constraint>
    <web-resource-collection>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>*</role-name>
    </auth-constraint>
</security-constraint>

<!--
    Allow only admin users to access /worker/* URLs e.g. to prevent
    normal user to access gtask web hooks.
-->
<security-constraint>
    <web-resource-collection>
        <url-pattern>/worker/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>

</web-app>

```

41.7. GTASK

gtask Component

The **gtask** component contributes to the [Camel Components for Google App Engine \(GAE\)](#). It supports asynchronous message processing on GAE by using the [task queueing service](#) as message queue. For adding messages to a queue it uses the task queue API. For receiving messages from a queue it installs an HTTP callback handler. The handler is called by an HTTP POST callback (a [web hook](#)) initiated by the task queueing service. Whenever a new task is added to a queue a callback will be sent. The **gtask** component abstracts from these details and supports endpoint URIs that make message queueing on GAE as easy as

message queueing with [JMS](#) or [SEDA](#).

URI format

`gtask://queue-name`

Options

| Name | Default Value | Context | Description |
|--------------------------------|--|----------|--|
| <code>workerRoot</code> | <code>worker</code> | Producer | The servlet mapping for callback handlers. By default, this component requires a callback servlet mapping of <code>/worker/*</code> . If another servlet mapping is used e.g. <code>/myworker/*</code> it must be set as option on the producer side: <code>to("gtask:myqueue?workerRoot=myworker")</code> . |
| <code>inboundBindingRef</code> | reference to <code>GTaskBinding</code> | Consumer | Reference to an <code>InboundBinding<GTaskEndpoint, HttpServletRequest, HttpServletResponse></code> in the Registry for customizing the binding of an <code>Exchange</code> to the Servlet API. The referenced binding is used as post-processor to <code>org.apache.camel.component.http.HttpBinding</code> . |

| | | | |
|---------------------------|----------------------------------|----------|--|
| outboundBindingRef | reference to GTaskBinding | Producer | Reference to an OutboundBinding<GTaskEndpoint, TaskOptions, void> in the Registry for customizing the binding of an Exchange to the task queueing service. |
|---------------------------|----------------------------------|----------|--|

On the consumer-side, all options of the [Servlet component](#) are supported.

Message headers

On the consumer-side all headers of the [Servlet component](#) component are supported plus the following.

| Name | Type | Context | Description |
|---------------------------------------|---------------|----------|-------------------------------------|
| GTaskBinding.GTASK_QUEUE_NAME | String | Consumer | Name of the task queue. |
| GTaskBinding.GTASK_TASK_NAME | String | Consumer | Name of the task (generated value). |
| GTaskBinding.GTASK_RETRY_COUNT | int | Consumer | Number of callback retries. |

Message body

On the producer side the **in** message body is converted to **abyte[]** and is POSTed to the callback handler as content-type **application/octet-stream**.

Usage

Setting up tasks queues is an administrative task on Google App Engine. Only one queue is pre-configured and can be referenced by name out-of-the-box: the **default** queue. This queue will be used in the following examples. Please note that when using task queues on the local development server, tasks must be executed manually from the [developer console](#).

Default queue

```
...
.to(gtask:default) // add message to default queue

from(gtask:default) // receive message from default queue (via a web hook)
...
```

This example requires the following servlet mapping.



WEB.XML

```

...
<servlet>
  <servlet-name>CamelServlet</servlet-name>
  <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServ
let</servlet-class>
  ...
</servlet>
...
<servlet-mapping>
  <servlet-name>CamelServlet</servlet-name>
  <url-pattern>/worker/*</url-pattern>
</servlet-mapping>
...

```

Dependencies

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



POM.XML

```

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

```

where **`${camel-version}`** must be replaced by the actual version of Apache Camel (2.1.0 or higher).

CHAPTER 42. GEOCODER

GEOCODER COMPONENT

Available as of Camel 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>
```

URI FORMAT

```
geocoder:address:name[?options]
geocoder:latlng:latitude,longitude[?options]
```

OPTIONS

| Property | Default | Description |
|--------------------|--------------|---|
| language | en | The language to use. |
| headersOnly | false | Whether to only enrich the Exchange with headers, and leave the body as-is. |
| clientId | | To use google premium with this client id |
| clientKey | | To use google premium with this client key |

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

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](#).

MESSAGE HEADERS

| Header | Description |
|--|---|
| <code>CamelGeoCoderStatus</code> | Mandatory. Status code from the geocoder library. If status is <code>GeocoderStatus.OK</code> then additional headers is enriched |
| <code>CamelGeoCoderAddress</code> | The formatted address |
| <code>CamelGeoCoderLat</code> | The latitude of the location. |
| <code>CamelGeoCoderLng</code> | The longitude of the location. |
| <code>CamelGeoCoderLatLng</code> | The latitude and longitude of the location. Separated by comma. |
| <code>CamelGeoCoderCity</code> | The city long name. |
| <code>CamelGeoCoderRegionCode</code> | The region code. |
| <code>CamelGeoCoderRegionName</code> | The region name. |
| <code>CamelGeoCoderCountryLong</code> | The country long name. |
| <code>CamelGeoCoderCountryShort</code> | The country short name. |

Notice not all headers may be provided depending on available data and mode in use (address vs latlng).

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 43. GUAVA EVENTBUS

GUAVA EVENTBUS COMPONENT

Available since Camel 2.10.0

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>
```

URI FORMAT

```
guava-eventbus:busName[?options]
```

Where **busName** represents the name of the `com.google.common.eventbus.EventBus` instance located in the Camel registry.

OPTIONS

| Name | Default Value | Description |
|-------------------|---------------|---|
| eventClass | null | Camel 2.10: 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. |

| | | |
|--------------------------|-------------|--|
| listenerInterface | null | Camel 2.11: 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. |
|--------------------------|-------------|--|

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 messageHandler(String message) {
        System.out.println("Message received from the Camel: " + message);
    }
});
```

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 snippet 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");
```

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");
```

CHAPTER 44. HAWTDB

HAWTDB

Available as of Apache Camel 2.3

[HawtDB](#) is a very lightweight and embeddable key value database. It allows together with Apache Camel to provide persistent support for various Apache Camel features such as [section "Aggregator" in "Apache Camel Development Guide"](#).



DEPRECATED

The [HawtDB](#) project is being deprecated and replaced by [leveldb](#) as the lightweight and embeddable 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.

Current features it provides:

- [HawtDBAggregationRepository](#)

USING HAWTDBAGGREGATIONREPOSITORY

[HawtDBAggregationRepository](#) 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](#).

It has the following options:

| Option | Type | Description |
|---------------------------------|--------|---|
| <code>repositoryName</code> | String | A mandatory repository name. Allows you to use a shared HawtDBFile for multiple repositories. |
| <code>persistentFileName</code> | 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. |
| 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 Apache Camel section "Aggregator" in "Apache Camel Development Guide" 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.

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.

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 Apache 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 Apache 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](#).

USING HAWTDBAGGREGATIONREPOSITORY IN JAVA DSL

In this example we want to persist aggregated messages in the `target/data/hawtdb.dat` file.

```
public void configure() throws Exception {
    // create the hawtdb repo
    HawtDBAggregationRepository repo = new
    HawtDBAggregationRepository("repo1", "target/data/hawtdb.dat");

    // here is the Camel route where we aggregate
    from("direct:start")
        .aggregate(header("id"), new MyAggregationStrategy())
        // use our created hawtdb repo as aggregation repository
        .completionSize(5).aggregationRepository(repo)
        .to("mock:aggregated");
}
```

USING HAWTDBAGGREGATIONREPOSITORY IN SPRING XML

The same example but using Spring XML instead:

```
<!-- a persistent aggregation repository using camel-hawtdb -->
<bean id="repo"
class="org.apache.camel.component.hawtdb.HawtDBAggregationRepository">
    <!-- store the repo in the hawtdb.dat file -->
    <property name="persistentFileName" value="target/data/hawtdb.dat"/>
    <!-- and use repo2 as the repository name -->
    <property name="repositoryName" value="repo2"/>
</bean>

<!-- aggregate the messages using this strategy -->
<bean id="myAggregatorStrategy"
class="org.apache.camel.component.hawtdb.HawtDBSpringAggregateTest$MyAggregationStrategy"/>
```



```

<!-- this is the camel routes -->
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

    <route>
        <from uri="direct:start"/>
        <!-- aggregate using our strategy and hawtdb repo, and complete
when we have 5 messages aggregated -->
        <aggregate strategyRef="myAggregatorStrategy"
aggregationRepositoryRef="repo" completionSize="5">
            <!-- correlate by header with the key id -->
            <correlationExpression><header>id</header>
        </correlationExpression>
        <!-- send aggregated messages to the mock endpoint -->
        <to uri="mock:aggregated"/>
    </aggregate>
</route>

</camelContext>

```

DEPENDENCIES

To use [HawtDB](#) in your Apache 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 version](#)).

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hawtdb</artifactId>
    <version>2.3.0</version>
</dependency>

```

See Also:

- [section "Aggregator" in "Apache Camel Development Guide"](#)
- [SQL](#)
- [Components](#)

CHAPTER 45. HAZELCAST COMPONENT

HAZELCAST COMPONENT

Available as of Apache Camel 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, eviction 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>
```

URI FORMAT

```
hazelcast:[ map | multimap | queue | seda | set | atomicvalue |
instance]:cachename[?options]
```



WARNING

You have to use the second prefix to define which type of data store you want to use.

SECTIONS

1. Usage of [#map](#)
2. Usage of [#multimap](#)
3. Usage of [#queue](#)
4. Usage of [#list](#)
5. Usage of [#seda](#)

6. Usage of [atomic number](#)
7. Usage of [#cluster](#) support (instance)

USAGE OF MAP

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 5 operations (put, get, update, delete, query). For the first 4 you have to provide the operation inside the "hazelcast.operation.type" header variable. In Java DSL you can use the constants from `org.apache.camel.component.hazelcast.HazelcastConstants`.

Header Variables for the request message:

| Name | Type | Description |
|---------------------------------------|---------------|---|
| <code>hazelcast.operation.type</code> | String | valid values are: put, delete, get, update, query |
| <code>hazelcast.objectId</code> | String | the object id to store / find your object inside the cache (not needed for the query operation) |



WARNING

Header variables have changed in Apache Camel 2.8

| Name | Type | Description |
|--|---------------|--|
| <code>CamelHazelcastOperationType</code> | String | valid values are: put, delete, get, update, query Version 2.8 |
| <code>CamelHazelcastObjectId</code> | String | the object id to store / find your object inside the cache (not needed for the query operation) Version 2.8 |

You can call the samples with:

```
template.sendBodyAndHeader("direct:[put|get|update|delete|query]", "my-foo", HazelcastConstants.OBJECT_ID, "4711");
```

SAMPLE FOR PUT:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUT_OPERATION))
.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 GET:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
.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>
```

SAMPLE FOR UPDATE:

Java DSL:

```
from("direct:update")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.UPDATE_OPERATION))
.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>
```

SAMPLE FOR DELETE:

Java DSL:

```
from("direct:delete")
  .setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.DELETE_OPERATION))
  .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>
```

SAMPLE FOR QUERY

Java DSL:

```
from("direct:query")
  .setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.QUERY_OPERATION))
  .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);
```

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 **evict**. 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 |
|--|---------------|--|
| <code>hazelcast.listener.time</code> | Long | time of the event in millis |
| <code>hazelcast.listener.type</code> | String | the map consumer sets here "cachelister" |
| <code>hazelcast.listener.action</code> | String | type of event - here added , updated , evicted and removed |
| <code>hazelcast.objectId</code> | String | the oid of the object |
| <code>hazelcast.cache.name</code> | String | the name of the cache - e.g. "foo" |
| <code>hazelcast.cache.type</code> | String | the type of the cache - here map |



WARNING

Header variables have changed in Apache Camel 2.8

| Name | Type | Description |
|------|------|-------------|
|------|------|-------------|

| | | |
|-------------------------------------|---------------|--|
| CamelHazelcastListenerTime | Long | time of the event in millis Version 2.8 |
| CamelHazelcastListenerType | String | the map consumer sets here "cachelister" Version 2.8 |
| CamelHazelcastListenerAction | String | type of event - here added, updated, envicted and removed . Version 2.8 |
| CamelHazelcastObjectId | String | the oid of the object Version 2.8 |
| CamelHazelcastCacheName | String | the name of the cache - e.g. "foo" Version 2.8 |
| CamelHazelcastCacheType | String | the type of the cache - here map Version 2.8 |

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!");

```

USAGE OF MULTI MAP

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 |
|---------------------------------------|--------|--|
| <code>hazelcast.operation.type</code> | String | valid values are: put, get, removevalue, delete |
| <code>hazelcast.objectId</code> | String | the object id to store / find your object inside the cache |



WARNING

Header variables have changed in Apache Camel 2.8

Header Variables for the request message in Apache Camel 2.8:

| Name | Type | Description |
|--|--------|--|
| <code>CamelHazelcastOperationType</code> | String | valid values are: put, delete, get, update, query Available as of Apache Camel 2.8 |
| <code>CamelHazelcastObjectId</code> | String | the object id to store / find your object inside the cache (not needed for the query operation) Version 2.8 |

You can call the samples with:

```
template.sendBodyAndHeader("direct:[put|get|update|delete|query]", "my-foo", HazelcastConstants.OBJECT_ID, "4711");
```

SAMPLE FOR PUT:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUT_OPERATION))
```



```
.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 GET:

Java DSL:

```
from("direct:get")
  .setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
  .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>
```

SAMPLE FOR UPDATE:

Java DSL:

```
from("direct:update")
  .setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.UPDATE_OPERATION))
  .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>

```

SAMPLE FOR DELETE:

Java DSL:

```

from("direct:delete")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.DELETE_OPERATION))
  .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>

```

SAMPLE FOR QUERY

Java DSL:

```

from("direct:query")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.QUERY_OPERATION))
  .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);
```

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 **evict**. 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 |
|--|---------------|--|
| <code>hazelcast.listener.time</code> | Long | time of the event in millis |
| <code>hazelcast.listener.type</code> | String | the map consumer sets here "cachelister" |
| <code>hazelcast.listener.action</code> | String | type of event - here added , updated , evicted and removed |
| <code>hazelcast.objectId</code> | String | the oid of the object |
| <code>hazelcast.cache.name</code> | String | the name of the cache - e.g. "foo" |
| <code>hazelcast.cache.type</code> | String | the type of the cache - here map |



WARNING

Header variables have changed in Apache Camel 2.8

| Name | Type | Description |
|---|---------------|---|
| <code>CamelHazelcastListenerTime</code> | Long | time of the event in millis Version 2.8 |
| <code>CamelHazelcastListenerType</code> | String | the map consumer sets here "cachelister" Version 2.8 |

| | | |
|-------------------------------------|---------------|---|
| CamelHazelcastListenerAction | String | type of event - here added, updated, evicted and removed . Version 2.8 |
| CamelHazelcastObjectId | String | the oid of the object Version 2.8 |
| CamelHazelcastCacheName | String | the name of the cache - e.g. "foo" Version 2.8 |
| CamelHazelcastCacheType | String | the type of the cache - here map Version 2.8 |

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.ENVICED))
    .log("...evicted")
    .to("mock:evicted")

  .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!");
```

USAGE OF MULTI MAP

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 |
|---------------------------------------|--------|--|
| <code>hazelcast.operation.type</code> | String | valid values are: put, get, removevalue, delete |
| <code>hazelcast.objectId</code> | String | the object id to store / find your object inside the cache |



WARNING

Header variables have changed in Apache Camel 2.8

| Name | Type | Description |
|--|--------|---|
| <code>CamelHazelcastOperationType</code> | String | valid values are: put, get, removevalue, delete Version 2.8 |
| <code>CamelHazelcastObjectId</code> | String | the object id to store / find your object inside the cache Version 2.8 |

SAMPLE FOR PUT:

Java DSL:

```
from("direct:put")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.PUT_OPERATION))
  .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>
```

SAMPLE FOR REMOVEVALUE:

Java DSL:

```
from("direct:removevalue")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.REMOVEVALUE_OPERATION))
  .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 } **you have to put "my-foo" inside the message body to remove the "my-foo" value.**

SAMPLE FOR GET:

Java DSL:

```
from("direct:get")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.GET_OPERATION))
  .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>
```

SAMPLE FOR DELETE:

Java DSL:

```
from("direct:delete")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.DELETE_OPERATION))
  .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");
```

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 eviction 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("...evicted")
        //          .to("mock:evicted")

.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 |
|--|--------|---|
| <code>hazelcast.listener.time</code> | Long | time of the event in millis |
| <code>hazelcast.listener.type</code> | String | the map consumer sets here "cachelister" |
| <code>hazelcast.listener.action</code> | String | type of event - here added and removed (and soon evicted) |
| <code>hazelcast.objectId</code> | String | the oid of the object |
| <code>hazelcast.cache.name</code> | String | the name of the cache - e.g. "foo" |
| <code>hazelcast.cache.type</code> | String | the type of the cache - here multimap |

Eviction will be added as feature, soon (this is a Hazelcast issue).



WARNING

Header variables have changed in Apache Camel 2.8

| Name | Type | Description |
|------|------|-------------|
|------|------|-------------|

| | | |
|------------------------------|--------|--|
| CamelHazelcastListenerTime | Long | time of the event in millis Version 2.8 |
| CamelHazelcastListenerType | String | the map consumer sets here "cachelister" Version 2.8 |
| CamelHazelcastListenerAction | String | type of event - here added and removed (and soon evicted) Version 2.8 |
| CamelHazelcastObjectId | String | the oid of the object Version 2.8 |
| CamelHazelcastCacheName | String | the name of the cache - e.g. "foo" Version 2.8 |
| CamelHazelcastCacheType | String | the type of the cache - here multimap Version 2.8 |

USAGE OF QUEUE

QUEUE PRODUCER – TO("HAZELCAST:QUEUE:FOO")

The queue producer provides 6 operations (add, put, poll, peek, offer, removevalue).

SAMPLE FOR ADD:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.ADD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

SAMPLE FOR PUT:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

SAMPLE FOR POLL:

```

from("direct:poll")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.POLL_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);

```

SAMPLE FOR PEEK:

```

from("direct:peek")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.PEEK_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);

```

SAMPLE FOR OFFER:

```

from("direct:offer")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.OFFER_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);

```

SAMPLE FOR REMOVEVALUE:

```

from("direct:removevalue")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.REMOVEVALUE_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);

```

QUEUE CONSUMER – FROM("HAZELCAST:QUEUE:FOO")

The queue consumer provides 2 operations (add, remove).

```

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!");

```

USAGE OF LIST

LIST PRODUCER – TO("HAZELCAST:LIST:F00")

The list producer provides 4 operations (add, set, get, removevalue).

SAMPLE FOR ADD:

```
from("direct:add")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.ADD_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

SAMPLE FOR GET:

```
from("direct:get")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.GET_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX)
  .to("seda:out");
```

SAMPLE FOR SETVALUE:

```
from("direct:set")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.SETVALUE_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

SAMPLE FOR REMOVEVALUE:

```
from("direct:removevalue")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.REMOVEVALUE_OPERATION))
  .toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

**WARNING**

Please note that `set`, `get` and `removevalue` and not yet supported by hazelcast, will be added in the future..

LIST CONSUMER – FROM("HAZELCAST:LIST:F00")

The list consumer provides 2 operations (add, remove).

```
fromF("hazelcast:%s", 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!");
```

USAGE OF SEDA

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.

SEDA PRODUCER – TO("HAZELCAST:SEDA:F00")

The SEDA producer provides no operations. You only send data to the specified queue.

| Name | default value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|-------------------------------|--------------------|--|
| <code>transferExchange</code> | <code>false</code> | Apache Camel 2.8.0: if set to true the whole Exchange will be transferred. If header or body contains not serializable objects, they will be skipped. |
|-------------------------------|--------------------|--|

Java DSL :

```
from("direct:foo")
.to("hazelcast:seda:foo");
```

Spring DSL :

```
<route>
  <from uri="direct:start" />
  <to uri="hazelcast:seda:foo" />
</route>
```

SEDA CONSUMER – FROM("HAZELCAST:SEDA:FOO")

The SEDA consumer provides no operations. You only retrieve data from the specified queue.

| Name | default value | Description |
|----------------------------------|--------------------|---|
| <code>pollInterval</code> | <code>1000</code> | How frequent to poll from the SEDA queue |
| <code>concurrentConsumers</code> | <code>1</code> | To use concurrent consumers polling from the SEDA queue. |
| <code>transferExchange</code> | <code>false</code> | Camel 2.8.0: if set to true the whole Exchange will be transferred. If header or body contains not serializable objects, they will be skipped. |

| | | |
|------------|-------|--|
| transacted | false | Camel 2.10.4: 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. |
|------------|-------|--|

Java DSL :

```
from("hazelcast:seda:foo")
.to("mock:result");
```

Spring DSL:

```
<route>
  <from uri="hazelcast:seda:foo" />
  <to uri="mock:result" />
</route>
```

USAGE OF ATOMIC NUMBER



WARNING

There is no consumer for this endpoint\!

ATOMIC NUMBER PRODUCER - TO("HAZELCAST:ATOMICNUMBER:FOO")

An atomic number is an object that simply provides a grid wide number (long). The operations for this producer are setvalue (set the number with a given value), get, increase (+1), decrease (-1) and destroy.

Header Variables for the request message:

| Name | Type | Description |
|--------------------------|--------|--|
| hazelcast.operation.type | String | valid values are: setvalue, get, increase, decrease, destroy |



WARNING

Header variables have changed in Apache Camel 2.8

| Name | Type | Description |
|-----------------------------|--------|--|
| CamelHazelcastOperationType | String | valid values are: setvalue, get, increase, decrease, destroy Available as of Apache Camel version 2.8 |

SAMPLE FOR SET:

Java DSL:

```
from("direct:set")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.SETVALUE_OPERATION))
  .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);`

SAMPLE FOR GET:

Java DSL:

```

from("direct:get")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.GET_OPERATION))
.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);`.

SAMPLE FOR INCREMENT:

Java DSL:

```

from("direct:increment")
.setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.INCREMENT_OPERATION))
.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.

SAMPLE FOR DECREMENT:

Java DSL:

```

from("direct:decrement")
  .setHeader(HazelcastConstants.OPERATION,
constant(HazelcastConstants.DECREMENT_OPERATION))
  .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.

SAMPLE FOR DESTROY

**WARNING**

There's a bug inside Hazelcast. So this feature may not work properly. Will be fixed in 1.9.3.

Java DSL:

```
from("direct:destroy")
  .setHeader(HazelcastConstants.OPERATION,
    constant(HazelcastConstants.DESTROY_OPERATION))
  .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>
```

CLUSTER SUPPORT**WARNING**

This endpoint provides no producer\!

INSTANCE CONSUMER - FROM("HAZELCAST:INSTANCE:FOO")

Hazelcast makes sense in one single "server node", but it's extremely powerful in a clustered environment. 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 |
|--|---------|---|
| <code>hazelcast.listener.time</code> | Long | time of the event in millis |
| <code>hazelcast.listener.type</code> | String | the map consumer sets here "instancelistener" |
| <code>hazelcast.listener.action</code> | String | type of event - here added or removed |
| <code>hazelcast.instance.host</code> | String | host name of the instance |
| <code>hazelcast.instance.port</code> | Integer | port number of the instance |



WARNING

Header variables have changed in Apache Camel 2.8

| Name | Type | Description |
|------------------------------|---------|---|
| CamelHazelcastListenerTime | Long | time of the event in millis Version 2.8 |
| CamelHazelcastListenerType | String | the map consumer sets here "instancelistener" Version 2.8 |
| CamelHazelcastListenerAction | String | type of event - here added or removed . Version 2.8 |
| CamelHazelcastInstanceHost | String | host name of the instance Version 2.8 |
| CamelHazelcastInstancePort | Integer | port number of the instance Version 2.8 |

CHAPTER 46. HBASE

HBASE COMPONENT

Available as of Camel 2.10

This component provides an idempotent 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>
```

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](#).

CAMEL AND HBASE

When using a data store inside a camel route, there is always the challenge of specifying how the camel message will be stored to the data store. 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.

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](#)

HBASE PRODUCER

As mentioned above `camel` provides producers 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**

SUPPORTED URI OPTIONS ON PRODUCER

| Name | Default Value | Description |
|----------------------------------|----------------------------|--|
| <code>operation</code> | <code>CamelHBasePut</code> | The HBase operation to perform. Supported values: <code>CamelHBasePut</code> , <code>CamelHBaseGet</code> , <code>CamelHBaseDelete</code> , and <code>CamelHBaseScan</code> . |
| <code>maxResults</code> | <code>100</code> | The maximum number of rows to scan. Supported operations: <code>CamelHBaseScan</code> . |
| <code>mappingStrategyName</code> | <code>header</code> | The strategy to use for mapping Camel messages to HBase columns. Supported values: <code>header</code> , or <code>body</code> . |

| | | |
|---------------------------------|-------------|---|
| mappingStrategyClassName | null | The class name of a custom mapping strategy implementation. |
| filters | null | A list of filters. Supported operations: CamelHBaseScan . |

Header mapping options:

| Name | Default Value | Description |
|------------------|----------------------|--|
| rowId | | The id of the row. This has limited use as the row usually changes per Exchange. |
| rowType | String | 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 | String | The value type. Supports a number suffix for referring to more than one columns. Supported operations: CamelHBaseGet , and CamelHBaseScan . |

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&amily=myfamily&ualifier=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:

```
<route>
  <from uri="direct:in"/>
  <!-- Set the HBase Row 1st column -->
  <setHeader headerName="CamelHBaseRowId">
    <el>${in.body.id}</el>
  </setHeader>
  <!-- Set the HBase Row 2nd column -->
  <setHeader headerName="CamelHBaseRowId">
    <el>${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&amily=myfamily&ualifier=myqualifier&amily2=myfamil
y&ualifier2=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.

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?
opertaion=CamelHBaseGet&amily=myfamily&ualifier=myqualifier&alueType=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.

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?opertaion=CamelHBaseDelete"/>
</route>

```

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?
opertaion=CamelHBaseScan&amily=myfamily&ualifier=myqualifier&alueType=java
.lang.Long&owType=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?
opertaion=CamelHBaseScan&ilters=#myFilterList"/>
</route>

<bean id="myFilters" class="java.util.ArrayList">
    <constructor-arg>
        <list>
            <bean
class="org.apache.camel.component.hbase.filters.ModelAwareColumnMatchingFi
lter"/>
        </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.

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=java.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.

SUPPORTED URI OPTIONS ON CONSUMER

| Name | Default Value | Description |
|---------------------------------------|------------------------------------|--|
| <code>initialDelay</code> | <code>1000</code> | Milliseconds before the first polling starts. |
| <code>delay</code> | <code>500</code> | Milliseconds before the next poll. |
| <code>useFixedDelay</code> | <code>true</code> | Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details. |
| <code>timeUnit</code> | <code>TimeUnit.MILLISECONDS</code> | time unit for <code>initialDelay</code> and <code>delay</code> options. |
| <code>runLoggingLevel</code> | <code>TRACE</code> | Camel 2.8: The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that. |
| <code>operation</code> | <code>CamelHBasePut</code> | The HBase operation to perform. Supported values: <code>CamelHBasePut</code> , <code>CamelHBaseGet</code> , <code>CamelHBaseDelete</code> , and <code>CamelHBaseScan</code> . |
| <code>maxResults</code> | <code>100</code> | The maximum number of rows to scan. Supported operations: <code>CamelHBaseScan</code> . |
| <code>mappingStrategyName</code> | <code>header</code> | The strategy to use for mapping Camel messages to HBase columns. Supported values: <code>header</code> , or <code>body</code> . |
| <code>mappingStrategyClassName</code> | <code>null</code> | The class name of a custom mapping strategy implementation. |

| | | |
|----------------|-------------|--|
| filters | null | A list of filters. Supported operations: CamelHBaseScan |
|----------------|-------------|--|

Header mapping options:

| Name | Default Value | Description |
|------------------|----------------------|--|
| rowId | | The id of the row. This has limited use as the row usually changes per Exchange. |
| rowType | String | The type to covert row id to. Supported operations: CamelHBaseScan |
| family | | The column family. *upports 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 |
| rowModel | String | An instance of org.apache.camel.component.hbase.model.HBaseRow which describes how each row should be modeled |

If the role of the `rowModel` is not clear, it allows you to construct the `HBaseRow` modle programmatically instead of "describing" it with uri options (such as `family`, `qualifier`, `type` etc).

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");
```

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.

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 |
|---------------------|-------------|
| CamelHBaseRowId | myrow |
| CamelHBaseFamily | myfamily |
| CamelHBaseQualifier | myqualifier |
| CamelHBaseValue | 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 example:

| Header | Value |
|---------------------|-------------|
| CamelHBaseFamily | myfamily |
| CamelHBaseQualifier | myqualifier |
| CamelHBaseValueType | 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.

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?mappingStrategy=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.addRow().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.

SEE ALSO

- [Polling Consumer](#)
- [Apache HBase](#)

CHAPTER 47. HDFS

HDFS COMPONENT

Available as of Camel 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>
```

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 `seg0`, `seg1`, `seg2`, etc.

OPTIONS

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|-----------------------|--------------------|---|
| overwrite | true | The file can be overwritten |
| append | false | Append to existing file. Notice that not all HDFS file systems support the append option. |
| bufferSize | 4096 | The buffer size used by HDFS |
| replication | 3 | The HDFS replication factor |
| blockSize | 67108864 | The size of the HDFS blocks |
| fileType | NORMAL_FILE | It can be SEQUENCE_FILE, MAP_FILE, ARRAY_FILE, or BLOOMMAP_FILE, see Hadoop |
| fileSystemType | HDFS | It can be LOCAL for local filesystem |
| keyType | NULL | The type for the key in case of sequence or map files. See below. |
| valueType | TEXT | The type for the key in case of sequence or map files. See below. |
| splitStrategy | | A string describing the strategy on how to split the file based on different criteria. See below. |
| openedSuffix | opened | When a file is opened for reading/ writing the file is renamed with this suffix to avoid to read it during the writing phase. |
| readSuffix | read | Once the file has been read is renamed with this suffix to avoid to read it again. |
| initialDelay | 0 | For the consumer, how much to wait (milliseconds) before to start scanning the directory. |
| delay | 0 | The interval (milliseconds) between the directory scans. |
| pattern | * | The pattern used for scanning the directory |

| | | |
|-------------------------|-------------|---|
| chunkSize | 4096 | When reading a normal file, this is split into chunks producing a message per chunk |
| connectOnStartup | true | Camel 2.9.3/2.10.1: 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. |

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`, in this case is written in a sequence file or a map file as a sequence of bytes.

SPLITTING STRATEGY

In the current version of Hadoop opening a file in append mode is disabled, since it's not reliable enough. 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 actual file name will become a directory name and a `<file name>/seg0` will be initially created.
- Every time a splitting condition is met a new file is created with name `<original file name>/segN` where N is 1, 2, 3, etc. 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

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/simplefile you'll find the following files seg0, seg1, seg2, etc.

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 explicit 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 is various strategy that control when the stream is closed.

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 48. HL7

HL7 COMPONENT

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](#)
- Agnostic data format using either plain String objects or HAPI HL7 model objects.
- [Type Converter](#) from/to HAPI and String
- HL7 DataFormat using HAPI library
- Even more ease-of-use as it's integrated well with the [Chapter 76, MINA \[Deprecated\]](#) (Camel 2.11: [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>
```

HL7 MLLP PROTOCOL

HL7 is often used with the HL7 MLLP protocol that is a text based TCP socket based protocol. This component ships with a Mina Codec that conforms

to the MLLP protocol so you can easily expose a HL7 listener that accepts HL7 requests over the TCP transport.

To expose a HL7 listener service we reuse the existing mina/mina2 component where we just use the HL7MLLPCodec as codec.

The HL7 MLLP codec has the following options:

| 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 (is a charset name) to use for the codec. If not provided, Camel will use the JVM default Charset . |
| convertLFtoCR | true (Camel 2.11:false) | Will convert \n to \r (0x0d, 13 decimal) as HL7 stipulates \r as segment terminators. The HAPI library requires the use of \r. |
| validate | true | Whether HAPI Parser should validate or not. |
| parser | ca.uhn.hl7v2.parser.PipeParser | *Camel 2.11:* To use a custom parser. Must be of type <code>ca.uhn.hl7v2.parser.Parser</code> . |

EXPOSING A HL7 LISTENER

In our Spring XML file, we configure an endpoint to listen for HL7 requests using TCP:

```
<endpoint id="hl7listener" uri="mina:tcp://localhost:8888?"
```



```
sync=true&odec=#hl7codec"/>
  <!-- Camel 2.11: uri="mina2:tcp... -->
```

Notice that we use TCP on localhost on port 8888. We use `sync=true` to indicate that this listener is synchronous and therefore will return a HL7 response to the caller. Then we setup mina to use our HL7 codec with `codec=#hl7codec`. Notice that `hl7codec` is just a Spring bean ID, so we could have named it `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>
```

Above we also configure the charset encoding to use (`iso-8859-1`).

The endpoint `hl7listener` can then be used in a route as a consumer, as this Java DSL example illustrates:

```
from("hl7listener").to("patientLookupService");
```

This is a very simple route that will listen for HL7 and route it to a service named `patientLookupService` that is also a Spring bean ID we have configured in the Spring XML as:

```
<bean id="patientLookupService"
class="com.mycompany.healthcare.service.PatientLookupService"/>
```

Another powerful feature of Camel is that we can have our business logic in POJO classes that is not tied to 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
}

```

Notice that this class uses just imports from the HAPI library and not from Camel.

HL7 MODEL USING JAVA.LANG.STRING

The HL7MLLP codec uses plain String as its data format. Camel uses its [Type Converter](#) to convert to/from strings to the HAPI HL7 model objects. However, you can use plain String objects if you prefer, for instance if you wish to parse the data yourself.

See samples for such an example.

HL7V2 MODEL USING HAPI

The HL7v2 model uses Java objects from the HAPI library. Using this library, we can encode and decode from the EDI format (ER7) that is mostly used with HL7v2. With this model you can code with Java objects instead of the EDI based HL7 format that can be hard for humans to read and understand.

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 we can work with the data as a `ca.uhn.hl7v2.model.Message` object. To retrieve the patient ID in the message above, you can do this in Java code:

```

Message msg = exchange.getIn().getBody(Message.class);
QRD qrd = (QRD)msg.get("QRD");
String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();

```

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();

```

Camel has built-in type converters, so when this operation is invoked:

```

Message msg = exchange.getIn().getBody(Message.class);

```

Camel will convert the received HL7 data from String to Message. This is powerful when combined with the HL7 listener, then you as the end-user don't have to work with byte[], String or any other simple object formats. You can just use the HAPI HL7v2 model objects.

MESSAGE HEADERS

The `unmarshal` operation adds these MSH fields as headers on the Camel message:

| Key | MSH field | Example |
|------------------------------|-----------|----------------|
| CamelHL7SendingApplication | MSH-3 | MYSERVER |
| 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 |

| | | |
|-------------------------------------|----------------------|-------------------|
| <code>CamelHL7TriggerEvent</code> | <code>MSH-9-2</code> | <code>A01</code> |
| <code>CamelHL7MessageControl</code> | <code>MSH-10</code> | <code>1234</code> |
| <code>CamelHL7ProcessingId</code> | <code>MSH-11</code> | <code>P</code> |
| <code>CamelHL7VersionId</code> | <code>MSH-12</code> | <code>2.4</code> |

All headers are String types. If a header value is missing, its value is null.

OPTIONS

The HL7 Data Format supports the following options:

| Option | Default | Description |
|-----------------------|--|--|
| <code>validate</code> | <code>true</code> | Whether the HAPI Parser should validate using the default validation rules. Camel 2.11: better use the <code>parser</code> option and initialize the parser with the desired HAPI <code>ValidationContext</code> |
| <code>parser</code> | <code>ca.uhn.hl7v2.parser.GenericParser</code> | *Camel 2.11:* To use a custom parser. Must be of type <code>ca.uhn.hl7v2.parser.Parser</code> . Note that <code>GenericParser</code> also allows to parse XML-encoded HL7v2 messages. |

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 since Version 0.6 has been 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 a 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>1.2</version>
  <!-- use the same version as your hapi-base version -->
</dependency>
<dependency>
  <groupId>ca.uhn.hapi</groupId>
  <artifactId>hapi-structures-v25</artifactId>
  <version>1.2</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>1.2</version>
</dependency>
```

TERSER LANGUAGE (CAMEL 2.11)

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");
```

HL7 VALIDATION PREDICATE (CAMEL 2.11)

Often it is preferable to parse a HL7v2 message and validate it against a HAPI [ValidationContext](#) in a separate step afterwards.

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");

```

HL7 ACKNOWLEDGEMENT EXPRESSION (CAMEL 2.11)

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())

```

MORE SAMPLES

In the following example we send a HL7 request to a HL7 listener and retrieves a response. We use plain String types in this example:

```

String line1 =
"MSH|^~\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|1234|P
|2.4";
String line2 = "QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||";

StringBuilder in = new StringBuilder();
in.append(line1);
in.append("\n");

```

```
in.append(line2);
```

```
String out = (String)template.requestBody("mina2:tcp://127.0.0.1:8888?
sync=true&codec=#hl7codec", in.toString());
```

In the next sample, we want to route HL7 requests from our HL7 listener to our business logic. We have our business logic in a plain POJO that we have registered in the registry as `hl7service` = for instance using Spring and letting the bean id = `hl7service`.

Our business logic is a plain POJO only using the HAPI library so we have these operations defined:

```
public class MyHL7BusinessLogic {

    // This is a plain POJO that has NO imports whatsoever on Apache
    Camel.
    // its a plain POJO only importing the HAPI library so we can much
    easier work with the HL7 format.

    public Message handleA19(Message msg) throws Exception {
        // here you can have your business logic for A19 messages
        assertTrue(msg instanceof QRY_A19);
        // just return the same dummy response
        return createADR19Message();
    }

    public Message handleA01(Message msg) throws Exception {
        // here you can have your business logic for A01 messages
        assertTrue(msg instanceof ADT_A01);
        // just return the same dummy response
        return createADT01Message();
    }
}
```

Then we set up the Camel routes using the `RouteBuilder` as follows:

```
DataFormat hl7 = new HL7DataFormat();
// we setup or HL7 listener on port 8888 (using the hl7codec) and in sync
mode so we can return a response
from("mina2:tcp://127.0.0.1:8888?sync=true&codec=#hl7codec")
    // we use the HL7 data format to unmarshal from HL7 stream to the HAPI
    Message model
    // this ensures that the camel message has been enriched with hl7
    specific headers to
    // make the routing much easier (see below)
    .unmarshal(hl7)
    // using choice as the content base router
    .choice()
        // where we choose that A19 queries invoke the handleA19 method on
```



```

our hl7service bean
    .when(header("CamelHL7TriggerEvent").isEqualTo("A19"))
        .beanRef("hl7service", "handleA19")
        .to("mock:a19")
    // and A01 should invoke the handleA01 method on our hl7service
bean
    .when(header("CamelHL7TriggerEvent").isEqualTo("A01")).to("mock:a01")
        .beanRef("hl7service", "handleA01")
        .to("mock:a19")
    // other types should go to mock:unknown
    .otherwise()
        .to("mock:unknown")
    // end choice block
    .end()
    // marshal response back
    .marshal(hl7);

```

Notice that we use the HL7 DataFormat to enrich our Camel Message with the MSH fields preconfigured on the Camel Message. This lets us much more easily define our routes using the fluent builders. If we do not use the HL7 DataFormat, then we do not gain these headers and we must resort to a different technique for computing the MSH trigger event (= what kind of HL7 message it is). This is a big advantage of the HL7 DataFormat over the plain HL7 type converters.

SAMPLE USING PLAIN STRING OBJECTS

In this sample we use plain String objects as the data format, that we send, process and receive. As the sample is part of a unit test, there is some code for assertions, but you should be able to understand what happens. First we send the plain string, Hello World, to the HL7MLLPCodec and receive the response as a plain string, Bye World.

```

MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedBodiesReceived("Bye World");

// send plain hello world as String
Object out = template.requestBody("mina2:tcp://127.0.0.1:8888?
sync=true&codec=#hl7codec", "Hello World");

assertMockEndpointsSatisfied();

// and the response is also just plain String
assertEquals("Bye World", out);

```

Here we process the incoming data as plain String and send the response also as plain String:

```
from("mina2:tcp://127.0.0.1:8888?sync=true&codec=#hl7codec")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // use plain String as message format
            String body = exchange.getIn().getBody(String.class);
            assertEquals("Hello World", body);

            // return the response as plain string
            exchange.getOut().setBody("Bye World");
        }
    })
    .to("mock:result");
```

CHAPTER 49. HTTP

HTTP COMPONENT

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>
```

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](#).

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

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

Spring DSL

```
<from uri="direct:start"/>
<setHeader
headerName="CamelHttpMethod">
  <constant>POST</constant>
</setHeader>
<to uri="http://www.google.com"/>
<to uri="mock:results"/>
```

HTTPENDPOINT OPTIONS

| Name | Default Value | Description |
|--------------------------------------|--------------------|---|
| <code>throwExceptionOnFailure</code> | <code>true</code> | 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. |
| <code>bridgeEndpoint</code> | <code>false</code> | If the option is true , <code>HttpProducer</code> will ignore the <code>Exchange.HTTP_URI</code> header, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the <code>HttpProducer</code> send all the fault response back. Camel 2.3: If the option is true, <code>HttpProducer</code> and <code>CamelServlet</code> will skip the gzip processing if the content-encoding is "gzip". |
| <code>disableStreamCache</code> | <code>false</code> | <code>DefaultHttpBinding</code> will copy the request input stream into a stream cache and put it into message body if this option is false to support read it twice, otherwise <code>DefaultHttpBinding</code> will set the request input stream direct into the message body. |
| <code>httpBindingRef</code> | <code>null</code> | Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component.http.HttpBinding</code> in the Registry . Use the <code>httpBinding</code> option instead. |
| <code>httpBinding</code> | <code>null</code> | Reference to a <code>org.apache.camel.component.http.HttpBinding</code> in the Registry . |

| | | |
|--------------------------------------|--------------------|---|
| <code>httpClientConfigurerRef</code> | <code>null</code> | Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry . Use the <code>httpClientConfigurer</code> option instead. |
| <code>httpClientConfigurer</code> | <code>null</code> | Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry . |
| <code>httpClient.XXX</code> | <code>null</code> | Setting options on the HttpClientParams . For instance <code>httpClient.soTimeout=5000</code> will set the <code>SO_TIMEOUT</code> to 5 seconds. |
| <code>clientConnectionManager</code> | <code>null</code> | To use a custom <code>org.apache.http.conn.ClientConnectionManager</code> . |
| <code>transferException</code> | <code>false</code> | Camel 2.6: 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 <code>application/x-java-serialized-object</code> 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 <code>HttpOperationFailedException</code> . The caused exception is required to be serialized. |
| <code>headerFilterStrategy</code> | <code>null</code> | Camel 2.11: Reference to a instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> in the Registry . It will be used to apply the custom <code>headerFilterStrategy</code> on the new create <code>HttpEndpoint</code> . |

| | | |
|-------------------------|-------------------|---|
| <code>urlRewrite</code> | <code>null</code> | Camel 2.11:Producer only Refers to a custom <code>org.apache.camel.component.http.UrlRewrite</code> which allows you to rewrite urls when you bridge/proxy endpoints. See more details at UrlRewrite and How to use Camel as a HTTP proxy between a client and server . |
|-------------------------|-------------------|---|

AUTHENTICATION AND PROXY

The following authentication options can also be set on the `HttpEndpoint`:

| Name | Default Value | Description |
|---------------------------------|-------------------|--|
| <code>authMethod</code> | <code>null</code> | Authentication method, either as Basic , Digest or NTLM . |
| <code>authMethodPriority</code> | <code>null</code> | Priority of authentication methods. Is a list separated with comma. For example: Basic, Digest to exclude NTLM . |
| <code>authUsername</code> | <code>null</code> | Username for authentication |
| <code>authPassword</code> | <code>null</code> | Password for authentication |
| <code>authDomain</code> | <code>null</code> | Domain for NTML authentication |
| <code>authHost</code> | <code>null</code> | Optional host for NTML authentication |
| <code>proxyHost</code> | <code>null</code> | The proxy host name |
| <code>proxyPort</code> | <code>null</code> | The proxy port number |
| <code>proxyAuthMethod</code> | <code>null</code> | Authentication method for proxy, either as Basic , Digest or NTLM . |

| | | |
|--------------------------------|-------------------|---|
| <code>proxyAuthUsername</code> | <code>null</code> | Username for proxy authentication |
| <code>proxyAuthPassword</code> | <code>null</code> | Password for proxy authentication |
| <code>proxyAuthDomain</code> | <code>null</code> | Domain for proxy NTLM authentication |
| <code>proxyAuthHost</code> | <code>null</code> | Optional host for proxy NTLM authentication |

When using authentication you must provide the choice of method for the `authMethod` or `authProxyMethod` options. You can configure the proxy and authentication details on either the `HttpComponent` or the `HttpEndpoint`. Values provided on the `HttpEndpoint` will take precedence over `HttpComponent`. Its most likely best to configure this on the `HttpComponent` which allows you to do this once.

The [HTTP](#) component uses convention over configuration which means that if you have not explicit set a `authMethodPriority` then it will fallback and use the `select(ed)` `authMethod` as priority as well. So if you use `authMethod.Basic` then the `authMethodPriority` will be `Basic` only.

HTTPCOMPONENT OPTIONS

| Name | Default Value | Description |
|------------------------------------|-------------------|---|
| <code>httpBinding</code> | <code>null</code> | To use a custom <code>org.apache.camel.component.http.HttpBinding</code> . |
| <code>httpClientConfigurer</code> | <code>null</code> | To use a custom <code>org.apache.camel.component.http.HttpClientConfigurer</code> . |
| <code>httpConnectionManager</code> | <code>null</code> | To use a custom <code>org.apache.commons.httpclient.HttpConnectionManager</code> . |

| | | |
|--------------------------------|-------------------|--|
| <code>httpConfiguration</code> | <code>null</code> | To use a custom <code>org.apache.camel.component.http.HttpConfiguration</code> |
|--------------------------------|-------------------|--|

MESSAGE HEADERS

| Name | Type | Description |
|---|---------------------|---|
| <code>Exchange.HTTP_URI</code> | <code>String</code> | URI to call. Will override existing URI set directly on the endpoint. |
| <code>Exchange.HTTP_METHOD</code> | <code>String</code> | HTTP Method / Verb to use (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) |
| <code>Exchange.HTTP_PATH</code> | <code>String</code> | Request URI's path, the header will be used to build the request URI with the <code>HTTP_URI</code> . Camel 2.3.0: If the path is start with "/", http producer will try to find the relative path based on the <code>Exchange.HTTP_BASE_URI</code> header or the <code>exchange.getFromEndpoint().getEndpointUri()</code> ; |
| <code>Exchange.HTTP_QUERY</code> | <code>String</code> | URI parameters. Will override existing URI parameters set directly on the endpoint. |
| <code>Exchange.HTTP_RESPONSE_CODE</code> | <code>int</code> | The HTTP response code from the external server. Is 200 for OK. |
| <code>Exchange.HTTP_CHARACTER_ENCODING</code> | <code>String</code> | Character encoding. |
| <code>Exchange.CONTENT_TYPE</code> | <code>String</code> | The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as <code>text/html</code> . |

| | | |
|---|---------------------|---|
| <code>Exchange.CONTENT_ENCODING</code> | String | The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as gzip . |
| <code>Exchange.HTTP_SERVLET_REQUEST</code> | HttpServletRequest | The HttpServletRequest object. |
| <code>Exchange.HTTP_SERVLET_RESPONSE</code> | HttpServletResponse | The HttpServletResponse object. |
| <code>Exchange.HTTP_PROTOCOL_VERSION</code> | 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, <code>HttpProducer</code> 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.

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.

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.

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

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.

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(HttpServletRequestResponse.class);
```

USING CLIENT TIMEOUT - SO_TIMEOUT

See the unit test in [this link](#)

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.

USING PROXY SETTINGS OUTSIDE OF URI

| Java DSL | Spring DSL |
|--|--|
| <pre>context.getProperties().put("http.proxyHost", "172.168.18.9"); context.getProperties().put("http.proxyPort", "8080");</pre> | <pre><camelContext> <properties> <property key="http.proxyHost" value="172.168.18.9"/> <property key="http.proxyPort" value="8080"/> </properties> </camelContext></pre> |

Options on Endpoint will override options on the context.

CONFIGURING CHARSET

If you are using POST to send data you can configure the charset

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

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");
```

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 `HttpProducer.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(HttpProducer.HTTP_RESPONSE_CODE,
Integer.class);
```

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.

```
// We set throwExceptionOnFailure to false to let Camel return any
response from the remove HTTP server without thrown
// HttpOperationException in case of failures.
// This allows us to handle all responses in the aggregation strategy
where we can check the HTTP response code
// and decide what to do. As this is based on an unit test we assert the
code is 404
from("direct:start").enrich("http://localhost:{{port}}/myserver?
throwExceptionOnFailure=false&user=Camel", new AggregationStrategy() {
    public Exchange aggregate(Exchange original, Exchange resource) {
```

```

        // get the response code
        Integer code =
resource.getIn().getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
        assertEquals(404, code.intValue());
        return resource;
    }
}).to("mock:result");

// this is our jetty server where we simulate the 404
from("jetty://http://localhost:{{port}}/myserver")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            exchange.getOut().setBody("Page not found");
            exchange.getOut().setHeader(Exchange.HTTP_RESPONSE_CODE,
404);
        }
    });

```

DISABLING COOKIES

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option: `httpClient.cookiePolicy=ignoreCookies`

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.

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.

```

<bean id="http" class="org.apache.camel.component.http.HttpComponent">
    <property name="camelContext" ref="camel"/>

```

```

    <property name="httpConnectionManager" ref="myHttpConnectionManager"/>
</bean>

<bean id="myHttpConnectionManager"
class="org.apache.commons.httpclient.MultiThreadedHttpConnectionManager">
    <property name="params" ref="myHttpConnectionManagerParams"/>
</bean>

<bean id="myHttpConnectionManagerParams"
class="org.apache.commons.httpclient.params.HttpConnectionManagerParams">
    <property name="defaultMaxConnectionsPerHost" value="5"/>
</bean>

```

And then we can just use it as we normally do in our routes:

```

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"
trace="true">
    <route>
        <from uri="direct:start"/>
        <to uri="http://www.google.com"/>
        <to uri="mock:result"/>
    </route>
</camelContext>

```

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`

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.

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:

```
HttpClientComponent httpClient = getContext().getComponent("http",
HttpClientComponent.class);
httpClient.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.

- [Jetty](#)

CHAPTER 50. HTTP4

HTTP4 COMPONENT

Available as of Camel 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 HttpClient 4.x while camel-http uses HttpClient 3.x.

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.

HTTPCOMPONENT OPTIONS

| Name | Default Value | Description |
|--------------------------------------|-------------------|---|
| <code>maxTotalConnections</code> | <code>200</code> | The maximum number of connections. |
| <code>connectionsPerRoute</code> | <code>20</code> | The maximum number of connections per route. |
| <code>cookieStore</code> | <code>null</code> | Camel 2.11.2/2.12.0: To use a custom <code>org.apache.http.client.CookieStore</code> . By default the <code>org.apache.http.impl.client.BasicCookieStore</code> is used which is an in-memory only cookie store. Notice if <code>bridgeEndpoint=true</code> then the cookie store is forced to be a noop cookie store as cookies shouldn't be stored as we are just bridging (eg acting as a proxy). |
| <code>httpClientConfigurer</code> | <code>null</code> | Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry . |
| <code>clientConnectionManager</code> | <code>null</code> | To use a custom <code>org.apache.http.conn.ClientConnectionManager</code> . |
| <code>httpBinding</code> | <code>null</code> | To use a custom <code>org.apache.camel.component.http.HttpBinding</code> . |

| | | |
|-----------------------------------|--|---|
| <code>httpContext</code> | <code>null</code> | Camel 2.9.2: To use a custom <code>org.apache.http.protocol.HttpContext</code> when executing requests. |
| <code>sslContextParameters</code> | <code>null</code> | Camel 2.8: To use a custom <code>org.apache.camel.util.jsse.SSLContextParameters</code> . See Using the JSSE Configuration Utility . |
| <code>x509HostnameVerifier</code> | <code>BrowserCompatHostnameVerifier</code> | Camel 2.7: You can refer to a different <code>org.apache.http.conn.ssl.X509HostnameVerifier</code> instance in the Registry such as <code>org.apache.http.conn.ssl.StrictHostnameVerifier</code> or <code>org.apache.http.conn.ssl.AllowAllHostnameVerifier</code> . |
| <code>connectionTimeToLive</code> | <code>-1</code> | Camel 2.11.0: The time for connection to live, the time unit is millisecond, the default value is always keep alive. |

HTTPEndpoint OPTIONS

| Name | Default Value | Description |
|--------------------------------------|-------------------|---|
| <code>throwExceptionOnFailure</code> | <code>true</code> | Option to disable throwing the <code>HttpOperationFailedException</code> in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code. |

| | | |
|----------------------------|--------------|--|
| bridgeEndpoint | false | If true, <code>HttpProducer</code> will ignore the <code>Exchange.HTTP_URI</code> header, and use the endpoint's URI for request. You may also set the <code>throwExcpetionOnFailure</code> to be false to let the <code>HttpProducer</code> send all fault responses back. Also if set to true <code>HttpProducer</code> and <code>CamelServlet</code> will skip the gzip processing if the content-encoding is "gzip". |
| clearExpiredCookies | true | Camel 2.11.2/2.12.0: 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. |
| cookieStore | null | Camel 2.11.2/2.12.0: To use a custom <code>org.apache.http.client.CookieStore</code> . By default the <code>org.apache.http.impl.client.BasicCookieStore</code> is used which is an in-memory only cookie store. Notice if <code>bridgeEndpoint=true</code> then the cookie store is forced to be a noop cookie store as cookies shouldn't be stored as we are just bridging (eg acting as a proxy). |
| disableStreamCache | false | <code>DefaultHttpBinding</code> will copy the request input stream into a stream cache and put it into the message body if this option is false to support multiple reads, otherwise <code>DefaultHttpBinding</code> will set the request input stream directly in the message body. |

| | | |
|--------------------------------------|-------------------|--|
| <code>headerFilterStrategy</code> | <code>null</code> | Camel 2.10.4: Reference to a instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> in the Registry . It will be used to apply the custom <code>headerFilterStrategy</code> on the new create <code>HttpEndpoint</code> . |
| <code>httpBindingRef</code> | <code>null</code> | Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component.http.HttpBinding</code> in the Registry . Use the <code>httpBinding</code> option instead. |
| <code>httpBinding</code> | <code>null</code> | To use a custom <code>org.apache.camel.component.http.HttpBinding</code> . |
| <code>httpClientConfigurerRef</code> | <code>null</code> | Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry . Use the <code>httpClientConfigurer</code> option instead. |
| <code>httpClientConfigurer</code> | <code>null</code> | Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry . |
| <code>httpContextRef</code> | <code>null</code> | Deprecated and will be removed in Camel 3.0:Camel 2.9.2: Reference to a custom <code>org.apache.http.protocol.HttpContext</code> in the Registry . Use the <code>httpContext</code> option instead. |
| <code>httpContext</code> | <code>null</code> | Camel 2.9.2: To use a custom <code>org.apache.http.protocol.HttpContext</code> when executing requests. |

| | | |
|--------------------------------------|--------------------|--|
| <code>httpClient.XXX</code> | <code>null</code> | Setting options on the BasicHttpParams . For instance <code>httpClient.soTimeout=5000</code> will set the <code>SO_TIMEOUT</code> to 5 seconds. Look on the setter methods of the following parameter beans for a complete reference: AuthParamBean , ClientParamBean , ConnConnectionParamBean , ConnRouteParamBean , CookieSpecParamBean , HttpConnectionParamBean and HttpProtocolParamBean |
| <code>clientConnectionManager</code> | <code>null</code> | To use a custom <code>org.apache.http.conn.ClientConnectionManager</code> . |
| <code>transferException</code> | <code>false</code> | If enabled and an Exchange failed processing on the consumer side, and if the caused <code>Exception</code> was send back serialized in the response as a <code>application/x-java-serialized-object</code> 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 <code>HttpOperationFailedException</code> . The caused exception is required to be serialized. |

| | | |
|--------------------------------|--------------------------------------|---|
| sslContextParametersRef | null | <p>Deprecated and will be removed in Camel 3.0:Camel 2.8: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry. Important: Only one instance of <code>org.apache.camel.util.jsse.SSLContextParameters</code> is supported per <code>HttpComponent</code>. If you need to use 2 or more different instances, you need to define a new <code>HttpComponent</code> per instance you need. See further below for more details. See Using the JSSE Configuration Utility. Use the sslContextParameters option instead.</p> |
| sslContextParameters | null | <p>Camel 2.11.1: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry. Important: Only one instance of <code>org.apache.camel.util.jsse.SSLContextParameters</code> is supported per <code>HttpComponent</code>. If you need to use 2 or more different instances, you need to define a new <code>HttpComponent</code> per instance you need. See further below for more details. See Using the JSSE Configuration Utility.</p> |
| x509HostnameVerifier | BrowserCompatHostnameVerifier | <p>Camel 2.7: You can refer to a different <code>org.apache.http.conn.ssl.X509HostnameVerifier</code> instance in the Registry such as <code>org.apache.http.conn.ssl.StrictHostnameVerifier</code> or <code>org.apache.http.conn.ssl.AllowAllHostnameVerifier</code>.</p> |

| | | |
|-------------------------|-------------------|---|
| <code>urlRewrite</code> | <code>null</code> | Camel 2.11:Producer only Refers to a custom <code>org.apache.camel.component.http4.UrlRewrite</code> which allows you to rewrite urls when you bridge/proxy endpoints. See more details at UrlRewrite and How to use Camel as a HTTP proxy between a client and server. |
|-------------------------|-------------------|---|

The following authentication options can also be set on the `HttpEndpoint`:

SETTING BASIC AUTHENTICATION AND PROXY

Before Camel 2.8.0

| Name | Default Value | Description |
|----------------------------|-------------------|-------------------------------------|
| <code>username</code> | <code>null</code> | Username for authentication. |
| <code>password</code> | <code>null</code> | Password for authentication. |
| <code>domain</code> | <code>null</code> | The domain name for authentication. |
| <code>host</code> | <code>null</code> | The host name authentication. |
| <code>proxyHost</code> | <code>null</code> | The proxy host name |
| <code>proxyPort</code> | <code>null</code> | The proxy port number |
| <code>proxyUsername</code> | <code>null</code> | Username for proxy authentication |
| <code>proxyPassword</code> | <code>null</code> | Password for proxy authentication |
| <code>proxyDomain</code> | <code>null</code> | The proxy domain name |
| <code>proxyNtHost</code> | <code>null</code> | The proxy Nt host name |
| Name | Default Value | Description |

| | | |
|--------------------------|-------------|---|
| authUsername | null | Username for authentication |
| authPassword | null | Password for authentication |
| authDomain | null | The domain name for authentication |
| authHost | null | The host name authentication |
| proxyAuthHost | null | The proxy host name |
| proxyAuthPort | null | The proxy port number |
| proxyAuthScheme | null | The proxy scheme, will fallback and use the scheme from the endpoint if not configured. |
| proxyAuthUsername | null | Username for proxy authentication |
| proxyAuthPassword | null | Password for proxy authentication |
| proxyAuthDomain | null | The proxy domain name |
| proxyAuthNtHost | null | The proxy Nt host name |

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. |
| Exchange.HTTP_QUERY | String | URI parameters. Will override existing URI parameters set directly on the endpoint. |

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

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.

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.

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

CALLING USING GET OR POST

The following algorithm is used to determine whether the 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.

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);
```

CONFIGURING URI TO CALL

You can set the HTTP producer's URI directly from 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`. Where `Constants` is the class, `org.apache.camel.component.http4.Constants`. If the `http4` endpoint is working in bridge mode, it will ignore the message header of `Exchange.HTTP_URI`.

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");
```

HOW TO SET THE HTTP METHOD (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) TO THE HTTP PRODUCER

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>
```

USING CLIENT TIMEOUT - SO_TIMEOUT

See the [HttpS0TimeoutTest](#) unit test.

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.

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.

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");
```

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");
```

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);
```

URI PARAMETERS FROM THE MESSAGE

```
Map headers = new HashMap();
headers.put(HttpProducer.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.

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 `HttpProducer.HTTP_RESPONSE_CODE`.

```
Exchange exchange = template.send("http4://www.google.com/search", new
Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getIn().setHeader(HttpProducer.QUERY,
constant("hl=en&q=activemq"));
    }
});
Message out = exchange.getOut();
int responseCode = out.getHeader(HttpProducer.HTTP_RESPONSE_CODE,
Integer.class);
```


DISABLING COOKIES

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option: `httpClient.cookiePolicy=ignoreCookies`

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.

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("http4",
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/?
sslContextParametersRef=sslContextParameters"/>...

```

CONFIGURING APACHE HTTP CLIENT DIRECTLY

Basically camel-http4 component is built on the top of [Apache HTTP client](#). 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 `HttpContext`s:

```
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 51. IBATIS

IBATIS

The `ibatis:` component allows you to query, poll, insert, update and delete data in a relational database using [Apache iBATIS](#).

**PREFER MYBATIS**

The Apache `iBatis` project is no longer active. The project is moved outside Apache and is now known as the `MyBatis` project. Therefore we encourage users to use [MyBatis](#) instead. This `camel-ibatis` component will be removed in Camel 3.0.

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

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

URI FORMAT

```
ibatis:statementName[?options]
```

Where `statementName` is the name in the `iBATIS` XML configuration 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 `iBatis SqlMapConfig` file from the root of the classpath and expected named as `SqlMapConfig.xml`. It uses Spring resource loading so you can define it using `classpath`, `file` or `http` as prefix

to load resources with those schemes. In Camel 2.2 you can configure this on the `iBatisComponent` with the `setSqlMapConfig(String)` method.

OPTIONS

| Option | Type | Default | Description |
|---|---------------|---------|---|
| <code>consumer.onConsume</code> | String | null | Statements to run after consuming. Can be used, for example, to update rows after they have been consumed and processed in Apache Camel. See sample later. Multiple statements can be separated with comma. |
| <code>consumer.useIterator</code> | boolean | true | If true each row returned when polling will be processed individually. If false the entire List of data is set as the IN body. |
| <code>consumer.routeEmptyResultSet</code> | boolean | false | Apache Camel 2.0: Sets whether empty result set should be routed or not. By default, empty result sets are not routed. |
| <code>statementType</code> | StatementType | null | Apache Camel 1.6.1/2.0: Mandatory to specify for <code>IbatisProducer</code> to control which <code>iBatis SqlMapClient</code> method to invoke. The enum values are: QueryForObject, QueryForList, Insert, Update, Delete. |

| | | | |
|---------------------------------|---------------------|--|---|
| <code>maxMessagesPerPoll</code> | <code>int</code> | <code>0</code> | Apache Camel 2.0: An integer 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. |
| <code>isolation</code> | <code>String</code> | <code>TRANSACTION_REPEATABLE_READ</code> | *Camel 2.9:* A String the defines the transaction isolation level of the will be used. Allowed values are TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE |
| <code>isolation</code> | <code>String</code> | <code>TRANSACTION_REPEATABLE_READ</code> | *Camel 2.9:* A String the defines the transaction isolation level of the will be used. Allowed values are TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE |

MESSAGE HEADERS

Apache Camel will populate the result message, either IN or OUT with a header with the operationName used:

| Header | Type | Description |
|---------------------------------------|--------|--|
| <code>CamelIBatisStatementName</code> | String | Apache Camel 2.0: The statementName used (for example: <code>insertAccount</code>). |
| <code>CamelIBatisResult</code> | Object | Apache Camel 1.6.2/2.0: The response returned from iBatis in any of the operations. For instance an INSERT could return the auto-generated key, or number of rows etc. |

MESSAGE BODY

Apache Camel 1.6.2/2.0: The response from iBatis will only be set as body if it's a SELECT statement. That means, for example, for INSERT statements Apache Camel will not replace the body. This allows you to continue routing and keep the original body. The response from iBatis is always stored in the header with the key `CamelIBatisResult`.

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("ibatis:insertAccount?statementType=Insert");
```

Notice we have to specify the `statementType`, as we need to instruct Apache Camel which `SqlMapClient` operation to invoke.

Where `insertAccount` is the iBatis ID in the SQL map file:

```
<!-- Insert example, using the Account parameter class -->
<insert id="insertAccount" parameterClass="Account">
  insert into ACCOUNT (
    ACC_ID,
    ACC_FIRST_NAME,
    ACC_LAST_NAME,
    ACC_EMAIL
```



```

    )
    values (
        #id#, #firstName#, #lastName#, #emailAddress#
    )
</insert>

```

USING STATEMENTTYPE FOR BETTER CONTROL OF IBATIS

Available as of Apache Camel 1.6.1/2.0 When routing to an iBatis endpoint you want more fine grained control so you can control whether the SQL statement to be executed is a SELEECT, UPDATE, DELETE or INSERT etc. This is now possible in Apache Camel 1.6.1/2.0. So for instance if we want to route to an iBatis endpoint in which the IN body contains parameters to a SELECT statement we can do:

```

from("direct:start")
    .to("ibatis:selectAccountById?statementType=QueryForObject")
    .to("mock:result");

```

In the code above we can invoke the iBatis 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 `QueryForList`:

```

from("direct:start")
    .to("ibatis:selectAllAccounts?statementType=QueryForList")
    .to("mock:result");

```

And the same for UPDATE, where we can send an Account object as IN body to iBatis:

```

from("direct:start")
    .to("ibatis:updateAccount?statementType=Update")
    .to("mock:result");

```

SCHEDULED POLLING EXAMPLE

Since this component does not support scheduled polling, you need to 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("ibatis:selectAllAccounts?statementType=QueryForList").to("activemq:queue:allAccounts");
```

And the iBatis SQL map file used:

```
<!-- Select with no parameters using the result map for Account class. ->
<select id="selectAllAccounts" resultMap="AccountResult">
  select * from ACCOUNT
</select>
```

USING ONCONSUME

This component supports executing statements after data have been consumed and processed by Apache Camel. This allows you to do post updates in the database. Notice all statements must be UPDATE statements. Apache Camel supports executing multiple statements whose name should be separated by comma.

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.

```
from("ibatis:selectUnprocessedAccounts?consumer.onConsume=consumeAccount").to("mock:results");
```

And the statements in the sqlmap file:

```
<select id="selectUnprocessedAccounts" resultMap="AccountResult">
  select * from ACCOUNT where PROCESSED = false
</select>
```

```
<update id="consumeAccount" parameterClass="Account">
  update ACCOUNT set PROCESSED = true where ACC_ID = #id#
</update>
```

CHAPTER 52. IRC

IRC COMPONENT

The `irc` component implements an [IRC](#) (Internet Relay Chat) transport.

URI FORMAT

```
irc:nick@host[:port]/#room[?options]
```

In Apache Camel 2.0, you can also use the following format:

```
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&...`

OPTIONS

| Name | Description | Example | Default Value |
|-----------------------|---|---|-------------------|
| <code>channels</code> | New in 2.0, comma separated list of IRC channels to join. | <code>channels=#channel1,#channel2</code> | <code>null</code> |
| <code>nickname</code> | The nickname used in chat. | <code>irc:MyNick@irc.server.org#channel1</code> or <code>irc:irc.server.org#channel?nickname=MyUser</code> | <code>null</code> |
| <code>username</code> | The IRC server user name. | <code>irc:MyUser@irc.server.org#channel1</code> or <code>irc:irc.server.org#channel?username=MyUser</code> | Same as nickname. |

| | | | |
|---------------------|---|--|---|
| password | The IRC server password. | password=somepassword | <i>None</i> |
| realname | The IRC user's actual name. | realname=MyName | <i>None</i> |
| colors | Whether or not the server supports color codes. | true, false | true |
| onReply | Whether or not to handle general responses to commands or informational messages. | true, false | false |
| onNick | Handle nickname change events. | true, false | true |
| onQuit | Handle user quit events. | true, false | true |
| onJoin | Handle user join events. | true, false | true |
| onKick | Handle kick events. | true, false | true |
| onMode | Handle mode change events. | true, false | true |
| onPart | Handle user part events. | true, false | true |
| onTopic | Handle topic change events. | true, false | true |
| onPrivmsg | Handle message events. | true, false | true |
| trustManager | New in 2.0, the trust manager used to verify the SSL server's certificate. | trustManager=#referenceToTrustManagerBean | The default trust manager, which accepts <i>all</i> certificates, will be used. |

| | | | |
|----------------------|---|---|------|
| keys | Camel 2.2: Comma separated list of IRC channel keys. Important to be listed in same order as channels. When joining multiple channels with only some needing keys just insert an empty value for that channel. | irc:MyNick@irc.server.org/#channel?keys=chankey | null |
| sslContextParameters | *Camel 2.9:* 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 this setting overrides the <code>trustManager</code> option. | \#mySslContextParameters | null |

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"/>...

```

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
```

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
```

CHAPTER 53. JASYPT

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 automatic be decrypted on-the-fly by Camel. This ensures that human eyes can't easily spot sensitive information such as usernames and passwords.

Maven users will need to add the following dependency to their 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>
```

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)

```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7K003Ww==)
```

TOOLING DEPENDENCIES FOR CAMEL 2.5 AND 2.6

The tooling requires the following JARs in the classpath, which has been enlisted in the MANIFEST.MF file of camel-jasypt with optional/ as prefix.

Hence why the `java` cmd above can pickup the needed JARs from the Apache Distribution in the optional directory.

```
jasypt-1.6.jar commons-lang-2.4.jar commons-codec-1.4.jar icu4j-4.0.1.jar
```



JAVA 1.5 USERS

The `icu4j-4.0.1.jar` is only needed when running on JDK 1.5.

This JAR is not distributed by Apache Camel and you have to download it manually and copy it to the `lib/optional` directory of the Camel distribution. You can download it from [Apache Central Maven repo](#).

TOOLING DEPENDENCIES FOR CAMEL 2.7 OR BETTER

Jasypt 1.7 onwards is now fully standalone, so no additional JARs are needed.

URI OPTIONS

The options below are exclusive for the [Jasypt](#) component.

| Name | Default Value | Type | Description |
|------------------------|-------------------|---------------------|--|
| <code>password</code> | <code>null</code> | <code>String</code> | Specifies the master password to use for decrypting. This option is mandatory. See below for more details. |
| <code>algorithm</code> | <code>null</code> | <code>String</code> | Name of an optional algorithm to use. |

PROTECTING THE MASTER PASSWORD

The master password used by [Jasypt](#) must be provided, so its capable of decrypting the values. However having this master password out in the opening may not be an ideal solution. Therefore you could for example provided 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`.

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:

```
// create the jasypt properties parser
JasyptPropertiesParser jasypt = new JasyptPropertiesParser();
// and set the master password
jasypt.setPassword("secret");

// create the properties component
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation("classpath:org/apache/camel/component/jasypt/myproperties.p
roperties");
// and use the jasypt properties parser so we can decrypt values
pc.setPropertiesParser(jasypt);

// add properties component to camel context
context.addComponent("properties", pc);
```

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)`

```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7K003Ww==)
```

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` have 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>
```

SEE ALSO

- [Security](#)
- [Properties](#)
- [Encrypted passwords in ActiveMQ](#) - ActiveMQ has a similar feature as this camel-jasypt component

CHAPTER 54. JAVASPACE

JAVASPACE COMPONENT

The `javaspace` component is a transport for working with any JavaSpace compliant implementation and this component has been tested with both the [Blitz implementation](#) and the [GigaSpace implementation](#). This component can be used for sending and receiving any object inheriting from the `Jini net.jini.core.entry.Entry` class. It is also possible to pass the bean ID of a template that can be used for reading/taking the entries from the space. This component can be used for sending/receiving any serializable object acting as a sort of generic transport. The JavaSpace component contains a special optimization for dealing with the `BeanExchange`. It can be used to invoke a POJO remotely, using a JavaSpace as a transport. This latter feature can provide a simple implementation of the master/worker pattern, where a POJO provides the business logic for the worker. Look at the test cases for examples of various use cases for this component.

URI FORMAT

```
javaspace:jini://host[?options]
```

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

OPTIONS

| Name | Default Value | Description |
|-----------------------------------|-----------------------------|---|
| <code>spaceName</code> | <code>null</code> | Specifies the JavaSpace name. |
| <code>verb</code> | <code>take</code> | Specifies the verb for getting JavaSpace entries. The values can be: take or read . |
| <code>transactional</code> | <code>false</code> | If true , sending and receiving entries is performed within a transaction. |
| <code>transactionalTimeout</code> | <code>Long.MAX_VALUE</code> | Specifies the transaction timeout. |
| <code>concurrentConsumers</code> | <code>1</code> | Specifies the number of concurrent consumers getting entries from the JavaSpace. |

| | | |
|------------|------|---|
| templateId | null | If present, this option specifies the Spring bean ID of the template to use for reading/taking entries. |
|------------|------|---|

SENDING AND RECEIVING ENTRIES

```
//Sending route
from("direct:input").to("javaspace:jini://localhost?spaceName=mySpace");

//Receiving Route
from("javaspace:jini://localhost?
spaceName=mySpace&templateId=template&verb=take&concurrentConsumers=1")
```

In this case the payload can be any object that inherits from the Jini Entry type.

SENDING AND RECEIVING SERIALIZABLE OBJECTS

Using the preceding routes, it is also possible to send and receive any serializable object. The JavaSpace component detects that the payload is not a Jini Entry and then it automatically wraps the payload with a Camel Jini Entry. In this way, a JavaSpace can be used as a generic transport mechanism.

USING JAVASPACE AS A REMOTE INVOCATION TRANSPORT

The JavaSpace component has been tailored to work in combination with the Camel bean component. It is therefore possible to call a remote POJO using JavaSpace as the transport:

```
from("direct:input").to("javaspace:jini://localhost?spaceName=mySpace");
//Client side

from("javaspace:jini://localhost?
concurrentConsumers=10&spaceName=mySpace").to("pojo:pojo"); //Server side
```

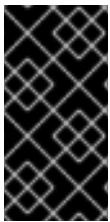
In the code there are two test cases showing how to use a POJO to realize the master/worker pattern. The idea is to use the POJO to provide the business logic and rely on Apache Camel for sending/receiving requests/replies with the proper correlation.

CHAPTER 55. JBI

55.1. JBI COMPONENT

Overview

The `jbi` component is implemented by the [ServiceMix Camel module](#) and provides integration with a JBI Normalized Message Router, such as the one provided by [Apache ServiceMix](#).

**IMPORTANT**

See below for information about how to use `StreamSource` types from [ServiceMix](#) in Camel.

The following code:

```
from("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint")
```

Automatically exposes a new endpoint to the bus, where the service QName is `{http://foo.bar.org}MyService` and the endpoint name is `MyEndpoint` (see [URI-format](#)).

When a JBI endpoint appears at the end of a route, for example:

```
to("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint")
```

The messages sent by this producer endpoint are sent to the already deployed JBI endpoint.

URI format

```
jbi:service:serviceNamespace[sep]serviceName[?options]  
jbi:endpoint:serviceNamespace[sep]serviceName[sep]endpointName[?options]  
jbi:name:endpointName[?options]
```


The separator that should be used in the endpoint URL is:

-

`/` (forward slash), if `serviceNameSpace` starts with `http://`, or

-

`:` (colon), if `serviceNameSpace` starts with `urn:foo:bar`.

For more details of valid JBI URIs see the [ServiceMix URI Guide](#).

Using the `jbi:service:` or `jbi:endpoint:` URI formats sets the service QName on the JBI endpoint to the one specified. Otherwise, the default Camel JBI Service QName is used, which is:

```
{http://activemq.apache.org/camel/schema/jbi}endpoint
```

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

Examples

```
jbi:service:http://foo.bar.org/MyService
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint
jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint
jbi:name:cheese
```

URI options

| Name | Default value | Description |
|-------------------------|---|--|
| mep | MEP of the Camel Exchange | Allows users to override the MEP set on the Exchange object. Valid values for this option are in-only , in-out , robust-in-out and in-optional-out . |
| operation | Value of the jbi.operation header property | Specifies the JBI operation for the MessageExchange . If no value is supplied, the JBI binding will use the value of the jbi.operation header property. |
| serialization | basic | Default value (basic) will check if headers are serializable by looking at the type, setting this option to strict will detect objects that can not be serialized although they implement the Serializable interface. Set to nocheck to disable this check altogether, note that this should only be used for in-memory transports like SEDAFlow, otherwise you can expect to get NotSerializableException thrown at runtime. |
| convertException | false | false : send any exceptions thrown from the Camel route back unmodified true : convert all exceptions to a JBI FaultException (can be used to avoid non-serializable exceptions or to implement generic error handling) |

Examples

```

jbi:service:http://foo.bar.org/MyService?mep=in-out      (override the
MEP, use InOut JBI MessageExchanges)
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint?mep=in    (override the
MEP, use InOnly JBI MessageExchanges)
jbi:endpoint:urn:foo:bar:MyService:MyEndpoint?operation=

```

```
{http://www.mycompany.org}AddNumbers
(override the operation for the JBI Exchange to
{http://www.mycompany.org}AddNumbers)
```

Using Stream bodies

If you are using a stream type as the message body, you should be aware that a stream is only capable of being read once. So if you enable `DEBUG` logging, the body is usually logged and thus read. To deal with this, Camel has a `streamCaching` option that can cache the stream, enabling you to read it multiple times.

```
from("jbi:endpoint:http://foo.bar.org/MyService/MyEndpoint").streamCaching
().to("xslt:transform.xsl", "bean:doSomething");
```

The stream caching is default enabled, so it is not necessary to set the `streamCaching()` option. We store big input streams (by default, over 64K) in a temp file using `CachedOutputStream`. When you close the input stream, the temp file will be deleted.

Creating a JBI Service Unit

If you have some Camel routes that you want to deploy inside JBI as a Service Unit, you can use the [JBI Service Unit Archetype](#) to create a new Maven project for the Service Unit.

If you have an existing Maven project that you need to convert into a JBI Service Unit, you may want to consult [ServiceMix Maven JBI Plugins](#) for further help. The key steps are as follows:

-

Create a Spring XML file at `src/main/resources/camel-context.xml` to bootstrap your routes inside the JBI Service Unit.

-

Change the POM file's packaging to `jbi-service-unit`.

Your `pom.xml` should look something like this to enable the `jbi-service-unit` packaging:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>myGroupId</groupId>
    <artifactId>myArtifactId</artifactId>
    <packaging>jbi-service-unit</packaging>
    <version>1.0-SNAPSHOT</version>

    <name>A Camel based JBI Service Unit</name>

    <url>http://www.myorganization.org</url>

    <properties>
        <camel-version>x.x.x</camel-version>
        <servicemix-version>3.3</servicemix-version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.servicemix</groupId>
            <artifactId>servicemix-camel</artifactId>
            <version>${servicemix-version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.servicemix</groupId>
            <artifactId>servicemix-core</artifactId>
            <version>${servicemix-version}</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <defaultGoal>install</defaultGoal>

        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
```

```

    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>

  <!-- creates the JBI deployment unit -->
  <plugin>
    <groupId>org.apache.servicemix.tooling</groupId>
    <artifactId>jbi-maven-plugin</artifactId>
    <version>${servicemix-version}</version>
    <extensions>>true</extensions>
  </plugin>
</plugins>
</build>
</project>

```

-

[ServiceMix Camel module](#)

-

[Using Camel with ServiceMix](#)

-

[Cookbook on using Camel with ServiceMix](#)

55.2. JBI SERVICE UNIT ARCHETYPE

JBI Service Unit Archetype

If you want to spin up your own project to use Camel to perform some

smart routing inside your JBI based ESB you can use the Maven archetype to get up to speed quickly.

Just type the following into a console...

```
mvn archetype:create \
  -DarchetypeGroupId=org.apache.camel \
  -DarchetypeArtifactId=camel-jbi-service-unit \
  -DarchetypeVersion=1.0-SNAPSHOT \
  -DgroupId=myGroupId \
  -DartifactId=myArtifactId
```

This will create a maven project which can be run immediately via the [Camel Maven Plugin](#) as follows

```
cd myArtifactId
mvn install
```

The configuration file is in `src/main/resources/camel-context.xml`.

The routing rules lives at `src/main/java/myGroupId/MyRouteBuilder.java`

CHAPTER 56. JCLOUDS

JCLOUDS COMPONENT

Available as of Camel 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 specifies 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>
```

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.

URI FORMAT


```
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&...`

BLOBSTORE URI OPTIONS

| Name | Default Value | Description |
|------------------------|---------------|--|
| <code>operation</code> | PUT | *Producer Only*. Specifies the type of operation that will be performed to the blobstore. Allowed values are PUT, GET. |
| <code>container</code> | null | The name of the blob container. |
| <code>blobName</code> | null | The name of the blob. |

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.

MESSAGE HEADERS FOR BLOBSTORE

| Header | Description |
|------------------------------------|---|
| <code>CamelJcloudsOperation</code> | The operation to be performed on the blob. The valid options are <ul style="list-style-type: none">• PUT• GET |
| <code>CamelJcloudsContainer</code> | The name of the blob container. |
| <code>CamelJcloudsBlobName</code> | The name of the blob. |

BLOBSTORE USAGE SAMPLES

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>

```

EXAMPLE 2: GETTING/READING FROM A BLOB

This example will show you how you can read the content 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>

```

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>

```

COMPUTE SERVICE URI OPTIONS

| Name | Default Value | Description |
|-------------------|---------------|--|
| operation | PUT | Specifies the type of operation that will be performed to the compute service. Allowed values are CREATE_NODE, RUNS_CRIPT, DESTROY_NODE, LIST_NODES, LIST_IMAGES, LIST_HARDWARE. |
| imageId | null | *CREATE_NODE operation only* The imageId that will be used for creating a node. Values depend on the actual cloud provider. |
| locationId | null | *CREATE_NODE operation only* The location that will be used for creating a node. Values depend on the actual cloud provider. |
| hardwareId | null | *CREATE_NODE operation only* The hardware that will be used for creating a node. Values depend on the actual cloud provider. |
| group | null | *CREATE_NODE operation only* The group that will be assigned to the newly created node. Values depend on the actual cloud provider. |

| | | |
|---------------|------|--|
| nodeId | null | *RUN_SCRIPT & DESTROY_NODE operation only* The id of the node that will run the script or destroyed. |
| user | null | *RUN_SCRIPT operation only* The user on the target node that will run the script. |

The combination of parameters for use with the compute service depend on the operation.

```
jclouds:compute:aws-ec2?
operation=CREATE_NODE&imageId=AMI_XXXXX&locationId=eu-west-1&group=mygroup
```

COMPUTE USAGE SAMPLES

Below are some examples that demonstrate the use of jclouds compute producer in java dsl and spring/blueprint xml.

EXAMPLE 1: LISTING THE AVAILABLE IMAGES.

```
from("jclouds:compute:aws-ec2" +
    "&operation=LIST_IMAGES")
    .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=LIST_IMAGES"/>
  <to uri="direct:next"/>
</route>
```

EXAMPLE 2: CREATE A NEW NODE.

```
from("direct:start").
to("jclouds:compute:aws-ec2" +
   "?operation=CREATE_NODE" +
   "&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=CREATE_NODE&imageId=AMI_XXXXX&locationId=XXXXX&group=myGroup"/>
</route>
```

EXAMPLE 3: RUN A SHELL SCRIPT ON RUNNING NODE.

```
from("direct:start").
to("jclouds:compute:aws-ec2" +
   "?operation=RUN_SCRIPT" +
   "?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=RUN_SCRIPT&?
nodeId=10&user=ubuntu"/>
</route>
```

SEE ALSO

If you want to find out more about jclouds here is list of interesting resources [Jclouds Blobstore wiki](#) [Jclouds Compute wiki](#)

CHAPTER 57. JCR

JCR COMPONENT

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>
```

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.

USAGE

The `repository` element of the URI is used to look up the JCR Repository object in the Camel context registry.

PRODUCER

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|--------------------------|-----------------------|--|
| CamelJcrOperation | CamelJcrInsert | CamelJcrInsert or CamelJcrGetById operation to use |
| CamelJcrNodeName | null | Used to determine the node name to use. |

When a message is sent to a JCR producer endpoint:

- If the operation is **CamelJcrInsert**: A new node is created in the content repository, all the message properties of the IN message are transformed to JCR Value instances and added to the new node and the node's UUID is returned in the OUT message.
- If the operation is **CamelJcrGetById**: A new node is retrieved from the repository using the message body as node identifier.

CONSUMER

The consumer will connect to JCR periodically and return a `List<javax.jcr.observation.Event>` in the message body.

| Name | Default Value | Description |
|-------------------|----------------------|---|
| eventTypes | 0 | A combination of one or more event types encoded as a bit mask value such as <code>javax.jcr.observation.Event.NO_DE_ADDED</code> , <code>javax.jcr.observation.Event.NO_DE_REMOVED</code> , etc. |
| deep | false | When it is true, events whose associated parent node is at current path or within its subgraph are received. |
| uuids | null | Only events whose associated parent node has one of the identifiers in the comma separated uuid list will be received. |

| | | |
|--|--------------|---|
| nodeTypeNames | null | 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. |
| noLocal | false | If noLocal is true , then events generated by the session through which the listener was registered are ignored. Otherwise, they are not ignored. |
| sessionLiveCheckInterval | 60000 | Interval in milliseconds to wait before each session live checking. |
| sessionLiveCheckIntervalOnStart | 3000 | Interval in milliseconds to wait before the first session live checking. |

EXAMPLE

The snippet below creates a node named `node` under the `/home/test` node in the content repository. One additional attribute is added to the node as well: `my.contents.property` which will contain the body of the message being sent.

```
from("direct:a").setProperty(JcrConstants.JCR_NODE_NAME, constant("node"))
    .setProperty("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>
```

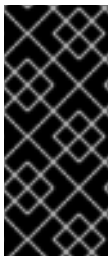
CHAPTER 58. JDBC

JDBC COMPONENT

The `jdbc` component enables you to access databases through JDBC, where SQL queries and operations are sent in the message body. This component uses the standard JDBC API, unlike the [SQL Component](#) component, which uses `spring-jdbc`.

**WARNING**

This component can only be used to define producer endpoints, which means that you cannot use the JDBC component in a `from()` statement.

**IMPORTANT**

This component can not be used as a [Transactional Client](#). If you need transaction support in your route, you should use the [SQL component](#) instead.

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&...`

OPTIONS

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|--|-------------------|--|
| <code>readSize</code> | <code>0</code> | The default maximum number of rows that can be read by a polling query. |
| <code>statement.<xxx></code> | <code>null</code> | Apache Camel 2.1: Sets additional options on the <code>java.sql.Statement</code> that is used behind the scenes to execute the queries. For instance, <code>statement.maxRows=10</code> . For detailed documentation, see the java.sql.Statement javadoc documentation. |
| <code>useJDBC4ColumnNameAndLabelSemantics</code> | <code>true</code> | Sets whether to use JDBC 4/3 column label/name semantics. You can use this option to turn it <code>false</code> in case you have issues with your JDBC driver to select data. This only applies when using <code>SQL SELECT</code> using aliases (e.g. <code>SQL SELECT id as identifier, name as given_name from persons</code>). |
| <code>resetAutoCommit</code> | <code>true</code> | Camel 2.9: Camel will set the <code>autoCommit</code> on the JDBC connection to be <code>false</code> , commit the change after executed the statement and reset the <code>autoCommit</code> flag of the connection at the end, if the <code>resetAutoCommit</code> is <code>true</code> . If the JDBC connection doesn't support to reset the <code>autoCommit</code> flag, you can set the <code>resetAutoCommit</code> flag to be <code>false</code> , and Camel will not try to reset the <code>autoCommit</code> flag. |
| <code>allowNamedParameters</code> | <code>true</code> | Camel 2.12: Whether to allow using named parameters in the queries. |

| | | |
|---------------------------------------|-------------------------|---|
| <code>prepareStatementStrategy</code> | | Camel 2.12: Allows to plugin to use a custom <code>org.apache.camel.component.jdbc.JdbcPreparedStatementStrategy</code> to control preparation of the query and prepared statement. |
| <code>useHeadersAsParameters</code> | <code>false</code> | Camel 2.12: Set this option to <code>true</code> to use the <code>prepareStatementStrategy</code> with named parameters. This allows to define queries with named placeholders, and use headers with the dynamic values for the query placeholders. |
| <code>outputType</code> | <code>SelectList</code> | Camel 2.12.1: Make the output of the producer to <code>SelectList</code> as List of Map, or <code>SelectOne</code> 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 <code>SELECT COUNT(*) FROM PROJECT</code> 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 <code>outputClass</code> 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 a non-unique result exception. |
| <code>outputClass</code> | <code>null</code> | Camel 2.12.1: Specify the full package and class name to use as conversion when <code>outputType=SelectOne</code> . |

| | | |
|---------------|--|---|
| beanRowMapper | | <p>Camel 2.12.1: To use a custom <code>org.apache.camel.component.jdbc.BeanRowMapper</code> when using <code>outputClass</code>. The default implementation will lower case the row names and skip underscores, and dashes. For example <code>"CUST_ID"</code> is mapped as <code>"custId"</code>.</p> |
|---------------|--|---|

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.



NOTE

This component fetches `ResultSetMetaData` to be able to return the column name as the key in the Map.

MESSAGE HEADERS

| Header | Description |
|-----------------------------------|---|
| CamelJdbcRowCount | If the query is a SELECT , the row count is returned in this OUT header. |
| CamelJdbcUpdateCount | If the query is an UPDATE , 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 <code>ResultSet</code> as a <code>java.util.Set</code> type. |

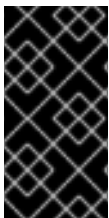
| | |
|---------------------|---|
| CamelJdbcParameters | Camel 2.12: A <code>java.util.Map</code> which has the headers to be used if <code>useHeadersAsParameters</code> has been enabled. |
|---------------------|---|

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](#).



IMPORTANT

Using generated keys does not work with together with named parameters.

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`.

SAMPLES

In the following example, we fetch the rows from the customer table.

First we register our datasource in the Apache Camel registry as `testdb`:

```

JndiRegistry reg = super.createRegistry();
reg.bind("testdb", db);
return reg;

```

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:

```

// lets add simple route
public void configure() throws Exception {
    from("direct:hello").to("jdbc:testdb?readSize=100");
}

```

Or you can create a `DataSource` in Spring like this:

```

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <!-- trigger every second -->
    <from uri="timer://kickoff?period=1s"/>
    <setBody>
      <constant>select * from customer</constant>
    </setBody>
    <to uri="jdbc:testdb"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
<!-- Just add a demo to show how to bind a date source for camel in
Spring-->

```



```
<jdbc:embedded-database id="testdb" type="DERBY">
  <jdbc:script location="classpath:sql/init.sql"/>
</jdbc:embedded-database>
```

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:

```
// first we create our exchange using the endpoint
Endpoint endpoint = context.getEndpoint("direct:hello");
Exchange exchange = endpoint.createExchange();
// then we set the SQL on the in body
exchange.getIn().setBody("select * from customer order by ID");

// now we send the exchange to the endpoint, and receives the response from
Camel
Exchange out = template.send(endpoint, exchange);

// assertions of the response
assertNotNull(out);
assertNotNull(out.getOut());
List<Map<String, Object>> data = out.getOut().getBody(List.class);
assertNotNull(data);
assertEquals(3, data.size());
Map<String, Object> row = data.get(0);
assertEquals("cust1", row.get("ID"));
assertEquals("jstrachan", row.get("NAME"));
row = data.get(1);
assertEquals("cust2", row.get("ID"));
assertEquals("nsandhu", row.get("NAME"));
```

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:

```
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
    .to("jdbc:testdb").split(body()).to("mock:result");
```

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");
```

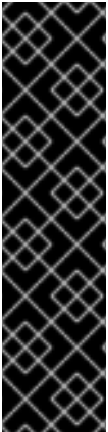
See also:

- [SQL](#)

CHAPTER 59. JETTY

JETTY COMPONENT

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

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 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.

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>
```

URI FORMAT

```
jetty:http://hostname[:port][/resourceUri][?options]
```

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

OPTIONS

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|-----------------------------------|--------------------|---|
| <code>sessionSupport</code> | <code>false</code> | Specifies whether to enable the session manager on the server side of Jetty. |
| <code>httpClient.XXX</code> | <code>null</code> | Configuration of Jetty's HttpClient . For example, setting <code>httpClient.idleTimeout=30000</code> sets the idle timeout to 30 seconds. |
| <code>httpClient</code> | <code>null</code> | To use a shared <code>org.eclipse.jetty.client.HttpClient</code> for all producers created by this endpoint. This option should only be used in special circumstances. |
| <code>httpClientMinThreads</code> | <code>null</code> | Camel 2.11:Producer only: To set a value for minimum number of threads in <code>HttpClient</code> thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured. |
| <code>httpClientMaxThreads</code> | <code>null</code> | Camel 2.11:Producer only: To set a value for maximum number of threads in <code>HttpClient</code> thread pool. This setting override any setting configured on component level. Notice that both a min and max size must be configured. |
| <code>httpBindingRef</code> | <code>null</code> | Reference to an <code>org.apache.camel.component.http.HttpBinding</code> in the Registry . <code>HttpBinding</code> can be used to customize how a response should be written for the consumer. |

| | | |
|----------------------------------|--------------------|--|
| <code>jettyHttpBindingRef</code> | <code>null</code> | Camel 2.6.0+: Reference to an <code>org.apache.camel.component.jetty.JettyHttpBinding</code> in the Registry . <code>JettyHttpBinding</code> can be used to customize how a response should be written for the producer. |
| <code>matchOnUriPrefix</code> | <code>false</code> | Whether or not the <code>CamelServlet</code> should try to find a target consumer by matching the URI prefix if no exact match is found. See here How do I let Jetty match wildcards . |
| <code>handlers</code> | <code>null</code> | Specifies a comma-delimited set of <code>org.mortbay.jetty.Handler</code> instances in your Registry (such as your Spring <code>ApplicationContext</code>). These handlers are added to the Jetty servlet context (for example, to add security). |
| <code>chunked</code> | <code>true</code> | Camel 2.2: If this option is false Jetty servlet will disable the HTTP streaming and set the content-length header on the response |
| <code>enableJmx</code> | <code>false</code> | Camel 2.3: If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details. |

| | | |
|--------------------------------|--------------|---|
| disableStreamCache | false | Camel 2.3: Determines whether or not the raw input stream from Jetty 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 Jetty 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 Jetty 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. |
| throwExceptionOnFailure | true | 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. |

| | | |
|------------------------------------|--------------------|--|
| <code>transferException</code> | <code>false</code> | Camel 2.6: 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. |
| <code>bridgeEndpoint</code> | <code>false</code> | > Camel 2.1: If the option is true , <code>HttpProducer</code> will ignore the <code>Exchange.HTTP_URI</code> header, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the <code>HttpProducer</code> send all the fault response back. Camel 2.3: If the option is true, <code>HttpProducer</code> and <code>CamelServlet</code> will skip the gzip processing if the content-encoding is "gzip". Also consider setting disableStreamCache to true to optimize when bridging. |
| <code>enableMultipartFilter</code> | <code>true</code> | Camel 2.5: Whether Jetty <code>org.eclipse.jetty.servlets.MultiPartFilter</code> is enabled or not. You should set this value to <code>false</code> when bridging endpoints, to ensure multipart requests is proxied/bridged as well. |
| <code>multipartFilterRef</code> | <code>null</code> | Camel 2.6: Allows using a custom multipart filter. Note: setting multipartFilterRef forces the value of enableMultipartFilter to <code>true</code> . |
| <code>FiltersRef</code> | <code>null</code> | Camel 2.9: Allows using a custom filters which is putted into a list and can be find in the Registry |

| | | |
|--------------------------------------|--------------------|---|
| <code>continuationTimeout</code> | <code>null</code> | Camel 2.6: Allows to set a timeout in millis when using Jetty as consumer (server). By default Jetty uses 30000. You can use a value of <code><= 0</code> 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 . |
| <code>useContinuation</code> | <code>true</code> | Camel 2.6: Whether or not to use Jetty continuations for the Jetty Server. |
| <code>sslContextParametersRef</code> | <code>null</code> | Camel 2.8: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the CAMEL:Registry. This reference overrides any configured <code>SSLContextParameters</code> at the component level. See Using the JSSE Configuration Utility . |
| <code>traceEnabled</code> | <code>false</code> | Specifies whether to enable HTTP TRACE for this Jetty consumer. By default TRACE is turned off. |
| <code>headerFilterStrategy</code> | <code>null</code> | Camel 2.11: Reference to a instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> in the Registry . It will be used to apply the custom <code>headerFilterStrategy</code> on the new create <code>HttpJettyEndpoint</code> . |
| <code>urlRewrite</code> | <code>null</code> | Camel 2.11:Producer only Refers to a custom <code>org.apache.camel.component.http.UrlRewrite</code> which allows you to rewrite urls when you bridge/proxy endpoints. See more details at UrlRewrite and How to use Camel as a HTTP proxy between a client and server . |

| | | |
|---------------------------------|-------------------|--|
| <code>responseBufferSize</code> | <code>null</code> | Camel 2.12: To use a custom buffer size on the <code>javax.servlet.ServletResponse</code> . |
|---------------------------------|-------------------|--|

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 chunked 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.

USAGE

The Jetty component supports both consumer and producer endpoints. Another option for producing to other HTTP endpoints, is to use the [HTTP Component](#)

COMPONENT OPTIONS

The `JettyHttpComponent` provides the following options:

| Name | Default Value | Description |
|-----------------------------|--------------------|--|
| <code>enableJmx</code> | <code>false</code> | Camel 2.3: If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details. |
| <code>sslKeyPassword</code> | <code>null</code> | Consumer only: The password for the keystore when using SSL. |

| | | |
|---|-------------------|---|
| <code>sslPassword</code> | <code>null</code> | Consumer only: The password when using SSL. |
| <code>sslKeystore</code> | <code>null</code> | Consumer only: The path to the keystore. |
| <code>minThreads</code> | <code>null</code> | Camel 2.5 Consumer only: To set a value for minimum number of threads in server thread pool. |
| <code>maxThreads</code> | <code>null</code> | Camel 2.5 Consumer only: To set a value for maximum number of threads in server thread pool. |
| <code>threadPool</code> | <code>null</code> | Camel 2.5 Consumer only: To use a custom thread pool for the server. |
| <code>sslSocketConnectors</code> | <code>null</code> | Camel 2.3 Consumer only: A map which contains per port number specific SSL connectors. See section <i>SSL support</i> for more details. |
| <code>socketConnectors</code> | <code>null</code> | Camel 2.5 Consumer only: A map which contains per port number specific HTTP connectors. Uses the same principle as <code>sslSocketConnectors</code> and therefore see section <i>SSL support</i> for more details. |
| <code>sslSocketConnectorProperties</code> | <code>null</code> | Camel 2.5 Consumer only. A map which contains general SSL connector properties. See section <i>SSL support</i> for more details. |
| <code>socketConnectorProperties</code> | <code>null</code> | Camel 2.5 Consumer only. A map which contains general HTTP connector properties. Uses the same principle as <code>sslSocketConnectorProperties</code> and therefore see section <i>SSL support</i> for more details. |

| | | |
|-----------------------------------|-------------------|--|
| <code>httpClient</code> | <code>null</code> | Deprecated:Producer only: To use a custom <code>HttpClient</code> with the jetty producer. This option is removed from Camel 2.11 onwards, instead you can set the option on the endpoint instead. |
| <code>httpClientMinThreads</code> | <code>null</code> | Producer only: To set a value for minimum number of threads in <code>HttpClient</code> thread pool. Notice that both a min and max size must be configured. |
| <code>httpClientMaxThreads</code> | <code>null</code> | Producer only: To set a value for maximum number of threads in <code>HttpClient</code> thread pool. Notice that both a min and max size must be configured. |
| <code>httpClientThreadPool</code> | <code>null</code> | Deprecated:Producer only: To use a custom thread pool for the client. This option is removed from Camel 2.11 onwards. |
| <code>sslContextParameters</code> | <code>null</code> | Camel 2.8: To configure a custom SSL/TLS configuration options at the component level. See Using the JSSE Configuration Utility for more details. |
| <code>requestBufferSize</code> | <code>null</code> | Camel 2.11.2: Allows to configure a custom value of the request buffer size on the Jetty connectors. |
| <code>requestHeaderSize</code> | <code>null</code> | Camel 2.11.2: Allows to configure a custom value of the request header size on the Jetty connectors. |
| <code>responseBufferSize</code> | <code>null</code> | Camel 2.11.2: Allows to configure a custom value of the response buffer size on the Jetty connectors. |

| | | |
|--------------------|------|--|
| responseHeaderSize | null | Camel 2.11.2: Allows to configure a custom value of the response header size on the Jetty connectors. |
|--------------------|------|--|

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("jetty://http://www.google.com");
```

or in Spring XML

```
<route>
  <from uri="direct:start"/>
  <to uri="jetty://http://www.google.com"/>
</route>
```

CONSUMER EXAMPLE

In this sample we define a route that exposes a HTTP service at <http://localhost:8080/myapp/myservice>:

```
from("jetty:http://localhost:{{port}}/myapp/myservice").process(new
MyBookService());
```



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.

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.

```
public class MyBookService implements Processor {
    public void process(Exchange exchange) throws Exception {
        // just get the body as a string
        String body = exchange.getIn().getBody(String.class);

        // we have access to the HttpServletRequest here and we can grab
it if we need it
        HttpServletRequest req =
exchange.getIn().getBody(HttpServletRequest.class);
        assertNotNull(req);

        // for unit testing
        assertEquals("bookid=123", body);

        // send a html response
        exchange.getOut().setBody("<html><body>Book 123 is Camel in
Action</body></html>");
    }
}
```

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`.

```
from("jetty:" + serverUri)
    .choice()
    .when().simple("${header.one}").to("mock:one")
    .otherwise()
    .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 `EL` or `OGNL`--we could also test for the parameter value and do routing based on the header value as well.

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();
    ...
}
```

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/?
sslContextParametersRef=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.keystorespecifies` 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>
```



```

        </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.

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>
```

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")
```

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>

```

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 `HttpProducer.HTTP_RESPONSE_CODE` header of the OUT message.

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:

```

public class MyHttpBinding extends DefaultHttpBinding {
    public MyHttpBinding(HttpEndpoint ep) {
        super(ep);
    }
    @Override
    public void doWriteExceptionResponse(Throwable exception,
HttpServletResponse response) throws IOException {
        // we override the doWriteExceptionResponse as we only want to
alter the binding how exceptions is
        // written back to the client.

        // we just return HTTP 200 so the client thinks its okay
response.setStatus(200);
        // and we return this fixed text
response.getWriter().write("Something went wrong but we dont

```

```
care");
    }
}
```

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/myervice?
httpBindingRef=mybinding"/><to uri="bean:doSomething"/></route>
```

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.

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:

```

from("jetty://http://localhost:{{port}}/myserver")
  // use onException to catch all exceptions and return a custom reply
  message
  .onException(Exception.class)
    .handled(true)
  // create a custom failure response

```

```

        .transform(constant("Dude something went wrong"))
        // we must remember to set error code 500 as handled(true)
        // otherwise would let Camel think its a OK response (200)
        .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
    .end()
    // now just force an exception immediately
    .throwException(new IllegalArgumentException("I cannot do this"));

```

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.

```

// Set the jetty temp directory which store the file for multi part form
// camel-jetty will clean up the file after it handled the request.
// The option works rightly from Camel 2.4.0
getContext().getProperties().put("CamelJettyTempDir", "target");

from("jetty://http://localhost:{{port}}/test").process(new Processor() {

    public void process(Exchange exchange) throws Exception {
        Message in = exchange.getIn();
        assertEquals("Get a wrong attachment size", 1,
in.getAttachments().size());
        // The file name is attachment id
        DataHandler data = in.getAttachment("NOTICE.txt");

        assertNotNull("Should get the DataHandle NOTICE.txt", data);
        // This assert is wrong, but the correct content-type
(application/octet-stream)
        // will not be returned until Jetty makes it available - currently
the content-type
        // returned is just the default for FileDataHandler (for the
implentation being used)
        //assertEquals("Get a wrong content type", "text/plain",
data.getContentType());
        assertEquals("Got the wrong name", "NOTICE.txt", data.getName());

        assertTrue("We should get the data from the DataHandle",
data.getDataSource()
            .getInputStream().available() > 0);

        // The other form date can be get from the message header
exchange.getOut().setBody(in.getHeader("comment"));
    }
});

```

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.

-

[HTTP](#)

CHAPTER 60. JING

JING COMPONENT

The Jing component uses the [Jing Library](#) to perform XML validation of the message body using either:

- [RelaxNG XML Syntax](#)
- [RelaxNG Compact Syntax](#)

Note that the [MSV](#) component can also support RelaxNG XML syntax.

URI FORMAT

```
rng:someLocalOrRemoteResource
rnc:someLocalOrRemoteResource
```

Where `rng` means use the [RelaxNG XML Syntax](#) whereas `rnc` means use [RelaxNG Compact Syntax](#). The following examples show possible URI values

| Example | Description |
|---|--|
| <code>rng:foo/bar.rng</code> | References the XML file foo/bar.rng on the classpath |
| <code>rnc:http://foo.com/bar.rnc</code> | References the RelaxNG Compact Syntax file from the URL, http://foo.com/bar.rnc . |

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

OPTIONS

| Option | Default | Description |
|--------|---------|-------------|
|--------|---------|-------------|

| | | |
|--------|-------|--|
| useDom | false | Apache Camel 2.0: Specifies whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator. |
|--------|-------|--|

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).

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <doTry>
      <to
uri="rnc:org/apache/camel/component/validator/jing/schema.rnc"/>
      <to uri="mock:valid"/>

      <doCatch>

<exception>org.apache.camel.ValidationException</exception>
      <to uri="mock:invalid"/>
    </doCatch>
    <doFinally>
      <to uri="mock:finally"/>
    </doFinally>
    </doTry>
  </route>
</camelContext>
```

CHAPTER 61. JMS

JMS COMPONENT



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.

TRANSACTIONED AND CACHING

See section *Transactions and Cache Levels* below if you are using transactions with [JMS](#) as it can impact performance.

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.

The JMS component allows messages to be sent to (or consumed from) a [JMS](#) Queue or Topic. The implementation of the JMS Component uses Spring's JMS support for declarative transactions, using 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>
```

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&...`

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.

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.

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](#).

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

OPTIONS

You can configure many different properties on the JMS endpoint which map to properties on the [JMSConfiguration POJO](#).



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.

The options are divided into two tables, the first one with the most common options used. The latter contains the rest.

MOST COMMONLY USED OPTIONS

| Option | Default Value | Description |
|--------|---------------|-------------|
|--------|---------------|-------------|

| | | |
|--------------------------------|--------------|--|
| clientId | null | 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. You may prefer to use Virtual Topics instead. |
| concurrentConsumers | 1 | Specifies the default number of concurrent consumers. From Camel 2.10.3 onwards this option can also be used when doing request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. |
| disableReplyTo | false | If true , a producer will behave like a InOnly exchange with the exception that JMSReplyTo header is sent out and not be suppressed like in the case of InOnly . Like InOnly the producer will not wait for a reply. A consumer with this flag will behave like InOnly . This feature can be used to bridge InOut requests to another queue so that a route on the other queue will send its response directly back to the original JMSReplyTo . |
| durableSubscriptionName | null | The durable subscriber name for specifying durable topic subscriptions. The clientId option must be configured as well. |
| maxConcurrentConsumers | 1 | Specifies the maximum number of concurrent consumers. From Camel 2.10.3 onwards this option can also be used when doing request/reply over JMS. See also the maxMessagesPerTask option to control dynamic scaling up/down of threads. |

| | | |
|---------------------------|--------------|--|
| maxMessagesPerTask | \-1 | 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. |
| preserveMessageQos | false | 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. |
| replyTo | null | Provides an explicit ReplyTo destination, which overrides any incoming value of Message.getJMSReplyTo() . If you do Request Reply over JMS then make sure to read the section <i>Request-reply over JMS</i> further below for more details, and the replyToType option as well. |

| | | |
|-----------------------------|--------------------|---|
| <code>replyToType</code> | <code>null</code> | <p>Camel 2.9: Allows for explicitly specifying which kind of strategy to use for <code>replyTo</code> queues when doing request/reply over JMS. Possible values are: Temporary, Shared, or Exclusive. By default Camel will use temporary queues. However if <code>replyTo</code> has been configured, then Shared is used by default. This option allows you to use exclusive queues instead of shared ones. See further below 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.</p> |
| <code>requestTimeout</code> | <code>20000</code> | <p>Producer only: The timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds). The default is 20 seconds. See below in section <i>About time to live</i> for more details. See also the <code>requestTimeoutCheckerInterval</code> option.</p> |
| <code>selector</code> | <code>null</code> | <p>Sets the JMS Selector, which is an SQL 92 predicate that is used to filter messages within the broker. You may have to encode special characters such as = as %3D Before Camel 2.3.0, we don't support this option in <code>CamelConsumerTemplate</code></p> |
| <code>timeToLive</code> | <code>null</code> | <p>When sending messages, specifies the time-to-live of the message (in milliseconds). See below in section <i>About time to live</i> for more details.</p> |

| | | |
|--------------------------------------|--------------------|---|
| <code>transacted</code> | <code>false</code> | Specifies whether to use transacted mode for sending/receiving messages using the InOnly Exchange Pattern . |
| <code>testConnectionOnStartup</code> | <code>false</code> | Camel 2.1: 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. From Camel 2.8 onwards also the JMS producers is tested as well. |

ALL THE OTHER OPTIONS

| Option | Default Value | Description |
|--|--------------------|---|
| <code>acceptMessagesWhileStopping</code> | <code>false</code> | 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. |

| | | |
|--------------------------------|-------------------------|--|
| acknowledgementModeName | AUTO_ACKNOWLEDGE | The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE |
| acknowledgementMode | \-1 | 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. |
| allowNullBody | true | Camel 2.9.3/2.10.1: Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown. |
| alwaysCopyMessage | false | 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) |

| | | |
|---------------------------|--------------|--|
| asyncConsumer | false | <p>Camel 2.9: 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 transactions must be executed synchronously (Camel 3.0 may support async transactions).</p> |
| asyncStartListener | false | <p>Camel 2.10: 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.</p> |

| | | |
|--------------------------|---|---|
| asyncStopListener | false | Camel 2.10: Whether to stop the JmsConsumer message listener asynchronously, when stopping a route. |
| autoStartup | true | Specifies whether the consumer container should auto-startup. |
| cacheLevelName | CACHE_AUTO (Camel >= 2.8.0) CACHE_CONSUMER (Camel <= 2.7.1) | 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 for Camel 2.8 and newer is CACHE_AUTO . For Camel 2.7.1 and older the default is CACHE_CONSUMER . See the Spring documentation and Transactions Cache Levels for more information. |
| cacheLevel | | Sets the cache level by ID for the underlying JMS resources. See cacheLevelName option for more details. |

| | | |
|--------------------------|----------------|---|
| consumerType | Default | <p>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 messageListenerContainerFactoryRef option will determine what org.springframework.jms.listener.AbstractMessageListenerContainer to use (new option in Camel 2.11 and 2.10.2). This option was temporary removed in Camel 2.7 and 2.8. But has been added back from Camel 2.9 onwards.</p> |
| connectionFactory | null | <p>The default JMS connection factory to use for the listenerConnectionFactory and templateConnectionFactory, if neither is specified.</p> |

| | | |
|--------------------------------|-------------------|--|
| defaultTaskExecutorType | (see description) | Camel 2.10.4: 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. |
| deliveryPersistent | true | Specifies whether persistent delivery is used by default. |
| destination | null | Specifies the JMS Destination object to use on this endpoint. |
| destinationName | null | Specifies the JMS destination name to use on this endpoint. |
| destinationResolver | null | 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). |

| | | |
|---------------------------------|--------------|--|
| disableTimeToLive | false | <p>Camel 2.8: 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 <i>About time to live</i> for more details.</p> |
| eagerLoadingOfProperties | false | <p>Enables eager loading of JMS properties as soon as a message is received, which is generally inefficient, because the JMS properties might not be required. But this feature can sometimes catch early any issues with the underlying JMS provider and the use of JMS properties. This feature can also be used for testing purposes, to ensure JMS properties can be understood and handled correctly.</p> |
| exceptionListener | null | <p>Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.</p> |

| | | |
|--|--------------------|--|
| <code>errorHandler</code> | <code>null</code> | > Camel 2.8.2, 2.9: Specifies a <code>org.springframework.util.ILogHandler</code> 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. From Camel 2.9.1: onwards you can configure logging level and whether stack traces should be logged using the below two options. This makes it much easier to configure, than having to code a custom errorHandler . |
| <code>errorHandlerLoggingLevel</code> | <code>WARN</code> | Camel 2.9.1: Allows to configure the default errorHandler logging level for logging uncaught exceptions. |
| <code>errorHandlerLogStackTrace</code> | <code>true</code> | Camel 2.9.1: Allows to control whether stacktraces should be logged or not, by the default errorHandler . |
| <code>explicitQosEnabled</code> | <code>false</code> | 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. |
| <code>exposeListenerSession</code> | <code>true</code> | Specifies whether the listener session should be exposed when consuming messages. |

| | | |
|---------------------------------------|--------------------|--|
| <code>forceSendOriginalMessage</code> | <code>false</code> | <p>>Camel 2.7: When using <code>mapJmsMessage=false</code> 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 <code>true</code> to force Camel to send the original JMS message that was received.</p> |
| <code>idleTaskExecutionLimit</code> | <code>1</code> | <p>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 <code>maxConcurrentConsumers</code> setting).</p> |
| <code>idleConsumerLimit</code> | <code>1</code> | <p>Camel 2.8.2, 2.9: Specify the limit for the number of consumers that are allowed to be idle at any given time.</p> |
| <code>includeSentJMSMessageID</code> | <code>false</code> | <p>Camel 2.10.3: Only applicable when sending to JMS destination using <code>InOnly</code> (eg <code>fire and forget</code>). Enabling this option will enrich the Camel <code>Exchange</code> with the actual <code>JMSMessageID</code> that was used by the JMS client when the message was sent to the JMS destination.</p> |
| <code>includeAllJMSXProperties</code> | <code>false</code> | <p>Camel 2.11.2/2.12: Whether to include all <code>JMSXxxx</code> properties when mapping from JMS to Camel Message. Setting this to <code>true</code> will include properties such as <code>JMSXAppID</code>, and <code>JMSXUserID</code> etc. Note: If you are using a custom <code>headerFilterStrategy</code> then this option does not apply.</p> |

| | | |
|-------------------------------------|----------------|---|
| jmsMessageType | null | 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. |
| jmsKeyFormatStrategy | default | 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. |
| jmsOperations | null | 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. |
| lazyCreateTransactionManager | true | If true , Camel will create a JmsTransactionManager , if there is no transactionManager injected when option transacted=true . |

| | | |
|---|-------------|---|
| listenerConnectionFactory | null | The JMS connection factory used for consuming messages. |
| mapJmsMessage | true | Specifies whether Camel should auto map the received JMS message to an appropriate payload type, such as javax.jms.TextMessage to a String etc. See section about how mapping works below for more details. |
| maximumBrowseSize | \-1 | Limits the number of messages fetched at most, when browsing endpoints using Browse or JMX API. |
| messageConverter | null | To use a custom Spring org.springframework.jms.support.converter.MessageConverter so you can be 100% in control how to map to/from a javax.jms.Message . |
| messageIdEnabled | true | When sending, specifies whether message IDs should be added. |
| messageListenerContainerFactoryRef | null | Camel 2.10.2: 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 . |
| messageTimestampEnabled | true | Specifies whether timestamps should be enabled by default on sending messages. |
| password | null | The password for the connector factory. |

| | | |
|---|-----------------------------|---|
| <code>priority</code> | <code>4</code> | 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. |
| <code>pubSubNoLocal</code> | <code>false</code> | Specifies whether to inhibit the delivery of messages published by its own connection. |
| <code>receiveTimeout</code> | <code>None</code> | The timeout for receiving messages (in milliseconds). |
| <code>recoveryInterval</code> | <code>5000</code> | 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. |
| <code>replyToCacheLevelName</code> | <code>CACHE_CONSUMER</code> | Camel 2.9.1: 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. |
| <code>replyToDestinationSelectorName</code> | <code>null</code> | 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). |

| | | |
|--|---|---|
| <code>replyToDeliveryPersistent</code> | <code>true</code> | Specifies whether to use persistent delivery by default for replies. |
| <code>requestTimeoutCheckerInterval</code> | <code>1000</code> | Camel 2.9.2: 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 <code>requestTimeout</code> . |
| <code>subscriptionDurable</code> | <code>false</code> | @deprecated: Enabled by default, if you specify a <code>durableSubscriberName</code> and a <code>clientId</code> . |
| <code>taskExecutor</code> | <code>null</code> | Allows you to specify a custom task executor for consuming messages. |
| <code>taskExecutorSpring2</code> | <code>null</code> | Camel 2.6: To use when using Spring 2.x with Camel. Allows you to specify a custom task executor for consuming messages. |
| <code>templateConnectionFactory</code> | <code>null</code> | The JMS connection factory used for sending messages. |
| <code>transactedInOut</code> | <code>false</code> | @deprecated: Specifies whether to use transacted mode for sending messages using the InOut Exchange Pattern . Applies only to producer endpoints. See section Enabling Transacted Consumption for more details. |
| <code>transactionManager</code> | <code>null</code> | The Spring transaction manager to use. |
| <code>transactionName</code> | <code>"JmsConsumer[destinationName]"</code> | The name of the transaction to use. |

| | | |
|---------------------------|--------------|---|
| transactionTimeout | null | The timeout value of the transaction (in seconds), if using transacted mode. |
| transferException | false | 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. |
| transferExchange | false | 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. |
| username | null | The username for the connector factory. |

| | | |
|--|--------------------|---|
| <code>useMessageIDAsCorrelationID</code> | <code>false</code> | Specifies whether JMSMessageID should always be used as JMSCorrelationID for InOut messages. |
| <code>useVersion102</code> | <code>false</code> | @deprecated (removed from Camel 2.5 onwards): Specifies whether the old JMS API should be used. |

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 |
|-----------------------------------|--------------------------------------|--|
| <code>String</code> | <code>javax.jms.TextMessage</code> | |
| <code>org.w3c.dom.Node</code> | <code>javax.jms.TextMessage</code> | The DOM will be converted to String . |
| <code>Map</code> | <code>javax.jms.MapMessage</code> | |
| <code>java.io.Serializable</code> | <code>javax.jms.ObjectMessage</code> | |
| <code>byte[]</code> | <code>javax.jms.BytesMessage</code> | |
| <code>java.io.File</code> | <code>javax.jms.BytesMessage</code> | |
| <code>java.io.Reader</code> | <code>javax.jms.BytesMessage</code> | |
| <code>java.io.InputStream</code> | <code>javax.jms.BytesMessage</code> | |
| <code>java.nio.ByteBuffer</code> | <code>javax.jms.BytesMessage</code> | |

When receiving a JMS message, Camel converts the JMS message to the following body type:

| JMS Message | Body Type |
|--------------------------------------|--|
| <code>javax.jms.TextMessage</code> | <code>String</code> |
| <code>javax.jms.BytesMessage</code> | <code>byte[]</code> |
| <code>javax.jms.MapMessage</code> | <code>Map<String, Object></code> |
| <code>javax.jms.ObjectMessage</code> | <code>Object</code> |

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.

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.

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`.

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 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}
```

MESSAGE FORMAT WHEN RECEIVING

Camel adds the following properties to the Exchange when it receives a message:

| Property | Type | Description |
|--|------------------------------------|------------------------|
| <code>org.apache.camel.jms.replyDestination</code> | <code>javax.jms.Destination</code> | The reply destination. |

Camel adds the following JMS properties to the In message headers when it receives a JMS message:

| Header | Type | Description |
|-------------------------------|------------------------------------|--|
| <code>JMSCorrelationID</code> | <code>String</code> | The JMS correlation ID. |
| <code>JMSDeliveryMode</code> | <code>int</code> | The JMS delivery mode. |
| <code>JMSDestination</code> | <code>javax.jms.Destination</code> | The JMS destination. |
| <code>JMSExpiration</code> | <code>long</code> | The JMS expiration. |
| <code>JMSMessageID</code> | <code>String</code> | The JMS unique message ID. |
| <code>JMSPriority</code> | <code>int</code> | 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. |

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.

JMSPRODUCER

The `JmsProducer` behaves as follows, depending on configuration:

| Exchange Pattern | Other options | Description |
|------------------|---------------|-------------|
|------------------|---------------|-------------|

| | | |
|---------------|--------------------------|---|
| <i>InOut</i> | \- | 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. |
| <i>InOut</i> | 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. |
| <i>InOnly</i> | \- | Camel will send the message and not expect a reply. |
| <i>InOnly</i> | 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. |

JMSCONSUMER

The `JmsConsumer` behaves as follows, depending on configuration:

| Exchange Pattern | Other options | Description |
|------------------|---------------|---|
| <i>InOut</i> | \- | Camel will send the reply back to the JMSReplyTo queue. |
| <i>InOnly</i> | \- | Camel will not send a reply back, as the pattern is <i>InOnly</i> . |

| | | |
|----|----------------------------|---------------------------------|
| \- | 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");
```

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.

CONFIGURING DIFFERENT JMS PROVIDERS

You can configure your JMS provider in [Spring XML](#) as follows:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <jmxAgent id="agent" disabled="true"/>
</camelContext>

<bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL" value="vm://localhost?
broker.persistent=false&roker.useJmx=false"/>
        </bean>
    </property>
</bean>
```

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.

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.

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`.

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 | <p>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.</p> |
| Exclusive | Fast | No | <p>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</p> |

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 a 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.

| | | | |
|-------------------------------------|------|-----|--|
| <code>concurrentConsumers</code> | Fast | Yes | <p>Camel 2.10.3: Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the <code>concurrentConsumers</code> and <code>maxConcurrentConsumers</code> options.</p> <p>Notice: That using Shared reply queues may not work as well with concurrent listeners, so use this option with care.</p> |
| <code>maxConcurrentConsumers</code> | Fast | Yes | <p>Camel 2.10.3: Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the <code>concurrentConsumers</code> and <code>maxConcurrentConsumers</code> options.</p> <p>Notice: That using Shared reply queues may not work as well with concurrent listeners, so use this option with care.</p> |

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 `SpringDefaultMessageListenerContainer` 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:

```
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);
```

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.

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 selectors 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.

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.

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.

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.

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 Transaction 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](#).

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.");
    }
});
```



```

    }
}

```

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:

```

// send a in-out with a timeout for 5 sec
Object out = template.requestBody("activemq:queue:slow?
requestTimeout=5000", "Hello World");

```

SAMPLES

JMS is used in many examples for other components as well. But we provide a few samples below to get started.

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");

```

SENDING TO A 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");
```

USING ANNOTATIONS

Camel also has annotations so you can use [POJO Consuming](#) and [POJO Producing](#).

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>
```

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](#).

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();
```

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.

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.

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.DetailedJMSEException: 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 JMSEException {
        MQQueueSession wmqSession = (MQQueueSession) session;
        return wmqSession.createQueue("queue:/// " + destinationName + "?
targetClient=1");
    }
});

```

- [Transactional Client](#)
- [Bean Integration](#)
- [Tutorial-JmsRemoting](#)
- [JMSTemplate gotchas](#)

CHAPTER 62. JMX

JMX COMPONENT

The JMX component enables consumers to subscribe to an MBean's notifications. The component supports passing the Notification object directly through the exchange or serializing it to XML according to the schema provided within this project. This is a consumer-only component. Exceptions are thrown if you attempt to create a producer for it.

URI FORMAT

The component can connect to the local platform MBean server with the following URI:

```
jmx://platform?options
```

A remote MBean server URL can be specified after the `jmx:` scheme prefix, as follows:

```
jmx:service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi?options
```

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

URI OPTIONS

| Property | Required | Default | Description |
|---------------------|----------|------------------|---|
| <code>format</code> | | <code>xml</code> | Format for the message body. Either <code>xml</code> or <code>raw</code> . If <code>xml</code> , the notification is serialized to XML. If <code>raw</code> , the raw java object is set as the body. |

| | | | |
|--------------------------------|-----|-------------|---|
| password | | | Credentials for making a remote connection. |
| objectDomain | Yes | | The domain of the MBean you are connecting to. |
| objectName | | | The name key for the MBean you are connecting to. Either this property of a list of keys must be provided (but not both). For more details, see the section called "ObjectName Construction" . |
| notificationFilter | | | Reference to a bean that implements the NotificationFilter interface. The #beanID syntax should be used to reference the bean in the registry. |
| handback | | | Value to hand back to the listener when a notification is received. This value will be put into the jmx.handback message header. |
| testConnectionOnStartup | | true | *Camel 2.11* If true, the consumer will throw an exception when 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 <i>reconnectDelay</i> . |

| | | | |
|-------------------------------------|--|--------------|--|
| reconnectOnConnectionFailure | | false | *Camel 2.11* 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 <i>reconnectDelay</i> . |
| reconnectDelay | | 10 | *Camel 2.11* The number of seconds to wait before retrying creation of the initial connection or before reconnecting a lost connection. |

OBJECTNAME CONSTRUCTION

The URI must always have the `objectDomain` property. In addition, the URI must contain either `objectName` or one or more properties that start with `key`.

DOMAIN WITH NAME PROPERTY

When the `objectName` property is provided, the following constructor is used to build the `ObjectName` instance for the MBean:

```
ObjectName(String domain, String key, String value)
```

The key value in the preceding constructor must be name and the value is the value of the `objectName` property.

DOMAIN WITH HASHTABLE

-


```
ObjectName(String domain, Hashtable<String,String> table)
```

The `Hashtable` is constructed by extracting properties that start with key. The properties will have the key prefix stripped prior to building the `Hashtable`. This allows the URI to contain a variable number of properties to identify the `MBean`.

EXAMPLE

```
from("jmx:platform?objectDomain=jmxExample&key.name=simpleBean").
  to("log:jmxEvent");
```

FULL EXAMPLE

A complete example using the JMX component is available under the `examples/camel-example-jmx` directory.

MONITOR TYPE CONSUMER

Available as of Camel 2.8 One popular use case for JMX is creating a monitor bean to monitor an attribute on a deployed bean. This requires writing a few lines of Java code to create the JMX monitor and deploy it. As shown below:

```
CounterMonitor monitor = new CounterMonitor();
monitor.addObservedObject(makeObjectName("simpleBean"));
monitor.setObservedAttribute("MonitorNumber");
monitor.setNotify(true);
monitor.setInitThreshold(1);
monitor.setGranularityPeriod(500);
registerBean(monitor, makeObjectName("counter"));
monitor.start();
```

The 2.8 version introduces a new type of consumer that automatically creates and registers a monitor bean for the specified `objectName` and attribute. Additional endpoint attributes allow the user to specify the attribute to monitor, type of monitor to create, and any other required properties. The code snippet above is condensed into a set of endpoint properties. The consumer uses these properties to create the `CounterMonitor`, register it, and then subscribe to its changes. All of the JMX monitor types are supported.

EXAMPLE

■

```
from("jmx:platform?objectDomain=myDomain&objectName=simpleBean&" +
    "monitorType=counter&observedAttribute=MonitorNumber&initThreshold=1&" +
    "granularityPeriod=500").to("mock:sink");
```

The example above will cause a new Monitor Bean to be created and deployed to the local mbean server that monitors the MonitorNumber attribute on the simpleBean. Additional types of monitor beans and options are detailed below. The newly deployed monitor bean is automatically undeployed when the consumer is stopped.

URI OPTIONS FOR MONITOR TYPE

| property | type | applies to | description |
|-------------------|---------|----------------|--|
| monitorType | enum | all | one of the counters, guage, string |
| observedAttribute | string | all | the attribute being observed |
| granularityPeriod | long | all | granularity period (in millis) for the attribute being observed. As per JMX, default is 10 seconds |
| initThreshold | number | counter | initial threshold value |
| offset | number | counter | offset value |
| modulus | number | counter | modulus value |
| differenceMode | boolean | counter, gauge | true if difference should be reported, false for actual value |
| notifyHigh | boolean | gauge | high notification on/off switch |
| notifyLow | boolean | gauge | low notification on/off switch |
| highThreshold | number | gauge | threshold for reporting high notification |

| | | | |
|------------------------|----------------|--------|---|
| lowThreshold | number | gauge | threshold for reporting low notificaton |
| notifyDiffer | boolean | string | true to fire notification when string differs |
| notifyMatch | boolean | string | true to fire notification when string matches |
| stringToCompare | string | string | string to compare against the attribute value |

The monitor style consumer is only supported for the local mbean server. JMX does not currently support remote deployment of mbeans without either having the classes already remotely deployed or an adapter library on both the client and server to facilitate a proxy deployment.

CHAPTER 63. JPA

JPA COMPONENT

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.

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 an array of entity beans.

If the body does not contain one of the preceding types, put a `MessageTranslator` in front of the endpoint to perform the necessary conversion first.

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).

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&...`

OPTIONS

| Name | Default Value | Description |
|--------------------------------|------------------------------|---|
| <code>entityType</code> | <code>entityClassName</code> | Overrides the <code>entityClassName</code> from the URI. |
| <code>persistenceUnit</code> | <code>camel</code> | The JPA persistence unit used by default. |
| <code>consumeDelete</code> | <code>true</code> | JPA consumer only: If <code>true</code> , the entity is deleted after it is consumed; if <code>false</code> , the entity is not deleted. |
| <code>consumeLockEntity</code> | <code>true</code> | JPA consumer only: Specifies whether or not to set an exclusive lock on each entity bean while processing the results from polling. |
| <code>flushOnSend</code> | <code>true</code> | JPA producer only: Flushes the EntityManager after the entity bean has been persisted. |
| <code>maximumResults</code> | <code>-1</code> | JPA consumer only: Set the maximum number of results to retrieve on the Query . |

| | | |
|-------------------------------------|--------------------|---|
| <code>transactionManager</code> | <code>null</code> | Apache Camel 1.6.1/2.0: Specifies the transaction manager to use. If none provided, Apache Camel will use a JpaTransactionManager by default. Can be used to set a JTA transaction manager (for integration with an EJB container). |
| <code>consumer.delay</code> | <code>500</code> | JPA consumer only: Delay in milliseconds between each poll. |
| <code>consumer.initialDelay</code> | <code>1000</code> | JPA consumer only: Milliseconds before polling starts. |
| <code>consumer.useFixedDelay</code> | <code>false</code> | JPA consumer only: Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |
| <code>maxMessagesPerPoll</code> | <code>0</code> | Apache Camel 2.0;JPA consumer only: 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. |
| <code>consumer.query</code> | | JPA consumer only: To use a custom query when consuming data. |
| <code>consumer.namedQuery</code> | | JPA consumer only: To use a named query when consuming data. |
| <code>consumer.nativeQuery</code> | | JPA consumer only: To use a custom native query when consuming data. |

| | | |
|------------------------------------|--------------------|---|
| <code>consumer.parameters</code> | | Camel 2.12: JPA consumer only: the parameters map which will be used for building the query. The parameters is an instance of Map which key is String and value is Object. |
| <code>consumer.resultClass</code> | | Camel 2.7: JPA consumer only: Defines the type of the returned payload (we will call <code>entityManager.createNativeQuery(nativeQuery, resultClass)</code> instead of <code>entityManager.createNativeQuery(nativeQuery)</code>). Without this option, we will return an object array. Only has an affect when using in conjunction with native query when consuming data. |
| <code>consumer.transacted</code> | <code>false</code> | *Camel 2.7.5/2.8.3/2.9: JPA consumer only:* 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. |
| <code>consumer.lockModeType</code> | <code>WRITE</code> | Camel 2.11.2/2.12: To configure the lock mode on the consumer. The possible values is defined in the enum <code>javax.persistence.LockModeType</code> . |
| <code>usePersist</code> | <code>false</code> | Camel 2.5: JPA producer only: Indicates to use <code>entityManager.persist(entity)</code> instead of <code>entityManager.merge(entity)</code> . Note: <code>entityManager.persist(entity)</code> doesn't work for detached entities (where the EntityManager has to execute an UPDATE instead of an INSERT query)! |

MESSAGE HEADERS

Apache Camel adds the following message headers to the exchange:

| Header | Type | Description |
|------------------|-------------|--|
| CamelJpaTemplate | JpaTemplate | Apache Camel 2.0: 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. |

CONFIGURING ENTITYMANAGERFACTORY

You are strongly advised to configure the JPA component to use a specific `EntityManagerFactory` instance. If you do not do so, each `JpaEndpoint` will auto-create its own `EntityManagerFactory` instance. 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`.

CONFIGURING TRANSACTIONMANAGER

You are strongly advised to specify the `TransactionManager` instance used by the JPA component. If you do not do so, each `JpaEndpoint` will auto-create its own instance of `TransactionManager`. For example, you can instantiate 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>
```


In Camel 2.3 the `JpaComponent` will auto lookup the `TransactionManager` 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`.

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");
```

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");
```

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.

EXAMPLE

See the Tracer Example for an example using JPA to store traced messages into a database.

USING THE JPA BASED IDEMPOTENT REPOSITORY

In this section we will use the JPA based idempotent repository.

First we need to setup a persistence-unit in the persistence.xml file:

```
<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"/>
  </properties>
</persistence-unit>
```

Second we have to setup a `org.springframework.orm.jpa.JpaTemplate` which is used by the `org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository`:

```
<!-- this is standard spring JPA configuration -->
<bean id="jpaTemplate" class="org.springframework.orm.jpa.JpaTemplate">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <!-- we use idempotentDB as the persintence unit name defined in the
persistence.xml file -->
  <property name="persistenceUnitName" value="idempotentDb"/>
</bean>
```

Afterwards we can configure our `org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository`:

```
<!-- 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 spring jpaTemplate -->
  <constructor-arg index="0" ref="jpaTemplate"/>
  <!-- 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 finally we can create our JPA 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="jpaStore">
      <header>messageId</camel:header>
      <to uri="mock:result" />
    </idempotentConsumer>
  </route>
</camelContext>
```

CHAPTER 64. JSCH

JSCH

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>
```

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).

OPTIONS

| Name | Description | Example | Default Value |
|----------|--|---------|---------------|
| username | Specifies the username to use to log in to the remote file system. | | null |
| password | Specifies the password to use to log in to the remote file system. | | null |

| | | | |
|------------------------------|---|--|-------------|
| knownHostsFile | Sets the known_hosts file, so that the scp endpoint can do host key verification. | | null |
| strictHostKeyChecking | Sets whether to use strict host key checking. Possible values are: no , yes | | no |
| chmod | Allows you to set chmod on the stored file. For example chmod=664 . | | null |

LIMITATIONS

Currently camel-jsch only supports a **Producer** (i.e. copy files to another host). The reason is that the scp protocol does not offer the possibility to scan (list) the content of a directory. As such a polling consumer cannot watch for changes and trigger events on changes. If continuous monitoring of a directory on a remote host and secure transfer is required, you can consider using the [sftp protocol](#).

CHAPTER 65. JT400

JT/400 COMPONENT

The `jt400` component allows you to exchanges messages with an AS/400 system using data queues.

URI FORMAT

```
jt400://user:password@system/QSYS.LIB/LIBRARY.LIB/QUEUE.DTAQ[?options]
```

To call a 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&...`

URI OPTIONS

For the data queue message exchange:

| Name | Default value | Description |
|------------------------------------|----------------------|---|
| <code>ccsid</code> | default system CCSID | Specifies the CCSID to use for the connection with the AS/400 system. |
| <code>format</code> | <code>text</code> | Specifies the data format for sending messages valid options are: <code>text</code> (represented by <code>String</code>) and <code>binary</code> (represented by <code>byte[]</code>) |
| <code>consumer.delay</code> | <code>500</code> | Delay in milliseconds between each poll. |
| <code>consumer.initialDelay</code> | <code>1000</code> | Milliseconds before polling starts. |

| | | |
|--------------------------------------|---|---|
| <code>consumer.userFixedDelay</code> | <code>false</code> | <code>true</code> to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |
| <code>guiAvailable</code> | <code>false</code> | Camel 2.8: Specifies whether AS/400 prompting is enabled in the environment running Camel. |
| <code>keyed</code> | <code>false</code> | *Camel 2.10:* Whether to use keyed or non-keyed data queues. |
| <code>searchKey</code> | <code>null</code> | *Camel 2.10:* Search key for keyed data queues. |
| <code>searchType</code> | <code>EQ</code> | *Camel 2.10:* Search type which can be a value of EQ , NE , LT , LE , GT , or GE . |
| <code>connectionPool</code> | <code>AS400ConnectionPool</code> instance | *Camel 2.10:* Reference to an <code>com.ibm.as400.access.AS400ConnectionPool</code> instance in the Registry. This is used for obtaining connections to the AS/400 system. The look up notation ('#' character) should be used. |

For the remote program call (Camel 2.7):

| Name | Default value | Description |
|------------------------------|---------------|---|
| <code>outputFieldsIdx</code> | | Specifies which fields (program parameters) are output parameters. |
| <code>fieldsLength</code> | | Specifies the fields (program parameters) length as in the AS/400 program definition. |

| | | |
|-----------------------|-------------------------------------|---|
| format | text | *Camel 2.10:* Specifies the data format for sending messages valid options are: text (represented by String) and binary (represented by byte[]) |
| guiAvailable | false | *Camel 2.8:* Specifies whether AS/400 prompting is enabled in the environment running Camel. |
| connectionPool | AS400ConnectionPool instance | *Camel 2.10:* Reference to an <code>com.ibm.as400.access.AS400ConnectionPool</code> instance in the Registry. This is used for obtaining connections to the AS/400 system. The look up notation ('#' character) should be used. |

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.

CONNECTION POOL

Available as of Camel 2.10

Connection pooling is in use from Camel 2.10 onwards. You can explicitly configure a connection pool on the `Jt400Component`, or as an `uri` option on the endpoint.

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!

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");
    }
}
```

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/co
        mpute.PGM?fieldsLength=10,10,512&ouputFieldsIdx=2,3").to("direct:play");
    }
}
```

WRITING TO KEYED DATA QUEUES

```
from("jms:queue:input")  
  .to("jt400://username:password@system/lib.lib/MSGINDQ.DTAQ?keyed=true");
```

READING FROM KEYED DATA QUEUES

```
from("jt400://username:password@system/lib.lib/MSGOUTDQ.DTAQ?  
keyed=true&searchKey=MYKEY&searchType=GE")  
  .to("jms:queue:output");
```

CHAPTER 66. KESTREL

KESTREL COMPONENT

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.

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
```

OPTIONS

You can configure properties on each Kestrel endpoint individually by specifying them in the `?parameters` portion of the endpoint URI. Any `?` parameters that are omitted will default to what is configured on the KestrelComponent's base KestrelConfiguration. The following properties may be set on KestrelConfiguration and/or each individual endpoint:

| Option | Default Value | Description |
|----------------------------------|------------------|--|
| <code>concurrentConsumers</code> | <code>1</code> | Specifies the number of concurrent consumer threads. |
| <code>waitTimeMs</code> | <code>100</code> | Specifies the <code>/t=...</code> wait time passed to Kestrel on GET requests. |

NOTE: If `waitTimeMs` is set to zero (or negative), the `/t=...` specifier does not get passed to the server on GET requests. When a queue is empty, the GET call returns immediately with no value. In order to prevent "tight looping" in the polling phase, this component will do a `Thread.sleep(100)` whenever nothing is returned from the GET request (only when nothing is returned). You are highly encouraged to configure a positive non-zero value for `waitTimeMs`.

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>

```

EXAMPLE 1: CONSUMING

```
from("kestrel://kserver02:22133/massive?
concurrentConsumers=10&waitTimeMs=500")
    .bean("myConsumer", "onMessage");

public class MyConsumer {
    public void onMessage(String message) {
        ...
    }
}
```

EXAMPLE 2: PRODUCING

```
public class MyProducer {
    @EndpointInject(uri =
"kestrel://kserver01:22133,kserver02:22133/myqueue")
    ProducerTemplate producerTemplate;

    public void produceSomething() {
        producerTemplate.sendBody("Hello, world.");
    }
}
```

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"/>
    </route>
</camelContext>
```

```

    <to uri="kestrel://ks02:22133/stuff"/>
  </route>
</camelContext>

```

```

public class MyBean {
    public void onMessage(String message) {
        ...
    }
}

```

DEPENDENCIES

The Kestrel component has the following dependencies:

-

spymemcached 2.5 (or greater)

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.



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.

CHAPTER 67. KRATI

KRATI COMPONENT

Available as of Camel 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>
```

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&...

KRATI URI OPTIONS

| Name | Default Value | Description |
|---------------------------|------------------------|---|
| operation | CamelKратиPut | Producer Only. Specifies the type of operation that will be performed to the datastore. Allowed values are CamelKратиPut, CamelKратиGet, CamelKратиDelete & CamelKратиDeleteAll. |
| initialCapacity | 100 | The initial capacity of the store. |
| keySerializer | KратиDefaultSerializer | The serializer that will be used to serialize the key. |
| valueSerializer | KратиDefaultSerializer | The serializer that will be used to serialize the value. |
| segmentFactory | ChannelSegmentFactory | The segment factory to use. Allowed instance classes: ChannelSegmentFactory, MemorySegmentFactory, MappedSegmentFactory & WriteBufferSegmentFactory. |
| hashFunction | FnvHashFunction | The hash function to use. Allowed instance classes: FnvHashFunction, Fnv1Hash32, FnvHash64, Fnv1aHash32, Fnv1aHash64, JenkisHashFunction, MurmurHashFunction |
| maxMessagesPerPoll | | Camel 2.10.5/2.11.1: 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. |

For producer endpoint you can override all of the above URI options by passing the appropriate headers to the message.

MESSAGE HEADERS FOR DATASTORE

| Header | Description |
|----------------------------|---|
| CamelKратиOperation | The operation to be performed on the datastore. The valid options are <ul style="list-style-type: none"> • CamelKратиAdd • CamelKратиGet • CamelKратиDelete • CamelKратиDeleteAll |
| CamelKратиKey | The key. |
| CamelKратиValue | The value. |

USAGE SAMPLES

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/producerstest");
```

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>
```

```
</route>
```

EXAMPLE 2: GETTING/READING FROM A DATASTORE

This example will show you how you can read the content of a datastore.

```
from("direct:get")
    .setHeader(KratiConstants.KRATI_OPERATION,
constant(KratiConstants.KRATI_OPERATION_GET))
    .to("krati:target/test/producerstest");
```

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>
```

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>
```

IDEMPOTENT REPOSITORY

As already mentioned this component also offers an idempotent repository which can be used for filtering out duplicate messages.

```
from("direct://in").idempotentConsumer(header("messageId"), new  
KратиIdempotentRepositroy("/tmp/idempotent").to("log://out");
```

SEE ALSO

[Kрати Websitre](#)

CHAPTER 68. LANGUAGE

LANGUAGE

Available as of Camel 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.

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]
```

URI FORMAT

```
language://languageName[:script][?options]
```

URI OPTIONS

The component supports the following options.

| Name | Default Value | Type | Description |
|--------------|---------------|--------|---|
| languageName | null | String | The name of the Language to use, such as simple , groovy , javascript etc. This option is mandatory. |

| | | | |
|---------------------|-------------|----------------|--|
| script | null | String | The script to execute. |
| transform | true | boolean | Whether or not the result of the script should be used as the new message body. By setting to false the script is executed but the result of the script is discarded. |
| contentCache | true | boolean | Camel 2.9: Whether to cache the script if loaded from a resource. Note: from Camel 2.10.3 a cached script can be forced to reload at runtime via JMX using the <code>clearContentCache</code> operation. |

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. |

EXAMPLES

For example you can use the [Simple](#) language to Message TranslatorMessage Translator a message:

```
String script = URLEncoder.encode("Hello ${body}", "UTF-8");
from("direct:start").to("language:simple:" + script).to("mock:result");
```

In case you want to convert the message body type you can do this as well:

```
String script = URLEncoder.encode("${mandatoryBodyAs(String)}", "UTF-8");
from("direct:start").to("language:simple:" + script).to("mock:result");
```

You can also use the [Groovy](#) language, such as this example where the input message will be multiplied with 2:

```
from("direct:start").to("language:groovy:request.body *
2").to("mock:result");
```

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);
```

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:

```
from("direct:start")
    // load the script from the classpath
    .to("language:simple:classpath:org/apache/camel/component/language/mysimple
script.txt")
    .to("mock:result");
```


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("direct:start")
    // the script will be loaded on each message, as we disabled cache
    .to("language:simple:file:target/script/myscript.txt?
contentCache=false")
    .to("mock:result");
```

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:

```
from("direct:start")
    // load the script from the classpath

.to("language:simple:resource:classpath:org/apache/camel/component/language/
mysimplescript.txt")
    .to("mock:result");
```

- [Languages](#)
- [Routing Slip](#)[Routing Slip](#)
- [Dynamic Router](#)[Dynamic Router](#)

CHAPTER 69. LDAP

LDAP COMPONENT

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.

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&...`

OPTIONS

| Name | Default Value | Description |
|-----------------------|------------------------|---|
| <code>base</code> | <code>ou=system</code> | The base DN for searches. |
| <code>scope</code> | <code>subtree</code> | Specifies how deeply to search the tree of entries, starting at the base DN. Value can be object , onelevel , or subtree . |
| <code>pageSize</code> | No paging used. | 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, an LdapContext (subclass of DirContext) has to be passed in as ldapServerBean (otherwise an exception is thrown) |

| | | |
|---------------------------------|---|---|
| <code>returnedAttributes</code> | Depends on LDAP Server (could be all or none) . | Comma-separated list of attributes that should be set in each entry of the result |
|---------------------------------|---|---|

RESULT

The result is returned in the Out body as a `ArrayList<javax.naming.directory.SearchResult>` object.

DIRCONTEXT

The URI, `ldap:ldapservers`, references a Spring bean with the ID, `ldapservers`. The `ldapservers` bean may be defined as follows:

```
<bean id="ldapservers" 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.

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:ldapservers?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=*)
```

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();
```

CHAPTER 70. LEVELDB

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

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. |
| 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.

WHAT IS PRESERVED WHEN PERSISTING

`LevelDBAggregationRepository` 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.

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 |
|--|---------|--|
| <code>Exchange.REDELIVERED</code> | Boolean | Is set to true to indicate the Exchange is being redelivered. |
| <code>Exchange.REDELIVERY_COUNTER</code> | 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](#).

USING LEVELDBAGGREGATIONREPOSITORY IN JAVA DSL

In this example we want to persist aggregated messages in the `target/data/leveldb.dat` file.

```
public void configure() throws Exception {
    // create the leveldb repo
    LevelDBAggregationRepository repo = new
    LevelDBAggregationRepository("repo1", "target/data/leveldb.dat");
```

```

// here is the Camel route where we aggregate
from("direct:start")
    .aggregate(header("id"), new MyAggregationStrategy())
        // use our created leveldb repo as aggregation repository
        .completionSize(5).aggregationRepository(repo)
        .to("mock:aggregated");
}

```

USING LEVELDBAGGREGATIONREPOSITORY IN SPRING XML

The same example but using Spring XML instead:

```

<!-- a persistent aggregation repository using camel-leveldb -->
<bean id="repo"
class="org.apache.camel.component.leveldb.LevelDBAggregationRepository">
    <!-- store the repo in the leveldb.dat file -->
    <property name="persistentFileName" value="target/data/leveldb.dat"/>
    <!-- and use repo2 as the repository name -->
    <property name="repositoryName" value="repo2"/>
</bean>

<!-- aggregate the messages using this strategy -->
<bean id="myAggregatorStrategy"
class="org.apache.camel.component.leveldb.LevelDBSpringAggregateTest$MyAggregationStrategy"/>

<!-- this is the camel routes -->
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

    <route>
        <from uri="direct:start"/>
        <!-- aggregate using our strategy and leveldb repo, and complete
when we have 5 messages aggregated -->
        <aggregate strategyRef="myAggregatorStrategy"
aggregationRepositoryRef="repo" completionSize="5">
            <!-- correlate by header with the key id -->
            <correlationExpression><header>id</header>
        </correlationExpression>
        <!-- send aggregated messages to the mock endpoint -->
        <to uri="mock:aggregated"/>
    </aggregate>
    </route>

</camelContext>

```

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>
```

-

Aggregator

-

HawtDB

-

Components

CHAPTER 71. LIST

LIST COMPONENT

deprecated: is renamed to the [Browse](#) component in Apache Camel 2.0

The List 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.

URI FORMAT

```
list:someName
```

Where someName can be any string to uniquely identify the endpoint.

SAMPLE

In the route below we have the list component to be able to browse the Exchanges that is passed through:

```
from("activemq:order.in").to("list:orderReceived").to("bean:processOrder")  
;
```

Then we will be able to inspect the received exchanges from java code:

```
private CamelContext context;  
  
public void inspectRecievedOrders() {  
    BrowsableEndpoint browse =  
context.getEndpoint("list: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();  
        ...  
    }  
}
```

See also:

- [Browse](#)

CHAPTER 72. LOG

LOG COMPONENT

The `log:` component logs message exchanges to the underlying logging mechanism.

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&...`

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 Apache Camel also ships with the Throughput logger, which is used whenever the `groupSize` option is specified.

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](#).

OPTIONS

| Option | Default | Type | Description |
|--------------------|-------------------|---------------------|--|
| <code>level</code> | <code>INFO</code> | <code>String</code> | Logging level to use. Possible values: ERROR, WARN, INFO, DEBUG, TRACE, OFF |

| | | | |
|------------------------|-------------|----------------|--|
| marker | null | String | Camel 2.9: An optional Marker name to use. |
| groupSize | null | Integer | An integer that specifies a group size for throughput logging. |
| groupInterval | null | Integer | If specified will group message stats by this time interval (in millis) |
| groupDelay | 0 | Integer | Set the initial delay for stats (in millis) |
| groupActiveOnly | true | boolean | If true, will hide stats when no new messages have been received for a time interval, if false, show stats regardless of message traffic |

FORMATTING

The log formats the execution of exchanges to log lines. By default, the log uses `LogFormatter` to format the log output, where `LogFormatter` has the following options:

| Option | Default | Description |
|----------------------------|----------------|---|
| showAll | false | Quick option for turning all options on (multiline, maxChars has to be manually set if to be used). |
| showExchangeId | false | Show the unique exchange ID. |
| showExchangePattern | true | Shows the Message Exchange Pattern (or MEP for short). |
| showProperties | false | Show the exchange properties. |
| showHeaders | false | Show the In message headers. |

| | | |
|----------------------------------|--------------------|--|
| <code>showBodyType</code> | <code>true</code> | Show the In body Java type. |
| <code>showBody</code> | <code>true</code> | Show the In body. |
| <code>showOut</code> | <code>false</code> | If the exchange has an Out message, show the Out message. |
| <code>showException</code> | <code>false</code> | Apache Camel 2.0: If the exchange has an exception, show the exception message (no stack trace). |
| <code>showCaughtException</code> | <code>false</code> | Apache Camel 2.0: If the exchange has a caught exception, show the exception message (no stack trace). A caught exception is stored as a property on the exchange and for instance a doCatch can catch exceptions. See Try Catch Finally . |
| <code>showStackTrace</code> | <code>false</code> | Apache Camel 2.0: Show the stack trace, if an exchange has an exception. Only effective if one of showAll , showException or showCaughtException are enabled. |
| <code>showFiles</code> | <code>false</code> | Camel 2.9: Whether Camel should show file bodies or not (eg such as <code>java.io.File</code>). |
| <code>showFuture</code> | <code>false</code> | Whether Camel should show java.util.concurrent.Future bodies or not. If enabled Camel could potentially wait until the Future task is done. Will by default not wait. |

| | | |
|--------------------------|--------------------|---|
| <code>showStreams</code> | <code>false</code> | Camel 2.8: Whether Camel should show stream bodies or not (eg such as <code>java.io.InputStream</code>). 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 . |
| <code>multiline</code> | <code>false</code> | If true , each piece of information is logged on a new line. |
| <code>maxChars</code> | | Limits the number of characters logged per line. The default value is 10000 from Camel 2.9 onwards. |

LOGGING STREAM BODIES

For older versions of Camel that do not support the `showFiles` or `showStreams` properties above, you can set the following property instead on the `CamelContext` to log both stream and file bodies:

```
camelContext.getProperties().put(Exchange.LOG_DEBUG_BODY_STREAMS, true);
```

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>
```

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");
```

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");
```

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:processOrder");
```

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"
```

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>
```

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 `logForamtter` 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&aram2=100"/>  
...  
<to uri="log:bar?param1=bar&aram2=200"/>
```

CHAPTER 73. LUCENE

LUCENE (INDEXER AND SEARCH) COMPONENT

Available as of Apache Camel 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 Apache Camel

This component only supports producer endpoints.

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&...

INSERT OPTIONS

| Name | Default Value | Description |
|------------------|-------------------------|---|
| analyzer | StandardAnalyzer | 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 |
| indexPath | ./indexPath | A file system directory in which index files are created upon analysis of the document by the specified analyzer |
| srcDir | null | An optional directory containing files to be used to be analyzed and added to the index at producer startup. |

QUERY OPTIONS

| Name | Default Value | Description |
|------------------|-------------------------|---|
| analyzer | StandardAnalyzer | 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 |
| indexPath | ./indexPath | A file system directory in which index files are created upon analysis of the document by the specified analyzer |
| maxHits | 10 | An integer value that limits the result set of the search operation |

MESSAGE HEADERS

| Header | Description |
|--------|---|
| QUERY | The Lucene Query to performed on the index. The query may include wildcards and phrases |

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

LUCENE PROCESSOR

There is a processor called `LuceneQueryProcessor` available to perform queries against lucene without the need to create a producer.

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");
    }
};
```

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());

```

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");
    }
};

```

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 74. MAIL

MAIL COMPONENT

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>
```

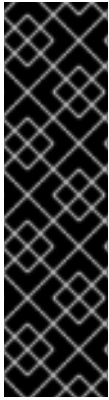
**GERONIMO MAIL .JAR**

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.

POP3 OR IMAP

POP3 has some limitations and end users are encouraged to use IMAP if possible.



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.

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&...`

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
```

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 |

OPTIONS

| Property | Default | Description |
|----------|----------------------------------|--|
| host | | The host name or IP address to connect to. |
| port | See DefaultPorts | The TCP port number to connect on. |
| username | | The user name on the email server. |
| password | null | The password on the email server. |

| | | |
|------------------------|------------------------|---|
| ignoreUriScheme | false | If false , Camel uses the scheme to determine the transport protocol (POP, IMAP, SMTP etc.) |
| defaultEncoding | null | The default encoding to use for Mime Messages. |
| contentType | text/plain | The mail message content type. Use text/html for HTML mails. |
| folderName | INBOX | The folder to poll. |
| destination | username@host | @deprecated Use the to option instead. The TO recipients (receivers of the email). |
| to | username@host | The TO recipients (the receivers of the mail). Separate multiple email addresses with a comma. |
| replyTo | alias@host | As of Camel 2.8.4, 2.9.1\+ , the Reply-To recipients (the receivers of the response mail). Separate multiple email addresses with a comma. |
| CC | null | The CC recipients (the receivers of the mail). Separate multiple email addresses with a comma. |
| BCC | null | The BCC recipients (the receivers of the mail). Separate multiple email addresses with a comma. |
| from | camel@localhost | The FROM email address. |
| subject | | As of Camel 2.3 , the Subject of the message being sent. Note: Setting the subject in the header takes precedence over this option. |

| | | |
|---------------|--------------|---|
| delete | false | 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. |
| unseen | true | It is possible to configure a consumer endpoint so that it processes only unseen messages (that is, new messages) or all messages. Note that Camel always skips deleted messages. The default option of true will filter to only unseen messages. POP3 does not support the SEEN flag, so this option is not supported in POP3; use IMAP instead. Important: This option is not in use if you also use searchTerm options. Instead if you want to disable unseen when using searchTerm 's then add searchTerm.unseen=false as a term. |
| copyTo | null | Camel 2.10: Consumer only. 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. |

| | | |
|-------------------------------|---------------------------------|--|
| fetchSize | \-1 | 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. |
| alternativeBodyHeader | CamelMailAlternativeBody | 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. |
| debugMode | false | Enable debug mode on the underlying mail framework. The SUN Mail framework logs the debug messages to System.out by default. |
| connectionTimeout | 30000 | The connection timeout in milliseconds. Default is 30 seconds. |
| consumer.initialDelay | 1000 | Milliseconds before the polling starts. |
| consumer.delay | 60000 | Camel will poll the mailbox only once a minute by default to avoid overloading the mail server. |
| consumer.useFixedDelay | false | Set to true to use a fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |

| | | |
|-----------------------------|--------------------|--|
| <code>disconnect</code> | <code>false</code> | Camel 2.8.3/2.9: Whether the consumer should disconnect after polling. If enabled this forces Camel to connect on each poll. |
| <code>closeFolder</code> | <code>true</code> | Camel 2.10.4: 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. |
| <code>mail.XXX</code> | <code>null</code> | Set any additional java mail properties . For instance if you want to set a special property when using POP3 you can now provide the option directly in the URI such as: mail.pop3.forgettopheaders=true . You can set multiple such options, for example: mail.pop3.forgettopheaders=true&mail.mime.encodefilename=true . |
| <code>mapMailMessage</code> | <code>true</code> | Camel 2.8: 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) . |

| | | |
|---------------------------------|--------------|--|
| maxMessagesPerPoll | 0 | 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. |
| javaMailSender | null | Specifies a pluggable org.springframework.mail.javamail.JavaMailSender instance in order to use a custom email implementation. If none provided, Camel uses the default org.springframework.mail.javamail.JavaMailSenderImpl . |
| ignoreUnsupportedCharset | false | 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. |
| sslContextParameters | null | Camel 2.10: 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 . |
| searchTerm | null | Camel 2.11: 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. See further below for examples. |

| | | |
|----------------|------|---|
| searchTerm.xxx | null | Camel 2.11: To configure search terms directly from the endpoint uri, which supports a limited number of terms defined by the <code>org.apache.camel.component.mail.SimpleSearchTerm</code> class. See further below for examples. |
|----------------|------|---|

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.

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(...)
    .to("smtps://smtp.google.com?
username=user@gmail.com&password=password&sslContextParameters=#sslContext
Parameters");

```

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=#sslContext
Parameters"/>...

```

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).

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:

```

from("direct:a").setHeader("subject",
constant(subject)).to("smtp://james2@localhost");

```

The same applies for other `MimeMessage` headers such as recipients, so you can use a header property as To:

```

Map<String, Object> map = new HashMap<String, Object>();
map.put("To", "davsclaus@apache.org");
map.put("From", "jstrachan@apache.org");
map.put("Subject", "Camel rocks");

String body = "Hello Claus.\nYes it does.\n\nRegards James.";
template.sendBodyAndHeaders("smtp://davsclaus@apache.org", body, map);

```

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.

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);
```

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.

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");
```

SUN JAVAMAIL

[SUN JavaMail](#) 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.

- [SUN POP3 API](#)
- [SUN IMAP API](#)
- And generally about the [MAIL Flags](#)

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:

-

```
// all the recipients of this mail are:
// To: camel@riders.org , easy@riders.org
// CC: me@you.org
// BCC: someone@somewhere.org
String recipients =
"&To=camel@riders.org,easy@riders.org&CC=me@you.org&BCC=someone@somewhere.
org";

from("direct:a").to("smtp://you@mymailserver.com?
password=secret&From=you@apache.org" + recipients);
```

SENDING MAIL WITH ATTACHMENT SAMPLE



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.

```
// create an exchange with a normal body and attachment to be produced as
email
Endpoint endpoint = context.getEndpoint("smtp://james@mymailserver.com?
password=secret");

// create the exchange with the mail message that is multipart with a file
and a Hello World text/plain message.
Exchange exchange = endpoint.createExchange();
Message in = exchange.getIn();
in.setBody("Hello World");
in.addAttachment("logo.jpeg", new DataHandler(new
FileDataSource("src/test/data/logo.jpeg")));

// create a producer that can produce the exchange (= send the mail)
Producer producer = endpoint.createProducer();
// start the producer
producer.start();
```

```
// and let it go (processes the exchange by sending the email)
producer.process(exchange);
```

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=[...
```

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.

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:

```
from("pop3://james@mymailserver.com?password=secret&consumer.delay=1000")
  .to("log:email")
  // use the SplitAttachmentsExpression which will split the message per
  attachment
  .split(new SplitAttachmentsExpression())
  // each message going to this mock has a single attachment
  .to("mock:split")
  .end();
```

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>
```

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://mymailserver?
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://mymailserver?
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://mymailserver?
username=foo&password=secret&searchTerm.subject=Camel&searchTerm.fromSentD
ate=now-24h"/>
  <to uri="bean:myBean"/>
</route>
```

| Option | Default | Description |
|---------------|-------------|--|
| unseen | true | Whether to limit by unseen mails only. |
| subjectOrBody | null | To limit by subject or body to contain the word. |
| subject | null | The subject must contain the word. |
| body | null | The body must contain the word. |
| from | null | The mail must be from a given email pattern. |
| to | null | The mail must be to a given email pattern. |

| | | |
|--------------|-------------|---|
| fromSentDate | null | The mail must be sent after or equals (GE) a given date. The date pattern is yyyy-MM-dd HH:mm:ss , eg use "2012-01-01 00:00:00" to be from the year 2012 onwards. You can use "now" for current timestamp. The "now" syntax supports an optional offset, that can be specified as either + or - with a numeric value. For example for last 24 hours, you can use "now - 24h" or without spaces "now-24h" . Notice that Camel supports shorthands for hours, minutes, and seconds. |
| toSentDate | null | The mail must be sent before or equals (BE) a given date. The date pattern is yyyy-MM-dd HH:mm:ss , eg use "2012-01-01 00:00:00" to be before the year 2012. You can use "now" for current timestamp. The "now" syntax supports an optional offset, that can be specified as either + or - with a numeric value. For example for last 24 hours, you can use "now - 24h" or without spaces "now-24h" . Notice that Camel supports shorthands for hours, minutes, and seconds. |

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://mymailserver?
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();
```

CHAPTER 75. MASTER COMPONENT

Abstract

The Master component provides a way to ensure that only a single consumer in a cluster consumes from a given endpoint; with automatic failover if that JVM dies. This feature can be useful if you need to consume from a legacy back-end that doesn't support concurrent consumption or, due to commercial or stability reasons, you can have only a single connection to the back-end at any point in time.

DEPENDENCIES

The Master component can only be used in the context of a fabric-enabled Red Hat JBoss Fuse container. You must ensure that the `fabric-camel` feature is installed.

In the context of Fabric, you install a feature by adding it to the relevant profile. For example, if you are using a profile called `my-master-profile`, you would add the `fabric-camel` feature by entering the following console command:

```
karaf@root> fabric:profile-edit --features fabric-camel my-master-profile
```

URI FORMAT

A Master endpoint can only be used as a *consumer endpoint*. It has the following URI format:

```
master:ClusterID:EndpointURI[?Options]
```

Where the `URI`, `EndpointURI`, is published in the fabric registry and associated with the `ClusterId` cluster.

URI OPTIONS

The Master component itself does *not* support any URI options. Any options on the URI are, therefore, applied to the specified consumer endpoint, `EndpointURI`.

HOW TO USE THE MASTER COMPONENT

The Master component is useful in cases where you need to poll messages from an endpoint, but you are only allowed to make *one connection* to that endpoint. In this case, you can use the Master component to define a failover cluster of consumer endpoints. Each Master endpoint in the cluster is capable of consuming messages from the given endpoint, but only *one* of the Master endpoints is active at any time (the master), while the other Master endpoints are waiting (the slaves).

For example, to set up a cluster of Master endpoints that can consume from the `seda:bar` endpoint, you would proceed as follows:

1. Define the Master endpoints with the following URI (where each endpoint in the cluster uses *exactly* the same URI):

```
master:mysedalock:seda:bar
```

Each of the Master endpoints in the cluster tries to get the `mysedalock` lock (implemented as a key in the Zookeeper registry). The Master endpoint that succeeds in getting the lock becomes active (the master) and starts consuming messages from the `seda:bar` endpoint. The other Master endpoints enter a waiting state and continue to try the lock (the slaves).

2. You must remember to include the `fabric-camel` feature in the profile that deploys a Master endpoint.

3. In the Blueprint XML file that defines a Master endpoint, you must define a reference to the Apache Curator service (the Master component uses this service to connect to the Zookeeper registry). For example, in Blueprint XML you can define the Curator reference as follows:

```
<?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">
  ...
  <reference id="curator"
            interface="org.apache.curator.framework.CuratorFramework"/>
  ...
</blueprint>
```

- 4.

In Blueprint XML, you can define a Master endpoint at the start of a Camel route, as follows:

```
<?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">
  ...
  <camelContext id="camel"
    xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="master:mysedalock:seda:bar"/>
      ...
    </route>
  </camelContext>
  ...
</blueprint>
```

EXAMPLE OF A MASTER-SLAVE CLUSTER POLLING A JMS ACTIVEMQ BROKER

For example, a typical way to use the Master component is to create a cluster of exclusive consumers for consuming messages from a JMS queue. Only one of the Master endpoints consumes from the queue at any time, and if that Master endpoint goes down, one of the other Master endpoints takes over (becomes the new master). In this example, we create a cluster of two Camel routes, where each route starts with a Master endpoint that is capable of consuming from the specified queue, FABRIC.DEMO.

STEPS TO CREATE A CLUSTER THAT POLLS MESSAGES FROM AN ACTIVEMQ BROKER

To create a master-slave cluster that polls messages from an ActiveMQ broker, based on the Master component, perform the following steps:

1. If you do not already have a fabric, enter the following console command to create one:

```
JBossFuse:karaf@root> fabric:create --new-user AdminUser --new-user-
password AdminPass
--zookeeper-password ZooPass --wait-for-provisioning
```

The `--new-user` and `--new-user-password` options specify the credentials for a new administrator user. The Zookeeper password is used to protect sensitive data in the Fabric registry service (all of the nodes under `/fabric`).

**NOTE**

If you use a VPN (virtual private network) on your local machine, it is advisable to log off VPN *before* you create the fabric and to stay logged off while you are using the local container. A local Fabric Server is permanently associated with a fixed IP address or hostname. If VPN is enabled when you create the fabric, the underlying Java runtime is liable to detect and use the VPN hostname instead of your permanent local hostname. This can also be an issue with multi-homed machines. To be absolutely sure about the hostname, you could specify the IP address explicitly—see [chapter "Creating a New Fabric" in "Fabric Guide"](#).

2.

For this example, you must have access to a running instance of an Apache ActiveMQ broker and you must know the IP port of the broker's OpenWire connector. For example, you might get access to an ActiveMQ broker in one of the following ways:

•

You just created the fabric on a clean installation of JBoss Fuse (after a cold restart). In this case, the root container ought to include the `jboss-fuse-full` profile by default. You can check whether this is the case by entering the `fabric:container-list` console command, as follows:

```
JBossFuse:karaf@root> fabric:container-list
[id]    [version] [connected] [profiles]
[provision status]
root*   1.0      true      fabric, fabric-ensemble-0000-1,
jboss-fuse-full  success
```

By default, the `jboss-fuse-full` profile instantiates an ActiveMQ broker that listens on port 61616. You can use this broker for the current example.

•

If no broker is running in the root container (or any other container), you can quickly install a broker into a new fabric child container, `broker1`, by entering the following fabric command at the console prompt:

```
JBossFuse:karaf@root> fabric:container-create-child --profile mq-
default root broker1
```

In this case, you can use the browser-based Fuse Management

Console to discover the IP port of the OpenWire connector on the broker.

3.

Create the master-example profile, which will be used to deploy a simple Apache Camel route that uses the Master component. Enter the following console command to create the profile:

```
JBossFuse:karaf@root> fabric:profile-create --parents default
master-example
```

4.

Add the requisite Karaf features to the master-example profile. Enter the following console commands:

```
fabric:profile-edit --features fabric-camel master-example
fabric:profile-edit --features activemq-camel master-example
```

5.

Define the simple Camel route as a resource in the master-example profile. Invoke the built-in text editor to create a new camel.xml resource, as follows:

```
fabric:profile-edit --resource camel.xml master-example
```

Copy and paste the following content into the built-in text editor:

```
<?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">

  <reference id="curator"
interface="org.apache.curator.framework.CuratorFramework"/>

  <camelContext id="camel"
xmlns="http://camel.apache.org/schema/blueprint">
    <route id="fabric-server">
      <from
uri="master:lockhandle:activemq:queue:FABRIC.DEMO"/>
      <log message="Message received : ${body}"/>
    </route>
  </camelContext>

  <bean id="activemq"
```

```

class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="brokerURL"
value="tcp://localhost:OpenWirePort"/>
  <property name="userName" value="UserName"/>
  <property name="password" value="Password"/>
</bean>

</blueprint>

```

Remember to customize the route configuration by replacing *OpenWirePort* with the port number of the OpenWire connector on the broker, and by replacing *UserName* and *Password* by any valid JAAS credentials on the container (for example, you could substitute the *AdminUser* and *AdminPass* credentials created in Step 1 of these instructions).

To save and exit from the text editor, type Ctrl-S, Ctrl-X.

6.

Configure the master-example profile to deploy the camel.xml resource as an OSGi bundle. Enter the following console command to create a new entry in the master-example agent properties:

```

fabric:profile-edit --bundles blueprint:profile:camel.xml master-
example

```



NOTE

The `blueprint:` prefix tells Fabric to deploy the specified resource as a Blueprint XML file, and the `profile:` prefix tells Fabric where to find the resource (that is, in the current version of the current profile).

7.

Create two new child containers, so that you can deploy the master-example profile as a cluster (one master and one slave). Enter the following console command:

```

fabric:container-create-child root child 2

```

8.

Now deploy both the master-example profile and the mq-client profile to each of the child containers, as follows:

```
fabric:container-change-profile child1 master-example mq-client
fabric:container-change-profile child2 master-example mq-client
```

9.

If you now send some messages to the FABRIC.DEMO queue on the broker, the messages are consumed by one (and only one) of the deployed master endpoints. For example, you can easily create and send messages to the broker using the browser-based Fuse Management console.

10.

If you stop the container that hosts the current master (initially, the child1 container), the slave will be promoted to be the new master (in the child2 container) and will start consuming messages from the FABRIC.DEMO queue. For example, assuming that child2 contains the current master, you can stop it by entering the following console command:

```
fabric:container-stop child2
```

OSGI BUNDLE PLUG-IN CONFIGURATION

When defining an OSGi bundle that uses Master endpoints, the Import-Package bundle header must be configured to import the following Java packages:

```
io.fabric8.zookeeper.spring
io.fabric8.zookeeper
```

For example, assuming that you use Maven to build your application, [Example 75.1, “Maven Bundle Plug-In Configuration”](#) shows how you can configure the Maven bundle plug-in to import the required packages.

Example 75.1. Maven Bundle Plug-In Configuration

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>>true</extensions>
        <configuration>
          <instructions>
            <Bundle-
SymbolicName>${project.groupId}.${project.artifactId}</Bundle-
```

```
SymbolicName>
    <Import-Package>
        io.fabric8.zookeeper.spring,
        io.fabric8.zookeeper,
        *
    </Import-Package>
</instructions>
</configuration>
</plugin>
</plugins>
</build>
...
</project>
```

CHAPTER 76. MINA [DEPRECATED]

MINA COMPONENT

**DEPRECATED**

The MINA component is deprecated, because the Apache Mina 1.x project is end-of-life. Use [Netty](#) instead.

The `mina:` component is a transport for working with [Apache MINA](#)

URI FORMAT

```
mina:tcp://hostname[:port][?options]
mina:udp://hostname[:port][?options]
mina: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. See the [MINA VM-Pipe API documentation](#) for details.

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&...`

OPTIONS

| Option | Default Value | Description |
|--------------------------------|----------------------|--|
| <code>codec</code> | <code>null</code> | You can refer to a named ProtocolCodecFactory instance in your Registry such as your Spring ApplicationContext , which is then used for the marshalling. |
| <code>codec</code> | <code>null</code> | Apache Camel 2.0: You must use the <code>#</code> notation to look up your codec in the Registry . For example, use <code>#myCodec</code> to look up a bean with the <code>id</code> value, <code>myCodec</code> . |
| <code>disconnect</code> | <code>false</code> | Camel 2.3: Whether or not to disconnect(close) from Mina session right after use. Can be used for both consumer and producer. |
| <code>textline</code> | <code>false</code> | 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 <code>false</code> , then Object Serialization is assumed over TCP. |
| <code>textlineDelimiter</code> | <code>DEFAULT</code> | Only used for TCP and if <code>textline=true</code> . Sets the text line delimiter to use. Possible values are: DEFAULT , AUTO , WINDOWS , UNIX or MAC . If none provided, Apache Camel will use DEFAULT . This delimiter is used to mark the end of text. |

| | | |
|----------------------------|------------------------|---|
| sync | true | You can configure the exchange pattern to be either InOnly (default) or InOut. Setting sync=true means a synchronous exchange (InOut), where the client can read the response from MINA (the exchange Out message). The default value has changed in Apache Camel 1.5 to true . In older releases, the default value is false . |
| lazySessionCreation | <i>See description</i> | Sessions can be lazily created to avoid exceptions, if the remote server is not up and running when the Apache Camel producer is started. From Apache Camel 2.0 onwards, the default is true . In Apache Camel 1.x, the default is false . |
| timeout | 30000 | 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. The timeout is only used for Mina producer. |
| encoding | <i>JVM Default</i> | You can configure the encoding (a charset name) to use for the TCP textline codec and the UDP protocol. If not provided, Apache Camel will use the JVM default Charset . |
| transferExchange | false | 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 <i>serializable</i> . Apache Camel will exclude any non-serializable objects and log it at WARN level. |

| | | |
|-----------------------------|--------------|--|
| minaLogger | false | You can enable the Apache MINA logging filter. Apache MINA uses slf4j logging at INFO level to log all input and output. |
| filters | null | As of 2.0, you can set a list of Mina IoFilters to register. The filters value must be one of the following: <ul style="list-style-type: none"> • Camel 2.2: comma-separated list of bean references (e.g. #filterBean1,#filterBean2) where each bean must be of type org.apache.mina.common.IoFilter. • Camel 2.0: a reference to a bean of type List<org.apache.mina.common.IoFilter>. |
| encoderMaxLineLength | -1 | As of 2.1, you can set the textline protocol encoder max line length. By default the default value of Mina itself is used which are Integer.MAX_VALUE . |
| decoderMaxLineLength | -1 | As of 2.1, you can set the textline protocol decoder max line length. By default the default value of Mina itself is used which are 1024. |
| allowDefaultCodec | true | 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. |

| | | |
|----------------------------------|-------------------|---|
| <code>disconnectOnNoReply</code> | <code>true</code> | Camel 2.3: If sync is enabled then this option dictates <code>MinaConsumer</code> if it should disconnect where there is no reply to send back. |
| <code>noReplyLogLevel</code> | <code>WARN</code> | Camel 2.3: If sync is enabled this option dictates <code>MinaConsumer</code> which logging level to use when logging a there is no reply to send back. Values are: FATAL , ERROR , INFO , DEBUG , OFF . |

USING A CUSTOM CODEC

See the [Mina documentation](#) 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.

SAMPLE WITH SYNC=FALSE

In this sample, Apache 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("mina:tcp://localhost:6200?
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("mina:tcp://localhost:6200?textline=true&sync=false",
"Hello World");

assertMockEndpointsSatisfied();
```

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("mina:tcp://localhost:6201?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("mina:tcp://localhost:6201?
textline=true&sync=true", "World");
assertEquals("Bye World", response);
```

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="mina: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"
}
```

CONFIGURING MINA ENDPOINTS USING SPRING BEAN STYLE

Configuration of Mina endpoints is now possible using regular Spring bean style configuration in the Spring DSL.

However, in the underlying Apache Mina toolkit, it is relatively difficult to set up the acceptor and the connector, because you can *not* use simple setters. To resolve this difficulty, we leverage the `MinaComponent` as a Spring factory bean to configure this for us. If you really need to configure this yourself, there are setters on the `MinaEndpoint` to set these when needed.

The sample below shows the factory approach:

```
<!-- Creating mina endpoints is a bit complex so we reuse MinaComponnet
      as a factory bean to create our endpoint, this is the easiest to do -
->
<bean id="myMinaFactory"
class="org.apache.camel.component.mina.MinaComponent">
    <!-- we must provide a camel context so we refer to it by its id -->
    <constructor-arg index="0" ref="myCamel"/>
</bean>

<!-- This is our mina endpoint configured with spring, we will use the
factory above
      to create it for us. The goal is to invoke the createEndpoint method
with the
      mina configuration parameter we defined using the constructor-arg
option -->
<bean id="myMinaEndpoint"
      factory-bean="myMinaFactory"
      factory-method="createEndpoint">
    <!-- and here we can pass it our configuration -->
    <constructor-arg index="0" ref="myMinaConfig"/>
</bean>

<!-- this is our mina configuration with plain properties -->
<bean id="myMinaConfig"
class="org.apache.camel.component.mina.MinaConfiguration">
    <property name="protocol" value="tcp"/>
    <property name="host" value="localhost"/>
    <property name="port" value="1234"/>
    <property name="sync" value="false"/>
</bean>
```

And then we can refer to our endpoint directly in the route, as follows:

```
<route>
    <!-- here we route from or mina endpoint we have defined above -->
    <from ref="myMinaEndpoint"/>
    <to uri="mock:result"/>
</route>
```

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 Apache 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("mina: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(MinaConstants.MINA_CLOSE_SESSION_WHEN_COMPLETE
, true);
    }
});

```

GET THE IOSESSION FOR MESSAGE

Available since Apache Camel 2.1 You can get the `IoSession` from the message header with this key `MinaEndpoint.HEADER_MINA_IOSESSION`, and also get the local host address with the key `MinaEndpoint.HEADER_LOCAL_ADDRESS` and remote host address with the key `MinaEndpoint.HEADER_REMOTE_ADDRESS`.

CONFIGURING MINA FILTERS

Available since Apache Camel 2.0

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`.



NOTE

If using the `SslFilter` you need to add the `mina-filter-ssl` JAR to the classpath.

For instance, the example below will send a keep-alive message after 10 seconds of inactivity:

```
public class KeepAliveFilter extends IoFilterAdapter {
    @Override
    public void sessionCreated(NextFilter nextFilter, IoSession session)
        throws Exception {
        session.setIdleTime(IdleStatus.BOTH_IDLE, 10);

        nextFilter.sessionCreated(session);
    }

    @Override
    public void sessionIdle(NextFilter nextFilter, IoSession session,
        IdleStatus status) throws Exception {
        session.write("NOOP"); // NOOP is a FTP command for keep alive
        nextFilter.sessionIdle(session, status);
    }
}
```

As Apache Camel Mina may use a request-reply scheme, the endpoint as a client would like to drop some message, such as greeting when the connection is established. For example, when you connect to an FTP server, you will get a 220 message with a greeting (220 Welcome to Pure-FTPd). If you don't drop the message, your request-reply scheme will be broken.

```
public class DropGreetingFilter extends IoFilterAdapter {

    @Override
    public void messageReceived(NextFilter nextFilter, IoSession session,
        Object message) throws Exception {
        if (message instanceof String) {
            String ftpMessage = (String) message;
            // "220" is given as greeting. "200 Zzz" is given as a
response to "NOOP" (keep alive)
            if (ftpMessage.startsWith("220") || or
ftpMessage.startsWith("200 Zzz")) {
                // Dropping greeting
                return;
            }
        }
        nextFilter.messageReceived(session, message);
    }
}
```

Then, you can configure your endpoint using Spring DSL:

```
<bean id="myMinaFactory"
class="org.apache.camel.component.mina.MinaComponent">
```

```
        <constructor-arg index="0" ref="camelContext" />
</bean>

<bean id="myMinaEndpoint"
      factory-bean="myMinaFactory"
      factory-method="createEndpoint">
  <constructor-arg index="0" ref="myMinaConfig"/>
</bean>

<bean id="myMinaConfig"
      class="org.apache.camel.component.mina.MinaConfiguration">
  <property name="protocol" value="tcp" />
  <property name="host" value="localhost" />
  <property name="port" value="2121" />
  <property name="sync" value="true" />
  <property name="minaLogger" value="true" />
  <property name="filters" ref="listFilters"/>
</bean>

<bean id="listFilters" class="java.util.ArrayList" >
  <constructor-arg>
    <list value-type="org.apache.mina.common.IoFilter">
      <bean class="com.example.KeepAliveFilter"/>
      <bean class="com.example.DropGreetingFilter"/>
    </list>
  </constructor-arg>
</bean>
```

CHAPTER 77. MINA2 [DEPRECATED]

MINA 2 COMPONENT



DEPRECATED

The MINA2 component is deprecated. Use [Netty](#) instead.

Available as of Camel 2.10

The `mina2:` component is a transport for working with [Apache MINA 2.x](#)

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>
```

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&...`

OPTIONS

| Option | Default Value | Description |
|-------------------------|--------------------|--|
| <code>codec</code> | <code>null</code> | You can refer to a named ProtocolCodecFactory instance in your Registry such as your Spring ApplicationContext , which is then used for the marshalling. |
| <code>disconnect</code> | <code>false</code> | Whether or not to <code>disconnect(close)</code> from Mina session right after use. Can be used for both consumer and producer. |

| | | |
|----------------------------|--------------------|---|
| textline | false | 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. |
| textlineDelimiter | DEFAULT | Only used for TCP and if textline=true . Sets the text line delimiter to use. Possible values are: DEFAULT, AUTO, WINDOWS, UNIX or MAC . If none provided, Camel will use DEFAULT . This delimiter is used to mark the end of text. |
| sync | true | Setting to set endpoint as one-way or request-response. |
| lazySessionCreation | true | Sessions can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started. |
| timeout | 30000 | 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. The timeout is only used for Mina producer. |
| encoding | <i>JVM Default</i> | 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 . |

| | | |
|-----------------------------|--------------|--|
| transferExchange | false | 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 <i>serializable</i> . Camel will exclude any non-serializable objects and log it at WARN level. |
| minaLogger | false | You can enable the Apache MINA logging filter. Apache MINA uses slf4j logging at INFO level to log all input and output. |
| filters | null | You can set a list of Mina IoFilters to register. The filters can be specified as a comma-separate list of bean references (e.g. <code>\#filterBean1,#filterBean2</code>) where each bean must be of type org.apache.mina.common.IoFilter . |
| encoderMaxLineLength | \-1 | Set the textline protocol encoder max line length. By default the default value of Mina itself is used which are Integer.MAX_VALUE . |
| decoderMaxLineLength | \-1 | Set the textline protocol decoder max line length. By default the default value of Mina itself is used which are 1024. |

| | | |
|----------------------------------|-------------|--|
| maximumPoolSize | 16 | The TCP producer is thread safe and supports concurrency much better. This option allows you to configure the number of threads in its thread pool for concurrent producers. Note: Camel has a pooled service which ensured it was already thread safe and supported concurrency already. |
| allowDefaultCodec | true | 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. |
| disconnectOnNoReply | true | If sync is enabled then this option dictates MinaConsumer if it should disconnect where there is no reply to send back. |
| noReplyLogLevel | WARN | If sync is enabled this option dictates MinaConsumer which logging level to use when logging a there is no reply to send back. Values are: FATAL, ERROR, INFO, DEBUG, OFF . |
| orderedThreadPoolExecutor | true | Whether to use ordered thread pool, to ensure events are processed orderly on the same channel. |
| sslContextParameters | null | SSL configuration using an org.apache.camel.util.jsse.SSLContextParameters instance. See Using the JSSE Configuration Utility . |
| autoStartTls | true | Whether to auto start SSL handshake. |

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.

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();
```

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);

```

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"
}

```

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_COMPLET
E, true);
    }
});

```

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`.

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.

- See also:

[Netty](#)

CHAPTER 78. MOCK

MOCK COMPONENT

The Mock component provides a powerful declarative testing mechanism, which is similar to [jMock](#) in that it allows declarative expectations to be created on any Mock endpoint before a test begins. Then the test is run, which typically fires messages to one or more endpoints, and finally the expectations can be asserted in a test case to ensure the system worked as expected.

This allows you to test various things like:

- The correct number of messages are received on each endpoint,
- The correct payloads are received, in the right order,
- Messages arrive on an endpoint in order, using some [Expression](#) to create an order testing function,
- Messages arrive match some kind of [Predicate](#) such as that specific headers have certain values, or that parts of the messages match some predicate, such as by evaluating an [XPath](#) or [XQuery Expression](#).

Note that there is also the [Test endpoint](#) which is a Mock endpoint, but which uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. In other words, it's a Mock endpoint that automatically sets up its assertions from some sample messages in a [File](#) or [database](#), for example.



MOCK ENDPOINTS KEEP RECEIVED EXCHANGES IN MEMORY INDEFINITELY

Remember that Mock is designed for testing. When you add Mock endpoints to a route, each [Exchange](#) sent to the endpoint will be stored (to allow for later validation) in memory until explicitly reset or the JVM is restarted. If you are sending high volume and/or large messages, this may cause excessive memory use. If your goal is to test deployable routes inline, consider using [NotifyBuilder](#) or [AdviceWith](#) in your tests instead of adding Mock endpoints to routes directly.

From Camel 2.10 onwards there are two new options `retainFirst`, and `retainLast` that can be used to limit the number of messages the Mock endpoints keep in memory.

URI FORMAT

```
mock:someName[?options]
```

Where `someName` can be any string that uniquely identifies the endpoint.

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

OPTIONS

| Option | Default | Description |
|--------------------------|-------------------|---|
| <code>reportGroup</code> | <code>null</code> | A size to use a throughput logger for reporting |

SIMPLE EXAMPLE

Here's a simple example of Mock endpoint in use. First, the endpoint is resolved on the context. Then we set an expectation, and then, after the test has run, we assert that our expectations have been met.

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo",
MockEndpoint.class);

resultEndpoint.expectedMessageCount(2);

// send some messages
...

// now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

You typically always call the `assertIsSatisfied()` method to test that the expectations were met after running a test.

Apache Camel will by default wait 10 seconds when the `assertIsSatisfied()` is invoked. This can be configured by setting the `setResultWaitTime(millis)` method.

USING ASSERTPERIOD

Available as of Camel 2.7 When the assertion is satisfied then Camel will stop waiting and continue from the `assertIsSatisfied` method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the `setAssertPeriod` method, for example:

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo",
MockEndpoint.class);
resultEndpoint.setAssertPeriod(5000);
resultEndpoint.expectedMessageCount(2);

// send some messages
...

// now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

SETTING EXPECTATIONS

You can see from the javadoc of `MockEndpoint` the various helper methods you can use to set expectations. The main methods are as follows:

| Method | Description |
|--------|-------------|
|--------|-------------|

| | |
|---|--|
| <code>expectedMessageCount(int)</code> | To define the expected message count on the endpoint. |
| <code>expectedMinimumMessageCount(int)</code> | To define the minimum number of expected messages on the endpoint. |
| <code>expectedBodiesReceived(...)</code> | To define the expected bodies that should be received (in order). |
| <code>expectedHeaderReceived(...)</code> | To define the expected header that should be received |
| <code>expectsAscending(Expression)</code> | To add an expectation that messages are received in order, using the given Expression to compare messages. |
| <code>expectsDescending(Expression)</code> | To add an expectation that messages are received in order, using the given Expression to compare messages. |
| <code>expectsNoDuplicates(Expression)</code> | To add an expectation that no duplicate messages are received; using an Expression to calculate a unique identifier for each message. This could be something like the JMSMessageID if using JMS, or some unique reference number within the message. |

Here's another example:

```
resultEndpoint.expectedBodiesReceived("firstMessageBody",
"secondMessageBody", "thirdMessageBody");
```

ADDING EXPECTATIONS TO SPECIFIC MESSAGES

In addition, you can use the `message(int messageIndex)` method to add assertions about a specific message that is received.

For example, to add expectations of the headers or body of the first message (using zero-based indexing like `java.util.List`), you can use the following code:

```
resultEndpoint.message(0).header("foo").isEqualTo("bar");
```

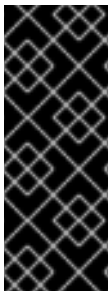
There are some examples of the Mock endpoint in use in the [camel-core](#)

processor tests.

MOCKING EXISTING ENDPOINTS

Available as of Camel 2.7

Camel now allows you to automatic mock existing endpoints in your Camel routes.



HOW IT WORKS

Important: The endpoints are still in action, what happens is that a **Mock** endpoint is injected and receives the message first, it then delegate the message to the target endpoint. You can view this as a kind of intercept and delegate or endpoint listener.

Suppose you have the given route below:

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {

            from("direct:start").to("direct:foo").to("log:foo").to("mock:result");

                from("direct:foo").transform(constant("Bye World"));
            }
        };
    }
}
```

You can then use the `adviceWith` feature in Camel to mock all the endpoints in a given route from your unit test, as shown below:

```
public void testAdvisedMockEndpoints() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new
AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock all endpoints
            mockEndpoints();
        }
    });
}
```

```

    }
    });

    getMockEndpoint("mock:direct:start").expectedBodiesReceived("Hello
World");
    getMockEndpoint("mock:direct:foo").expectedBodiesReceived("Hello
World");
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");
    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // all the endpoints was mocked
    assertNotNull(context.hasEndpoint("mock:direct:start"));
    assertNotNull(context.hasEndpoint("mock:direct:foo"));
    assertNotNull(context.hasEndpoint("mock:log:foo"));
}

```

Notice that the mock endpoints is given the uri `mock:<endpoint>`, for example `mock:direct:foo`. Camel logs at INFO level the endpoints being mocked:

```
INFO  Adviced endpoint [direct://foo] with mock endpoint [mock:direct:foo]
```



MOCKED ENDPOINTS ARE WITHOUT PARAMETERS

Endpoints which are mocked will have their parameters stripped off. For example the endpoint `log:foo?showAll=true` will be mocked to the following endpoint `mock:log:foo`. Notice the parameters has been removed.

Its also possible to only mock certain endpoints using a pattern. For example to mock all log endpoints you do as shown:

```

public void testAdvisedMockEndpointsWithPattern() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new
AdviceWithRouteBuilder() {
        @Override

```

```

        public void configure() throws Exception {
            // mock only log endpoints
            mockEndpoints("log*");
        }
    });

    // now we can refer to log:foo as a mock and set our expectations
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");

    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // only the log:foo endpoint was mocked
    assertNotNull(context.hasEndpoint("mock:log:foo"));
    assertNull(context.hasEndpoint("mock:direct:start"));
    assertNull(context.hasEndpoint("mock:direct:foo"));
}

```

The pattern supported can be a wildcard or a regular expression. See more details about this at [Intercept](#) as its the same matching function used by Camel.



IMPORTANT

Mind that mocking endpoints causes the messages to be copied when they arrive on the mock. That means Camel will use more memory. This may not be suitable when you send in a lot of messages.

MOCKING EXISTING ENDPOINTS USING THE CAMEL-TEST COMPONENT

Instead of using the `adviceWith` to instruct Camel to mock endpoints, you can easily enable this behavior when using the camel-test Test Kit. The same route can be tested as follows. Notice that we return `*` from the `isMockEndpoints` method, which tells Camel to mock all endpoints. If you only want to mock all log endpoints you can return `"log"` instead.

```

public class IsMockEndpointsJUnit4Test extends CamelTestSupport {

    @Override
    public String isMockEndpoints() {

```

```

        // override this method and return the pattern for which endpoints
to mock.
        // use * to indicate all
        return "*";
    }

    @Test
    public void testMockAllEndpoints() throws Exception {
        // notice we have automatic mocked all endpoints and the name of
the endpoints is "mock:uri"
        getMockEndpoint("mock:direct:start").expectedBodiesReceived("Hello
World");
        getMockEndpoint("mock:direct:foo").expectedBodiesReceived("Hello
World");
        getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye
World");
        getMockEndpoint("mock:result").expectedBodiesReceived("Bye
World");

        template.sendBody("direct:start", "Hello World");

        assertMockEndpointsSatisfied();

        // additional test to ensure correct endpoints in registry
        assertNotNull(context.hasEndpoint("direct:start"));
        assertNotNull(context.hasEndpoint("direct:foo"));
        assertNotNull(context.hasEndpoint("log:foo"));
        assertNotNull(context.hasEndpoint("mock:result"));
        // all the endpoints was mocked
        assertNotNull(context.hasEndpoint("mock:direct:start"));
        assertNotNull(context.hasEndpoint("mock:direct:foo"));
        assertNotNull(context.hasEndpoint("mock:log:foo"));
    }

    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            @Override
            public void configure() throws Exception {
                from("direct:start").to("direct:foo").to("log:foo").to("mock:result");

                from("direct:foo").transform(constant("Bye World"));
            }
        };
    }
}

```

MOCKING EXISTING ENDPOINTS WITH XML DSL

If you do not use the camel-test component for unit testing (as shown above) you can use a different approach when using XML files for routes. The

solution is to create a new XML file used by the unit test and then include the intended XML file which has the route you want to test.

Suppose we have the route in the camel-route.xml file:

```

<!-- this camel route is in the camel-route.xml file -->
<camelContext xmlns="http://camel.apache.org/schema/spring">

  <route>
    <from uri="direct:start"/>
    <to uri="direct:foo"/>
    <to uri="log:foo"/>
    <to uri="mock:result"/>
  </route>

  <route>
    <from uri="direct:foo"/>
    <transform>
      <constant>Bye World</constant>
    </transform>
  </route>

</camelContext>

```

Then we create a new XML file as follows, where we include the camel-route.xml file and define a spring bean with the class `org.apache.camel.impl.InterceptSendToMockEndpointStrategy` which tells Camel to mock all endpoints:

```

<!-- the Camel route is defined in another XML file -->
<import resource="camel-route.xml"/>

<!-- bean which enables mocking all endpoints -->
<bean id="mockAllEndpoints"
class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy"/>

```

Then in your unit test you load the new XML file (`test-camel-route.xml`) instead of `camel-route.xml`.

To only mock all `log` endpoints you can define the pattern in the constructor for the bean:

```

<bean id="mockAllEndpoints"
class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy">
  <constructor-arg index="0" value="log*"/>
</bean>

```


MOCKING ENDPOINTS AND SKIP SENDING TO ORIGINAL ENDPOINT

Available as of Camel 2.10

Sometimes you want to easily mock and skip sending to a certain endpoints. So the message is detoured and send to the mock endpoint only. From Camel 2.10 onwards you can now use the `mockEndpointsAndSkip` method using [AdviceWith](#) or the [Test Kit](#). The example below will skip sending to the two endpoints "direct:foo", and "direct:bar".

```
public void testAdvisedMockEndpointsWithSkip() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new
AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock sending to direct:foo and direct:bar and skip send to
it
            mockEndpointsAndSkip("direct:foo", "direct:bar");
        }
    });

    getMockEndpoint("mock:result").expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:direct:foo").expectedMessageCount(1);
    getMockEndpoint("mock:direct:bar").expectedMessageCount(1);

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // the message was not send to the direct:foo route and thus not sent
to the seda endpoint
    SedaEndpoint seda = context.getEndpoint("seda:foo",
SedaEndpoint.class);
    assertEquals(0, seda.getCurrentQueueSize());
}
```

The same example using the [Test Kit](#)

```
public class IsMockEndpointsAndSkipJUnit4Test extends CamelTestSupport {

    @Override
    public String isMockEndpointsAndSkip() {
        // override this method and return the pattern for which endpoints
```

```

to mock,
    // and skip sending to the original endpoint.
    return "direct:foo";
}

@Test
public void testMockEndpointAndSkip() throws Exception {
    // notice we have automatic mocked the direct:foo endpoints and
the name of the endpoints is "mock:uri"
    getMockEndpoint("mock:result").expectedBodiesReceived("Hello
World");
    getMockEndpoint("mock:direct:foo").expectedMessageCount(1);

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // the message was not send to the direct:foo route and thus not
sent to the seda endpoint
    SedaEndpoint seda = context.getEndpoint("seda:foo",
SedaEndpoint.class);
    assertEquals(0, seda.getCurrentQueueSize());
}

@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:start").to("direct:foo").to("mock:result");

            from("direct:foo").transform(constant("Bye
World")).to("seda:foo");
        }
    };
}
}

```

LIMITING THE NUMBER OF MESSAGES TO KEEP

Available as of Camel 2.10

The [Mock](#) endpoints will by default keep a copy of every [Exchange](#) that it received. So if you test with a lot of messages, then it will consume memory. From Camel 2.10 onwards we have introduced two options `retainFirst` and `retainLast` that can be used to specify to only keep N'th of the first and/or last [Exchanges](#).

For example in the code below, we only want to retain a copy of the first 5 and last 5 [Exchanges](#) the mock receives.

```
MockEndpoint mock = getMockEndpoint("mock:data");
mock.setRetainFirst(5);
mock.setRetainLast(5);
mock.expectedMessageCount(2000);

...

mock.assertIsSatisfied();
```

Using this has some limitations. The `getExchanges()` and `getReceivedExchanges()` methods on the `MockEndpoint` will return only the retained copies of the [Exchanges](#). So in the example above, the list will contain 10 [Exchanges](#); the first five, and the last five. The `retainFirst` and `retainLast` options also have limitations on which expectation methods you can use. For example the `expectedXXX` methods that work on message bodies, headers, etc. will only operate on the retained messages. In the example above they can test only the expectations on the 10 retained messages.

TESTING WITH ARRIVAL TIMES

Available as of Camel 2.7

The [Mock](#) endpoint stores the arrival time of the message as a property on the [Exchange](#).

```
Date time = exchange.getProperty(Exchange.RECEIVED_TIMESTAMP, Date.class);
```

You can use this information to know when the message arrived on the mock. But it also provides foundation to know the time interval between the previous and next message arrived on the mock. You can use this to set expectations using the `arrives` DSL on the [Mock](#) endpoint.

For example to say that the first message should arrive between 0-2 seconds before the next you can do:

```
mock.message(0).arrives().noLaterThan(2).seconds().beforeNext();
```

You can also define this as that 2nd message (0 index based) should arrive

no later than 0-2 seconds after the previous:

```
mock.message(1).arrives().noLaterThan(2).seconds().afterPrevious();
```

You can also use `between` to set a lower bound. For example suppose that it should be between 1-4 seconds:

```
mock.message(1).arrives().between(1, 4).seconds().afterPrevious();
```

You can also set the expectation on all messages, for example to say that the gap between them should be at most 1 second:

```
mock.allMessages().arrives().noLaterThan(1).seconds().beforeNext();
```

TIME UNITS

In the example above we use seconds as the time unit, but Camel offers milliseconds, and minutes as well.

CHAPTER 79. MONGODB

CAMEL MONGODB COMPONENT

Available as of Camel 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.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI FORMAT

```
mongodb:connectionBean?
```

```
database=databaseName&collection=collectionName&operation=operationName[&moreOptions...]
```

ENDPOINT OPTIONS

MongoDB endpoints support the following options, depending on whether they are acting like a Producer or as a Consumer (options vary based on the consumer type too).

| Name | Default Value | Description | Producer | Tailable Cursor Consumer |
|-----------------|---------------|---|----------|--------------------------|
| database | none | Required. The name of the database to which this endpoint will be bound. All operations will be executed against this database unless dynamicity is enabled and the CamelMongoDbDatabase header is set. | (/) | (/) |

| | | | | |
|------------------------|------|---|-----|-----|
| collection | none | Required. The name of the collection (within the specified database) to which this endpoint will be bound. If operations will be executed against this database unless dynamicity is enabled and the CamelMongoOdbDatabase header is set. | (/) | (/) |
| collectionIndex | none | Camel 2.12: An optional index to create when inserting new collections. | (/) | |

| | | | | |
|-------------------------|------|---|-----|--|
| operation | none | <p>Required for producers. The id of the operation this endpoint will execute. Pick from the following:</p> <ul style="list-style-type: none"> • Query operations: findById, findOneByQuery, findAll, count • Write operations: insert, save, update • Delete operations: remove • Other operations: getDbStats, getColStats | (/) | |
| createCollection | true | Determines whether the collection will be automatically created in the MongoDB database during endpoint initialisation if it doesn't exist already. If this option is false and the collection doesn't exist, an initialisation exception will be thrown. | (/) | |

| | | | | |
|---------------------------|--|---|-----|--|
| invokeGetLastError | false (behaviour may be inherited from connections WriteConcern) | Instructs the MongoDB Java driver to invoke getLastError () after every call. Default behaviour in version 2.7.2 of the MongoDB Java driver is that only network errors will cause the operation to fail, because the actual operation is executed asynchronously in the MongoDB server without holding up the client - to increase performance. The client can obtain the real result of the operation by explicitly invoking getLastError () on the WriteResult object returned or by setting the appropriate WriteConcern . If the backend operation has not finished yet, the client will block until the result is available. Setting this option to true will make the endpoint behave synchronously and return an Exception if the underlying operation failed. | (/) | |
|---------------------------|--|---|-----|--|

| | | | | |
|------------------------|-------------------------|--|-----|--|
| writeConcern | none (driver's default) | Set a WriteConcern on the operation out of MongoDB's parameterised values. See WriteConcern.valueOf(String) . | (/) | |
| writeConcernRef | none | Sets a custom WriteConcern that exists in the Registry. Specify the bean name. | (/) | |
| readPreference | none | Sets a ReadPreference on the connection. Accepted values: the name of any inner subclass of ReadPreference . For example: PrimaryReadPreference , SecondaryReadPreference , TaggedReadPreference . | (/) | |

| | | | | |
|----------------------------|-------|---|-----|--|
| dynamicity | false | If set to true, the endpoint will inspect the CamelMongoDb Database and CamelMongoDb Collection headers of the incoming message, and if any of them exists, the target collection and/or database will be overridden for that particular operation. Set to false by default to avoid triggering the lookup on every Exchange if the feature is not desired. | (/) | |
| writeResultAsHeader | false | Available as of Camel 2.10.3 and 2.11: In write operations (save, update, insert, etc.), instead of replacing the body with the WriteResult object returned by MongoDB, keep the input body untouched and place the WriteResult in the CamelMongoWriteResult header (constant MongoDbConstants.WRITERESULT). | (/) | |

| | | | | |
|-------------------------------|-------|--|--|-----|
| persistentTailTracking | false | Enables or disables persistent tail tracking for Tailable Cursor consumers. See below for more information. | | (/) |
| persistentId | none | Required if persistent tail tracking is enabled. The id of this persistent tail tracker, to separate its records from the rest on the tail-tracking collection. | | (/) |

| | | | | |
|------------------------------------|--------------------|--|--|-----|
| tailTrackingIncreasingField | none | <p>Required if persistent tail tracking is enabled.</p> <p>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: <code>tailTrackIncreasingField > lastValue</code> (where <code>lastValue</code> is possibly recovered from persistent tail tracking). Can be of type Integer, Date, String, etc.</p> <p>NOTE: No support for dot notation at the current time, so the field should be at the top level of the document.</p> | | (/) |
| cursorRegenerationDelay | 1000ms | Establishes how long the endpoint will wait to regenerate the cursor after it has been killed by the MongoDB server (normal behaviour). | | (/) |
| tailTrackDb | same as endpoint's | Database on which the persistent tail tracker will store its runtime information. | | (/) |

| | | | | |
|----------------------------|-------------------|---|--|-----|
| tailTrackCollection | camelTailTracking | Collection on which the persistent tail tracker will store its runtime information. | | (/) |
| tailTrackField | lastTrackingValue | Field in which the persistent tail tracker will store the last tracked value. | | (/) |

MONGODB OPERATIONS - PRODUCER ENDPOINTS

QUERY OPERATIONS

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> and <http://www.mongodb.org/display/DOCS/Java+Types>.

```
from("direct:findById")
  .to("mongodb:myDb?
database=flights&collection=tickets&operation=findById")
  .to("mock:resultFindById");
```

SUPPORTS FIELDS FILTER

This operation supports specifying a fields filter. See [Specifying a fields filter](#).

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");
```

SUPPORTS FIELDS FILTER

This operation supports specifying a fields filter. See [Specifying a fields filter](#).

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 MongoDB API doc) | Expected type |
|---------------------------|----------------------------------|--|---------------|
| CamelMongoDbNumT oSkip | MongoDbConstants .NUM_TO_SKIP | Discards a given number of elements at the beginning of the cursor. | int/Integer |
| CamelMongoDbLimi t | 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 |
|------------------------------|------------------------------------|---|-------------|

Additionally, you can set a `sortBy` criteria by putting the relevant `DBObject` describing your sorting in the `CamelMongoDbSortBy` header, quick constant: `MongoDbConstants.SORT_BY`.

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 |
|--|---|---|-------------|
| <code>CamelMongoDbResultTotalSize</code> | <code>MongoDbConstants.RESULT_TOTAL_SIZE</code> | Number of objects matching the query. This does not take limit/skip into consideration. | int/Integer |
| <code>CamelMongoDbResultPageSize</code> | <code>MongoDbConstants.RESULT_PAGE_SIZE</code> | Number of objects matching the query. This does not take limit/skip into consideration. | int/Integer |

SUPPORTS FIELDS FILTER

This operation supports specifying a fields filter. See [Specifying a fields filter](#).

SPECIFYING A FIELDS FILTER

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 `DBObject`s. 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, MongoClientConstants.FIELDS_FILTER, fieldFilter);
```

CREATE/UPDATE OPERATIONS

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 it is unique (and this is a good practice too).

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");
```

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.



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 |
| 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);
```

DELETE OPERATIONS

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()`).

OTHER OPERATIONS

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);
```

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.

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.

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);
```

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 `DBObject`s in natural order to your tailable cursor consumer, who will transform them to an Exchange and will trigger your route logic.

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=departu
reTime")
    .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.

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.

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=departu
reTime&persistentTailTracking=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=departu
reTime&persistentTailTracking=true" +

"&persistentId=cancellationsTracker"&tailTrackDb=trackers&tailTrackCollect
ion=camelTrackers" +
    "&tailTrackField=lastProcessedDepartureTime")
    .id("tailableCursorConsumer3")
    .autoStartup(false)
    .to("mock:test");
```

TYPE CONVERSIONS

The `MongoDbBasicConverters` type converter included with the `camel-mongodb` component provides the following conversions:

| Name | From type | To type | How? |
|--------------------------------------|----------------------|-----------------|---|
| <code>fromMapToDBObject</code> | Map | DBObject | constructs a new BasicDBObject via the new BasicDBObject(Map m) constructor |
| <code>fromBasicDBObjectToMap</code> | BasicDBObject | Map | BasicDBObject already implements Map |
| <code>fromStringToDBObject</code> | String | DBObject | uses <code>com.mongodb.util.JSON.parse(String s)</code> |
| <code>fromAnyObjectToDBObject</code> | 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.

SEE ALSO

-

[MongoDB website](#)

-

[NoSQL Wikipedia article](#)

-

[MongoDB Java driver API docs - current version](#)

-

[Unit tests](#) for more examples of usage

CHAPTER 80. MQTT

MQTT COMPONENT

Available as of Camel 2.10

The `mqtt:` component is used for communicating with [MQTT](#) compliant message brokers, like [Apache ActiveMQ](#) or [Mosquitto](#)

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-mqtt</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI FORMAT

```
mqtt://name[?options]
```

Where `name` is the name you want to assign the component.

OPTIONS

| Property | Default | Description |
|-----------------------------------|-----------------------------|--|
| host | tcp://127.0.0.1:1883 | The URI of the MQTT broker to connect too - this component also supports SSL - e.g. <i>ssl://127.0.0.1:8883</i> |
| localAddress | | The local InetAddress and port to use |
| connectAttemptsMax | -1 | The maximum number of attempts to establish an initial connection - -1 in infinite. |
| reconnectAttemptsMax | -1 | The maximum number of attempts to re-establish a connection after failure - -1 in infinite. |
| reconnectDelay | 10 | The time in milliseconds between attempts to reestablish an initial or failed connection |
| reconnectBackOffMultiplier | 2.0 | The multiplier to use to the delay between connection attempts for successive failed connection attempts |
| reconnectDelayMax | 30000 | The maximum time in milliseconds between a new attempt to establish a connection. So even using the reconnectBackOffMultiplier , this property will define the maximum delay before another connection attempt to the MQTT broker |
| QoS | AtLeastOnce | The MQTT Quality of Service to use for message exchanges. It can be one of AtMostOnce , AtLeastOnce or ExactlyOnce |
| subscribeTopicName | | The name of the Topic to subscribe to for messages |

| | | |
|--------------------------------|--------------------------------|---|
| publishTopicName | camel/mqtt/test | The default Topic to publish messages on |
| byDefaultRetain | false | The default retain policy to be used on messages sent to the MQTT broker |
| mqttTopicPropertyName | _MQTTTopicPropertyName+ | The property name to look for on an Exchange for an individual published message. If this is set - the name will be used as the Topic to publish a message to the MQTT message broker. |
| mqttRetainPropertyName | MQTTRetain | 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. |
| mqttQosPropertyName | MQTTQos | 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. |
| connectWaitInSeconds | 10 | Delay in seconds the Component will wait for a connection to be established to the MQTT broker |
| disconnectWaitInSeconds | 5 | the number of seconds the Component will wait for a valid disconnect on stop() from the MQTT broker |
| sendWaitInSeconds | 5 | The maximum time the Component will wait for a receipt from the MQTT broker to acknowledge a published message before throwing an exception |

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

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")
```

CHAPTER 81. MSV

MSV COMPONENT

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](#).

Note that the [Jing](#) component also supports [RelaxNG Compact Syntax](#)

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&...`

OPTIONS

| Option | Default | Description |
|---------------------|-------------------|--|
| <code>useDom</code> | <code>true</code> | Whether DOMSource/DOMResult or SaxSource/SaxResult should be used by the validator. Note: DOM must be used by the MSV component. |

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).

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <doTry>
      <to
uri="msv:org/apache/camel/component/validator/msv/schema.rng"/>
      <to uri="mock:valid"/>

      <doCatch>

<exception>org.apache.camel.ValidationException</exception>
      <to uri="mock:invalid"/>
      </doCatch>
      <doFinally>
        <to uri="mock:finally"/>
      </doFinally>
    </doTry>
  </route>
</camelContext>
```

CHAPTER 82. MUSTACHE

MUSTACHE

Available as of Camel 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>
```

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&...`

OPTIONS

| Option | Default | Description |
|-----------------------------|-------------------|--|
| <code>encoding</code> | <code>null</code> | Character encoding of the resource content. |
| <code>startDelimiter</code> | <code>{{</code> | Characters used to mark template code beginning. |
| <code>endDelimiter</code> | <code>}}</code> | Characters used to mark template code end. |

MUSTACHE CONTEXT

Camel will provide exchange information in the Mustache context (just a Map). The Exchange is transferred as:

| key | value |
|----------------------------------|--|
| <code>exchange</code> | The Exchange itself. |
| <code>exchange.properties</code> | The Exchange properties. |
| <code>headers</code> | The headers of the In message. |
| <code>camelContext</code> | The Camel Context. |
| <code>request</code> | The In message. |
| <code>body</code> | The In message body. |
| <code>response</code> | The Out message (only for InOut message exchange pattern). |

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. | |
| MustacheConstants.MUSTACHE_TEMPLATE | String | The template to use instead of the endpoint configured. | |

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/te  
mplate.mustache").  
to("mustache:dummy");
```

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}}
```


CHAPTER 83. MVEL COMPONENT

MVEL COMPONENT

Available as of Camel 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>
```

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&...`

OPTIONS

| Option | Default | Description |
|---------------------------|-------------------|---|
| <code>contentCache</code> | <code>true</code> | Cache for the resource content when it is loaded. The cached resource content can be cleared via JMX using the endpoint's <code>clearContentCache</code> operation. |
| <code>encoding</code> | <code>null</code> | Character encoding of the resource content. |

MESSAGE HEADERS

The `mvel` component sets a couple headers on the message.

| Header | Description |
|-----------------------------------|--|
| <code>CamelMvelResourceUri</code> | The <code>templateName</code> as a <code>String</code> object. |

MVEL CONTEXT

Camel will provide exchange information in the MVEL context (just a `Map`). The Exchange is transferred as:

| key | value |
|----------------------------------|---------------------------------------|
| <code>exchange</code> | The <code>Exchange</code> itself. |
| <code>exchange.properties</code> | The <code>Exchange</code> properties. |
| <code>headers</code> | 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). |
| response | The Out message (only for InOut message exchange pattern). |

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.

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 |
|-----------------------------------|--------|--|
| <code>CamelMvelResourceUri</code> | String | A URI for the template resource to use instead of the endpoint configured. |
| <code>CamelMvelTemplate</code> | String | The template to use instead of the endpoint configured. |

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");
```

CHAPTER 84. MYBATIS

MYBATIS

Available as of Camel 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>
```

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.

OPTIONS

| Option | Type | Default | Description |
|---|---------------|---------|--|
| <code>consumer.onConsume</code> | String | null | Statements to run after consuming. Can be used, for example, to update rows after they have been consumed and processed in Camel. See sample later. Multiple statements can be separated with commas. |
| <code>consumer.useIterator</code> | boolean | true | If true each row returned when polling will be processed individually. If false the entire List of data is set as the IN body. |
| <code>consumer.routeEmptyResultSet</code> | boolean | false | Sets whether empty result sets should be routed. |
| <code>statementType</code> | StatementType | null | Mandatory to specify for the producer to control which kind of operation to invoke. The enum values are: SelectOne , SelectList , Insert , InsertList , Update , UpdateList , Delete , and DeleteList . Notice:InsertList is available as of Camel 2.10, and UpdateList , DeleteList is available as of Camel 2.11. |

| | | | |
|---------------------------|---------------|-------------|--|
| maxMessagesPerPoll | int | 0 | An integer to define the 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 disable it. |
| executorType | String | null | Camel 2.11: The executor type to be used while executing statements. The supported values are: simple, reuse, batch. By default, the value is not specified and is equal to what MyBatis uses, i.e. simple . simple executor does nothing special. reuse executor reuses prepared statements. batch executor reuses statements and batches updates. |

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. |

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`.

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>
```

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:


```

from("direct:start")
  .to("mybatis:selectAccountById?statementType=SelectOne")
  .to("mock:result");

```

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`:

```

from("direct:start")
  .to("mybatis:selectAllAccounts?statementType=SelectList")
  .to("mock:result");

```

And the same for UPDATE, where we can send an Account object as the IN body to MyBatis:

```

from("direct:start")
  .to("mybatis:updateAccount?statementType=Update")
  .to("mock:result");

```

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:

```

<!-- Batch Insert example, using the Account parameter class -->
<insert id="batchInsertAccount" parameterType="java.util.List">
  insert into ACCOUNT (
    ACC_ID,
    ACC_FIRST_NAME,
    ACC_LAST_NAME,
    ACC_EMAIL
  )
  values (
    <foreach item="Account" collection="list" open="" close=""
separator=""),(">
      #{Account.id}, #{Account.firstName}, #{Account.lastName}, #
{Account.emailAddress}

```

```

    </foreach>
  )
</insert>

```

Then you can insert multiple rows, by sending a Camel message to the mybatis endpoint which uses the InsertList statement type, as shown below:

```

from("direct:start")
  .to("mybatis:batchInsertAccount?statementType=InsertList")
  .to("mock:result");

```

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");

```

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");
```

NOTICE ON INSERTLIST, UPDATELIST AND DELETEDLIST 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.

SCHEDULED POLLING EXAMPLE

Since this component does not support scheduled polling, you need to 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("mbatis: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>
```

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.

```
from("mybatis:selectUnprocessedAccounts?
consumer.onConsume=consumeAccount").to("mock:results");
```

And the statements in the sqlmap file:

```
<select id="selectUnprocessedAccounts" resultMap="AccountResult">
  select * from ACCOUNT where PROCESSED = false
</select>
```

```
<update id="consumeAccount" parameterType="Account">
  update ACCOUNT set PROCESSED = true where ACC_ID = #{id}
</update>
```

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-spring SqlSessionFactoryBean` 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>
```

CHAPTER 85. NAGIOS

NAGIOS

Available as of Apache Camel 2.3

The [Nagios](#) component allows you to send passive checks to [Nagios](#).

URI FORMAT

```
nagios://host[:port][?Options]
```

Apache Camel provides two abilities with the [Nagios](#) component. You can send passive check messages by sending a message to its endpoint. Apache Camel also provides a [EventNotifier](#) which allows you to send notifications to Nagios.

OPTIONS

| Name | Default Value | Description |
|--------------------------|---------------|---|
| host | none | This is the address of the Nagios host where checks should be send. |
| port | | The port number of the host. |
| password | | Password to be authenticated when sending checks to Nagios. |
| connectionTimeout | 5000 | Connection timeout in millis. |
| timeout | 5000 | Sending timeout in millis. |

| | | |
|-------------------------|-------------|---|
| nagiosSettings | | To use an already configured <code>com.googlecode.jsendnsca.core.NagiosSettings</code> object. Then any of the other options are not in use, if using this. |
| sendSync | true | Whether or not to use synchronous when sending a passive check. Setting it to false will allow Apache Camel to continue routing the message and the passive check message will be send asynchronously. |
| encryptionMethod | No | *Camel 2.9:* To specify an encryption method. Possible values: No , Xor , or TripleDes . |

HEADERS

| Name | Description |
|-------------------------------|---|
| CamelNagiosHostName | This is the address of the Nagios host where checks should be send. This header will override any existing hostname configured on the endpoint. |
| CamelNagiosLevel | This is the severity level. You can use values CRITICAL , WARNING , OK . Apache Camel will by default use OK . |
| CamelNagiosServiceName | The servie name. Will default use the CamelContext name. |

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);
```

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 Apache Camel will pick it up as documented here: [Advanced configuration of CamelContext using Spring](#).

CHAPTER 86. NETTY

NETTY COMPONENT

Available as of Camel 2.3

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>
```

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&...

OPTIONS

| Name | Default Value | Description |
|-----------------------------|--------------------|--|
| <code>keepAlive</code> | <code>true</code> | Setting to ensure socket is not closed due to inactivity |
| <code>tcpNoDelay</code> | <code>true</code> | Setting to improve TCP protocol performance |
| <code>backlog</code> | | Camel 2.9.6/2.10.4/2.11: 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. |
| <code>broadcast</code> | <code>false</code> | Setting to choose Multicast over UDP |
| <code>connectTimeout</code> | <code>10000</code> | Time to wait for a socket connection to be available. Value is in millis. |
| <code>reuseAddress</code> | <code>true</code> | Setting to facilitate socket multiplexing |
| <code>sync</code> | <code>true</code> | Setting to set endpoint as one-way or request-response |
| <code>synchronous</code> | <code>false</code> | Camel 2.10: Whether Asynchronous Routing Engine is not in use. false then the Asynchronous Routing Engine is used, true to force processing synchronous. |

| | | |
|-----------------------------------|--------------------------|--|
| <code>ssl</code> | <code>false</code> | Setting to specify whether SSL encryption is applied to this endpoint |
| <code>sslClientCertHeaders</code> | <code>false</code> | Camel 2.12: 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. |
| <code>sendBufferSize</code> | <code>65536 bytes</code> | The TCP/UDP buffer sizes to be used during outbound communication. Size is bytes. |
| <code>receiveBufferSize</code> | <code>65536 bytes</code> | The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes. |
| <code>option.XXX</code> | <code>null</code> | Camel 2.11/2.10.4: 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. |
| <code>corePoolSize</code> | <code>10</code> | The number of allocated threads at component startup. Defaults to 10. Note: This option is removed from Camel 2.9.2 onwards. As we rely on Netty's default settings. |
| <code>maxPoolSize</code> | <code>100</code> | The maximum number of threads that may be allocated to this endpoint. Defaults to 100. Note: This option is removed from Camel 2.9.2 onwards. As we rely on Netty's default settings. |

| | | |
|--------------------------------------|--------------|---|
| disconnect | false | Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer. |
| lazyChannelCreation | true | Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started. |
| transferExchange | false | 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. |
| disconnectOnNoReply | true | If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back. |
| noReplyLogLevel | WARN | If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back. Values are: FATAL, ERROR, INFO, DEBUG, OFF. |
| serverExceptionCaughtLogLevel | WARN | Camel 2.11.1: If the server (NettyConsumer) catches an exception then its logged using this logging level. |

| | | |
|---|--------------------|--|
| <code>serverClosedChannelExceptionCaughtLogLevel</code> | <code>DEBUG</code> | Camel 2.11.1: If the server (NettyConsumer) catches an <code>java.nio.channels.ClosedChannelException</code> 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. |
| <code>allowDefaultCodec</code> | <code>true</code> | Camel 2.4: The netty component installs a default codec if both, encoder/decoder is null and textline is false. Setting <code>allowDefaultCodec</code> to false prevents the netty component from installing a default codec as the first element in the filter chain. |
| <code>textline</code> | <code>false</code> | Camel 2.4: 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. |
| <code>delimiter</code> | <code>LINE</code> | Camel 2.4: The delimiter to use for the textline codec. Possible values are <code>LINE</code> and <code>NULL</code> . |
| <code>decoderMaxLineLength</code> | <code>1024</code> | Camel 2.4: The max line length to use for the textline codec. |
| <code>autoAppendDelimiter</code> | <code>true</code> | Camel 2.4: Whether or not to auto append missing end delimiter when sending using the textline codec. |
| <code>encoding</code> | <code>null</code> | Camel 2.4: The encoding (a charset name) to use for the textline codec. If not provided, Camel will use the JVM default Charset. |

| | | |
|-----------------------------------|--------------|--|
| workerCount | null | Camel 2.9: When netty works on nio mode, it uses default workerCount parameter from Netty, which is <code>cpu_core_threads*2</code> . User can use this operation to override the default workerCount from Netty |
| sslContextParameters | null | Camel 2.9: SSL configuration using an <code>org.apache.camel.util.jsse.SSLContextParameters</code> instance. See Using the JSSE Configuration Utility. |
| receiveBufferSizePredictor | null | Camel 2.9: Configures the buffer size predictor. See details at Jetty documentation and this mail thread . |
| requestTimeout | 0 | Camel 2.11.1: 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. |
| needClientAuth | false | Camel 2.11: Configures whether the server needs client authentication when using SSL. |
| orderedThreadPoolExecutor | true | Camel 2.10.2: Whether to use ordered thread pool, to ensure events are processed orderly on the same channel. See details at the netty javadoc of <code>org.jboss.netty.handler.execution.OrderedMemoryAwareThreadPoolExecutor</code> for more details. |
| maximumPoolSize | 16 | Camel 2.10.2: The core pool size for the ordered thread pool, if its in use. |

| | | |
|-------------------------------------|--------------|--|
| producerPoolEnabled | true | Camel 2.10.4/Camel 2.11: Producer only. 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. |
| producerPoolMaxActive | -1 | Camel 2.10.3: Producer only. 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. |
| producerPoolMinIdle | 0 | Camel 2.10.3: Producer only. Sets the minimum number of instances allowed in the producer pool before the evictor thread (if active) spawns new objects. |
| producerPoolMaxIdle | 100 | Camel 2.10.3: Producer only. Sets the cap on the number of "idle" instances in the pool. |
| producerPoolMinEvictableIdle | 30000 | Camel 2.10.3: Producer only. 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. |
| bootstrapConfiguration | null | Camel 2.12: Consumer only. Allows to configure the Netty ServerBootstrap options using a org.apache.camel.component.netty.NettyServerBootstrapConfiguration instance. This can be used to reuse the same configuration for multiple consumers, to align their configuration more easily. |

| | | |
|-------------------------|-------------|--|
| bossPoll | null | Camel 2.12: To use a explicit <code>org.jboss.netty.channel.socket.nio.BossPool</code> 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. |
| workerPool | null | Camel 2.12: To use a explicit <code>org.jboss.netty.channel.socket.nio.WorkerPool</code> 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. |
| networkInterface | null | Camel 2.12: Consumer only. When using UDP then this option can be used to specify a network interface by its name, such as <code>eth0</code> to join a multicast group. |

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 org.jboss.netty.channel.ChannelDownStreamHandler . |
| 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 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.

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.

SENDING MESSAGES TO/FROM A NETTY ENDPOINT

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.

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.

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);
                }
            })
    }
};
```

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");
    }
};
```

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();

```

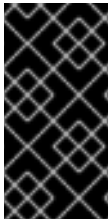
Remember to set `needClientAuth=true` to authenticate the client, otherwise `SSLSession` cannot access information about the client certificate, and you may get an exception `javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated`. You may also get this exception if the client certificate is 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`.

USING MULTIPLE CODECS

In certain cases it may be necessary to add chains of encoders and decoders to the netty pipeline. To add multiple 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.

**IMPORTANT**

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.netty.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="org.jboss.netty.handler.codec.string.StringDecoder"/>
</util:list>

<util:list id="encoders" list-class="java.util.LinkedList">
    <bean
class="org.jboss.netty.handler.codec.frame.LengthFieldPrepender">
    <constructor-arg value="4"/>
    </bean>
    <bean class="org.jboss.netty.handler.codec.string.StringEncoder"/>
</util:list>

    <bean id="length-encoder"
class="org.jboss.netty.handler.codec.frame.LengthFieldPrepender">
    <constructor-arg value="4"/>
    </bean>
    <bean id="string-encoder"
class="org.jboss.netty.handler.codec.string.StringEncoder"/>

    <bean id="length-decoder"
class="org.apache.camel.component.netty.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="org.jboss.netty.handler.codec.string.StringDecoder"/>

</beans>

```

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("netty:tcp://localhost:
{{port}}?encoders=#encoders&sync=false");

        from("netty:tcp://localhost:{{port}}?decoders=#length-
decoder,#string-decoder&sync=false").to("mock:multiple-codec");
    }
};
}
}

```


or via spring.

```
<camelContext id="multiple-netty-codecs-context"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:multiple-codec"/>
    <to uri="netty:tcp://localhost:5150?
encoders=#encoders&ync=false"/>
  </route>
  <route>
    <from uri="netty:tcp://localhost:5150?decoders=#length-
decoder,#string-decoder&ync=false"/>
    <to uri="mock:multiple-codec"/>
  </route>
</camelContext>
```

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);
        }
    }
});
```

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);
                }
            })
    }
});

```

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 create 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 produces.

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&ync=true&orkerPool=#sharedPool&rderedThreadPoolExecutor=fals
e"/>
  <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&ync=true&orkerPool=#sharedPool&rderedThreadPoolExecutor=fals
e"/>
  <to uri="log:result"/>
  ...
</route>
```

... and so forth.

SEE ALSO

- [Netty HTTP](#)
- [Mina](#)

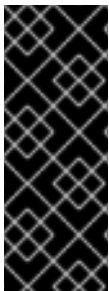
CHAPTER 87. NETTY HTTP

NETTY HTTP COMPONENT

Available as of Camel 2.12

The netty-http component is an extension to [Netty](#) component to facilitate HTTP transport with [Netty](#).

This camel component supports both producer and consumer endpoints.



UPGRADE TO NETTY 4.0 PLANNED

This component is intended to be upgraded to use Netty 4.0 when camel-netty4 component has finished being upgraded. At the time being this component is still based on Netty 3.x. The upgrade is intended to be as backwards compatible as possible.



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.

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>

```

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&...

HTTP OPTIONS



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.

| Name | Default Value | Description |
|--------------------------------------|---------------|--|
| <code>chunkedMaxContentLength</code> | 1mb | Value in bytes the max content length per chunked frame received on the Netty HTTP server. |

| | | |
|-----------------------------|--------------|--|
| compression | false | Allow using gzip/deflate for compression on the Netty HTTP server if the client supports it from the HTTP headers. |
| headerFilterStrategy | | To use a custom org.apache.camel.spi.HeaderFilterStrategy to filter headers. |
| httpMethodRestrict | | To disable HTTP methods on the Netty HTTP consumer. You can specify multiple separated by comma. |
| mapHeaders | true | 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. |
| matchOnUriPrefix | false | Whether or not Camel should try to find a target consumer by matching the URI prefix if no exact match is found. See further below for more details. |
| nettyHttpBinding | | To use a custom org.apache.camel.component.netty.http.NettyHttpBinding for binding to/from Netty and Camel Message API. |

| | | |
|--------------------------------|--------------|--|
| bridgeEndpoint | false | 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. |
| throwExceptionOnFailure | true | 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. |
| traceEnabled | false | Specifies whether to enable HTTP TRACE for this Netty HTTP consumer. By default TRACE is turned off. |
| transferException | false | 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. |

| | | |
|------------------------------|--------------|--|
| urlDecodeHeaders | true | 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. |
| nettySharedHttpServer | null | To use a shared Netty HTTP server. See Netty HTTP Server Example for more details. |
| disableStreamCache | false | Determines whether or not the raw input stream from Netty HttpRequest#getContent() 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. |
| securityConfiguration | null | Consumer only. Refers to a org.apache.camel.component.netty.http.NettyHttpSecurityConfiguration for configuring secure web resources. |

| | | |
|-----------------------------|-------------|---|
| send503whenSuspended | true | Consumer only. 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. |
|-----------------------------|-------------|---|

The `NettyHttpSecurityConfiguration` has the following options:

| Name | Default Value | Description |
|--------------------------------|----------------------|---|
| authenticate | true | Whether authentication is enabled. Can be used to quickly turn this off. |
| constraint | Basic | The constraint supported. Currently only Basic is implemented and supported. |
| realm | null | The name of the JAAS security realm. This option is mandatory. |
| securityConstraint | null | Allows to plugin a security constraint mapper where you can define ACL to web resources. |
| securityAuthenticator | null | Allows to plugin a authenticator that performs the authentication. If none has been configured then the org.apache.camel.component.netty.http.JAASecurityAuthenticator is used by default. |
| loginDeniedLoggingLevel | DEBUG | Logging level used when a login attempt failed, which allows to see more details why the login failed. |

| | | |
|----------------------|-------------|--|
| roleClassName | null | To specify FQN class names of Principal implementations that contains user roles. If none has been specified, then the Netty HTTP component will by default assume a Principal is role based if its FQN classname has the lower-case word role in its classname. You can specify multiple class names separated by comma. |
|----------------------|-------------|--|

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. |
| Content-Type | String | To set the content-type of the HTTP body. For example: text/plain; charset="UTF-8". |

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 |
|---|---------------------|--|
| <code>CamelHttpMethod</code> | <code>String</code> | The HTTP method used, such as GET, POST, TRACE etc. |
| <code>CamelHttpUrl</code> | <code>String</code> | The URL including protocol, host and port, etc: <code>http://0.0.0.0:8080/myapp</code> |
| <code>CamelHttpUri</code> | <code>String</code> | The URI without protocol, host and port, etc: <code>/myapp</code> |
| <code>CamelHttpQuery</code> | <code>String</code> | Any query parameters, such as foo=bar&beer=yes |
| <code>CamelHttpPath</code> | <code>String</code> | 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. |
| <code>CamelHttpCharacterEncoding</code> | <code>String</code> | The charset from the content-type header. |
| <code>CamelHttpAuthentication</code> | <code>String</code> | If the user was authenticated using HTTP Basic then this header is added with the value Basic . |
| <code>Content-Type</code> | <code>String</code> | The content type if provided. For example: text/plain; charset="UTF-8" . |

ACCESS TO NETTY TYPES

This component uses the `org.apache.camel.component.netty.http.NettyHttpRequestMessage` 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:

```
org.jboss.netty.handler.codec.http.HttpRequest request =  
exchange.getIn(NettyHttpRequestMessage.class).getHttpRequest();
```

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.

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:

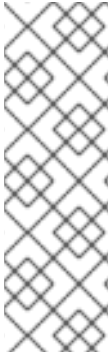
```
from("netty-http:http://0.0.0.0:8123?
matchOnUriPrefix=true").to("mock:foo");
```

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");
```

REUSING SAME SERVER BOOTSTRAP CONFIGURATION WITH MULTIPLE ROUTES

By configuring the common server bootstrap option in a 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="connectionTimeout" 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>

```

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.

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 LoginModules 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.

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&securityConfiguration.realm=karaf&securityConfiguratio
n.securityConstraint=#constraint"/>
  ...
</route>
```

-

Netty

-

Netty HTTP Server Example

-

Jetty

CHAPTER 88. NMR

NMR COMPONENT

The `nmr` component is an adapter to the Normalized Message Router (NMR) in `ServiceMix`, which is intended for use by Camel applications deployed directly into the OSGi container. You can exchange objects with NMR and not only XML like this is the case with the JBI specification. The interest of this component is that you can interconnect camel routes deployed in different OSGI bundles.

By contrast, the `JBI` component is intended for use by Camel applications deployed into the `ServiceMix` JBI container.

INSTALLING IN APACHE SERVICEMIX

The NMR component is provided with Apache ServiceMix. It is *not* distributed with Apache Camel. To install the NMR component in ServiceMix, enter the following command in the ServiceMix console window:

```
features:install nmr camel-nmr
```

INSTALLING IN PLAIN APACHE KARAF

In plain Karaf the `nmr` component can also be installed using the `servicemix` artifacts:

```
features:chooseurl camel <version>
features:addurl mvn:org.apache.servicemix.nmr/apache-servicemix-
nmr/1.5.0/xml/features
features:install camel-blueprint nmr camel-nmr
install -s
mvn:org.apache.servicemix.camel/org.apache.servicemix.camel.component/4.4.
2
```

CONFIGURATION

You also need to instantiate the NMR component. You can do this by editing your Spring configuration file, `META-INF/spring/*.xml`, and adding the

following bean instance:

```
<beans xmlns:osgi="http://www.springframework.org/schema/osgi" ... >
  ...
  <bean id="nmr"
class="org.apache.servicemix.camel.nmr.ServiceMixComponent">
    <property name="nmr">
      <osgi:reference interface="org.apache.servicemix.nmr.api.NMR"
/>
    </property>
  </bean>
  ...
</beans>
```

NMR CONSUMER AND PRODUCER ENDPOINTS

The following code:

```
from("nmr:MyServiceEndpoint")
```

Automatically exposes a new endpoint to the bus with endpoint name `MyServiceEndpoint` (see [#URI-format](#)).

When an NMR endpoint appears at the end of a route, for example:

```
to("nmr:MyServiceEndpoint")
```

The messages sent by this producer endpoint are sent to the already deployed NMR endpoint.

URI FORMAT

An NMR endpoint has the following URI format:

`nmr:endpointName`

URI OPTIONS

An NMR endpoint supports the following options:

| Option | Default Value | Description |
|--------------------------------------|--------------------|---|
| <code>synchronous</code> | <code>false</code> | When this is set to true on a consumer endpoint, an incoming, synchronous NMR Exchange will be handled on the sender's thread instead of being handled on a new thread of the NMR endpoint's thread pool |
| <code>runAsSubject</code> | <code>false</code> | Apache ServiceMix 4.4: When this is set to true on a consumer endpoint, the endpoint will be invoked on behalf of the Subject that is set on the Exchange (i.e. the call to Subject.getSubject(AccessControlContext) will return the Subject instance) |
| <code>timeout</code> | <code>0</code> | Apache ServiceMix 4.4: When this is set to a value greater than 0, the producer endpoint will timeout if it doesn't receive a response from the NMR within the given timeout period (in milliseconds). Configuring a timeout value will switch to using synchronous interactions with the NMR instead of the usual asynchronous messaging. |
| <code>throwExceptionOnFailure</code> | <code>true</code> | Apache ServiceMix 4.5.2: When this is set to false then NMR's exceptions (like <code>TimeoutException</code>) will be consumed silently. |

EXAMPLES

Consumer:

```
// consume nmr exchanges asynchronously
from("nmr:MyServiceEndpoint")
// consume nmr exchanges synchronously and use the same thread as defined
by NMR ThreadPool
from("nmr:MyServiceEndpoint?synchronous=true").to()
```

Producer:

```
// produce nmr exchanges asynchronously
from()...to("nmr:MyServiceEndpoint")
// produce nmr exchanges synchronously and wait till 10s to receive
response
from()...to("nmr:MyServiceEndpoint?timeout=10000")
```

USING STREAM BODIES

If you are using a stream type as the message body, you should be aware that a stream is only capable of being read once. So if you enable `DEBUG` logging, the body is usually logged and thus read. To deal with this, Camel has a `streamCaching` option that can cache the stream, enabling you to read it multiple times.

```
from("nmr:MyEndpoint").streamCaching().to("xslt:transform.xsl",
"bean:doSomething");
```

From Camel 1.5 onwards, the stream caching is default enabled, so it is not necessary to set the `streamCaching()` option. In Camel 2.0 we store big input streams (by default, over 64K) in a temp file using `CachedOutputStream`. When you close the input stream, the temp file will be deleted.

TESTING

NMR camel routes can be tested using the Apache Camel unit test approach even if they will be deployed next in different bundles in an OSGi runtime. With this aim in view, you will extend the `ServiceMixNMR` Mock class, `org.apache.servicemix.camel.nmr.AbstractComponentTest`, which will create an NMR bus, register the Apache Camel NMR Component and the endpoints defined into the Apache Camel routes.

```
public class ExchangeUsingNMRTTest extends AbstractComponentTest {
```



```
@Test
public void testProcessing() throws InterruptedException {
    MockEndpoint mock = getMockEndpoint("mock:simple");
    mock.expectedBodiesReceived("Simple message body");

    template.sendBody("direct:simple", "Simple message body");

    assertMockEndpointsSatisfied();
}

@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {

        @Override
        public void configure() throws Exception {
            from("direct:simple").to("nmr:simple");
            from("nmr:simple?synchronous=true").to("mock:simple");
        }
    };
}
}
```

CHAPTER 89. PAX-LOGGING

PAXLOGGING COMPONENT

Available in Camel 2.6

The `paxlogging` component can be used in an OSGi environment to receive [PaxLogging](#) events and process them.

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).

URI FORMAT

```
paxlogging:appender
```

where `appender` is the name of the pax appender that need to be configured in the `PaxLogging` service configuration.

URI OPTIONS

| Name | Default value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

MESSAGE HEADERS

| Name | Type | Message | Description |
|------|------|---------|-------------|
|------|------|---------|-------------|

MESSAGE BODY

The in message body will be set to the received PaxLoggingEvent.

EXAMPLE USAGE

```
<route>
  <from uri="paxlogging:camel"/>
  <to uri="stream:out"/>
```

`</route>`

Configuration:

`log4j.rootLogger=INFO, out, osgi:VmLogAppender, osgi:camel`

CHAPTER 90. PRINTER

PRINTER COMPONENT

Available as of Apache Camel 2.1

The printer component provides a way to direct payloads on a route to a printer. Obviously the payload has to be a formatted piece of payload in order for the component to appropriately print it. The objective is to be able to direct specific payloads as jobs to a line printer in a Apache Camel flow.

This component only supports a producer endpoint.

The functionality allows for the payload to be printed on a default printer, named local, remote or wirelessly linked printer using the javax printing API under the covers.

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

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

URI FORMAT

Since the URI scheme for a printer has not been standardized (the nearest thing to a standard being the IETF print standard) and therefore not uniformly applied by vendors, we have chosen "lpr" as the scheme.

```
lpr://localhost/default[?options]
lpr://remotehost:port/path/to/printer[?options]
```

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

OPTIONS

| Name | Default Value | Description |
|----------------------------|--------------------------------------|---|
| <code>mediaSize</code> | <code>MediaSizeName.NA_LETTER</code> | Sets the stationary as defined by enumeration settings in the javax.print.attribute.standard.MediaSizeName API . The default setting is to use North American Letter sized stationary |
| <code>copies</code> | <code>1</code> | Sets number of copies based on the javax.print.attribute.standard.Copies API |
| <code>sides</code> | <code>Sides.ONE_SIDED</code> | Sets one sided or two sided printing based on the javax.print.attribute.standard.Sides API |
| <code>flavor</code> | <code>DocFlavor.BYTE_ARRAY</code> | Sets <code>DocFlavor</code> based on the javax.print.DocFlavor API |
| <code>mimeType</code> | <code>AUTOSENSE</code> | Sets <code>mimeTypes</code> supported by the javax.print.DocFlavor API |
| <code>mediaTray</code> | <code>AUTOSENSE</code> | Since Camel 2.11.x sets MediaTray supported by the javax.print.DocFlavor API |
| <code>printerPrefix</code> | <code>null</code> | Since Camel 2.11.x sets the prefix name of the printer, it is useful when the printer name is not start with <code>//hostname/printer</code> |
| <code>sendToPrinter</code> | <code>true</code> | Setting this option to false prevents sending of the print data to the printer |

PRINTER PRODUCER

Sending data to the printer is very straightforward and involves creating a producer endpoint that can be sent message exchanges on in route.

EXAMPLE 1: PRINTING TEXT BASED PAYLOADS ON A DEFAULT PRINTER USING LETTER STATIONARY AND ONE-SIDED MODE

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from(file://inputdir/?delete=true)
        .to("lpr://localhost/default?copies=2" +
            "&flavor=DocFlavor.INPUT_STREAM" +
            "&mimeType=AUTOSENSE" +
            "&mediaSize=na-letter" +
            "&sides=one-sided")
    }
};
```

EXAMPLE 2: PRINTING GIF BASED PAYLOADS ON A REMOTE PRINTER USING A4 STATIONARY AND ONE-SIDED MODE

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from(file://inputdir/?delete=true)
        .to("lpr://remotehost/sales/salesprinter" +
            "?copies=2&sides=one-sided" +
            "&mimeType=GIF&mediaSize=iso-a4" +
            "&flavor=DocFlavor.INPUT_STREAM")
    }
};
```

EXAMPLE 3: PRINTING JPEG BASED PAYLOADS ON A REMOTE PRINTER USING JAPANESE POSTCARD STATIONARY AND ONE-SIDED MODE

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from(file://inputdir/?delete=true)
        .to("lpr://remotehost/sales/salesprinter" +
            "?copies=2&sides=one-sided" +
            "&mimeType=JPEG" +
            "&mediaSize=japanese-postcard" +
            "&flavor=DocFlavor.INPUT_STREAM")
    }
};
```

CHAPTER 91. PROPERTIES

PROPERTIES COMPONENT

Available as of Apache Camel 2.3

URI FORMAT

```
properties:key[?options]
```

Where `key` is the key for the property to lookup

OPTIONS

| Name | Type | Default | Description |
|------------------------------------|----------------------|--------------------|---|
| <code>cache</code> | <code>boolean</code> | <code>true</code> | Whether or not to cache loaded properties. |
| <code>locations</code> | <code>String</code> | <code>null</code> | 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. |
| <code>ignoreMissingLocation</code> | <code>boolean</code> | <code>false</code> | <i>*Camel 2.10:*</i> Whether to silently ignore if a location cannot be located, such as a properties file not found. |
| <code>propertyPrefix</code> | <code>String</code> | <code>null</code> | <i>*Camel 2.9*</i> Optional prefix prepended to property names before resolution. |

| | | | |
|--------------------------------------|----------------|-------------|--|
| propertySuffix | String | null | *Camel 2.9* Optional suffix appended to property names before resolution. |
| fallbackToUnaugmentedProperty | boolean | true | *Camel 2.9* 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. |
| prefixToken | String | {{ | *Camel 2.9* The token to indicate the beginning of a property token. |
| suffixToken | String | }} | *Camel 2.9* The token to indicate the end of a property token. |

RESOLVING PROPERTY FROM JAVA CODE

You can use the method `resolvePropertyPlaceholders` on the `CamelContext` to resolve a property from any Java code.

SEE ALSO

- [section "Property Placeholders" in "Apache Camel Development Guide"](#)
- [Jasypt](#) for using encrypted values (for example, passwords) in the properties

CHAPTER 92. QUARTZ

QUARTZ COMPONENT

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`).



NOTE

If you are using Quartz 2.x then from Camel 2.12 onwards there is a [Quartz2](#) component you should use

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&...`

OPTIONS

| Parameter | Default | Description |
|----------------------------------|-------------|--|
| <code>cron</code> | <i>None</i> | Specifies a cron expression (not compatible with the trigger.* or job.* options). |
| <code>trigger.repeatCount</code> | 0 | <code>SimpleTrigger</code> : How many times should the timer repeat? |

| | | |
|-------------------------------------|--------------------|--|
| <code>trigger.repeatInterval</code> | <code>0</code> | SimpleTrigger: The amount of time in milliseconds between repeated triggers. |
| <code>job.name</code> | <code>null</code> | Sets the job name. |
| <code>job.XXX</code> | <code>null</code> | Sets the job option with the <code>XXX</code> setter name. |
| <code>trigger.XXX</code> | <code>null</code> | Sets the trigger option with the <code>XXX</code> setter name. |
| <code>stateful</code> | <code>false</code> | Uses a Quartz StatefulJob instead of the default job. |
| <code>fireNow</code> | <code>false</code> | New to Camel 2.2.0, if it is true will fire the trigger when the route is start when using SimpleTrigger. |
| <code>deleteJob</code> | <code>true</code> | Camel 2.12: 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 <code>deleteJob</code> and <code>pauseJob</code> set to true. |
| <code>pauseJob</code> | <code>false</code> | Camel 2.12: 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 <code>deleteJob</code> and <code>pauseJob</code> set to true. |

For example, the following routing rule will fire two timer events to the `mock:results` endpoint:

```
from("quartz://myGroup/myTimerName?
trigger.repeatInterval=2&trigger.repeatCount=1").routeId("myRoute").to("mo
ck:result");
```

When using a `StatefulJob`, the `JobDataMap` is re-persisted after every execution of the job, thus preserving state for the next execution.



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.

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 |
|-----------------------------|-------------------|-------------------------|---|
| <code>properties</code> | <code>null</code> | <code>Properties</code> | Camel 2.4: You can configure a <code>java.util.Properties</code> instance. |
| <code>propertiesFile</code> | <code>null</code> | <code>String</code> | 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>
```

STARTING THE QUARTZ SCHEDULER

Available as of Camel 2.4

The `Quartz` component offers an option to let the Quartz scheduler be started delayed, or not auto started at all.

| Parameter | Default | Type | Description |
|----------------------------------|---------|---------|---|
| <code>startDelayedSeconds</code> | 0 | int | Camel 2.4: Seconds to wait before starting the quartz scheduler. |
| <code>autoStartScheduler</code> | true | boolean | Camel 2.4: Whether or not the scheduler should be auto started. |

To do this you can configure this in Spring XML as follows

```
<bean id="quartz"
class="org.apache.camel.component.quartz.QuartzComponent">
  <property name="startDelayedSeconds" value="5"/>
</bean>
```

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.

MESSAGE HEADERS

Apache Camel adds the getters from the Quartz Execution Context as header values. The following headers are added: `calendar`, `fireTime`, `jobDetail`, `jobInstance`, `jobRuntime`, `mergedJobDataMap`, `nextFireTime`, `previousFireTime`, `refireCount`, `result`, `scheduledFireTime`, `scheduler`, `trigger`, `triggerName`, `triggerGroup`.

The `fireTime` header contains the `java.util.Date` of when the exchange was fired.

USING CRON TRIGGERS

Available as of Apache Camel 2.0 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 |
|-----------------|----------------|
| <code>\+</code> | <i>Space</i> |

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`.

- [Quartz2](#)
- [Timer](#)

CHAPTER 93. QUARTZ2

QUARTZ2 COMPONENT

Available as of Camel 2.12.0

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.

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&...

OPTIONS

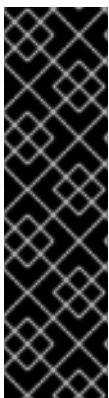
| Parameter | Default | Description |
|-------------------------------------|--------------|--|
| <code>cron</code> | <i>None</i> | Specifies a cron expression (not compatible with the trigger.* or job.* options). |
| <code>trigger.repeatCount</code> | 0 | SimpleTrigger: How many times should the timer repeat? |
| <code>trigger.repeatInterval</code> | 0 | SimpleTrigger: The amount of time in milliseconds between repeated triggers. |
| <code>job.name</code> | null | Sets the job name. |
| <code>job.XXX</code> | null | Sets the job option with the XXX setter name. |
| <code>trigger.XXX</code> | null | Sets the trigger option with the XXX setter name. |
| <code>stateful</code> | false | Uses a Quartz @PersistJobDataAfterExecution and @DisallowConcurrentExecution instead of the default job. |
| <code>fireNow</code> | false | If it is true will fire the trigger when the route is start when using SimpleTrigger. |

| | | |
|------------------|--------------|---|
| deleteJob | true | 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. |
| pauseJob | false | 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. |

For example, the following routing rule will fire two timer events to the `mock:result` endpoint:

```
from("quartz2://myGroup/myTimerName?
trigger.repeatInterval=2&trigger.repeatCount=1").routeId("myRoute").to("mo
ck:result");
```

When using `stateful=true`, the `JobDataMap` is re-persisted after every execution of the job, thus preserving state for the next execution.



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.

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 |
|-----------------------------|-------------------|-------------------------|---|
| <code>properties</code> | <code>null</code> | <code>Properties</code> | You can configure a <code>java.util.Properties</code> instance. |
| <code>propertiesFile</code> | <code>null</code> | <code>String</code> | 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.quartz2.QuartzComponent">
  <property name="propertiesFile"
value="com/mycompany/myquartz.properties"/>
</bean>
```

STARTING THE QUARTZ SCHEDULER

The [Quartz2](#) component offers an option to let the Quartz scheduler be started delayed, or not auto started at all.

| Parameter | Default | Type | Description |
|----------------------------------|----------------|------------------|---|
| <code>startDelayedSeconds</code> | <code>0</code> | <code>int</code> | Seconds to wait before starting the quartz scheduler. |

| | | | |
|---------------------------------|-------------------|----------------------|--|
| <code>autoStartScheduler</code> | <code>true</code> | <code>boolean</code> | Whether or not the scheduler should be auto started. |
|---------------------------------|-------------------|----------------------|--|

To do this you can configure this in Spring XML as follows

```
<bean id="quartz"  
class="org.apache.camel.component.quartz2.QuartzComponent">  
  <property name="startDelayedSeconds" value="5"/>  
</bean>
```

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.

MESSAGE HEADERS

Camel adds the getters from the Quartz Execution Context as header values. The following headers are added: `calendar`, `fireTime`, `jobDetail`, `jobInstance`, `jobRuntime`, `mergedJobDataMap`, `nextFireTime`, `previousFireTime`, `refireCount`, `result`, `scheduledFireTime`, `scheduler`, `trigger`, `triggerName`, `triggerGroup`.

The `fireTime` header contains the `java.util.Date` of when the exchange was fired.

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("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 |
|---------------|----------------|
| \+ | <i>Space</i> |

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`.

USING QUARTZSCHEDULEDPOLLCONSUMERSCHEDULER

The [Quartz2](#) component provides a [Polling Consumer](#) scheduler which allows to use cron based scheduling for [Polling Consumers](#) 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 |
|------------------------------|-------------------|-----------------------------------|---|
| <code>quartzScheduler</code> | <code>null</code> | <code>org.quartz.Scheduler</code> | To use a custom Quartz scheduler. If none configure then the shared scheduler from the Quartz2 component is used. |
| <code>cron</code> | <code>null</code> | <code>String</code> | Mandatory: To define the cron expression for triggering the polls. |
| <code>triggerId</code> | <code>null</code> | <code>String</code> | To specify the trigger id. If none provided then an UUID is generated and used. |

| | | | |
|---------------------------|---|-----------------------|--|
| <code>triggerGroup</code> | <code>QuartzScheduledPollConsumerScheduler</code> | <code>String</code> | To specify the trigger group. |
| <code>timeZone</code> | <code>Default</code> | <code>TimeZone</code> | 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");
```

-

Quartz

-

Timer

CHAPTER 94. QUICKFIX

QUICKFIX/J COMPONENT

Available as of Camel 2.0

The quickfix component adapts the [QuickFIX/J](#) FIX engine for using in Camel. This component uses the standard [Financial Interchange \(FIX\) protocol](#) for message transport.



PREVIOUS VERSIONS

The quickfix component was rewritten for Camel 2.5. For information about using the quickfix component prior to 2.5 see the documentation section 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-quickfix</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

URI FORMAT

```
quickfix:configFile[?sessionId=sessionID]
```

The configFile is the name of the QuickFIX/J configuration to use for the FIX engine (located as a resource found in your classpath). The optional sessionId identifies a specific FIX session. The format of the sessionId is:

```
(BeginString):(SenderCompID)[/(SenderSubID)[/(SenderLocationID)]]->
(TargetCompID)[/(TargetSubID)[/(TargetLocationID)]]
```

Example URIs:


```
quickfix:config.cfg
```

```
quickfix:config.cfg?sessionID=FIX.4.2:MyTradingCompany->SomeExchange
```

ENDPOINTS

FIX sessions are endpoints for the quickfix component. An endpoint URI may specify a single session or all sessions managed by a specific QuickFIX/J engine. Typical applications will use only one FIX engine but advanced users may create multiple FIX engines by referencing different configuration files in quickfix component endpoint URIs.

When a consumer does not include a session ID in the endpoint URI, it will receive exchanges for all sessions managed by the FIX engine associated with the configuration file specified in the URI. If a producer does not specify a session in the endpoint URI then it must include the session-related fields in the FIX message being sent. If a session is specified in the URI then the component will automatically inject the session-related fields into the FIX message.

EXCHANGE FORMAT

The exchange headers include information to help with exchange filtering, routing and other processing. The following headers are available:

| Header Name | Description |
|-----------------------|---|
| EventCategory | One of AppMessageReceived , AppMessageSent , AdminMessageReceived , AdminMessageSent , SessionCreated , SessionLogon , SessionLogoff . See the QuickfixjEventCategory enum. |
| SessionID | The FIX message SessionID |
| MessageType | The FIX MsgType tag value |
| DataDictionary | Specifies a data dictionary to used for parsing an incoming message. Can be an instance of a data dictionary or a resource path for a QuickFIX/J data dictionary file |

The **DataDictionary** header is useful if string messages are being received and need to be parsed in a route. QuickFIX/J requires a data dictionary to

parse certain types of messages (with repeating groups, for example). By injecting a `DataDictionary` header in the route after receiving a message string, the FIX engine can properly parse the data.

QUICKFIX/J CONFIGURATION EXTENSIONS

When using QuickFIX/J directly, one typically writes code to create instances of logging adapters, message stores and communication connectors. The quickfix component will automatically create instances of these classes based on information in the configuration file. It also provides defaults for many of the common required settings and adds additional capabilities (like the ability to activate JMX support).

The following sections describe how the quickfix component processes the QuickFIX/J configuration. For comprehensive information about QuickFIX/J configuration, see the [QFJ user manual](#).

COMMUNICATION CONNECTORS

When the component detects an initiator or acceptor session setting in the QuickFIX/J configuration file it will automatically create the corresponding initiator and/or acceptor connector. These settings can be in the default or in a specific session section of the configuration file.

| Session Setting | Component Action |
|---------------------------------------|-------------------------------|
| <code>ConnectionType=initiator</code> | Create an initiator connector |
| <code>ConnectionType=acceptor</code> | Create an acceptor connector |

The threading model for the QuickFIX/J session connectors can also be specified. These settings affect all sessions in the configuration file and must be placed in the settings default section.

| Default/Global Setting | Component Action |
|---|---|
| <code>ThreadModel=ThreadPerConnector</code> | Use <code>SocketInitiator</code> or <code>SocketAcceptor</code> (default) |
| <code>ThreadModel=ThreadPerSession</code> | Use <code>ThreadedSocketInitiator</code> or <code>ThreadedSocketAcceptor</code> |

LOGGING

The QuickFIX/J logger implementation can be specified by including the following settings in the default section of the configuration file. The `ScreenLog` is the default if none of the following settings are present in the configuration. It's an error to include settings that imply more than one log implementation.

| Default/Global Setting | Component Action |
|------------------------------------|---|
| <code>ScreenLogShowEvents</code> | Use a <code>ScreenLog</code> |
| <code>ScreenLogShowIncoming</code> | Use a <code>ScreenLog</code> |
| <code>ScreenLogShowOutgoing</code> | Use a <code>ScreenLog</code> |
| <code>SLF4J*</code> | Camel 2.6+ . Use a <code>SLF4JLog</code> . Any of the SLF4J settings will cause this log to be used. |
| <code>FileLogPath</code> | Use a <code>FileLog</code> |
| <code>JdbcDriver</code> | Use a <code>JdbcLog</code> |

MESSAGE STORE

The QuickFIX/J message store implementation can be specified by including the following settings in the default section of the configuration file. The `MemoryStore` is the default if none of the following settings are present in the configuration. It's an error to include settings that imply more than one message store implementation.

| Default/Global Setting | Component Action |
|-----------------------------------|-----------------------------------|
| <code>JdbcDriver</code> | Use a <code>JdbcStore</code> |
| <code>FileStorePath</code> | Use a <code>FileStore</code> |
| <code>SleepycatDatabaseDir</code> | Use a <code>SleepycatStore</code> |

MESSAGE FACTORY

A message factory is used to construct domain objects from raw FIX messages. The default message factory is `DefaultMessageFactory`. However,

advanced applications may require a custom message factory. This can be set on the QuickFIX/J component.

JMX

| Default/Global Setting | Component Action |
|------------------------|----------------------------------|
| UseJmx | if Y, then enable QuickFIX/J JMX |

OTHER DEFAULTS

The component provides some default settings for what are normally required settings in QuickFIX/J configuration files. `SessionStartTime` and `SessionEndTime` default to "00:00:00", meaning the session will not be automatically started and stopped. The `HeartBtInt` (heartbeat interval) defaults to 30 seconds.

MINIMAL INITIATOR CONFIGURATION EXAMPLE

```
[SESSION]
ConnectionType=initiator
BeginString=FIX.4.4
SenderCompID=YOUR_SENDER
TargetCompID=YOUR_TARGET
```

USING THE INOUT MESSAGE EXCHANGE PATTERN

Camel 2.8+

Although the FIX protocol is event-driven and asynchronous, there are specific pairs of messages that represent a request-reply message exchange. To use an InOut exchange pattern, there should be a single request message and single reply message to the request. Examples include an `OrderStatusRequest` message and `UserRequest`.

IMPLEMENTING INOUT EXCHANGES FOR CONSUMERS

Add "exchangePattern=InOut" to the QuickFIX/J endpoint URI. The `MessageOrderStatusService` in the example below is a bean with a synchronous service method. The method returns the response to the request (an

ExecutionReport in this case) which is then sent back to the requestor session.

```

    from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:MARKET-
    >TRADER&exchangePattern=InOut")

    .filter(header(QuickfixjEndpoint.MESSAGE_TYPE_KEY).isEqualTo(MsgType.ORDER
    _STATUS_REQUEST))
        .bean(new MarketOrderStatusService());

```

IMPLEMENTING INOUT EXCHANGES FOR PRODUCERS

For producers, sending a message will block until a reply is received or a timeout occurs. There is no standard way to correlate reply messages in FIX. Therefore, a correlation criteria must be defined for each type of InOut exchange. The correlation criteria and timeout can be specified using Exchange properties.

| Description | Key String | Key Constant | Default | Correlation Criteria | "CorrelationCriteria" | QuickfixjProducer.CORRELATION_CRITERIA_KEY | None |
|-------------------------------------|----------------------|---|---------|----------------------|-----------------------|--|------|
| Correlation Timeout in Milliseconds | "CorrelationTimeout" | QuickfixjProducer.CORRELATION_TIMEOUT_KEY | 1000 | | | | |

The correlation criteria is defined with a MessagePredicate object. The following example will treat a FIX ExecutionReport from the specified session where the transaction type is STATUS and the Order ID matches our request. The session ID should be for the *requestor*, the sender and target CompID fields will be reversed when looking for the reply.

```

exchange.setProperty(QuickfixjProducer.CORRELATION_CRITERIA_KEY,
    new MessagePredicate(new SessionID(sessionID),
    MsgType.EXECUTION_REPORT)
        .withField(ExecTransType.FIELD,
    Integer.toString(ExecTransType.STATUS))
        .withField(OrderID.FIELD, request.getString(OrderID.FIELD)));

```

EXAMPLE

The source code contains an example called `RequestReplyExample` that demonstrates the `InOut` exchanges for a consumer and producer. This example creates a simple HTTP server endpoint that accepts order status requests. The HTTP request is converted to a `FIX OrderStatusRequestMessage`, is augmented with a correlation criteria, and is then routed to a quickfix endpoint. The response is then converted to a JSON-formatted string and sent back to the HTTP server endpoint to be provided as the web response.

The Spring configuration have changed from Camel 2.9 onwards. See further below for example.

SPRING CONFIGURATION

Camel 2.6 - 2.8.x

The QuickFIX/J component includes a Spring `FactoryBean` for configuring the session settings within a Spring context. A type converter for QuickFIX/J session ID strings is also included. The following example shows a simple configuration of an acceptor and initiator session with default settings for both sessions.

```
<!-- camel route -->
  <camelContext id="quickfixjContext"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="quickfix:example"/>
    <filter>
      <simple>${in.header.EventCategory} == 'AppMessageReceived'</simple>
      <to uri="log:test"/>
    </filter>
  </route>
</camelContext>

<!-- quickfix component -->
<bean id="quickfix"
class="org.apache.camel.component.quickfixj.QuickfixjComponent">
  <property name="engineSettings">
    <util:map>
      <entry key="quickfix:example" value-ref="quickfixjSettings"/>
    </util:map>
  </property>
  <property name="messageFactory">
    <bean
class="org.apache.camel.component.quickfixj.QuickfixjSpringTest.CustomMess
```

```

ageFactory"/>
  </property>
</bean>

<!-- quickfix settings -->
<bean id="quickfixjSettings"

class="org.apache.camel.component.quickfixj.QuickfixjSettingsFactory">
  <property name="defaultSettings">
    <util:map>
      <entry key="SocketConnectProtocol" value="VM_PIPE"/>
      <entry key="SocketAcceptProtocol" value="VM_PIPE"/>
      <entry key="UseDataDictionary" value="N"/>
    </util:map>
  </property>
  <property name="sessionSettings">
    <util:map>
      <entry key="FIX.4.2:INITIATOR->ACCEPTOR">
        <util:map>
          <entry key="ConnectionType" value="initiator"/>
          <entry key="SocketConnectHost" value="localhost"/>
          <entry key="SocketConnectPort" value="5000"/>
        </util:map>
      </entry>
      <entry key="FIX.4.2:ACCEPTOR->INITIATOR">
        <util:map>
          <entry key="ConnectionType" value="acceptor"/>
          <entry key="SocketAcceptPort" value="5000"/>
        </util:map>
      </entry>
    </util:map>
  </property>
</bean>

```

Camel 2.9 onwards

The QuickFIX/J component includes a `QuickfixjConfiguration` class for configuring the session settings. A type converter for QuickFIX/J session ID strings is also included. The following example shows a simple configuration of an acceptor and initiator session with default settings for both sessions.

```

<!-- camel route -->
<camelContext id="quickfixjContext"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="quickfix:example"/>
    <filter>
      <simple>${in.header.EventCategory} ==
'AppMessageReceived'</simple>
      <to uri="log:test"/>
    </filter>
  </route>

```

```

</camelContext>

<!-- quickfix component -->
<bean id="quickfix"
class="org.apache.camel.component.quickfixj.QuickfixjComponent">
  <property name="configurations">
    <util:map>
      <entry key="example" value-ref="quickfixjConfiguration"/>
    </util:map>
  </property>
  <property name="messageFactory">
    <bean
class="org.apache.camel.component.quickfixj.QuickfixjSpringTest.CustomMess
ageFactory"/>
  </property>
</bean>

<!-- quickfix settings -->
<bean id="quickfixjConfiguration"
class="org.apache.camel.component.quickfixj.QuickfixjConfiguration">
  <property name="defaultSettings">
    <util:map>
      <entry key="SocketConnectProtocol" value="VM_PIPE"/>
      <entry key="SocketAcceptProtocol" value="VM_PIPE"/>
      <entry key="UseDataDictionary" value="N"/>
    </util:map>
  </property>
  <property name="sessionSettings">
    <util:map>
      <entry key="FIX.4.2:INITIATOR->ACCEPTOR">
        <util:map>
          <entry key="ConnectionType" value="initiator"/>
          <entry key="SocketConnectHost" value="localhost"/>
          <entry key="SocketConnectPort" value="5000"/>
        </util:map>
      </entry>
      <entry key="FIX.4.2:ACCEPTOR->INITIATOR">
        <util:map>
          <entry key="ConnectionType" value="acceptor"/>
          <entry key="SocketAcceptPort" value="5000"/>
        </util:map>
      </entry>
    </util:map>
  </property>
</bean>

```

EXCEPTION HANDLING

QuickFIX/J behavior can be modified if certain exceptions are thrown during processing of a message. If a `RejectLogon` exception is thrown while processing an incoming logon administrative message, then the logon will be rejected.

Normally, QuickFIX/J handles the logon process automatically. However, sometimes an outgoing logon message must be modified to include credentials required by a FIX counterparty. If the FIX logon message body is modified when sending a logon message (`EventCategory={{AdminMessageSent}}`) the modified message will be sent to the counterparty. It is important that the outgoing logon message is being processed *synchronously*. If it is processed asynchronously (on another thread), the FIX engine will immediately send the unmodified outgoing message when its callback method returns.

FIX SEQUENCE NUMBER MANAGEMENT

If an application exception is thrown during *synchronous* exchange processing, this will cause QuickFIX/J to not increment incoming FIX message sequence numbers and will cause a resend of the counterparty message. This FIX protocol behavior is primarily intended to handle *transport* errors rather than application errors. There are risks associated with using this mechanism to handle application errors. The primary risk is that the message will repeatedly cause application errors each time it's re-received. A better solution is to persist the incoming message (database, JMS queue) immediately before processing it. This also allows the application to process messages asynchronously without losing messages when errors occur.

Although it's possible to send messages to a FIX session before it's logged on (the messages will be sent at logon time), it is usually a better practice to wait until the session is logged on. This eliminates the required sequence number resynchronization steps at logon. Waiting for session logon can be done by setting up a route that processes the `SessionLogon` event category and signals the application to start sending messages.

See the FIX protocol specifications and the QuickFIX/J documentation for more details about FIX sequence number management.

ROUTE EXAMPLES

Several examples are included in the QuickFIX/J component source code (test subdirectories). One of these examples implements a trival trade execution simulation. The example defines an application component that uses the URI scheme "trade-executor".

The following route receives messages for the trade executor session and passes application messages to the trade executor component.

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:MARKET->TRADER").
```

```
filter(header(QuickfixjEndpoint.EVENT_CATEGORY_KEY).isEqualTo(QuickfixjEventCategory.AppMessageReceived)).
    to("trade-executor:market");
```

The trade executor component generates messages that are routed back to the trade session. The session ID must be set in the FIX message itself since no session ID is specified in the endpoint URI.

```
from("trade-executor:market").to("quickfix:examples/inprocess.cfg");
```

The trader session consumes execution report messages from the market and processes them.

```
from("quickfix:examples/inprocess.cfg?sessionID=FIX.4.2:TRADER->MARKET").
filter(header(QuickfixjEndpoint.MESSAGE_TYPE_KEY).isEqualTo(MsgType.EXECUTION_REPORT)).
    bean(new MyTradeExecutionProcessor());
```

QUICKFIX/J COMPONENT PRIOR TO CAMEL 2.5

Available since Camel 2.0

The quickfix component is an implementation of the [QuickFIX/J](#) engine for Java . This engine allows to connect to a FIX server which is used to exchange financial messages according to [FIX protocol](#) standard.

Note: The component can be used to send/receives messages to a FIX server.

URI FORMAT

```
quickfix-server:config file
quickfix-client:config file
```

Where config file is the location (in your classpath) of the quickfix configuration file used to configure the engine at the startup.

Note: Information about parameters available for quickfix can be found on [QuickFIX/J](#) web site.

The quickfix-server endpoint must be used to receive from FIX server FIX messages and quickfix-client endpoint in the case that you want to send messages to a FIX gateway.

EXCHANGE DATA FORMAT

The QuickFIX/J engine is like CXF component a messaging bus using MINA as protocol layer to create the socket connection with the FIX engine gateway.

When QuickFIX/J engine receives a message, then it create a QuickFix.Message instance which is next received by the camel endpoint. This object is a 'mapping object' created from a FIX message formatted initially as a collection of key value pairs data. You can use this object or you can use the method 'toString' to retrieve the original FIX message.

Note: Alternatively, you can use [camel bindy dataformat](#) to transform the FIX message into your own java POJO

When a message must be send to QuickFix, then you must create a QuickFix.Message instance.

SAMPLES

Direction : to FIX gateway

```
<route>
  <from uri="activemq:queue:fix"/>
  <bean ref="fixService" method="createFixMessage" /> // bean method in
charge to transform message into a QuickFix.Message
  <to uri="quickfix-client:META-INF/quickfix/client.cfg" /> // Quickfix
engine who will send the FIX messages to the gateway
</route>
```

Direction : from FIX gateway

```
<route>
  <from uri="quickfix-server:META-INF/quickfix/server.cfg"/> // QuickFix
engine who will receive the message from FIX gateway
```

```
<bean ref="fixService" method="parseFixMessage" /> // bean method
parsing the QuickFix.Message
  <to uri="uri="activemq:queue:fix"/>" />
</route>
```

CHAPTER 95. RABBITMQ

RABBITMQ COMPONENT

Available as of Camel 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>
```

URI FORMAT

```
rabbitmq://hostname[:port]/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.

OPTIONS

| Property | Default | Description |
|-----------------------|--------------------|--|
| autoAck | true | If messages should be auto acknowledged |
| autoDelete | true | If it is true, the message will be deleted when it is consumed |
| durable | true | If message is durable |
| queue | random uuid | The queue to receive messages from |
| routingKey | null | The routing key to use when binding a consumer queue to the exchange. For producer routing keys, you set the header (see header section) |
| threadPoolSize | 10 | The consumer uses a Thread Pool Executor with a fixed number of threads. This setting allows you to set that number of threads. |
| username | null | username in case of authenticated access |
| password | null | password for authenticated access |
| vhost | / | the vhost for the channel |

HEADERS

The following headers are set on exchanges when consuming messages.

| Property | Value |
|----------|-------|
|----------|-------|

| | |
|-------------------------------------|---|
| <code>rabbitmq.ROUTING_KEY</code> | The routing key that was used to receive the message, or the routing key that will be used when producing a message |
| <code>rabbitmq.EXCHANGE_NAME</code> | The exchange the message was received from |
| <code>rabbitmq.DELIVERY_TAG</code> | The rabbitmq delivery tag of the received message |

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 |
|--|--|
| <code>rabbitmq.ROUTING_KEY</code> | The routing key that will be used when sending the message |
| <code>rabbitmq.EXCHANGE_NAME</code> | The exchange the message was received from, or sent to |
| <code>rabbitmq.CONTENT_TYPE</code> | The contentType to set on the RabbitMQ message |
| <code>rabbitmq.PRIORITY</code> | The priority header to set on the RabbitMQ message |
| <code>rabbitmq.CORRELATIONID</code> | The correlationId to set on the RabbitMQ message |
| <code>rabbitmq.MESSAGE_ID</code> | The message id to set on the RabbitMQ message |
| <code>rabbitmq.DELIVERY_MODE</code> | If the message should be persistent or not |
| <code>rabbitmq.USERID</code> | The userId to set on the RabbitMQ message |
| <code>rabbitmq.CLUSTERID</code> | The clusterId to set on the RabbitMQ message |
| <code>rabbitmq.REPLY_TO</code> | The replyTo to set on the RabbitMQ message |
| <code>rabbitmq.CONTENT_ENCODING</code> | The contentEncoding to set on the RabbitMQ message |
| <code>rabbitmq.TYPE</code> | The type to set on the RabbitMQ message |

| | |
|----------------------------------|---|
| <code>rabbitmq.EXPIRATION</code> | The expiration to set on the RabbitMQ message |
| <code>rabbitmq.TIMESTAMP</code> | The timestamp to set on the RabbitMQ message |
| <code>rabbitmq.APP_ID</code> | 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 overridden.

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.

SAMPLES

To receive messages from a queue that is bound to an exchange A with the routing key B,

```
from("rabbitmq://localhost/A?routingKey=B")
```

To receive messages from a queue with a single thread with auto acknowledge disabled.

```
from("rabbitmq://localhost/A?routingKey=B&threadPoolSize=1&autoAck=false")
```

To send messages to an exchange called C

```
...to("rabbitmq://localhost/B")
```


■

CHAPTER 96. REF

REF COMPONENT

The `ref:` component is used for lookup of existing endpoints bound in the [Registry](#).

URI FORMAT

```
ref:someName
```

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.

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>
```

SAMPLE

In the sample below we use the `ref:` in the URI to reference the endpoint with the spring ID, `endpoint2`:

```
<bean id="mybean" class="org.apache.camel.spring.example.DummyBean">
  <property name="endpoint" ref="endpoint1"/>
</bean>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" disabled="true"/>
  <endpoint id="endpoint1" uri="direct:start"/>
  <endpoint id="endpoint2" uri="mock:end"/>

  <route>
    <from ref="endpoint1"/>
    <to uri="ref:endpoint2"/>
  </route>
</camelContext>
```

You could, of course, have used the `ref` attribute instead:

```
<to ref="endpoint2"/>
```

Which is the more common way to write it.

CHAPTER 97. RESTLET

RESTLET COMPONENT

The Restlet component provides [Restlet](#) based [endpoints](#) for consuming and producing RESTful resources.

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&...`

OPTIONS

| Name | Default Value | Description |
|--|---|--|
| <code>headerFilterStrategy=#refName</code> | An instance of RestletHeaderFilterStrategy | Use the # notation (headerFilterStrategy=#refName) to reference a header filter strategy in the Camel Registry. The strategy will be plugged into the restlet binding if it is HeaderFilterStrategyAware . |
| <code>restletBinding=# refName</code> | An instance of DefaultRestletBinding | The bean ID of a RestletBinding object in the Camel Registry. |

| | | |
|---|-------------|--|
| restletMethod | GET | On a producer endpoint, specifies the request method to use. On a consumer endpoint, specifies that the endpoint consumes only restletMethod requests. The string value is converted to org.restlet.data.Method by the Method.valueOf(String) method. |
| restletMethods | <i>None</i> | Consumer only 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. |
| restletRealm | null | Use the # notation (restletRealm=#refName) to specify the bean ID of the Realm Map in the Camel registry. |
| restletUriPatterns=#refName | <i>None</i> | Consumer only Specify one or more URI templates to be serviced by a restlet consumer endpoint, using the # notation to reference a List<String> 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. |
| throwExceptionOnFailure (2.6 or later) | true | <i>Producer only</i> Throws exception on a producer failure. |

COMPONENT OPTIONS

The Restlet component can be configured with the following options. Notice these are component options and cannot be configured on the endpoint, see

further below for an example.

| Name | Default Value | Description |
|------------------------------------|--------------------|---|
| <code>controllerDaemon</code> | <code>true</code> | Camel 2.10: Indicates if the controller thread should be a daemon (not blocking JVM exit). |
| <code>controllerSleepTimeMs</code> | <code>100</code> | Camel 2.10: Time for the controller thread to sleep between each control. |
| <code>inboundBufferSize</code> | <code>8192</code> | Camel 2.10: The size of the buffer when reading messages. |
| <code>minThreads</code> | <code>1</code> | Camel 2.10: Minimum threads waiting to service requests. |
| <code>maxThreads</code> | <code>10</code> | Camel 2.10: Maximum threads that will service requests. |
| <code>maxConnectionsPerHost</code> | <code>-1</code> | Camel 2.10: Maximum number of concurrent connections per host (IP address). |
| <code>maxTotalConnections</code> | <code>-1</code> | Camel 2.10: Maximum number of concurrent connections in total. |
| <code>outboundBufferSize</code> | <code>8192</code> | Camel 2.10: The size of the buffer when writing messages. |
| <code>persistingConnections</code> | <code>true</code> | Camel 2.10: Indicates if connections should be kept alive after a call. |
| <code>pipeliningConnections</code> | <code>false</code> | Camel 2.10: Indicates if pipelining connections are supported. |
| <code>threadMaxIdleTimeMs</code> | <code>60000</code> | Camel 2.10: Time for an idle thread to wait for an operation before being collected. |

| | | |
|------------------------------|--------------|--|
| useForwardedForHeader | false | Camel 2.10: Lookup the "X-Forwarded-For" header supported by popular proxies and caches and uses it to populate the <code>Request.getClientAddresses()</code> 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. |
| reuseAddress | true | Camel 2.10.5/2.11.1: Enable/disable the <code>SO_REUSEADDR</code> socket option. See <code>java.io.ServerSocket#reuseAddress</code> property for additional details. |

MESSAGE HEADERS

| Name | Type | Description |
|-------------------------|---------------|---|
| CamelContentType | 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. nbsp; 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 Apache Camel. |
| 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 Apache 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. |

| | | |
|----------------------------|--|---|
| <code>org.restlet.*</code> | | Attributes of a Restlet message that get propagated to Apache Camel IN headers. |
| <code>cache-control</code> | 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. |

MESSAGE BODY

Apache 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.

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.

```
from("restlet:http://localhost:9080/securedOrders?
restletMethod=post&restletRealm=#realm").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getOut().setBody(
            "received [" + exchange.getIn().getBody()
            + "] as an order id = "
            + exchange.getIn().getHeader("id"));
    }
});
```

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`.

```
<util:map id="realm">
  <entry key="admin" value="foo" />
  <entry key="bar" value="foo" />
</util:map>
```

The following sample starts a direct endpoint that sends requests to the server on <http://localhost:8080> (that is, our restlet consumer endpoint).

```
// Note: restletMethod and restletRealmRef are stripped
// from the query before a request is sent as they are
// only processed by Camel.
from("direct:start-auth").to("restlet:http://localhost:9080/securedOrders?
restletMethod=post");
```

That is all we need. We are ready to send a request and try out the restlet component:

```
final String id = "89531";

Map<String, Object> headers = new HashMap<String, Object>();
headers.put(RestletConstants.RESTLET_LOGIN, "admin");
headers.put(RestletConstants.RESTLET_PASSWORD, "foo");
headers.put("id", id);

String response = (String) template.requestBodyAndHeaders("direct:start-
auth",
    "<order foo='1'/>", headers);
```

The sample client sends a request to the `direct:start-auth` endpoint with the following headers:

- CamelRestletLogin (used internally by Apache Camel)
- CamelRestletPassword (used internally by Apache 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
```

SINGLE RESTLET ENDPOINT TO SERVICE MULTIPLE METHODS AND URI TEMPLATES (2.0 OR LATER)

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:

```
from("restlet:http://localhost:9080/users/{username}?
restletMethods=post,get,put")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // echo the method

exchange.getOut().setBody(exchange.getIn().getHeader(Exchange.HTTP_METHOD,
String.class));

        }
    });
```

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.

```
from("restlet:http://localhost:9080?
restletMethods=post,get&restletUriPatterns=#uriTemplates")
    .process(new Processor() {
        public void process(Exchange exchange) throws Exception {
            // echo the method
            String uri = exchange.getIn().getHeader(Exchange.HTTP_URI,
String.class);
            String out = exchange.getIn().getHeader(Exchange.HTTP_METHOD,
String.class);
            if ("http://localhost:9080/users/homer".equals(uri)) {
                exchange.getOut().setBody(out + " " +
exchange.getIn().getHeader("username", String.class));
            } else if
("http://localhost:9080/atom/collection/foo/component/bar".equals(uri)) {
                exchange.getOut().setBody(out + " " +
exchange.getIn().getHeader("id", String.class)
                    + " " +
exchange.getIn().getHeader("cid", String.class));
            }
        }
    });
```

```
    }
  });
```

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>
```

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](#):

```
from("restlet:http://localhost:" + portNum + "/users/{id}/like/{beer}")
  .process(new Processor() {
    public void process(Exchange exchange) throws Exception {
      // the Restlet request should be available if needed
      Request request =
exchange.getIn().getHeader(RestletConstants.RESTLET_REQUEST,
Request.class);
      assertNotNull("Restlet Request", request);

      // use Restlet API to create the response
      Response response =
exchange.getIn().getHeader(RestletConstants.RESTLET_RESPONSE,
Response.class);
      assertNotNull("Restlet Response", response);
      response.setStatus(Status.SUCCESS_OK);
      response.setEntity("<response>Beer is Good</response>",
MediaType.TEXT_XML);
      exchange.getOut().setBody(response);
    }
  });
```

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.RestletComponent">
  <property name="maxThreads" value="100"/>
</bean>
```

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 98. RMI

RMI COMPONENT

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>
```

URI FORMAT

```
rmi://rmi-registry-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&...`

OPTIONS

| Name | Default Value | Description |
|--------|---------------|---|
| method | null | As of Apache Camel 1.3 , you can set the name of the method to invoke. |

| | | |
|-------------------------------|-------------------|---|
| <code>remoteInterfaces</code> | <code>null</code> | Its now possible to use this option from Camel 2.7 : in the XML DSL. It can be a list of interface names separated by comma. |
|-------------------------------|-------------------|---|

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>
```


CHAPTER 99. ROUTEBOX

ROUTEBOX COMPONENT

Available as of Camel 2.6

**ROUTEBOX SUBJECT TO CHANGE**

The Routebox component will be revisited in upcoming releases to see if it can be further simplified, be more intuitive and user friendly. The related [Context](#) component may be regarded as the simpler component. This component might be deprecated in favor of [Context](#).

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>
```

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 Example |
|--------------------------|---|
| 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.

URI FORMAT

```
routebox:routeboxname[?options]
```

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

OPTIONS

| Name | Default Value | Description |
|-------------------------------|-------------------|--|
| <code>dispatchStrategy</code> | <code>null</code> | A string representing a key in the Camel Registry matching an object value implementing the interface <code>org.apache.camel.component.routebox.strategy.RouteboxDispatchStrategy</code> |

| | | |
|----------------------|---------------------|--|
| dispatchMap | null | A string representing a key in the Camel Registry matching an object value of the type <code>HashMap<String, String></code> . The <code>HashMap</code> key should contain strings that can be matched against the value set for the exchange header ROUTE_DISPATCH_KEY . The <code>HashMap</code> value should contain inner route consumer URI's to which requests should be directed. |
| innerContext | auto created | A string representing a key in the Camel Registry matching an object value of the type <code>org.apache.camel.CamelContext</code> . If a <code>CamelContext</code> is not provided by the user a <code>CamelContext</code> is automatically created for deployment of inner routes. |
| innerRegistry | null | A string representing a key in the Camel Registry matching an object value that implements the interface <code>org.apache.camel.spi.Registry</code> . If <code>Registry</code> values are utilized by inner routes to create endpoints, an <code>innerRegistry</code> parameter must be provided |
| routeBuilders | empty List | A string representing a key in the Camel Registry matching an object value of the type <code>List<org.apache.camel.builder.RouteBuilder></code> . If the user does not supply an <code>innerContext</code> pre-primed with inner routes, the <code>routeBuilders</code> option must be provided as a non-empty list of <code>RouteBuilders</code> containing inner routes |
| innerProtocol | Direct | The Protocol used internally by the <code>Routebox</code> component. Can be <code>Direct</code> or <code>SEDA</code> . The <code>Routebox</code> component currently offers protocols that are JVM bound. |

| | | |
|-----------------------------|------------------------|---|
| <code>sendToConsumer</code> | <code>true</code> | 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. |
| <code>forkContext</code> | <code>true</code> | The Protocol used internally by the Routebox component. Can be Direct or SEDA. The Routebox component currently offers protocols that are JVM bound. |
| <code>threads</code> | <code>20</code> | Number of threads to be used by the routebox to receive requests. Setting applicable only for innerProtocol SEDA. |
| <code>queueSize</code> | <code>unlimited</code> | Create a fixed size queue to receive requests. Setting applicable only for innerProtocol SEDA. |

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.

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());

```

STEP 2: OPTIONALLY 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.util.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;
    }
}

```

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");
}

```

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 100. RSS

RSS COMPONENT

The `rss:` component is used for polling RSS feeds. Apache Camel will default poll the feed every 60th seconds.

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](#).

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&...`

OPTIONS

| Property | Default | Description |
|----------|---------|-------------|
|----------|---------|-------------|

| | | |
|------------------------|-------------|---|
| splitEntries | true | If true , Apache Camel splits a feed into its individual entries and returns each entry, poll by poll. For example, if a feed contains seven entries, Apache Camel returns the first entry on the first poll, the second entry on the second poll, and so on. When no more entries are left in the feed, Apache Camel contacts the remote RSS URI to obtain a new feed. If false , Apache Camel obtains a fresh feed on every poll and returns all of the feed's entries. |
| filter | true | Use in combination with the splitEntries option in order to filter returned entries. By default, Apache Camel applies the UpdateDateFilter filter, which returns only new entries from the feed, ensuring that the consumer endpoint never receives an entry more than once. The filter orders the entries chronologically, with the newest returned last. |
| throttleEntries | true | Camel 2.5: 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 is set to true. |
| lastUpdate | null | Use in combination with the filter option to block entries earlier than a specific date/time (uses the entry.updated timestamp). The format is: yyyy-MM-ddTHH:MM:ss . Example: 2007-12-24T17:45:59 . |
| feedHeader | true | Specifies whether to add the ROME SyndFeed object as a header. |

| | | |
|--------------------------------------|--------------------|---|
| <code>sortEntries</code> | <code>false</code> | If <code>splitEntries</code> is <code>true</code> , this specifies whether to sort the entries by updated date. |
| <code>consumer.delay</code> | <code>60000</code> | Delay in milliseconds between each poll. |
| <code>consumer.initialDelay</code> | <code>1000</code> | Milliseconds before polling starts. |
| <code>consumer.userFixedDelay</code> | <code>false</code> | Set to <code>true</code> to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |

EXCHANGE DATA TYPES

Apache Camel initializes the In body on the Exchange with a ROME SyndFeed. Depending on the value of the `splitEntries` flag, Apache Camel returns either a SyndFeed with one SyndEntry or a `java.util.List` of SyndEntries.

| Option | Value | Behavior |
|---------------------------|--------------------|--|
| <code>splitEntries</code> | <code>true</code> | A single entry from the current feed is set in the exchange. |
| <code>splitEntries</code> | <code>false</code> | The entire list of entries from the current feed is set in the exchange. |

MESSAGE HEADERS

| Header | Description |
|---------------------------|--|
| <code>CamelRssFeed</code> | Apache Camel 2.0: The entire <code>SyncFeed</code> object. |

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 this would look something 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 Apache Camel's lovely built-in expressions for manipulating RSS messages. As shown below, an XPath expression can be used to filter the RSS message:

```
// only entries with Apache 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("m
ock:result");
```

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");`

FILTERING ENTRIES

You can filter out entries quite easily using XPath, as shown in the data format section above. You can also exploit Apache Camel's [Bean Integration](#) to implement your own conditions. For instance, a filter equivalent to the XPath example above would be:

```
// only entries with Camel in the title will get through the filter
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");  
    }  
}
```

SEE ALSO

- [Atom](#)

CHAPTER 101. SALESFORCE

SALESFORCE COMPONENT

Available as of Camel 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>
```

URI FORMAT

The URI scheme for a salesforce component is as follows

```
salesforce:topic?options
```

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

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.

REST API

-

`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

•

`search` - Runs a Salesforce SOSL query

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")...
```

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.

•

`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 SQL query

•

`getQueryResultIds` - Gets a list of Result Ids for a Batch Query

•

`getQueryResult` - Gets results for a Result Id

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")..
```

REST STREAMING API

Consumer endpoints can use the following syntax 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&sObjectQuery=SELECT Id, Name FROM Merchandise__c")...
```

To subscribe to an existing topic

```
from("salesforce:CamelTestTopic&sObjectName=Merchandise__c")...
```

CAMEL SALESFORCE MAVEN PLUGIN

This Maven plugin generates DTOs for the Camel [Salesforce](#).

USAGE

The plugin configuration has the following properties.

| Option | Description |
|--------------|--|
| clientId | Salesforce client Id for Remote API access |
| clientSecret | Salesforce client secret for Remote API access |
| userName | Salesforce account user name |
| password | Salesforce account password (including secret token) |
| version | Salesforce Rest API version, defaults to 25.0 |

| | |
|-----------------|--|
| outputDirectory | Directory where to place generated DTOs, defaults to \${project.build.directory}/generated-sources/camel-salesforce |
| includes | List of SObject types to include |
| excludes | List of SObject types to exclude |
| includePattern | Java RegEx for SObject types to include |
| excludePattern | Java RegEx for SObject types to exclude |
| packageName | Java package name for generated DTOs, defaults to org.apache.camel.salesforce.dto. |

```
mvn camel-salesforce:generate -DclientId=<clientid> -DclientSecret=
<clientsecret> -Dusername=<username> -Dpassword=<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.

CHAPTER 102. SAP COMPONENT

Abstract

The SAP Component enables outbound and inbound communication to and from SAP systems using synchronous remote function calls, sRFC. The component uses the SAP Java Connector (SAP JCo) library to facilitate bidirectional communication with SAP. The component supports two types of endpoints: *destination endpoints* and *server endpoints*.

102.1. 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>
```

Deploying in a Fuse Fabric

A prerequisite for using the SAP component is that the SAP Java Connector (SAP JCo) libraries must be installed into the lib/ directory of the Java runtime (sapjco3.jar and libsapjco3.jnilib).

In the case of a Fuse Fabric deployment, this requires some special configuration. There is no point in simply installing the SAP JCo libraries in the Java lib directory on a single machine, because Fabric containers need to be deployable anywhere in the network. The correct approach is to define a special profile that is capable of downloading and installing the SAP JCo libraries on whichever host it is running on.

You can define a profile for the SAP JCo libraries as follows:

1. Deploy the JCo libraries (sapjco3.jar and libsapjco3.jnilib) to a network accessible location. For example, you could install the libraries in a Web server, so that the JCo libraries can be downloaded

through HTTP URLs, `http://mywebserver/sapjco3.jar` and `http://mywebserver/libsapjco3.jnilib`.

2. Create a new profile, `camel-sap-profile`, by entering the following console command:

```
JBossFuse:karaf@root> profile-create camel-sap-profile
```

3. Edit the agent properties of the `camel-sap-profile` profile, by entering the following console command:

```
JBossFuse:karaf@root> profile-edit camel-sap-profile
```

4. The built-in profile editor starts up. Use this built-in text editor to add the following contents to the agent properties:

```
# Profile:my-camel-sap-profile
attribute.parents = feature-camel

# Deploy JCo3 Libs to Container
lib.sapjco3.jar = http://mywebserver/sapjco3.jar
lib.sapjco3.jnilib = http://mywebserver/libsapjco3.jnilib

# Append JCo3 Packages to OSGi system property
# in order to expose JCo3 classes to OSGi environment
config.org.osgi.framework.system.packages.extra= \
  ... Packages from etc/config.properties file ... \
  com.sap.conn.jco, \
  com.sap.conn.jco.ext, \
  com.sap.conn.jco.monitor, \
  com.sap.conn.jco.rt, \
  com.sap.conn.jco.server
```

Customize the property settings as follows:

`lib.sapjco3.jar`

Customize the HTTP URL to the actual location of the `sapjco3.jar` file on your Web server.

`lib.sapjco3.jnilib`

Customize the HTTP URL to the actual location of the `libsapjco3.jnilib` file on your Web server.

`lib.sapidoc3.jar`

Customize the HTTP URL to the actual location of the `sapidoc3.jar` file on your Web server.

`config.org.osgi.framework.system.packages.extra`

Open the container configuration properties file, `etc/config.properties`, of your JBoss Fuse installation and look for the `org.osgi.framework.system.packages.extra` property setting. Copy the list of packages from that setting and paste them into the profile's agent properties, replacing the line:

```
... Packages from etc/config.properties file ...\
```



NOTE

The `config.*` prefix in `config.org.osgi.framework.system.packages.extra` indicates to Fabric that you are setting a container configuration property in the profile.



NOTE

The backslash, `\`, is the line continuation character (UNIX convention) and must be followed immediately by a newline character.

Type `Ctrl-S` to save the properties when you are finished.

5.

You can now deploy the `camel-sap-profile` profile to any Fabric container where you want to run the SAP component. For example, to deploy the `camel-sap-profile` to the `sap-instance` container:

```
JBossFuse:karaf@root> container-add-profile sap-instance came-sap-profile
```

URI format

The URI scheme for an SAP component is as follows:

```
sap:[destination:destinationName|server:serverName]rfcName?options
```

The `destination:` prefix designates a destination endpoint 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 102.4, “Destination Configuration”](#). The `rfcName` in a destination endpoint URI is the name of the RFC invoked by the endpoint in the connected SAP instance.

The `server:` prefix designates a server endpoint 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 102.5, “Server Configuration”](#). The `rfcName` in a server endpoint URI is the name of the RFC handled by the endpoint when invoked from the connected SAP instance.



NOTE

Server endpoints do not currently support any options.

URI options for destination endpoints

The following table summarizes the options supported by the URI for destination endpoints:

| Name | Default | Description |
|-------------------------|--------------------|---|
| <code>transacted</code> | <code>false</code> | When true, the endpoint will join or initiate a SAP transaction invoking an RFC call. |

SAP destination endpoint

A destination endpoint supports outbound communication to SAP, which enable these endpoints to make RFC calls out to ABAP function modules in SAP. A destination endpoint is configured to make an RFC call to a specific ABAP function over a specific connection to an SAP instance. A destination is a logical designation for an outbound connection and has a unique name. A

destination is specified by a set of connection parameters called *destination data*.

A 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 destination endpoints only support outbound communication, a destination endpoint will only support the creation of producers.

SAP server endpoint

A 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 a server endpoint as if it were a remote function module. A server endpoint is configured to receive an RFC call to a specific RFC function over a specific connection from an SAP instance. A server is a logical designation for an inbound connection and has a unique name. A server is specified by a set of connection parameters called server data.

A 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 server endpoints only support inbound communication, a server endpoint will only support the creation of consumers.

Meta-data repositories

A meta-data repository is used to store and provide 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.

SAP destination and server endpoints thus require access to a repository in order to send and receive RFC calls and the meta-data for all function modules invoked and handled by the endpoints must reside within a repository. The location of the repository used by a destination and server endpoint is specified in destination and 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.

102.2. MESSAGE BODY

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 Digits | |
|------------|-----------------------------------|-------------|---------------------|---------------|-------------------------|
| CHAR | <code>java.lang.String</code> | 1 to 65535 | 1 to 65535 | - | ABAP Type |
| DATE | <code>java.util.Date</code> | 8 | 16 | - | ABAP Type |
| BCD | <code>java.math.BigDecimal</code> | 1 to 16 | 1 to 16 | 0 to 14 | ABAP Type number co |
| TIME | <code>java.util.Date</code> | 6 | 12 | - | ABAP Type |
| BYTE | <code>byte[]</code> | 1 to 65535 | 1 to 65535 | - | ABAP Type |
| NUM | <code>java.lang.String</code> | 1 to 65535 | 1 to 65535 | - | ABAP Type string |
| FLOAT | <code>java.lang.Double</code> | 8 | 8 | 0 to 15 | ABAP Type |
| INT | <code>java.lang.Integer</code> | 4 | 4 | - | ABAP Type |
| INT2 | <code>java.lang.Integer</code> | 2 | 2 | - | ABAP Type |
| INT1 | <code>java.lang.Integer</code> | 1 | 1 | - | ABAP Type |
| DECF16 | <code>java.math.BigDecimal</code> | 8 | 8 | 16 | ABAP Type Floating P |
| DECF34 | <code>java.math.BigDecimal</code> | 16 | 16 | 34 | ABAP Type Floating P |
| STRING | <code>java.lang.String</code> | 8 | 8 | - | ABAP Type string |
| XSTRING | <code>byte[]</code> | 8 | 8 | - | ABAP Type |

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 | |
|------------------|---|--|--|------------------------|-----------|
| STRUCTURE | <code>org.fusesource.camel.component.sap.model.rfc.Structure</code> | Total of individual field byte lengths | Total of individual field Unicode byte lengths | - | ABAP Type |
| TABLE | <code>org.fusesource.camel.component.sap.model.rfc.Table</code> | Byte length of row structure | Unicode byte length of row structure | - | ABAP Type |

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.

102.3. 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

```

<bean id="sap" class="org.fusesource.camel.component.sap.SAPComponent">
  <property name="destinationDataStore">
    <map>
      <entry key="nplDest" value-ref="nplDestinationData" />
    </map>
  </property>
  <property name="serverDataStore">
    <map>
      <entry key="nplServer" value-ref="nplServerData" />
    </map>
  </property>
  <property name="repositoryDataStore">
    <map>
      <entry key="nplServer" value-ref="nplRepositoryData"/>
    </map>
  </property>
</bean>

```

102.4. 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. The following tables list the configuration properties that may be set on this object.

Logon and authentication options

| Name | Default Value | Des |
|---------------------|---------------|------------------------------------|
| <code>client</code> | | SAP client, mandatory logon parame |

| | | |
|----------------------------|--|--|
| user | | Logon user, logon parameter for pas |
| aliasUser | | Logon user alias, can be used instea |
| userId | | User identity which is used for logon runtime, if the destination configura certificate, current user ,or SNC envi is mandatory, if neither user nor use to the SAP backend, it will be used b |
| passwd | | Logon password, logon parameter fo |
| lang | | Logon language, if not defined, the c |
| mysapso2 | | Use the specified SAP Cookie Version authentication |
| x509cert | | Use the specified X509 certificate fo |
| lcheck | | Postpone the authentication until the cases only . |
| useSapGui | | Use a visible, hidden, or do not use ? |
| codePage | | Additional logon parameter to define the logon parameters. Used in speci |
| getssso2 | | Order a SSO ticket after logon, the o destination attributes |
| denyInitialPassword | | If set to 1 , using initial passwords wi |

Connection options

| Name | Default Value | Des |
|------------------|---------------|---|
| saprouter | | SAP Router string for connection to s string contains the chain of SAP Rou form: (/H/<host>[/S/<port>])+ |
| sysnr | | System number of the SAP ABAP app connection |
| ashost | | SAP ABAP application server, manda |

| | | |
|---------------|--|--|
| mshost | | SAP message server, mandatory pro |
| msserv | | SAP message server port, optional p In order to resolve the service name is performed by the network layer of numbers instead of symbolic service no additional entries are needed. |
| gwhost | | Allows specifying a concrete gatewa the connection to an application ser application server is used |
| gwserv | | Should be set, when using gwhost. A gateway. If not specified the port of used. In order to resolve the service etc/services is performed by the net using port numbers instead of symb performed and no additional entries |
| r3name | | System ID of the SAP system, mandi connection. |
| group | | Group of SAP application servers, m connection |

Connection pool options

| Name | Default Value | Des |
|-------------------------|----------------------|---|
| peakLimit | 0 | Maximum number of active outbound destination simultaneously. A value active connections, otherwise if the jpoolCapacity , it will be automati setting is the value of poolCapacit being specified as well, the default i: |
| poolCapacity | 1 | Maximum number of idle outbound (o destination. A value of 0 has the eff (default is 1). |
| expirationTime | | Time in milliseconds after which a fr destination can be closed |
| expirationPeriod | | Period in milliseconds after which th connections for expiration. |

| | | |
|-------------------|--|--|
| maxGetTime | | Maximum time in milliseconds to wait for the allowed number of connections has been established. |
|-------------------|--|--|

Secure network connection options

| Name | Default Value | Description |
|-----------------------|---------------|---|
| sncMode | | Secure network connection (SNC) mode. |
| sncPartnername | | SNC partner, for example: p:CN=R3 |
| sncQop | | SNC level of security: 1 to 9 |
| sncMyname | | Own SNC name. Overrides environment variable. |
| sncLibrary | | Path to library that provides SNC services. |

Repository options

| Name | Default Value | Description |
|-------------------------|---------------|---|
| repositoryDest | | Specifies which destination should be used. |
| repositoryUser | | If a repository destination is not set, this is used as user for repository calls. This enables repository look-ups. |
| repositoryPasswd | | The password for a repository user. This is used. |
| repositorySnc | | <i>(Optional)</i> If SNC is used for this destination, if this property is set, it overrides the value of jco.client.snc_mode . For specification, see the snc_mode property. |

| | | |
|--|--|--|
| repositoryRoundtripOptimization | | <p>Enable the RFC_METADATA_GET API single round trip.</p> <p>1 Activates use of RFC_METADATA_</p> <p>0 Deactivates RFC_METADATA_GET</p> <p>If the property is not set, the destination whether RFC_METADATA_GET is available will use it.</p> <p>Note: If the repository is already initialized by some other destination) this property is related to the ABAP ! on all destinations pointing to the same backend prerequisites.</p> |
|--|--|--|

Trace configuration options

| Name | Default Value | Description |
|-----------|---------------|---|
| trace | | Enable/disable RFC trace (0 or 1) |
| cpicTrace | | Enable/disable CPIC trace [0..3] |

Destination data example

```

<bean id="nplDestinationData"
class="org.fusesource.camel.component.sap.model.rfc.impl.DestinationDataImpl">
  <property name="ashost" value="nplhost" />
  <property name="sysnr" value="42" />
  <property name="client" value="001" />
  <property name="user" value="developer" />
  <property name="passwd" value="ch4ngeme" />
  <property name="lang" value="en" />
</bean>

```

102.5. 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. The following tables lists the configuration properties that may be set on this object.

Required options

The required options for the server data configuration object are, as follows:

| Name | Default Value | Description |
|------------------------------|---------------|---|
| <code>gwhost</code> | | Gateway host on which the server connects. |
| <code>gwserv</code> | | Gateway service, which is the port on which the server connects in order to resolve the service names. This is performed by the network layer of the gateway. If the service names are numbers instead of symbolic service names, no additional entries are needed. |
| <code>progid</code> | | The program ID with which the gateway registers the destination in the SAP system. |
| <code>connectionCount</code> | | The number of connections that should be established. |

Secure network connection options

The secure network connection options for the server data configuration object are, as follows:

| Name | Default Value | Description |
|----------------------|---------------|---|
| <code>sncMode</code> | | Secure network connection (SNC) mode. |
| <code>sncQop</code> | | SNC level of security, 1 to 9 . |

| | | |
|------------------|--|---|
| sncMyname | | SNC name of your server. Overrides something like p:CN=JCoServer , |
| sncLib | | Path to library which provides SNC s the value of the jco.middleware . |

Other options

The other options for the server data configuration object are, as follows:

| Name | Default Value | Des |
|-----------------------------|---------------|---|
| saprouter | | SAP router string to use for a system therefore only be reached through a at the gateway of that ABAP System /H/firewall.hostname/H/ |
| maxStartupDelay | | The maximum time (in seconds) bet failures. The waiting time is doubled up failure until either the maximum started successfully. |
| trace | | Enable/disable RFC trace (0 or 1) |
| workerThreadCount | | The maximum number of threads us the value for the connectionCount The maximum number of threads ca |
| workerThreadMinCount | | The minimum number of threads us value for connectionCount is usec |

Server data example

```
<bean id="nplServerData"
class="org.fusesource.camel.component.sap.model.rfc.impl.ServerDataImpl">
  <property name="gwHost" value="nplHost" />
  <property name="gwServ" value="3342" />
  <property name="progId" value="JCO_SERVER" />
  <property name="repositoryDestination" value="nplDest" />
  <property name="connectionCount" value="2" />
</bean>
```

102.6. REPOSITORY CONFIGURATION

Overview

The configuration for repositories are maintain 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

```
<bean id="nplRepositoryData"
class="org.fusesource.camel.component.sap.model.rfc.impl.RepositoryDataImp
1">
  <property name="functionTemplates">
    <map>
      <entry key="BOOK_FLIGHT" value-
ref="bookFlightFunctionTemplate" />
    </map>
  </property>
</bean>
```

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 |
|------------------------------------|---|
| <code>importParameterList</code> | A list of list field meta-data objects, org.fusesource.camel.component.sap.taImpl . Specifies the parameters that are sent to a function module. |
| <code>changingParameterList</code> | A list of list field meta-data objects, org.fusesource.camel.component.sap.taImpl . Specifies the parameters that are sent to and returned by a function module. |
| <code>exportParameterList</code> | A list of list field meta-data objects, org.fusesource.camel.component.sap.taImpl . Specifies the parameters that are returned by a function module. |
| <code>tableParameterList</code> | A list of list field meta-data objects, org.fusesource.camel.component.sap.taImpl . Specifies the table parameters that are sent to and returned by a function module. |
| <code>exceptionList</code> | A list of ABAP exception meta-data objects, org.fusesource.camel.component.sap.taImpl . Specifies the ABAP exceptions potentially raised by a 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.FunctionTemplateI
mpl">
  <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 | Des |
|------|---------------|----------------------------------|
| name | - | The name of the parameter field. |

| | | |
|--------------------------|-------|--|
| type | - | The parameter type of the field. |
| byteLength | - | The field length in bytes for a non-UI parameter type. See Section 102.2 , |
| unicodeByteLength | - | The field length in bytes for a Unicode parameter type. See Section 102.2 , |
| decimals | 0 | The number of decimals in field value for BCD and FLOAT. See Section 102.2 , |
| optional | false | If true, the field is optional and needs |

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. The <code>org.fusesource.camel.component.structure.metaDataImpl</code> is passed to specify the structure. |
| optional | false | If true, the field is optional and needs |

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.structure.metaDataImpl.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.ListFieldMetaData
Impl">
  <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.ListFieldMetaData
Impl">
  <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 | Des |
|-------------------|---------------|-------------------------|
| <code>name</code> | - | The name of the record. |

| | | |
|----------------------------------|---|---|
| <code>recordFieldMetaData</code> | - | The list of record field meta-data objects of <code>org.fusesource.camel.component.sap.model.rfc.impl.RecordMetaDataImpl</code> . Specifies the fields contained in the record. |
|----------------------------------|---|---|

**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 within 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 |
|--------------------------------|---------------|---|
| <code>name</code> | - | The name of the parameter field |
| <code>type</code> | - | The parameter type of the field |
| <code>byteLength</code> | - | The field length in bytes for a non-Unicode parameter type. See Section 102.2 , |
| <code>unicodeByteLength</code> | - | The field length in bytes for a Unicode parameter type. See Section 102.2 , |
| <code>byteOffset</code> | - | The field offset in bytes for non-Unicode location of the field within the enclosing structure |
| <code>unicodeByteOffset</code> | - | The field offset in bytes for Unicode the field within the enclosing structure |
| <code>decimals</code> | 0 | The number of decimals in field value for BCD and FLOAT . See Section 102.2 , |

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 |
|--------------------------------|---------------|--|
| <code>name</code> | - | The name of the parameter field |
| <code>type</code> | - | The parameter type of the field |
| <code>byteOffset</code> | - | The field offset in bytes for non-Unicode location of the field within the enclosing structure |
| <code>unicodeByteOffset</code> | - | The field offset in bytes for Unicode the field within the enclosing structure |
| <code>recordMetaData</code> | - | The meta-data for the structure or table. The <code>org.fusesource.camel.component.table.RecordMetaDataImpl</code> is passed to specify the meta-data. |

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>
```

102.7. 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.

102.8. XML SERIALIZATION

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 102.11](#), “[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:complexType name="CONNECTION_INFO_STRUCTURE_TABLE">
    <xs:sequence>
      <xs:element
        name="row"
        minOccurs="0"
        maxOccurs="unbounded"
        type="CONNECTION_INFO_STRUCTURE"/>
      ...
    </xs:sequence>
  </xs:complexType>

  <xs:complexType 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 102.11, "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"
```

102.9. 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 a 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: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:destination:np1Dest: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.
    SAPEndpoint endpoint =
        exchange.getContext().getEndpoint("bean:returnFlightCustomerInfo",
            SAPEndpoint.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);
}
```

102.10. 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:destination:nplDest:BAPI_FLTRIP_GETLIST?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:destination:nplDest:BAPI_FLTRIP_GETLIST?
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 102.7, "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.

102.11. EXAMPLE 3: HANDLING REQUESTS FROM SAP

Overview

This example demonstrates a route which handles a request from SAP to the BOOK_FLIGHT RCF, 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: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 Spring DSL for the same route is as follows:

```
<route>
```

```

<from uri="sap: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 103. SAP NETWEAVER

SAP NETWEAVER GATEWAY COMPONENT

Available as of Camel 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>
```

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&...`

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.

COMPONENT AND ENDPOINT OPTIONS

| Name | Default Value | Description |
|-------------------------|-------------------|--|
| <code>username</code> | | Username for account. This is mandatory. |
| <code>password</code> | | Password for account. This is mandatory. |
| <code>json</code> | <code>true</code> | Whether to return data in JSON format. If this option is false, then XML is returned in Atom format. |
| <code>jsonAsMap</code> | <code>true</code> | To transform the JSON from a String to a Map in the message body. |
| <code>flattenMap</code> | <code>true</code> | If the JSON Map contains only a single entry, then flatten by storing that single entry value as the message body. |

MESSAGE HEADERS

The following headers can be used by the producer.

| Name | Type | Description |
|-----------------------|--------|---|
| CamelNetWeaverCommand | String | Mandatory: The command to execute in MS ADO.Net Data Service format. |

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")
  .toF("sap-netweaver:%s?username=%s&password=%s", url, username,
password)
  .to("log:response")
  .to("velocity:flight-info.vm")
```


Where url, username, and password 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=date
time'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
```

-

HTTP

CHAPTER 104. SEDA

SEDA COMPONENT

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](#).

SYNCHRONOUS

The [Direct](#) component provides synchronous invocation of any consumers when a producer sends a message exchange.

URI FORMAT

```
seda:queueName[?options]
```

Where `queueName` 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&...`

**NOTE**

When matching consumer endpoints to producer endpoints, only the `queueName` is considered and any option settings are ignored. That is, the identity of a consumer endpoint depends only on the `queueName`. If you want to attach multiple consumers to the same queue, use the approach described in [the section called “Using `multipleConsumers`”](#).

OPTIONS

| Name | Default | Description |
|----------------------------|------------------|---|
| size | <i>Unbounded</i> | The maximum capacity of the SEDA queue (i.e., the number of messages it can hold). 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. From Camel 2.11 onwards, a validation is taken place to ensure if using mixed queue sizes for the same queue name, Camel would detect this and fail creating the endpoint. |
| concurrentConsumers | 1 | Apache Camel 1.6.1/2.0: Number of concurrent threads processing exchanges. |

| | | |
|------------------------------------|------------------------------|--|
| <code>waitForTaskToComplete</code> | <code>IfReplyExpected</code> | 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 . See more information about Async messaging. |
| <code>timeout</code> | <code>30000</code> | Apache Camel 2.0: Timeout in millis a seda producer will at most waiting for an async task to complete. See <code>waitForTaskToComplete</code> and Async for more details. In Camel 2.2 you can now disable timeout by using 0 or a negative value. |
| <code>multipleConsumers</code> | <code>false</code> | Camel 2.2: Specifies whether multiple consumers are allowed or not. If enabled, you can use SEDA for a publish/subscribe style of messaging. Send a message to a SEDA queue and have multiple consumers receive a copy of the message. |

| | | | |
|---------------------------------|--------------|---|--|
| limitConcurrentConsumers | true | Camel 2.3: Whether to limit the concurrentConsumers to maximum 500. If its configured with a higher number an exception will be thrown. You can disable this check by turning this option off. | |
| blockWhenFull | 2.9 | false | 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. |
| queueSize | 2.9 | | Component only: The maximum default size (capacity of the number of messages it can hold) of the SEDA queue. This option is used if size is not in use. |
| pollTimeout | 2.9.3 | 1000 | <i>Consumer only</i> -- 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. |

| | | | |
|--------------------------|---------------|--------------|--|
| purgeWhenStopping | 2.11.1 | false | 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. |
| queue | 2.12.0 | null | Define the queue instance which will be used by seda endpoint |
| queueFactory | 2.12.0 | null | Define the QueueFactory which could create the queue for the seda endpoint |
| failIfNoConsumers | 2.12.0 | false | Whether the producer should fail by throwing an exception, when sending to a SEDA queue with no active consumers. |

CHOOSING BLOCKINGQUEUE IMPLEMENTATION

Available as of Camel 2.12

By default, the SEDA component always instantiates `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>
```

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.

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.

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(...)
```

DIFFERENCE BETWEEN THREAD POOLS AND CONCURRENT CONSUMERS

The *thread pool* is a pool that can increase/shrink dynamically at runtime depending on load, whereas the concurrent consumers are always fixed.

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 want to consider configuring 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.

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.

```
public void configure() throws Exception {
    from("direct:start")
        // send it to the seda queue that is async
        .to("seda:next")
}
```

```

        // return a constant response
        .transform(constant("OK"));

    from("seda:next").to("mock:result");
}

```

Here we send a Hello World message and expect 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 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.

USING MULTIPLECONSUMERS

Available as of Camel 2.2

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="seda:foo?multipleConsumers=true"/>
</camelContext>

```

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.

■

```
public class FooEventConsumer {  
  
    @EndpointInject(uri = "mock:result")  
    private ProducerTemplate destination;  
  
    @Consume(ref = "foo")  
    public void doSomething(String body) {  
        destination.sendBody("foo" + body);  
    }  
  
}
```

EXTRACTING QUEUE INFORMATION.

If you need it, you can also get information like queue size etc without using JMX like this:

```
SedaEndpoint seda = context.getEndpoint("seda:xxxx");  
int size = seda.getExchanges().size()
```

- [Disruptor](#)
- [VM](#)
- [Direct](#)

CHAPTER 105. SERVLET

SERVLET COMPONENT

The `servlet:` component provides HTTP based [endpoints](#) for consuming HTTP requests that arrive at a HTTP endpoint and this endpoint 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>
```

URI FORMAT

```
servlet://relative_path[?options]
```

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

OPTIONS

| Name | Default Value | Description |
|-------------------------------|--------------------|--|
| <code>httpBindingRef</code> | <code>null</code> | Reference to an <code>org.apache.camel.component.http.HttpBinding</code> in the Registry . A <code>HttpBinding</code> implementation can be used to customize how to write a response. |
| <code>matchOnUriPrefix</code> | <code>false</code> | Whether or not the <code>CamelServlet</code> should try to find a target consumer by matching the URI prefix, if no exact match is found. |

| | | |
|--------------------------|-------------------|--|
| <code>servletName</code> | <code>null</code> | Specifies the servlet name that the servlet endpoint will bind to. If there is no servlet name specified, the servlet endpoint will be bind to first published Servlet |
|--------------------------|-------------------|--|

MESSAGE HEADERS

Apache Camel will apply the same Message Headers as the [HTTP](#) component.

Apache Camel will also populate `allrequest.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`.

USAGE

You can only consume from endpoints generated by the Servlet component. Therefore, it should only be used as input into your Apache Camel routes. To issue HTTP requests against other HTTP endpoints, use the [HTTP Component](#)

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</servle
t-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</servle
t-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.

SAMPLE



IMPORTANT

From Camel 2.7 onwards its 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:

```
<web-app>

  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <display-name>Camel Http Transport Servlet</display-name>
    <servlet-class>
      org.apache.camel.component.servlet.CamelHttpTransportServlet
    </servlet-class>

  </servlet>

  <servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Then you can define your route as follows:

```
from("servlet:///hello?matchOnUriPrefix=true").process(new Processor() {
  public void process(Exchange exchange) throws Exception {
    String contentType =
exchange.getIn().getHeader(Exchange.CONTENT_TYPE, String.class);
    String path = exchange.getIn().getHeader(Exchange.HTTP_URI,
String.class);
    path = path.substring(path.lastIndexOf("/"));

    assertEquals("Get a wrong content type", CONTENT_TYPE,
contentType);
    // assert camel http header
    String charsetEncoding =
exchange.getIn().getHeader(Exchange.HTTP_CHARACTER_ENCODING,
String.class);
    assertEquals("Get a wrong charset name from the message heaer",
"UTF-8", charsetEncoding);
    // assert exchange charset
    assertEquals("Get a wrong charset naem from the exchange
property", "UTF-8", exchange.getProperty(Exchange.CHARSET_NAME));
    exchange.getOut().setHeader(Exchange.CONTENT_TYPE, contentType +
"; charset=UTF-8");
    exchange.getOut().setHeader("PATH", path);
    exchange.getOut().setBody("<b>Hello World</b>");
  }
});
```



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").
```

SAMPLE WHEN USING SPRING 3.X

The standalone Apache Camel package contains a demonstration of how to deploy the Servlet component in the Tomcat Web container. The demonstration is located in the `examples/camel-example-servlet-tomcat` directory. When deploying a Servlet component in the Web container, it is necessary to create a Spring application context explicitly by creating a Spring `ContextLoaderListener` instance in the `WEB-INF/web.xml` file.

For example, to create a Spring application context that loads Spring definitions (including the `camelContext` and route definitions) from the `camel-config.xml` file, define a `web.xml` file as follows:

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>My Web Application</display-name>

  <!-- location of spring xml files -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:camel-config.xml</param-value>
  </context-param>

  <!-- the listener that kick-starts Spring -->
  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>

  <!-- Camel servlet -->
  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-
class>org.apache.camel.component.servlet.CamelHttpTransportServlet</servle
t-class>
```



```

        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Camel servlet mapping -->
    <servlet-mapping>
        <servlet-name>CamelServlet</servlet-name>
        <url-pattern>/camel/*</url-pattern>
    </servlet-mapping>

</web-app>

```

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>

```

SAMPLE WHEN USING OSGI

From Camel 2.6.0, you can publish the [CamelHttpTransportServlet](#) as an OSGi service with help of SpringDM like this.

```

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

```

```

    <bean id="camelServlet"
class="org.apache.camel.component.servlet.CamelHttpTransportServlet">
    </bean>

    <!--
    Enlist it in OSGi service registry
    This will cause two things:
    1) As the pax web whiteboard extender is running the CamelServlet
will
    be registered with the OSGi HTTP Service
    2) It will trigger the HttpRegistry in other bundles so the
servlet is
    made known there too
    -->
    <osgi:service ref="camelServlet">
    <osgi:interfaces>
    <value>javax.servlet.Servlet</value>
    <value>org.apache.camel.component.http.CamelServlet</value>
    </osgi:interfaces>
    <osgi:service-properties>
    <entry key="alias" value="/camel/services" />
    <entry key="matchOnUriPrefix" value="true" />
    <entry key="servlet-name" value="CamelServlet"/>
    </osgi:service-properties>
    </osgi:service>
</beans>

```

Then use this service in your camel route like this:

```

<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"
xmlns:osgi="http://www.springframework.org/schema/osgi"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

    <osgi:reference id="servletref"
interface="org.apache.camel.component.http.CamelServlet">
    <osgi:listener bind-method="register" unbind-
method="unregister">
    <ref bean="httpRegistry"/>
    </osgi:listener>
    </osgi:reference>

    <bean id="httpRegistry"
class="org.apache.camel.component.servlet.DefaultHttpRegistry"/>

    <bean id="servlet"

```

```

class="org.apache.camel.component.servlet.ServletComponent">
    <property name="httpRegistry" ref="httpRegistry" />
</bean>

<bean id="servletProcessor"
class="org.apache.camel.itest.osgi.servlet.ServletProcessor" />

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <!-- notice how we can use the servlet scheme which is that
osgi:reference above -->
        <from uri="servlet:///hello"/>
        <process ref="servletProcessor"/>
    </route>
</camelContext>

</beans>

```

Alternatively - pre Camel 2.6 - you can use an Activator to publish the [CamelHttpTransportServlet](#) on the OSGi platform

```

import java.util.Dictionary;
import java.util.Hashtable;

import org.apache.camel.component.servlet.CamelHttpTransportServlet;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.service.http.HttpContext;
import org.osgi.service.http.HttpService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.osgi.context.BundleContextAware;

public final class ServletActivator implements BundleActivator,
BundleContextAware {
    private static final transient Logger LOG =
LoggerFactory.getLogger(ServletActivator.class);
    private static boolean registerService;

    /**
     * HttpService reference.
     */
    private ServiceReference httpServiceRef;

    /**
     * Called when the OSGi framework starts our bundle
     */
    public void start(BundleContext bc) throws Exception {
        registerServlet(bc);
    }

    /**
     * Called when the OSGi framework stops our bundle

```

```

    */
    public void stop(BundleContext bc) throws Exception {
        if (httpServiceRef != null) {
            bc.ungetService(httpServiceRef);
            httpServiceRef = null;
        }
    }

    protected void registerServlet(BundleContext bundleContext) throws
Exception {
        httpServiceRef =
bundleContext.getServiceReference(HttpService.class.getName());

        if (httpServiceRef != null && !registerService) {
            LOG.info("Register the servlet service");
            final HttpService httpService =
(HttpService)bundleContext.getService(httpServiceRef);
            if (httpService != null) {
                // create a default context to share between
registrations
                final HttpContext httpContext =
httpService.createDefaultHttpContext();
                // register the hello world servlet
                final Dictionary<String, String> initParams = new
Hashtable<String, String>();
                initParams.put("matchOnUriPrefix", "false");
                initParams.put("servlet-name", "CamelServlet");
                httpService.registerServlet("/camel/services", // alias
                    new CamelHttpTransportServlet(), // register servlet
                    initParams, // init params
                    httpContext // http context
                );
                registerService = true;
            }
        }
    }

    public void setBundleContext(BundleContext bc) {
        try {
            registerServlet(bc);
        } catch (Exception e) {
            LOG.error("Cannot register the servlet, the reason is " + e);
        }
    }
}

```

CHAPTER 106. SERVLETLISTENER COMPONENT

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.



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 bootstrap Camel using the `@WebListener`, and implement your own class, where you bootstrap 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>
```

USING

You would need to choose 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:

```
<web-app>

  <!-- the test parameter is only to be used for unit testing -->
  <!-- you should not use this option for production usage -->
  <context-param>
    <param-name>test</param-name>
    <param-value>true</param-value>
  </context-param>

  <!-- you can configure any of the properties on CamelContext, eg setName
  will be configured as below -->
  <context-param>
    <param-name>name</param-name>
    <param-value>MyCamel</param-value>
  </context-param>

  <!-- configure a route builder to use -->
  <!-- Camel will pickup any parameter names that start with routeBuilder
  (case ignored) -->
  <context-param>
    <param-name>routeBuilder-MyRoute</param-name>
    <param-
value>org.apache.camel.component.servletlistener.MyRoute</param-value>
  </context-param>

  <!-- register Camel as a listener so we can bootstrap Camel when the web
```

```

application starts -->
<listener>
  <listener-
class>org.apache.camel.component.servletlistener.SimpleCamelServletContext
Listener</listener-class>
  </listener>

</web-app>

```

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 <code>propertyPlaceholder.location</code> 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 <code>jmx.disabled</code> as name. You can configure all the options from <code>org.apache.camel.spi.ManagementAgent</code> . 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. |
| 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 . |
|-----|--|---|

EXAMPLES

See [Servlet Tomcat No Spring Example](#).

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".

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".

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>

```

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:

```

<?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.

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>

```

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`.

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.

```
/**
 * Our custom {@link CamelContextLifecycle} which allows us to enlist
 beans in the {@link JndiContext}
 * so the Camel application can lookup the beans in the {@link
 org.apache.camel.spi.Registry}.
 * <p/>
 * We can of course also do other kind of custom logic as well.
 */
public class MyLifecycle implements CamelContextLifecycle<SimpleRegistry>
{

    @Override
    public void beforeStart(ServletCamelContext camelContext,
 SimpleRegistry registry) throws Exception {
```

```

        // enlist our bean(s) in the registry
        registry.put("myBean", new HelloBean());
    }

    @Override
    public void afterStart(ServletCamelContext camelContext,
SimpleRegistry registry) throws Exception {
        // noop
    }

    @Override
    public void beforeStop(ServletCamelContext camelContext,
SimpleRegistry registry) throws Exception {
        // noop
    }

    @Override
    public void afterStop(ServletCamelContext camelContext, SimpleRegistry
registry) throws Exception {
        // unbind our bean when Camel has been stopped
        registry.remove("myBean");
    }
}

```

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`

SEE ALSO

-

[Servlet](#)

-

[Servlet Tomcat Example](#)

-

[Servlet Tomcat No Spring Example](#)

CHAPTER 107. SHIRO SECURITY

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>
```

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:*
```

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);
```

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 that only requests from the same user going forward are authenticated. |
| 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 is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required |

| | | | |
|----------------------------|--------------------|--|--|
| <code>cipherService</code> | <code>AES</code> | <code>org.apache.shiro.crypto.CipherService</code> | Shiro ships with AES & Blowfish based CipherServices. You may use one these or pass in your own Cipher implementation |
| <code>base64</code> | <code>false</code> | <code>boolean</code> | 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. |

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");
    }
};
}

```

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");
        }
    };
}

```

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");

```

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();
}

```

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.

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, of a base64 representation as a `String` (the latter is when you have set `base64=true`).

CHAPTER 108. SIP

SIP COMPONENT

Available as of Camel 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>
```

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&...

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.

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|--------------------------|---------------------------|--|
| <code>stackName</code> | <code>NAME_NOT_SET</code> | Name of the SIP Stack instance associated with an SIP Endpoint. |
| <code>transport</code> | <code>tcp</code> | Setting for choice of transport potocol. Valid choices are "tcp" or "udp". |
| <code>fromUser</code> | | Username of the message originator. Mandatory setting unless a registry based custom FromHeader is specified. |
| <code>fromHost</code> | | Hostname of the message originator. Mandatory setting unless a registry based FromHeader is specified |
| <code>fromPort</code> | | Port of the message originator. Mandatory setting unless a registry based FromHeader is specified |
| <code>toUser</code> | | Username of the message receiver. Mandatory setting unless a registry based custom ToHeader is specified. |
| <code>toHost</code> | | Hostname of the message receiver. Mandatory setting unless a registry based ToHeader is specified |
| <code>toPort</code> | | Portname of the message receiver. Mandatory setting unless a registry based ToHeader is specified |
| <code>maxforwards</code> | <code>0</code> | the number of intermediaries that may forward the message to the message receiver. Optional setting. May alternatively be set using as registry based MaxForwardsHeader |

| | | |
|-------------------------------|----------------|--|
| eventId | | Setting for a String based event Id. Mandatory setting unless a registry based FromHeader is specified |
| eventHeaderName | | Setting for a String based event Id. Mandatory setting unless a registry based FromHeader is specified |
| maxMessageSize | 1048576 | Setting for maximum allowed Message size in bytes. |
| cacheConnections | false | 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. |
| consumer | false | This setting is used to determine whether the kind of header (FromHeader,ToHeader etc) that needs to be created for this endpoint |
| automaticDialogSupport | off | Setting to specify whether every communication should be associated with a dialog. |
| contentType | text | Setting for contentType can be set to any valid MimeType. |
| contentSubType | xml | Setting for contentSubType can be set to any valid MimeSubType. |
| receiveTimeoutMillis | 10000 | Setting for specifying amount of time to wait for a Response and/or Acknowledgement can be received from another SIP stack |
| useRouterForAllUris | false | This setting is used when requests are sent to the Presence Agent via a proxy. |
| msgExpiration | 3600 | The amount of time a message received at an endpoint is considered valid |

| | | |
|----------------------|--------------|---|
| presenceAgent | false | 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. |
|----------------------|--------------|---|

REGISTRY BASED OPTIONS

SIP requires a number of headers to be sent/received as part of a request. These SIP header 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 |
|-------------------|---|
| fromHeader | a custom Header object containing message originator settings. Must implement the type <code>javax.sip.header.FromHeader</code> |
| toHeader | a custom Header object containing message receiver settings. Must implement the type <code>javax.sip.header.ToHeader</code> |
| viaHeaders | List of custom Header objects of the type <code>javax.sip.header.ViaHeader</code> . Each <code>ViaHeader</code> containing a proxy address for request forwarding. (Note this header is automatically updated by each proxy when the request arrives at its listener) |

| | |
|--------------------------|--|
| contentTypeHeader | a custom Header object containing message content details. Must implement the type <code>javax.sip.header.ContentTypeHeader</code> |
| callIdHeader | a custom Header object containing call details. Must implement the type <code>javax.sip.header.CallIdHeader</code> |
| maxForwardsHeader | a custom Header object containing details on maximum proxy forwards. This header places a limit on the <code>viaHeaders</code> possible. Must implement the type <code>javax.sip.header.MaxForwardsHeader</code> |
| eventHeader | a custom Header object containing event details. Must implement the type <code>javax.sip.header.EventHeader</code> |
| contactHeader | an optional custom Header object containing verbose contact details (email, phone number etc). Must implement the type <code>javax.sip.header.ContactHeader</code> |
| expiresHeader | a custom Header object containing message expiration details. Must implement the type <code>javax.sip.header.ExpiresHeader</code> |
| extensionHeader | a custom Header object containing user/application specific details. Must implement the type <code>javax.sip.header.ExtensionHeader</code> |

SENDING MESSAGES TO/FROM A SIP ENDPOINT

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&f  
romHost=localhost&fromPort=3534",  
    "EVENT_A",  
    "REQUEST_METHOD",  
    Request.PUBLISH);
```

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 set 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&even
tId=evtId")
                .to("mock:neverland");

            // Create Sip Consumer(Event Subscriber)
            from("sip://johndoe@localhost:5154?
stackName=Subscriber&toUser=agent&toHost=localhost&toPort=5152&eventHeader
Name=evtHdrName&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 109. SJMS

SJMS COMPONENT

Available as of Camel 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

-

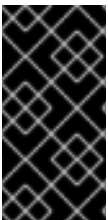
Batch Consumers and Producers

-

Transacted Batch Consumers & Producers

-

Support for Customizable Transaction Commit Strategies (Local JMS Transactions only)



WHY THE S IN SJMS

S stands for Simple and Standard and Springless. Also camel-jms was already taken. :-)

**WARNING**

This is a rather new component in a complex world of JMS messaging. So this component is ongoing development and hardening. The classic [JMS](#) component based on Spring JMS has been hardened and battle tested extensively.

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>
```

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&...`

COMPONENT OPTIONS AND CONFIGURATIONS

The SJMS Component supports the following configuration options:

| Option | Required | Default Value | Description |
|-----------------------------------|-----------------|--|---|
| <code>connectionCount</code> | | <code>1</code> | The maximum number of connections available to endpoints started under this component |
| <code>connectionFactory</code> | <code>()</code> | <code>null</code> | A ConnectionFactory is required to enable the <code>SjmsComponent</code> . It can be set directly or set as part of a <code>ConnectionResource</code> . |
| <code>connectionResource</code> | | <code>null</code> | A <code>ConnectionResource</code> is an interface that allows for customization and container control of the <code>ConnectionFactory</code> . See Pluggable Connection Resource Management for further details. |
| <code>headerFilterStrategy</code> | | <code>DefaultJmsKeyFormatStrategy</code> | |

| | | | |
|---------------------------|--|-----------------------------|--|
| keyFormatStrategy | | DefaultJmsKeyFormatStrategy | |
| transactionCommitStrategy | | null | |

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 `SjmsComponent` 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.setConnectionFactoryResource(connectionResource);
component.setMaxConnections(1);
```

PRODUCER CONFIGURATION OPTIONS

The `SjmsProducer` Endpoint supports the following properties:

| Option | Default Value | Description |
|--------|---------------|-------------|
|--------|---------------|-------------|

| | | |
|----------------------------|-------------------------|---|
| acknowledgementMode | AUTO_ACKNOWLEDGE | The JMS acknowledgement name, which is one of: SESSION_TRANSACTED , AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE . CLIENT_ACKNOWLEDGE is not supported at this time. |
| consumerCount | 1 | InOut only. Defines the number of MessageListener instances that for response consumers. |
| exchangePattern | InOnly | Sets the Producers message exchange pattern. |
| namedReplyTo | null | InOut only. Specifies a named reply to destination for responses. |
| persistent | true | Whether a message should be delivered with persistence enabled. |
| producerCount | 1 | Defines the number of MessageProducer instances. |
| responseTimeout | 5000 | InOut only. Specifies the amount of time an InOut Producer will wait for its response. |
| synchronous | true | Sets whether the Endpoint will use synchronous or asynchronous processing. |
| transacted | false | If the endpoint should use a JMS Session transaction. |
| ttl | \-1 | Disabled by default. Sets the Message time to live header. |

PRODUCER USAGE

INONLY PRODUCER - (DEFAULT)

The `InOnly` Producer is the default behavior of the `SJMS` Producer Endpoint.

```
from("direct:start")
.to("sjms:queue:bar");
```

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");
```

CONSUMERS CONFIGURATION OPTIONS

The `SjmsConsumer` Endpoint supports the following properties:

| Option | Default Value | Description |
|--------|---------------|-------------|
|--------|---------------|-------------|

| | | |
|--------------------------------|-------------------------|--|
| acknowledgementMode | AUTO_ACKNOWLEDGE | The JMS acknowledgement name, which is one of: TRANSACTIONED , AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE . CLIENT_ACKNOWLEDGE is not supported at this time. |
| consumerCount | 1 | Defines the number of MessageListener instances. |
| durableSubscriptionId | null | Required for a durable subscriptions. |
| exchangePattern | InOnly | Sets the Consumers message exchange pattern. |
| messageSelector | null | Sets the message selector. |
| synchronous | true | Sets whether the Endpoint will use synchronous or asynchronous processing. |
| transacted | false | If the endpoint should use a JMS Session transaction. |
| transactionBatchCount | 1 | The number of exchanges to process before committing a local JMS transaction. The transacted property must also be set to true or this property will be ignored. |
| transactionBatchTimeout | 5000 | The amount of time a the transaction will stay open between messages before committing what has already been consumed. Minimum value is 1000ms. |
| ttl | \-1 | Disabled by default. Sets the Message time to live header. |

CONSUMER USAGE

INONLY CONSUMER - (DEFAULT)

The `InOnly` Consumer is the default Exchange behavior of the `SJMS` Consumer Endpoint.

```
from("sjms:queue:bar")
.to("mock:result");
```

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"));
```

ADVANCED USAGE NOTES

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 pluggable `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](#).

SESSION, CONSUMER, & PRODUCER POOLING & CACHING MANAGEMENT

Coming soon ...

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 `BatchMessages`:

```
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);
```


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.

TRANSACTIONED BATCH CONSUMERS & PRODUCERS

The `SjmsComponent` 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 `SjmsConsumer` 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);
```

ADDITIONAL NOTES

MESSAGE HEADER FORMAT

The SJMS Component uses the same header format strategy that is used in the Camel JMS Component. This pluggable 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 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:

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.

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.

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).

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&xchangePattern=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 110. SMPP

SMPP COMPONENT

This component provides access to an SMSC (Short Message Service Center) over the [SMPP](#) protocol to send and receive SMS. The [JSMPP](#) is used.



USING APACHE CAMEL 2.2 ONWARDS

This component is only available for Apache Camel 2.2 or newer.

URI FORMAT

```
smpp://[username@]hostname[:port][?options]
smpps://[username@]hostname[:port][?options]
```

If no username is provided, then Apache Camel will provide the default value `smppclient`. If no port number is provided, then Apache Camel will provide the default value `2775`. Camel 2.3: If the protocol name is `smpps`, camel-smpp will 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&...`

URI OPTIONS

| Name | Default Value | Description |
|-------------------------|-----------------------|---|
| <code>password</code> | <code>password</code> | Specifies the password to use to log in to the SMSC. |
| <code>systemType</code> | <code>cp</code> | This parameter is used to categorize the type of ESME (External Short Message Entity) that is binding to the SMSC (max. 13 characters). |

| | | |
|-------------------------|-------------------|--|
| dataCoding | 0 | <p>Camel 2.11 Defines the data coding according the SMPP 3.4 specification, section 5.2.19. (Prior to Camel 2.9, this option is also supported.) Example data encodings are:</p> <ul style="list-style-type: none"> • 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) |
| alphabet | 0 | <p>Camel 2.5 Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. This option is mapped to Alphabet.java and used to create the byte[] which is send to the SMSC. Example alphabets are: 0: SMSC Default Alphabet 4: 8 bit Alphabet 8: UCS2 Alphabet</p> |
| encoding | ISO-8859-1 | <p>only for SubmitSm, ReplaceSm and SubmitMulti Defines the encoding scheme of the short message user data.</p> |
| enquireLinkTimer | 5000 | <p>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.</p> |
| transactionTimer | 10000 | <p>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).</p> |

| | | |
|------------------------------|-------------|--|
| initialReconnectDelay | 5000 | Defines the initial delay in milliseconds after the consumer/producer tries to reconnect to the SMSC, after the connection was lost. |
| reconnectDelay | 5000 | Defines the interval in milliseconds between the reconnect attempts, if the connection to the SMSC was lost and the previous was not succeed. |
| registeredDelivery | 1 | only for SubmitSm, ReplaceSm, SubmitMulti and DataSm 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. |
| serviceType | CMT | The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: <ul style="list-style-type: none"> • CMT: Cellular Messaging • CPT: Cellular Paging • VMN: Voice Mail Notification • VMA: Voice Mail Alerting • WAP: Wireless Application Protocol • USSD: Unstructured Supplementary Services Data |
| sourceAddr | 1616 | Defines the address of SME (Short Message Entity) which originated this message. |

| | | |
|----------------------|-------------|--|
| destAddr | 1717 | only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS. |
| sourceAddrTon | 0 | Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: <ul style="list-style-type: none"> • 0: Unknown • 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated |
| destAddrTon | 0 | only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the type of number (TON) to be used in the SME destination address parameters. The following TON values are defined: <ul style="list-style-type: none"> • 0: Unknown • 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated |

| | | |
|----------------------|----------|---|
| sourceAddrNpi | 0 | <p>Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined:</p> <ul style="list-style-type: none"> • 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) |
| destAddrNpi | 0 | <p>only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. The following NPI values are defined:</p> <ul style="list-style-type: none"> • 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) |

| | | |
|-----------------------------|----------|---|
| priorityFlag | 1 | <p>only for SubmitSm and SubmitMulti Allows the originating SME to assign a priority level to the short message. Four Priority Levels are supported:</p> <ul style="list-style-type: none"> • 0: Level 0 (lowest) priority • 1: Level 1 priority • 2: Level 2 priority • 3: Level 3 (highest) priority |
| replaceIfPresentFlag | 0 | <p>only for SubmitSm and SubmitMulti 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:</p> <ul style="list-style-type: none"> • 0: Don't replace • 1: Replace |
| dataCoding | 0 | <p>Camel 2.5 onwards Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. Example data encodings are: 0: SMSC Default Alphabet 4: 8 bit Alphabet 8: UCS2 Alphabet</p> |

| | | |
|-------------------------------|--------------|---|
| typeOfNumber | 0 | <p>Defines the type of number (TON) to be used in the SME. The following TON values are defined:</p> <ul style="list-style-type: none"> • 0: Unknown • 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated |
| numberingPlanIndicator | 0 | <p>Defines the numeric plan indicator (NPI) to be used in the SME. The following NPI values are defined:</p> <ul style="list-style-type: none"> • 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) |
| lazySessionCreation | false | <p>Camel 2.8 onwards Sessions can be lazily created to avoid exceptions, if the SMSC is not available when the Camel producer is started. Camel 2.11 onwards 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.</p> |

| | | |
|-----------------------------------|-------------------|--|
| <code>httpProxyHost</code> | <code>null</code> | Camel 2.9.1: If you need to tunnel SMPP through a HTTP proxy, set this attribute to the hostname or ip address of your HTTP proxy. |
| <code>httpProxyPort</code> | <code>3128</code> | Camel 2.9.1: If you need to tunnel SMPP through a HTTP proxy, set this attribute to the port of your HTTP proxy. |
| <code>httpProxyUsername</code> | <code>null</code> | Camel 2.9.1: If your HTTP proxy requires basic authentication, set this attribute to the username required for your HTTP proxy. |
| <code>httpProxyPassword</code> | <code>null</code> | Camel 2.9.1: If your HTTP proxy requires basic authentication, set this attribute to the password required for your HTTP proxy. |
| <code>sessionStateListener</code> | <code>null</code> | Camel 2.9.3: You can refer to a <code>org.jsmpp.session.SessionStateListener</code> in the Registry to receive callbacks when the session state changed. |
| <code>addressRange</code> | <code>""</code> | Camel 2.11: 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. |

You can have as many of these options as you like.

```
smpp://smppclient@localhost:2775?
password=password&enquireLinkTimer=3000&transactionTimer=5000&systemType=c
onsumer
```

PRODUCER MESSAGE HEADERS

The following message headers can be used to affect the behavior of the SMPP producer

| Header | Type | Description |
|-----------------------------------|-------------|--|
| <code>CamelSmppDestAddr</code> | 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. |
| <code>CamelSmppDestAddrTon</code> | Byte | only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the type of number (TON) to be used in the SME destination address parameters. The following TON values are defined: <ul style="list-style-type: none">• 0: Unknown• 1: International• 2: National• 3: Network Specific• 4: Subscriber Number• 5: Alphanumeric• 6: Abbreviated |

| | | |
|-------------------------------|---------------|---|
| CamelSmppDestAddrNpi | Byte | <p>only for SubmitSm, SubmitMulti, CancelSm and DataSm Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. The following NPI values are defined:</p> <ul style="list-style-type: none"> • 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) |
| CamelSmppSourceAddr | String | Defines the address of SME (Short Message Entity) which originated this message. |
| CamelSmppSourceAddrTon | Byte | <p>Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined:</p> <ul style="list-style-type: none"> • 0: Unknown • 1: International • 2: National • 3: Network Specific • 4: Subscriber Number • 5: Alphanumeric • 6: Abbreviated |

| | | |
|-------------------------------|---------------|---|
| CamelSmppSourceAddrNpi | Byte | Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: <ul style="list-style-type: none">• 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) |
| CamelSmppServiceType | String | The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: <ul style="list-style-type: none">• CMT: Cellular Messaging• CPT: Cellular Paging• VMN: Voice Mail Notification• VMA: Voice Mail Alerting• WAP: Wireless Application Protocol• USSD: Unstructured Supplementary Services Data |

| | | |
|------------------------------------|------------------------|--|
| CamelSmppRegisteredDelivery | Byte | <p>only for SubmitSm, ReplaceSm, SubmitMulti and DataSm Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. The following values are defined:</p> <ul style="list-style-type: none"> • 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. |
| CamelSmppPriorityFlag | Byte | <p>only for SubmitSm and SubmitMulti Allows the originating SME to assign a priority level to the short message. Four Priority Levels are supported:</p> <ul style="list-style-type: none"> • 0: Level 0 (lowest) priority • 1: Level 1 priority • 2: Level 2 priority • 3: Level 3 (highest) priority |
| CamelSmppValidityPeriod | String/{{Date}} | <p>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.</p> |

| | | |
|--|----------------------------------|--|
| CamelSmppReplaceIfPresentFlag | Byte | <p>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:</p> <ul style="list-style-type: none"> • 0: Don't replace • 1: Replace |
| CamelSmppAlphabet / CamelSmppDataCoding | Byte | <p>Camel 2.5For 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.</p> |
| CamelSmppOptionalParameters | Map<String, String> | <p>Deprecated and will be removed in Camel 2.13.0/3.0.0Camel 2.10.5 and 2.11.1 onwards and only for SubmitSm, SubmitMulti and DataSm The optional parameters send back by the SMSC.</p> |

| | | |
|----------------------------|--------------------|--|
| CamelSmppOptionalParameter | Map<Short, Object> | <p>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.COctetStringByte[] -> org.jsmpp.bean.OptionalParameter.OctetStringByte -> org.jsmpp.bean.OptionalParameter.ByteInteger -> org.jsmpp.bean.OptionalParameter.IntShort -> org.jsmpp.bean.OptionalParameter.Shortnull -> org.jsmpp.bean.OptionalParameter.Null</p> |
|----------------------------|--------------------|--|

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. |

| | | |
|---|--|---|
| <p>CamelSmppError</p> | <p>Map<String, List<Map<String, Object>>></p> | <p>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)).</p> |
| <p>CamelSmppOptionalParameters</p> | <p>Map<String, String></p> | <p>Deprecated and will be removed in Camel 2.13.0/3.0.0 From Camel 2.11.1 onwards only for DataSm The optional parameters which are returned from the SMSC by sending the message.</p> |
| <p>CamelSmppOptionalParameter</p> | <p>Map<Short, Object></p> | <p>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</p> |

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 | De |
|--------|------|----|
|--------|------|----|

| | | |
|--------------------------------|----------------|-----------------------|
| CamelSmppSequenceNumber | Integer | 01 re m |
| CamelSmppCommandId | Integer | 01 id 5. |
| CamelSmppSourceAddr | String | 01 (S |
| CamelSmppSourceAddrNpi | Byte | 01 be de |
| CamelSmppSourceAddrTon | Byte | 01 US |
| CamelSmppEsmeAddr | String | 01 te |

| | | |
|-------------------------------|-----------------------------|-----------------|
| CamelSmppEsmeAddrNpi | Byte | Optional |
| CamelSmppEsmeAddrTon | Byte | Optional |
| CamelSmppId | String | Optional |
| CamelSmppDelivered | Integer | Optional |
| CamelSmppDoneDate | Date | Optional |
| CamelSmppFinalStatus | DeliveryReceiptState | Optional |
| CamelSmppCommandStatus | Integer | Optional |

| | | |
|--------------------------------------|----------------|---|
| CamelSmppError | String | OI er er |
| CamelSmppSubmitDate | Date | OI su or |
| CamelSmppSubmitted | Integer | OI is va |
| CamelSmppDestAddr | String | OI te |
| CamelSmppScheduleDeliveryTime | String | OI m tir be re sr |
| CamelSmppValidityPeriod | String | OI tir lt ak sp |
| CamelSmppServiceType | String | OI in ge |
| CamelSmppRegisteredDelivery | Byte | OI ac |

| | | |
|------------------------------------|----------------------------------|---|
| CamelSmppDestAddrNpi | Byte | Or de |
| CamelSmppDestAddrTon | Byte | Or ac |
| CamelSmppMessageType | String | Ca Al m sh |
| CamelSmppOptionalParameters | Map<String, Object> | Di ar |
| CamelSmppOptionalParameter | Map<Short, Object> | Ca SE va or St by By Ir Sh |

JSMPP LIBRARY

See the documentation of the [JSMPP Library](#) for more details about the underlying library.

EXCEPTION HANDLING

This component supports the general Camel exception handling capabilities. 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=c
onsumer")
    .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.

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=p
roducer");
```

A route which sends an SMS using the Spring XML DSL:

```
<route>
  <from uri="direct:start"/>
  <to uri="smpp://smppclient@localhost:2775?
```

```
password=password&nquireLinkTimer=3000&ransactionTimer=5000&systemType=producer"/>
</route>
```

A route which receives an SMS using the Java DSL:

```
from("smp://smpclient@localhost:2775?
password=password&nquireLinkTimer=3000&transactionTimer=5000&systemType=consumer")
    .to("bean:foo");
```

A route which receives an SMS using the Spring XML DSL:

```
<route>
  <from uri="smp://smpclient@localhost:2775?
password=password&nquireLinkTimer=3000&ransactionTimer=5000&systemType=consumer"/>
  <to uri="bean:foo"/>
</route>
```

SMSC SIMULATOR

If you need an SMSC simulator for your test, you can use the simulator provided by [Logica](#).

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.smp=DEBUG
```

CHAPTER 111. SNMP

SNMP COMPONENT

The `snmp:` component gives you the ability to poll SNMP capable devices or receiving traps.

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&...`

OPTIONS

| Name | Default Value | Description |
|-----------------------|---------------|---|
| <code>type</code> | none | The type of action you want to perform. Actually you can enter here POLL or TRAP . The value POLL will instruct the endpoint to poll a given host for the supplied OID keys. If you put in TRAP you will setup a listener for SNMP Trap Events. |
| <code>address</code> | none | This is the IP address and the port of the host to poll or where to setup the Trap Receiver. Example: 127.0.0.1:162 |
| <code>protocol</code> | udp | Here you can select which protocol to use. You can use either udp or tcp . |
| <code>retries</code> | 2 | Defines how often a retry is made before canceling the request. |

| | | |
|----------------------|-------------------------------|--|
| timeout | 1500 | Sets the timeout value for the request in millis. |
| snmpVersion | 0 (which means SNMPv1) | Sets the snmp version for the request. |
| snmpCommunity | public | Sets the community octet string for the snmp request. |
| delay | 60 seconds | Defines the delay in seconds between to poll cycles. |
| oids | none | 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" |

THE RESULT OF A POLL

Given the situation, that I poll for the following 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:

```
<?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.

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 (*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");

```

CHAPTER 112. SOLR

SOLR COMPONENT

Available as of Camel 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>
```

URI FORMAT

```
solr://host[:port]/solr?[options]
```

ENDPOINT OPTIONS

The following [SolrServer](#) options may be configured on the Solr endpoint.

| name | default value | description |
|------------|---------------|---|
| maxRetries | 0 | maximum number of retries to attempt in the event of transient errors |

| | | |
|---|----------------------------|--|
| <code>soTimeout</code> | 1000 | read timeout on the underlying <code>HttpURLConnection</code> . This is desirable for queries, but probably not for indexing |
| <code>connectionTimeout</code> | 100 | <code>connectionTimeout</code> on the underlying <code>HttpURLConnection</code> |
| <code>defaultMaxConnectionsPerHost</code> | 2 | <code>maxConnectionsPerHost</code> on the underlying <code>HttpURLConnection</code> |
| <code>maxTotalConnections</code> | 20 | <code>maxTotalConnection</code> on the underlying <code>HttpURLConnection</code> |
| <code>followRedirects</code> | false | indicates whether redirects are used to get to the Solr server |
| <code>allowCompression</code> | false | server side must support gzip or deflate for this to have any effect |
| <code>requestHandler</code> | <code>/update (xml)</code> | set the request handler to be used |
| <code>streamingThreadCount</code> | 2 | Camel 2.9.2 set the number of threads for the StreamingUpdateSolrServer |
| <code>streamingQueueSize</code> | 10 | Camel 2.9.2 set the queue size for the StreamingUpdateSolrServer |

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 |
| 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 |
| OPTIMIZE | n/a | performs a commit on any pending index changes and then runs the optimize command |

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 message body to the insert route or to the delete route, 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);

```

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](#)

CHAPTER 113. SPRINGBATCH

SPRING BATCH COMPONENT

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>
```

URI FORMAT

```
spring-batch:jobName[?options]
```

Where `jobName` represents the name of the Spring Batch job located in the Camel registry.

**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.

OPTIONS

| Name | Default Value | Description |
|----------------|---------------|---|
| jobLauncherRef | null | Deprecated and will be removed in Camel 3.0! Camel 2.10: Use <code>jobLauncher=#theName</code> instead. |
| jobLauncher | null | Camel 2.11.1: Explicitly specifies a JobLauncher to be used from the Camel Registry . |

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.JobLauncher` 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`.

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 2.11.1JobExecution 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();
```

SUPPORT CLASSES

Apart from the Component, Camel Spring Batch provides also support classes, which can be used to hook into Spring Batch infrastructure.

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>

```

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="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>

```

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>

```

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.CamelJobExecutionLi
stener">
  <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>
```


CHAPTER 114. SPRINGINTEGRATION

SPRING INTEGRATION COMPONENT

The `spring-integration` component provides a bridge for Apache Camel components to talk to [spring integration endpoints](#).

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&...`

OPTIONS

| Name | Description | Example | Required | Default Value |
|---------------------------|--|--|----------|---------------|
| <code>inputChannel</code> | The Spring integration input channel name that this endpoint wants to consume from, where the specified channel name is defined in the Spring context. | <code>inputChannel=requestChannel</code> | No | |

| | | | | |
|--------------------------------|---|--------------------------------------|----|--|
| outputChannel | The Spring integration output channel name that is used to send messages to the Spring integration context. | outputChannel=replyChannel | No | |
| inOut | The exchange pattern that the Spring integration endpoint should use. | inOut=true | No | inOnly for the Spring integration consumer and outOnly for the Spring integration provider |
| consumer.delay | Delay in milliseconds between each poll. | consumer.delay=60000 | No | 500 |
| consumer.initialDelay | Milliseconds before polling starts. | consumer.initialDelay=10000 | No | 1000 |
| consumer.useRFixedDelay | Specify true to use fixed delay between polls, otherwise fixed rate is used. See the ScheduledExecutorService class for details. | consumer.useRFixedDelay=false | No | false |

USAGE

The Spring integration component is a bridge that connects Apache 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.

USING THE SPRING INTEGRATION ENDPOINT

You can set up a Spring integration endpoint using a URI, as follows:

```

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

  <channel id="inputChannel"/>
    <channel id="outputChannel"/>
    <channel id="onewayChannel"/>

  <service-activator input-channel="inputChannel"
    ref="helloService"
    method="sayHello"/>

  <service-activator input-channel="onewayChannel"
    ref="helloService"
    method="greet"/>

  <beans:bean id="helloService"
class="org.apache.camel.component.spring.integration.HelloWorldService"/>

  <camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="direct:twowayMessage"/>
      <!-- Using the &as the separator of & -->
      <to uri="spring-integration:inputChannel?
inOut=true&nputChannel=outputChannel"/>
    </route>
    <route>
      <from uri="direct:onewayMessage"/>
      <to uri="spring-integration:onewayChannel?inOut=false"/>
    </route>
  </camelContext>

  <channel id="requestChannel"/>
  <channel id="responseChannel"/>

  <beans:bean id="myProcessor"
class="org.apache.camel.component.spring.integration.MyProcessor"/>

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <!-- Using the &as the separator of & -->

```

```

    <from uri="spring-integration://requestChannel?
outputChannel=responseChannel&nOut=true"/>
    <process ref="myProcessor"/>
  </route>
</camelContext>

```

Or directly using a Spring integration channel name:

```

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

  <camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
    <route>
      <!-- camel will create a spring integration endpoint automatically
-->
      <from uri="outputChannel"/>
      <to uri="mock:result"/>
    </route>
  </camelContext>

```

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 Apache Camel endpoint or from a Apache Camel endpoint to a Spring integration channel.

This example uses the following namespaces:

```

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

```

```

    http://camel.apache.org/schema/spring/integration
    http://camel.apache.org/schema/spring/integration/camel-spring-
integration.xsd
    http://camel.apache.org/schema/spring
    http://camel.apache.org/schema/spring/camel-spring.xsd
">

```

You can bind your source or target to a Apache Camel endpoint as follows:

```

<!-- Create the camel context here -->
<camelContext id="camelTargetContext"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:EndpointA" />
        <to uri="mock:result" />
    </route>
    <route>
        <from uri="direct:EndpointC"/>
        <process ref="myProcessor"/>
    </route>
</camelContext>

<!-- We can bind the camelTarget to the camel context's endpoint by
specifying the camelEndpointUri attribute -->
<camel-si:camelTarget id="camelTargetA"
camelEndpointUri="direct:EndpointA" expectReply="false">
    <camel-si:camelContextRef>camelTargetContext</camel-
si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetB"
camelEndpointUri="direct:EndpointC" replyChannel="channelC"
expectReply="true">
    <camel-si:camelContextRef>camelTargetContext</camel-
si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetD"
camelEndpointUri="direct:EndpointC" expectReply="true">
    <camel-si:camelContextRef>camelTargetContext</camel-
si:camelContextRef>
</camel-si:camelTarget>

<beans:bean id="myProcessor"
class="org.apache.camel.component.spring.integration.MyProcessor"/>

<!-- spring integration channels -->
<channel id="channelA"/>
<channel id="channelB"/>
<channel id="channelC"/>

<!-- spring integration service activator -->
<service-activator input-channel="channelB" output-channel="channelC"

```

```
ref="helloService" method="sayHello"/>

<!-- custom bean -->
<beans:bean id="helloService"
class="org.apache.camel.component.spring.integration.HelloWorldService"/>

<camelContext id="camelSourceContext"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:OneWay"/>
    <to uri="direct:EndpointB" />
  </route>
  <route>
    <from uri="direct:TwoWay"/>
    <to uri="direct:EndpointC" />
  </route>
</camelContext>

<!-- camelSource will redirect the message coming for direct:EndpointB to
the spring requestChannel channelA -->

<camel-si:camelSource id="camelSourceA"
camelEndpointUri="direct:EndpointB"
requestChannel="channelA" expectReply="false">
  <camel-si:camelContextRef>camelSourceContext</camel-
si:camelContextRef>
</camel-si:camelSource>

<!-- camelSource will redirect the message coming for direct:EndpointC to
the spring requestChannel channelB
then it will pull the response from channelC and put the response message
back to direct:EndpointC -->

<camel-si:camelSource id="camelSourceB"
camelEndpointUri="direct:EndpointC"
requestChannel="channelB" replyChannel="channelC" expectReply="true">
  <camel-si:camelContextRef>camelSourceContext</camel-
si:camelContextRef>
</camel-si:camelSource>
```

CHAPTER 115. SPRING EVENT

SPRING EVENT COMPONENT

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](#).

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.

CHAPTER 116. SPRING LDAP

SPRING LDAP COMPONENT

Available since Camel 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>
```

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.

OPTIONS

| Name | Type | Description |
|------------------|--------|--|
| operation | String | The LDAP operation to be performed. Must be one of search , bind , or unbind . |
| scope | String | The scope of the search operation. Must be one of object , onelevel , or subtree , see also http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare |

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`). This map must contain at least an entry with the key `dn` 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` operation, 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,%20java.lang.String,%20int,%20org.springframework.ldap.core.AttributesMapper%29>.

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.

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.

UNBIND

No further entries necessary, the node with the specified dn is deleted.

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"
```

CHAPTER 117. SPRING REDIS

SPRING REDIS COMPONENT

Available as of Camel 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.



PREREQUISITES

In order to use this component, you must have a Redis server running.

URI FORMAT

```
spring-redis://host:port[?options]
```

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

URI OPTIONS

| Name | Default Value | Context | Description |
|------|---------------|---------|---|
| host | null | Both | The host where Redis server is running. |

| | | | |
|-------------------|-------------|----------|---|
| port | null | Both | Redis port number. |
| command | SET | Both | Default command, which can be overridden by message header. |
| channels | SET | Consumer | List of topic names or name patterns to subscribe to. |
| redisTemplate | null | Producer | Reference to a pre-configured <code>org.springframework.data.redis.core.RedisTemplate</code> instance in the Registry. |
| connectionFactory | null | Both | Reference to a <code>org.springframework.data.redis.connection.RedisConnectionFactory</code> instance in the Registry. |
| listenerContainer | null | Consumer | Reference to a <code>org.springframework.data.redis.listener.RedisMessageListenerContainer</code> instance in the Registry. |
| serializer | null | Consumer | Reference to a <code>org.springframework.data.redis.serializer.RedisSerializer</code> instance in the Registry. |

USAGE**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 |
| 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> |
| 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 |
| 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 |
| 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 |
| 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 |

| | | | |
|-------------------------|---|--|-------------|
| 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 |
| 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 |
| 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 |
| 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 |
| 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 |

REDIS CONSUMER

The consumer subscribes to a channel either by channel name using **SUBSCRIBE** or a string pattern using **PSUBSCRIBE** commands. When a message is sent to the channel using **PUBLISH** command, it will be consumed and the message will be available as Camel message body. The message is also serialized using configured serializer or the default `JdkSerializationRedisSerializer`.

Message headers set by the Consumer

| Header | Type | Description |
|---------------------------------|---------------------|---|
| <code>CamelRedis.Channel</code> | <code>String</code> | The channel name, where the message was received. |
| <code>CamelRedis.Pattern</code> | <code>String</code> | The pattern matching the channel, where the message was received. |

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).

CHAPTER 118. SPRING SECURITY

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.

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 |
|---------------------|-------------------|--|
| <code>id</code> | <code>null</code> | The unique Spring bean identifier which is used to reference the policy in routes (required) |
| <code>access</code> | <code>null</code> | The Spring Security authority name that is passed to the access decision manager (required) |

| | | |
|---------------------------------------|---|--|
| <code>authenticationManager</code> | <code>authenticationManager</code> | The name of the Spring Security AuthenticationManager object in the context |
| <code>accessDecisionManager</code> | <code>accessDecisionManager</code> | The name of the Spring Security AccessDecisionManager object in the context |
| <code>authenticationAdapter</code> | <code>DefaultAuthenticationAdapter</code> | Camel 2.4 The name of a camel-spring-securityAuthenticationAdapter object in the context that is used to convert a javax.security.auth.Subject into a Spring Security Authentication instance. |
| <code>useThreadSecurityContext</code> | <code>true</code> | If a javax.security.auth.Subject cannot be found in the In message header under <code>Exchange.AUTHENTICATION</code> , check the Spring Security SecurityContextHolder for an Authentication object. |
| <code>alwaysReauthenticate</code> | <code>false</code> | If set to true, the SpringSecurityAuthorizationPolicy will always call AuthenticationManager.authenticate() each time the policy is accessed. |

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:

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

```

    <bean id="accessDecisionManager"
class="org.springframework.security.access.vote.AffirmativeBased">
    <property name="allowIfAllAbstainDecisions" value="true"/>
    <property name="decisionVoters">
        <list>
            <bean
class="org.springframework.security.access.vote.RoleVoter"/>
        </list>
    </property>
</bean>

<spring-security:authentication-manager alias="authenticationManager">
    <spring-security:authentication-provider user-service-
ref="userDetailsService"/>
</spring-security:authentication-manager>

<spring-security:user-service id="userDetailsService">
    <spring-security:user name="jim" password="jimspassword"
authorities="ROLE_USER, ROLE_ADMIN"/>
    <spring-security:user name="bob" password="bobspassword"
authorities="ROLE_USER"/>
</spring-security:user-service>

</beans>

```

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:

```

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

    <!-- import the spring security configuration -->
    <import
resource="classpath:org/apache/camel/component/spring/security/commonSecurity.xml"/>

    <authorizationPolicy id="admin" access="ROLE_ADMIN"
authenticationManager="authenticationManager"
accessDecisionManager="accessDecisionManager"
xmlns="http://camel.apache.org/schema/spring-
security"/>

```

```

    <camelContext id="myCamelContext"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <!-- The exchange should be authenticated with the role of
ADMIN before it is send to mock:endpoint -->
        <policy ref="admin">
            <to uri="mock:end"/>
        </policy>
    </route>
</camelContext>

</beans>

```

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 `adminSpringSecurityAuthorizationPolicy`.

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 [camel-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(token);

    // 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 `camel-cxf` component populates the `Exchange.AUTHENTICATION` header.

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.AccessDeniedExcepti

```

```
on</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>
```

DEPENDENCIES

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

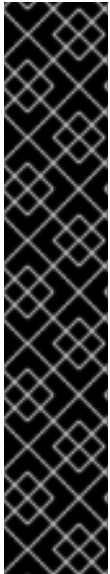
```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-security</artifactId>
  <version>2.4.0</version>
</dependency>
```

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.

CHAPTER 119. SPRING WEB SERVICES

SPRING WEB SERVICES COMPONENT

The `spring-ws:` component allows you to integrate with [Spring Web Services](#). It offers both *clientside* support, for accessing web services, and *serverside* support for creating your own contract-first web services.



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](#))

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 one of the following values:

| Mapping type | Description |
|-------------------------|---|
| <code>rootqname</code> | Offers the option to map web service requests based on the qualified name of the root element contained in the message. |
| <code>soapaction</code> | 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 a org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher 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 you are calling upon.

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

OPTIONS

| Name | Required? | Description |
|-------------------|-----------|---|
| soapAction | <i>No</i> | SOAP action to include inside a SOAP request when accessing remote web services |

| | | |
|---------------------------|---|--|
| wsAddressingAction | No | WS-Addressing 1.0 action header to include when accessing web services. The To header is set to the <i>address</i> of the web service as specified in the endpoint URI (default Spring-WS behavior). |
| expression | Only when <i>mapping-type</i> is xpathresult | XPath expression to use in the process of mapping web service requests, should match the result specified by xpathresult |
| timeout | No | <p>Camel 2.10: 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: <i>CommonsHttpMessageSender</i> and <i>HttpURLConnectionMessageSender</i>. 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. Camel 2.12: The built-in message sender <i>HttpComponentsMessageSender</i> is considered instead of <i>CommonsHttpMessageSender</i> which has been deprecated, see HttpComponentsMessageSender.setReadTimeout().</p> |

| | | |
|-----------------------------|----|---|
| sslContextParameters | No | <p>Camel 2.10: eference to an <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry. ee Using the JSSE Configuration Utility.</p> <p>This option works when using the built-in message sender implementations: <code>CommonsHttpMessageSender</code> and <code>HttpURLConnectionMessageSender</code>. ne of these implementations will be used by default for HTTP based services unless you customize the Spring WS configuration options supplied to the component. f you are using a non-standard sender, it is assumed that you will handle your own TLS configuration. Camel 2.12: he built-in message sender <code>HttpComponentsMessageSender</code> s considered instead of <code>CommonsHttpMessageSender</code> hich has been deprecated.</p> |
|-----------------------------|----|---|

REGISTRY BASED OPTIONS

The following options can be specified in the registry (most likely a Spring application context) and referenced from the endpoint URI using the `#beanID` notation.

| Name | Required? | Description |
|---------------------------|-----------|--|
| webServiceTemplate | No | 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. |

| | | |
|---------------------------|--|---|
| messageSender | No | Option to provide a custom WebServiceMessageSender . For example to perform authentication or use alternative transports |
| messageFactory | No | Option to provide a custom WebServiceMessageFactory . For example when you want Apache Axiom to handle web service messages instead of SAAJ |
| transformerFactory | No | Option to override the default TransformerFactory . The provided transformer factory must be of type javax.xml.transform.TransformerFactory |
| endpointMapping | Only when <i>mapping-type</i> is rootqname , soapaction , uri or xpathresult | Reference to 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) |
| messageFilter | No | Option to provide a custom MessageFilter since 2.10.3. For example when you want to process your headers or attachments by your own. |

MESSAGE HEADERS

| Name | Type | Description |
|--|--------|--|
| <code>CamelSpringWebserviceEndpointUri</code> | String | URI of the web service you are accessing as a client; overrides the <i>address</i> part of the endpoint URI. |
| <code>CamelSpringWebserviceSoapAction</code> | String | Header to specify the SOAP action of the message; overrides the soapAction option, if present |
| <code>CamelSpringWebserviceAddressingAction</code> | URI | Use this header to specify the WS-Addressing action of the message; overrides the wsAddressingAction option, if present |
| <code>CamelSpringWebserviceSoapHeader</code> | Source | Camel 2.11.1: Use this header to specify/access the SOAP headers of the message. |

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.

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");
```

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](#).

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 WebSdrviceMessage 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.

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: 1. 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>
```

2. 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>

```

3. 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))); }
}

```

4. Define endpoint (producer) as usual, for example like this:

```

from("direct:send")
.process(new Attachement())
.to("spring-ws:http://localhost:8089/mySoapService?
soapAction=mySoap&messageFactory=axiomMessageFactory");

```

5. Now, your producer will generate MTOM message with optimized attachments.

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 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.myMessageFilter"
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 doProcessSoapAttachments(Message inOrOut, SoapMessage
response)
{ your code /*no need to call super*/ }
```

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#saa-jboss -->
<bean id="messageFactory"
class="org.springframework.ws.soap.saa-j.Saa-jSoapMessageFactory">
```



```

<property name="messageFactory">
  <bean
class="com.sun.xml.messaging.saaj.soap.ver1_1.SOAPMessageFactory1_1Impl">
</bean>
</property>
</bean>

```

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</serv
let-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](#).

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 receives 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)

```

ALTERNATIVE CONFIGURATION, USING EXISTING ENDPOINT MAPPINGS

For every endpoint with mapping-type beaname 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 beaname 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 beaname 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 beaname:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="spring-ws:beaname: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"
/>
```

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 send and receive POJOs (Plain Old Java Objects), 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 POJOs and marshal the response message back to XML:

```
from("spring-ws:rootname:{http://example.com/}GetFoo?
endpointMapping=#endpointMapping").unmarshal(jaxb)
.to("mock:example").marshal(jaxb);
```

CHAPTER 120. SQL COMPONENT

SQL COMPONENT

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.

The SQL component also supports:

- a JDBC based repository for the [Idempotent Consumer](#) EIP pattern. See further below.
- a JDBC based repository for the [Aggregator](#) EIP pattern. See further below.

**IMPORTANT**

This component can be used as a [Transactional Client](#).

URI FORMAT

**WARNING**

In Camel 2.10 or older the SQL component can only be used as producer. From Camel 2.11 onwards this component can also be a consumer, eg `from()`.

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` 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, then an exception is thrown.

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.

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

OPTIONS

| Option | Type | Default | Description |
|--------------------|----------------------|--------------------|--|
| <code>batch</code> | <code>boolean</code> | <code>false</code> | Camel 2.7.5, 2.8.4 and 2.9: Execute SQL batch update statements. See notes below on how the treatment of the inbound message body changes if this is set to true . |

| | | | |
|-----------------------------------|----------------------|-------------------|---|
| <code>dataSourceRef</code> | <code>String</code> | <code>null</code> | Deprecated and will be removed in Camel 3.0: Reference to a DataSource to look up in the registry. Use dataSource=#theName instead. |
| <code>dataSource</code> | <code>String</code> | <code>null</code> | Camel 2.11: Reference to a DataSource to look up in the registry. |
| <code>placeholder</code> | <code>String</code> | <code>#</code> | Camel 2.4: Specifies a character that will be replaced to <code>?</code> in SQL query. Notice, that it is simple String.replaceAll() operation and no SQL parsing is involved (quoted strings will also change) |
| <code>template.<xxx></code> | | <code>null</code> | Sets additional options on the Spring JdbcTemplate that is used behind the scenes to execute the queries. For instance, template.maxRows=10 . For detailed documentation, see the JdbcTemplate javadoc documentation. |
| <code>allowNamedParameters</code> | <code>boolean</code> | <code>true</code> | Camel 2.11: Whether to allow using named parameters in the queries. |

| | | | |
|---------------------------------------|----------------------|--------------------|---|
| <code>processingStrategy</code> | | | Camel 2.11:SQL consumer only: Allows to plugin to use a custom <code>org.apache.camel.component.sql.SqlProcessingStrategy</code> to execute queries when the consumer has processed the rows/batch. |
| <code>prepareStatementStrategy</code> | | | Camel 2.11: Allows to plugin to use a custom <code>org.apache.camel.component.sql.SqlPreparedStatementStrategy</code> to control preparation of the query and prepared statement. |
| <code>consumer.delay</code> | <code>long</code> | <code>500</code> | Camel 2.11:SQL consumer only: Delay in milliseconds between each poll. |
| <code>consumer.initialDelay</code> | <code>long</code> | <code>1000</code> | Camel 2.11:SQL consumer only: Milliseconds before polling starts. |
| <code>consumer.useFixedDelay</code> | <code>boolean</code> | <code>false</code> | Camel 2.11:SQL consumer only: Set to <code>true</code> to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |
| <code>maxMessagesPerPoll</code> | <code>int</code> | <code>0</code> | Camel 2.11:SQL consumer only: An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set. |

| | | | |
|--|---------|-------|---|
| <code>consumer.useIterator</code> | boolean | true | Camel 2.11:SQL consumer only: If true each row returned when polling will be processed individually. If false the entire java.util.List of data is set as the IN body. |
| <code>consumer.routeEmptyResultSet</code> | boolean | false | Camel 2.11:SQL consumer only: Whether to route a single empty Exchange if there was no data to poll. |
| <code>consumer.onConsume</code> | String | null | Camel 2.11:SQL consumer only: 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. |
| <code>consumer.onConsumeFailed</code> | String | null | Camel 2.11:SQL consumer only: 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. |
| <code>consumer.onConsumeBatchComplete</code> | String | null | Camel 2.11:SQL consumer only: After processing the entire batch, this query can be executed to bulk update rows etc. The query cannot have parameters. |

| | | | |
|---|----------------------|--------------------|--|
| <code>consumer.expectedUpdateCount</code> | <code>int</code> | <code>\-1</code> | Camel 2.11:SQL consumer only: If using <code>consumer.onConsume</code> then this option can be used to set an expected number of rows being updated. Typically you may set this to <code>1</code> to expect one row to be updated. |
| <code>consumer.breakBatchOnConsumeFail</code> | <code>boolean</code> | <code>false</code> | Camel 2.11:SQL consumer only: If using <code>consumer.onConsume</code> and it fails, then this option controls whether to break out of the batch or continue processing the next row from the batch. |
| <code>alwaysPopulateStatement</code> | <code>boolean</code> | <code>false</code> | Camel 2.11:SQL producer only: If enabled then the <code>populateStatement</code> method from <code>org.apache.camel.component.sql.SqlPrepareStatementStrategy</code> is always invoked, also if there is no expected parameters to be prepared. When this is <code>false</code> then the <code>populateStatement</code> 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. |

| | | | |
|--------------------|---------------|-------------------|---|
| separator | char | , | Camel 2.11.1: 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. |
| outputType | String | SelectList | Camel 2.12.0: 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 a non-unique result exception. |
| outputClass | String | null | Camel 2.12.0: Specify the full package and class name to use as conversion when outputType=SelectOne . |

| | | | |
|------------------------------|----------------------|--------------------|--|
| <code>parametersCount</code> | <code>int</code> | <code>0</code> | Camel 2.11.2/2.12.0 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. |
| <code>noop</code> | <code>boolean</code> | <code>false</code> | Camel 2.12.0 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 |

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 other 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.

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`.

HEADER VALUES

When performing update operations, the SQL Component stores the update count in the following message headers:

| Header | Description |
|----------------------------------|--|
| <code>CamelSqlUpdateCount</code> | Apache Camel 2.0: The number of rows updated for update operations, returned as an Integer object. |
| <code>CamelSqlRowCount</code> | Apache Camel 2.0: The number of rows returned for select operations, returned as an Integer object. |
| <code>CamelSqlQuery</code> | 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 <i>are</i> represented by a <code>?</code> instead of a <code>#</code> symbol |

CONFIGURATION

You can now set a reference to a `DataSource` in the URI directly:

```
sql:select * from table where id=# order by name?dataSourceRef=myDS
```

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 java code:

```
// this is the database we create with some initial data for our unit test
```

```
db = new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.DERBY).addScript("sql/createAndPopulateDatabase.sql").build();
```

The SQL script `createAndPopulateDatabase.sql` we execute looks like as described below:

```
create table projects (id integer primary key, project varchar(10),
license varchar(5));
insert into projects values (1, 'Camel', 'ASF');
insert into projects values (2, 'AMQ', 'ASF');
insert into projects values (3, 'Linux', 'XXX');
```

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 in the registry, so we can use standard Spring XML to configure our `DataSource`.

```
from("direct:simple")
    .to("sql:select * from projects where license = # order by id?
dataSourceRef=jdbc/myDataSource")
    .to("mock:result");
```

And then we fire the message into the direct endpoint that will route it to our sql component that queries the database.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedMessageCount(1);

// send the query to direct that will route it to the sql where we will
// execute the query
// and bind the parameters with the data from the body. The body only
// contains one value
// in this case (XXX) but if we should use multi values then the body will
// be iterated
// so we could supply a List<String> instead containing each binding
// value.
template.sendBody("direct:simple", "XXX");

mock.assertIsSatisfied();

// the result is a List
List<?> received = assertInstanceOf(List.class,
mock.getReceivedExchanges().get(0).getIn().getBody());

// and each row in the list is a Map
```

```
Map<?, ?> row = assertIsInstanceOf(Map.class, received.get(0));

// and we should be able to get the project from the map that should be
Linux
assertEquals("Linux", row.get("PROJECT"));
```

We could configure the DataSource in Spring XML as follows:

```
<jee:jndi-lookup id="myDS" jndi-name="jdbc/myDataSource"/>
```

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")
```

USING THE JDBC BASED IDEMPOTENT REPOSITORY

In this section we will use the JDBC based idempotent repository.

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.

In Camel 2.8, we added the `createdAt` column:

```
CREATE TABLE CAMEL_MESSAGEPROCESSED (  
  processorName VARCHAR(255),  
  messageId VARCHAR(100),  
  createdAt TIMESTAMP  
)
```

We recommend to have a unique constraint on the columns `processorName` and `messageId`. Because the syntax for this constraint differs for database to database, we do not show it here.

Second we need to setup a `javax.sql.DataSource` in the spring XML file:

```
<jdbc:embedded-database id="dataSource" type="DERBY" />
```

And finally we can create our JDBC idempotent repository in the spring XML file as well:

```
<bean id="messageIdRepository"  
  class="org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository"  
>  
  <constructor-arg ref="dataSource" />  
  <constructor-arg value="myProcessorName" />  
</bean>  
  
<camel:camelContext>  
  <camel:errorHandler id="deadLetterChannel" type="DeadLetterChannel"  
    deadLetterUri="mock:error">  
    <camel:redeliveryPolicy maximumRedeliveries="0"  
      maximumRedeliveryDelay="0" logStackTrace="false" />  
  </camel:errorHandler>  
  
  <camel:route id="JdbcMessageIdRepositoryTest"
```



```

errorHandlerRef="deadLetterChannel">
  <camel:from uri="direct:start" />
  <camel:idempotentConsumer messageIdRepositoryRef="messageIdRepository">
    <camel:header>messageId</camel:header>
    <camel:to uri="mock:result" />
  </camel:idempotentConsumer>
</camel:route>
</camel:camelContext>

```

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 |
|-------------------------------------|--|---|
| <code>createTableIfNotExists</code> | <code>true</code> | Defines whether or not Camel should try to create the table if it doesn't exist. |
| <code>tableExistsString</code> | <code>SELECT 1 FROM CAMEL_MESSAGEPROCESSED WHERE 1 = 0</code> | 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. |
| <code>createString</code> | <code>CREATE TABLE CAMEL_MESSAGEPROCESSED (processorName VARCHAR(255), messageId VARCHAR(100), createdAt TIMESTAMP)</code> | The statement which is used to create the table. |
| <code>queryString</code> | <code>SELECT COUNT(*) FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ? AND messageId = ?</code> | 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). |

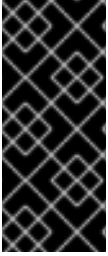
| | | |
|--------------|--|--|
| insertString | INSERT INTO CAMEL_MESSAGEPROCESSED (processorName, messageId, createdAt) VALUES (?, ?, ?) | The statement which is used to add the entry into the table. It takes three parameter. The first one is the processor name (String), the second one is the message id (String) and the third one is the timestamp (java.sql.Timestamp) when this entry was added to the repository. |
| deleteString | 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). |

A customized `org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository` could look like:

```
<bean id="messageIdRepository"
class="org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository"
>
  <constructor-arg ref="dataSource" />
  <constructor-arg value="myProcessorName" />
  <property name="tableExistsString" value="SELECT 1 FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE 1 = 0" />
  <property name="createString" value="CREATE TABLE
CUSTOMIZED_MESSAGE_REPOSITORY (processorName VARCHAR(255), messageId
VARCHAR(100), createdAt TIMESTAMP)" />
  <property name="queryString" value="SELECT COUNT(*) FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?"
/>
  <property name="insertString" value="INSERT INTO
CUSTOMIZED_MESSAGE_REPOSITORY (processorName, messageId, createdAt) VALUES
(?, ?, ?)" />
  <property name="deleteString" value="DELETE FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?"
/>
</bean>
```

USING THE JDBC BASED AGGREGATION REPOSITORY

Available as of Camel 2.6



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](#).

It has the following options:

| Option | Type | Description |
|---------------------------------|---------------------------------|--|
| <code>dataSource</code> | <code>DataSource</code> | Mandatory: The <code>javax.sql.DataSource</code> to use for accessing the database. |
| <code>repositoryName</code> | <code>String</code> | Mandatory: The name of the repository. |
| <code>transactionManager</code> | <code>TransactionManager</code> | Mandatory: The <code>org.springframework.transaction.PlatformTransactionManager</code> to manage transactions for the database. The <code>TransactionManager</code> must be able to support databases. |
| <code>lobHandler</code> | <code>LobHandler</code> | A <code>org.springframework.jdbc.support.lob.LobHandler</code> to handle <code>Lob</code> types in the database. Use this option to use a vendor specific <code>LobHandler</code> , for example when using Oracle. |

| | | |
|----------------------------|---------|---|
| 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. |
| storeBodyAsText | boolean | Camel 2.11: Whether to store the message body as String which is human readable. By default this option is false storing the body in binary format. |

| | | |
|--|---------------------------------|--|
| <code>headersToStoreAsText</code> | <code>List<String></code> | Camel 2.11: Allows to store headers as String which is human readable. By default this option is disabled, storing the headers in binary format. |
| <code>optimisticLocking</code> | <code>false</code> | Camel 2.12: To turn on optimistic locking, which often would be needed in clustered environments where multiple Camel applications shared the same JDBC based aggregation repository. |
| <code>jdbcOptimisticLockingExceptionHandler</code> | | Camel 2.12: Allows to plugin a custom <code>org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionHandler</code> to map vendor specific error codes to an optimistic locking error, for Camel to perform a retry. This requires <code>optimisticLocking</code> to be enabled. |

WHAT IS PRESERVED WHEN PERSISTING

`JdbcAggregationRepository` 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.

From Camel 2.11 onwards you can store the message body and select(ed) headers as String in separate columns.

RECOVERY

The `JdbcAggregationRepository` 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 |
|--|---------|---|
| <code>Exchange.REDELIVERED</code> | Boolean | Is set to true to indicate the Exchange is being redelivered. |
| <code>Exchange.REDELIVERY_COUNTER</code> | 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-sql`, for example [this test](#).

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)
);
```

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>

```

CODEC (SERIALIZATION)

Since 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.

TRANSACTION

A `Spring PlatformTransactionManager` is required to orchestrate transaction.

SERVICE (START/STOP)

The start method verify the connection of the database and the presence of the required tables. If anything is wrong it will fail during starting.

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.CommonsDbcpNativeJdbcEx
tractor"/>

<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>
```

OPTIMISTICK 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 optimistic 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.DefaultJdbcOptimisticLock
ingExceptionMapper">
```

```
<property name="classNames">
  <util:set>
    <value>com.foo.sql.MyViolationExceptoion</value>
    <value>com.foo.sql.MyOtherViolationExceptoion</value>
  </util:set>
</property>
</bean>
```

See also:

- [JDBC](#)

CHAPTER 121. SSH

SSH

Available as of Camel 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>
```

URI FORMAT

```
ssh:[username[:password]@]host[:port][?options]
```

OPTIONS

| Name | Default Value | Description |
|----------|---------------|---|
| host | | Hostname of SSH Server |
| port | 22 | Port of the SSH Server |
| username | | Username used for authenticating with SSH Server. |

| | | |
|------------------------|----------------|---|
| password | | Password used for authenticating with SSH Server. Used if keyPairProvider is null. |
| keyPairProvider | | Refers to a org.apache.sshd.common.KeyPairProvider to use for loading keys for authentication. If this option is used, then password is not used. |
| keyType | ssh-rsa | Refers to a key type to load from keyPairProvider . The key types can for example be "ssh-rsa" or "ssh-dss". |
| certResource | null | Camel 2.11: Path reference to a public key certificate. Prefix path with classpath: , file: , or http: . |
| certFilename | null | @deprecated: Use certResource instead. Refers to a filename to use within file based keyPairProvider . |
| timeout | 30000 | Milliseconds to wait before timing out connection to SSH Server. |

CONSUMER ONLY OPTIONS

| Name | Default Value | Description |
|---------------------|---------------|--|
| initialDelay | 1000 | Milliseconds before polling the SSH server starts. |
| delay | 500 | Milliseconds before the next poll of the SSH Server. |

| | | |
|----------------------------|-------------------|--|
| <code>useFixedDelay</code> | <code>true</code> | Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details. |
| <code>pollCommand</code> | <code>null</code> | Command to send to SSH Server during each poll cycle. You may need to end your command with a newline, and that must be URL encoded %0A. |

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>
```

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.

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&useFixedDelay=true&delay=5000&pollCommand=fea
tures: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>bcpkg-jdk15on</artifactId>
  <version>1.47</version>
</dependency>
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcpkix-jdk15on</artifactId>
  <version>1.47</version>
</dependency>
```

EXAMPLE

See the `examples/camel-example-ssh` and `examples/camel-example-ssh-security` in the Camel distribution.

CHAPTER 122. STAX

STAX COMPONENT

Available as of Camel 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>
```

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
```

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
    }
  });
```

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"/>
</records>

```

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");
```

THE PREVIOUS EXAMPLE WITH XML DSL

The example above could be implemented as follows in XML DSL

```
<!-- use STAXBuilder to create the expression we want to use in the route
below for splitting the XML file -->
<!-- notice we use the factory-method to define the stax method, and to
pass in the parameter as a constructor-arg -->
<bean id="staxRecord" class="org.apache.camel.component.stax.StAXBuilder"
factory-method="stax">
    <!-- FQN class name of the POJO with the JAXB annotations -->
    <constructor-arg index="0"
value="org.apache.camel.component.stax.model.Record"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <!-- pickup XML files -->
        <from uri="file:target/in"/>
        <split streaming="true">
            <!-- split the file using StAX (ref to bean above) -->
            <!-- and use streaming mode in the splitter -->
            <ref>staxRecord</ref>
            <!-- and send each splitted to a mock endpoint, which will be a
Record POJO instance -->
            <to uri="mock:records"/>
        </split>
    </route>
</camelContext>
```

CHAPTER 123. STOMP

STOMP COMPONENT

Available as of Camel 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>
```

URI FORMAT

```
stomp:queue:destination[?options]
```

Where `destination` is the name of the queue.

OPTIONS

| Property | Default | Description |
|------------------------|------------------------------------|---|
| <code>brokerURL</code> | <code>tcp://localhost:61613</code> | The URI of the Stomp broker to connect to |
| <code>login</code> | | The username |
| <code>passcode</code> | | The password |

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

SAMPLES

Sending messages:

```
from("direct:foo").to("stomp:queue:test");
```

Consuming messages:

```
from("stomp:queue:test").transform(body().convertToString()).to("mock:result")
```

CHAPTER 124. STREAM

STREAM COMPONENT

The `stream:` component provides access to the `System.in`, `System.out` and `System.err` streams as well as allowing streaming of file and URL.

URI FORMAT

```
stream:in[?options]
stream:out[?options]
stream:err[?options]
stream:header[?options]
```

In addition, the `file` and `url` endpoint URIs are supported in Apache Camel 2.0:

```
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&...`

OPTIONS

| Name | Default Value | Description |
|--------------------|----------------|---|
| <code>delay</code> | <code>0</code> | Initial delay in milliseconds before consuming or producing the stream. |

| | | |
|---------------------------|--------------------|--|
| encoding | <i>JVM Default</i> | As of 1.4, 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, Apache Camel uses the JVM default Charset . |
| promptMessage | null | Apache Camel 2.0: Message prompt to use when reading from stream:in ; for example, you could set this to Enter a command: |
| promptDelay | 0 | Apache Camel 2.0: Optional delay in milliseconds before showing the message prompt. |
| initialPromptDelay | 2000 | Apache Camel 2.0: 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. |
| fileName | null | Apache Camel 2.0: When using the stream:file URI format, this option specifies the filename to stream to/from. |
| url | null | 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. |
| scanStream | false | Apache Camel 2.0: To be used for continuously reading a stream such as the unix tail command. Camel 2.4 to Camel 2.6: will retry opening the file if it is overwritten, somewhat like tail --retry |

| | | |
|------------------------------|--------------------|--|
| <code>retry</code> | <code>false</code> | Camel 2.7: will retry opening the file if it's overwritten, somewhat like <code>tail --retry</code> |
| <code>scanStreamDelay</code> | <code>0</code> | Apache Camel 2.0: Delay in milliseconds between read attempts when using <code>scanStream</code> . |
| <code>groupLines</code> | <code>0</code> | Camel 2.5: 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. |
| <code>autoCloseCount</code> | <code>0</code> | Camel 2.10.0: (2.9.3 and 2.8.6) 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 <code>autoCloseCount</code> batch. |
| <code>closeOnDone</code> | <code>false</code> | Camel 2.11.0: 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. |

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.

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`.

```
private OutputStream mystream = new MyOutputStream();
private StringBuffer sb = new StringBuffer();

@Test
public void testStringContent() {
    template.sendBody("direct:in", "Hello");
    // StreamProducer appends \n in text mode
    assertEquals("Hello\n", sb.toString());
}

@Test
public void testBinaryContent() {
    template.sendBody("direct:in", "Hello".getBytes());
    // StreamProducer is in binary mode so no \n is appended
    assertEquals("Hello", sb.toString());
}

protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from("direct:in").setHeader("stream", constant(mystream)).
                to("stream:header");
        }
    };
}

private class MyOutputStream extends OutputStream {

    public void write(int b) throws IOException {
        sb.append((char)b);
    }
}
```

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");
```



NOTE

One difficulty with using the combination of `scanStream` and `retry` is that 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 or recreated.

CHAPTER 125. STRINGTEMPLATE

STRING TEMPLATE

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>
```

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&...`

OPTIONS

| Option | Default | Description |
|---------------------------|--------------------|---|
| <code>contentCache</code> | <code>false</code> | Cache for the resource content when its loaded. Note : as of Camel 2.9 cached resource content can be cleared via JMX using the endpoint's clearContentCache operation. |

| | | |
|-----------------------------|-------------------|--|
| <code>delimiterStart</code> | <code>null</code> | Since Camel 2.11.1 , configuring the variable start delimiter |
| <code>delimiterEnd</code> | <code>null</code> | Since Camel 2.11.1 , configuring the variable end delimiter |

HEADERS

Apache 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.

HOT RELOADING

The string template resource is by default hot-reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, Apache Camel loads the resource only once and hot-reloading is not possible. This scenario can be used in production when the resource never changes.

STRINGTEMPLATE ATTRIBUTES

Apache Camel will provide exchange information as attributes (just a `java.util.Map`) to the string template. The Exchange is transferred as:

| key | value |
|---------------------------|--|
| <code>exchange</code> | The Exchange itself. |
| <code>headers</code> | The headers of the In message. |
| <code>camelContext</code> | The Camel Context. |
| <code>request</code> | The In message. |
| <code>in</code> | The In message. |
| <code>body</code> | The In message body. |
| <code>out</code> | The Out message (only for InOut message exchange pattern). |

| | |
|----------|--|
| response | The Out message (only for InOut message exchange pattern). |
|----------|--|

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");
```

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:

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testVelocityLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Dear Ibsen, Claus! Thanks for the order
of Camel in Action. Regards Camel Riders Bookstore PS: Next beer is on me,
James");

    template.send("direct:a", createLetter());
}
```

```
        mock.assertIsSatisfied();
    }

    protected RouteBuilder createRouteBuilder() throws Exception {
        return new RouteBuilder() {
            public void configure() throws Exception {
                from("direct:a").to("string-
template:org/apache/camel/component/stringtemplate/letter.tm").to("mock:re
sult");
            }
        };
    }
}
```

CHAPTER 126. STUB

STUB COMPONENT

Available as of Camel 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.

URI FORMAT

```
stub:someUri
```

Where `someUri` can be any URI with any query parameters.

EXAMPLES

Here are a few samples:

-

`stub:smtp://somehost.foo.com?user=whatnot&something=else`

-

`stub:http://somehost.bar.com/something`

CHAPTER 127. TEST

TEST COMPONENT

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.

URI FORMAT

```
test:expectedMessagesEndpointUri
```

Where `expectedMessagesEndpointUri` refers to some other [Component](#) URI that the expected message bodies are pulled from before starting the test.

URI OPTIONS

| Name | Default Value | Description |
|---------|---------------|---|
| timeout | 2000 | Camel 2.12: The timeout to use when polling for message bodies from the URI. |

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.

Here is a [real example test case using Mock and Spring](#) along with its [Spring XML](#).

To see how you can set other expectations on the test endpoint, see the [Mock](#) component.

CHAPTER 128. TIMER

TIMER COMPONENT

The `timer:` component is used to generate message exchanges when a timer fires. You can only consume events from this endpoint.

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.

ADVANCED SCHEDULER

See also the [Quartz](#) component that supports much more advanced scheduling.

SPECIFY TIME IN HUMAN FRIENDLY FORMAT

In Camel 2.3 onwards you can specify the time in [human friendly syntax](#).

OPTIONS

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|--------------------|--------------|--|
| time | null | 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 . |
| pattern | null | Allows you to specify a custom Date pattern to use for setting the time option using URI syntax. |
| period | 1000 | If greater than 0, generate periodic events every period milliseconds. |
| delay | 1000 | 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 has been changed to 1000 from Camel 2.11 onwards. In older releases the default value is 0 . |
| fixedRate | false | Events take place at approximately regular intervals, separated by the specified period. |
| daemon | true | Specifies whether or not the thread associated with the timer endpoint runs as a daemon. |
| repeatCount | 0 | Camel 2.8: 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. |

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. |

MESSAGE HEADERS

When the timer is fired, it adds the following information as headers to the IN message

| Name | Type | Description |
|---------------------------|----------------|----------------------------------|
| Exchange.TIMER_FIRED_TIME | java.util.Date | The time when the consumer fired |

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 specify the more readable, `period=60s`.

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&period=60000"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

FIRING ONLY ONCE

Available as of Camel 2.8

You may want to fire a message in a Apache Camel route only once, such as when starting the route. To do that, you use the `repeatCount` option as follows:

```
<route>
  <from uri="timer://foo?repeatCount=1"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

See also:

- [Quartz](#)

CHAPTER 129. TWITTER

TWITTER

Available as of Camel 2.10

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>
```

URI FORMAT

```
twitter://endpoint[?options]
```

TWITTERCOMPONENT:

The twitter component can be configured with the Twitter account settings which is mandatory to configure before using. You can also configure these options directly in the endpoint.

| Option | Description |
|--------|-------------|
|--------|-------------|

| | |
|--------------------------------|-------------------------|
| <code>consumerKey</code> | The consumer key |
| <code>consumerSecret</code> | The consumer secret |
| <code>accessToken</code> | The access token |
| <code>accessTokenSecret</code> | The access token secret |

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 (once for each Status).

| Endpoint | Context | Body Type | Notice |
|------------------------------------|---|---------------------------------------|--|
| <code>directmessage</code> | direct, polling | <code>twitter4j.Direct Message</code> | |
| <code>search</code> | direct, polling | <code>twitter4j.Tweet</code> | |
| <code>streaming/filter</code> | event, polling | <code>twitter4j.Status</code> | |
| <code>streaming/sample</code> | event, polling | <code>twitter4j.Status</code> | |
| <code>timeline/home</code> | direct, polling | <code>twitter4j.Status</code> | |
| <code>timeline/mentions</code> | direct, polling | <code>twitter4j.Status</code> | |
| <code>timeline/public</code> | direct, polling | <code>twitter4j.Status</code> | @deprecated. Use <code>timeline/home</code> or <code>direct/home</code> instead. Removed from Camel 2.11 onwards. |
| <code>timeline/retweet some</code> | direct, polling | <code>twitter4j.Status</code> | |
| <code>timeline/user</code> | direct, polling | <code>twitter4j.Status</code> | |
| <code>trends/daily</code> | Camel 2.10.1: direct, polling | <code>twitter4j.Status</code> | @deprecated. Removed from Camel 2.11 onwards. |

| | | | |
|---------------|---|------------------|---|
| trends/weekly | Camel 2.10.1: direct, polling | twitter4j.Status | @deprecated. Removed from Camel 2.11 onwards. |
|---------------|---|------------------|---|

PRODUCER ENDPOINTS:

| Endpoint | Body Type |
|---------------|-----------------------|
| directmessage | String |
| search | List<twitter4j.Tweet> |
| timeline/user | String |

URI OPTIONS

| Name | Default Value | Description |
|----------------|---------------|---|
| type | direct | direct, event, or polling |
| delay | 60 | in seconds |
| consumerKey | null | Consumer Key. Can also be configured on the TwitterComponent level instead. |
| consumerSecret | null | Consumer Secret. Can also be configured on the TwitterComponent level instead. |
| accessToken | null | Access Token. Can also be configured on the TwitterComponent level instead. |

| | | |
|-------------------|-------------|---|
| accessTokenSecret | null | Access Token Secret. Can also be configured on the TwitterComponent level instead. |
| user | null | Username, used for user timeline consumption, direct message production, etc. |
| locations | null | 'lat,lon;lat,lon;...' Bounding boxes, created by pairs of lat/lons. Can be used for streaming/filter |
| keywords | null | 'foo1,foo2,foo3...' Can be used for search and streaming/filter. See Advanced search for keywords syntax for searching with for example OR. |
| userIds | null | 'username,username...' Can be used for streaming/filter |
| filterOld | true | Filter out old tweets, that has previously been polled. This state is stored in memory only, and based on last tweet id. Since Camel 2.11.0 The search producer supports this option |
| sinceId | 1 | Camel 2.11.0: 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. |
| lang | null | Camel 2.11.0: The lang string ISO_639-1 which will be used for searching |
| count | null | Camel 2.11.0: Limiting number of results per page. |
| numberOfPages | 1 | Camel 2.11.0: The number of pages result which you want camel-twitter to consume. |

MESSAGE HEADER

| Name | Description |
|---|---|
| <code>CamelTwitterKeywords</code> | This header is used by the search producer to change the search key words dynamically. |
| <code>CamelTwitterSearchLanguage</code> | Camel 2.11.0: This header can override the option of lang which set the search language for the search endpoint dynamically |
| <code>CamelTwitterCount</code> | Camel 2.11.0 This header can override the option of count which sets the max twitters that will be returned. |
| <code>CamelTwitterNumberOfPages</code> | Camel 2.11.0 This header can converrid the option of numberOfPages which sets how many pages we want to twitter returns. |

MESSAGE BODY

All message bodies utilize objects provided by the Twitter4J API.

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]);
```

TO POLL, EVERY 5 SEC., ALL STATUSES ON YOUR HOME TIMELINE:

```
from("twitter://timeline/home?type=polling&delay=5&consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&accessTokenSecret=[s]")
    .to("bean:blah");
```

TO SEARCH FOR ALL STATUSES WITH THE KEYWORD 'CAMEL':

```
from("twitter://search?type=direct&keywords=camel&consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&accessTokenSecret=[s]")
    .to("bean:blah");
```

SEARCHING USING A PRODUCER WITH STATIC KEYWORDS

```
from("direct:foo")
    .to("twitter://search?keywords=camel&consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&accessTokenSecret=[s]");
```

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]");
```

EXAMPLE

See also the [Twitter Websocket Example](#).

- [Twitter Websocket Example](#)

CHAPTER 130. URLREWRITE

URLREWRITE COMPONENT

Available as of Camel 2.11

The camel-urlrewrite component allows to plugin url rewrite functionality to [HTTP](#), [HTTP4](#), [Jetty](#), or [AHC](#) components. This component integrates the [UrlRewriteFilter](#) project with Apache Camel. This allows you to use the capabilities from the url rewrite project with your Camel routes.

This component requires that your Camel routes starts from a servlet based endpoint such as [Jetty](#) or [Servlet](#).

OPTIONS

The camel-urlrewrite component offers the following options

| Name | Default Value | Description |
|-------------------------|-------------------|--|
| <code>conf</code> | <code>null</code> | To use a custom <code>org.tuckey.web.filters.urlrewrite.Conf</code> as the configuration for the url rewrite filter. |
| <code>configFile</code> | <code>null</code> | Refers to a file name to load as the configuration file. The file should be XML file. |

| | | |
|---------------------------|----------------|--|
| modRewriteConfFile | null | Refers to a file name to load as the Apache mod_rewrite style based configuration file. This file should be a plain text file. |
| modRewriteConfText | null | Is used to quickly configure a Apache mod_rewrite style based configuration without using a file. |
| useQueryString | false | Whether to use query strings in the url rewrite rules. |
| useContext | false | Whether to include context-path or not in the matching. Will be default remove the context-path. |
| defaultMatchType | "regex" | Which match type to use. Can be either "wildcard" or "regex". |
| decodeUsing | null | To use a specific encoding such as "utf-8" or "iso-8859-1". |

USAGE

The following component producers supports using together with the camel-`urlrewrite` component: [HTTP](#), [HTTP4](#) and [Jetty](#).

| Component | Class |
|--------------------------|--|
| <code>camel-http</code> | <code>org.apache.camel.component.urlrewrite.http.HttpUrlRewrite</code> |
| <code>camel-jetty</code> | <code>org.apache.camel.component.urlrewrite.http.HttpUrlRewrite</code> |

| | |
|-------------|--|
| camel-http4 | org.apache.camel.component.urlrewrite.http4.HttpUrlRewrite |
|-------------|--|

You setup the url rewrite as a bean of the type `org.apache.camel.component.urlrewrite.http.HttpUrlRewrite` (when using [HTTP](#) component) as shown below:

```
HttpUrlRewrite myRewrite = new HttpUrlRewrite();
myRewrite.setConfigFile("example/urlrewrite2.xml");
```

And in XML DSL you can do:

```
<bean id="myRewrite"
class="org.apache.camel.component.urlrewrite.HttpUrlRewrite">
  <property name="configFile" value="example/urlrewrite2.xml"/>
</bean>
```

In the bean above we configure it with the `configFile` option which is a XML [UrlRewriteFilter](#) configuration file.

```
<urlrewrite>

  <!-- this is a rule where we map from /products/nnn to
        an url that is a jps page with the product id as a parameter -->
  <rule>
    <from>/products/([0-9]+)</from>
    <to>/products/index.jsp?product_id=$1</to>
  </rule>

</urlrewrite>
```

TIP

See more details at [UrlRewriteFilter](#) for how to configure the XML rules file.

In the Camel routes, you bridge the endpoints, and refer to the url rewrite on the producer endpoint as shown below. Notice how we refer to our url rewrite filter using the `urlRewrite` option in the endpoint uri.


```
from("jetty:http://localhost:{{port}}/myapp?matchOnUriPrefix=true")
  .to("jetty:http://localhost:{{port2}}/myapp2?
bridgeEndpoint=true&throwExceptionOnFailure=false&urlRewrite=#myRewrite");
```

USING LOAD BALANCING EIP

You can also use Camel's [Load Balancer](#) together with [UrlRewrite](#), as shown below. In this code we use the failover load balancer, to failover to the 2nd endpoint (which goes to myapp3) in case of any exception thrown.

```
// we want to use the failover loadbalancer
// to have it to react we must set throwExceptionOnFailure=true, which is
// also the default value
// so we can omit configuring this option
from("jetty:http://localhost:{{port}}/myapp?matchOnUriPrefix=true")
  .loadBalance().failover(Exception.class)
  .to("jetty:http://localhost:{{port2}}/myapp2?
bridgeEndpoint=true&throwExceptionOnFailure=true&urlRewrite=#myRewrite")
  .to("jetty:http://localhost:{{port2}}/myapp3?
bridgeEndpoint=true&throwExceptionOnFailure=true&urlRewrite=#myRewrite");
```

USING APACHE MOD_REWRITE STYLE CONFIGURATION

The [UrlRewriteFilter](#) project also supports Apache mod_rewrite style configuration.

For example to configure the bean using the modRewriteConfFile property to refer to a mod configuration file:

```
<bean id="myRewrite"
class="org.apache.camel.component.urlrewrite.HttpUrlRewrite">
  <property name="modRewriteConfFile" value="example/modrewrite.cxf"/>
</bean>
```

Which then can contain mod rewrite rules and the likes:

```
# our custom mod rewrite rule
RewriteRule page/([^\./]+)/?$ index.php?page=$1 [L]
```

You can have multiple RewriteRule rules and other configurations. See more details at [UrlRewriteFilter](#) project.

WRITING CUSTOM URL REWRITERS

You can implement your custom url rewriters by implementing either of the following interfaces from the component of choice:

Component(s): camel-http / camel-jetty

-

`org.apache.camel.component.http.UrlRewrite`

-

`org.apache.camel.component.http.HttpServletUrlRewrite`

Component(s): camel-http4

-

`org.apache.camel.component.http4.UrlRewrite`

-

`org.apache.camel.component.http4.HttpServletUrlRewrite`

The former is a simple and generic interface, which is not dependent on the Servlet API. The later is servlet based which requires the Camel route to start from a servlet based component such as [Jetty](#) or [Servlet](#). The servlet based is more feature rich and that's the API we use to integrate with the [UrlRewriteFilter](#) project in this camel-urlrewrite component.

A SIMPLE URL REWRITE FILTER

In this example we implement a custom `org.apache.camel.component.http.UrlRewrite` that just replaces yahoo with google. Mind this example is based on a unit test, and only for demonstration purposes. As its java code you have full power to implement more complex logic.

The `url` parameter contains the full url including `scheme://hostname:port/path?query`. The `relativeUrl` parameter is the url without the endpoint configured path. Notice this option may be null if the url doesn't start with the endpoint configured path.

```
/**
 * A very simple url rewrite that replaces yahoo with google in the url.
 * <p/>
 * This is only used for testing purposes.
 */
public class GoogleUrlRewrite implements UrlRewrite {

    @Override
    public String rewrite(String url, String relativeUrl, Producer
producer) {
        return url.replaceAll("yahoo", "google");
    }
}
```

•

Jetty

-

HTTP4

-

HTTP

-

AHC

CHAPTER 131. VALIDATION

VALIDATION COMPONENT

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](#).

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`.

OPTIONS

| Option | Default | Description |
|-------------------------------|--------------------|---|
| <code>resourceResolver</code> | <code>null</code> | Camel 2.9: Reference to a <code>org.w3c.dom.ls.LSResourceResolver</code> in the Registry . |
| <code>useDom</code> | <code>false</code> | Apache Camel 2.0: Whether <code>DOMSource</code> / <code>DOMResult</code> or <code>SaxSource</code> / <code>SaxResult</code> should be used by the validator. |
| <code>useSharedSchema</code> | <code>true</code> | Camel 2.3: 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. |
| <code>failOnNullBody</code> | <code>true</code> | Camel 2.9.5/2.10.3: Whether to fail if no body exists. |
| <code>headerName</code> | <code>null</code> | Camel 2.11: To validate against a header instead of the message body. |
| <code>failOnNullHeader</code> | <code>true</code> | Camel 2.11: Whether to fail if no header exists when validating against a header. |

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).

```
<route>
  <from uri="direct:start"/>
  <doTry>
    <to
uri="validator:org/apache/camel/component/validator/schema.xsd"/>
    <to uri="mock:valid"/>
    <doCatch>

<exception>org.apache.camel.ValidationException</exception>
    <to uri="mock:invalid"/>
    </doCatch>
    <doFinally>
    <to uri="mock:finally"/>
    </doFinally>
  </doTry>
</route>
```

CHAPTER 132. VELOCITY

VELOCITY

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.

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 (for example, `file://folder/myfile.vm`).

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

OPTIONS

| Option | Default | Description |
|-----------------------------|-------------------|--|
| <code>loaderCache</code> | <code>true</code> | Velocity based file loader cache. |
| <code>contentCache</code> | <code>true</code> | Cache for the resource content when it is loaded. Note : as of Camel 2.9 cached resource content can be cleared via JMX using the endpoint's clearContentCache operation. |
| <code>encoding</code> | <code>null</code> | Character encoding of the resource content. |
| <code>propertiesFile</code> | <code>null</code> | New option in Camel 2.1: The URI of the properties file which is used for VelocityEngine initialization. |

VELOCITY CONTEXT

Apache Camel will provide exchange information in the Velocity 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 instance. |
| request | The In message. |
| 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). |

HOT RELOADING

The Velocity template resource is, by default, hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, Apache 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.

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. |

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");
```

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:

```
private Exchange createLetter() {
    Exchange exchange = context.getEndpoint("direct:a").createExchange();
    Message msg = exchange.getIn();
    msg.setHeader("firstName", "Claus");
    msg.setHeader("lastName", "Ibsen");
    msg.setHeader("item", "Camel in Action");
    msg.setBody("PS: Next beer is on me, James");
    return exchange;
}

@Test
public void testVelocityLetter() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
```

```
    mock.expectedBodiesReceived("Dear Ibsen, Claus\n\nThanks for the order
of Camel in Action.\n\nRegards Camel Riders Bookstore\nPS: Next beer is on
me, James");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {

            from("direct:a").to("velocity:org/apache/camel/component/velocity/letter.v
m").to("mock:result");
        }
    };
}
```

CHAPTER 133. VERTX

VERTX COMPONENT

Available as of Camel 2.12

The vertx component is for working with the [Vertx EventBus](#).

The vertx [EventBus](#) sends and receives JSON events.

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>
```

URI FORMAT

```
vertx:channelName[?options]
```

At the moment, there are no options for this component. Any future release could change this, so please check back.

CHAPTER 134. VM

VM COMPONENT

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.

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&...`

BEFORE CAMEL 2.3 - SAME URI MUST BE USED FOR BOTH PRODUCER AND CONSUMER

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 example as follows:

```
from("direct:foo").to("vm:bar");  
from("vm:bar?concurrentConsumers=5").to("file://output");
```

OPTIONS

See the [SEDA](#) component for options and other important usage details as the same rules apply to the [VM](#) component.

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);
```

- [SEDA](#)

CHAPTER 135. WEATHER

WEATHER COMPONENT

Available as of Camel 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.

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>
```

URI FORMAT

```
weather://<unused name>[?options]
```

OPTIONS

| Property | Default | Description |
|-------------------|-------------|--|
| location | null | 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. |
| lat | null | Latitude of location. You can use lat and lon options instead of location. |
| lon | null | Longitude of location. You can use lat and lon options instead of location. |
| period | null | 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) |
| headerName | null | 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. |
| mode | JSON | The output format of the weather data. The possible values are HTML , JSON or XML |

| | | |
|--------------------------------------|----------------|--|
| <code>units</code> | METRIC | The units for temperature measurement. The possible values are IMPERIAL or METRIC |
| <code>consumer.delay</code> | 3600000 | Delay in millis between each poll (default is 1 hour) |
| <code>consumer.initialDelay</code> | 1000 | Millis before polling starts. |
| <code>consumer.userFixedDelay</code> | false | If true , use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details. |

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

EXCHANGE DATA FORMAT

Camel will deliver the body as a json formatted `java.lang.String` (see the `mode` option above).

MESSAGE HEADERS

| Header | Description |
|-----------------------------------|---|
| <code>CamelWeatherQuery</code> | The original query URL sent to the Open Weather Map site |
| <code>CamelWeatherLocation</code> | Used by the producer to override the endpoint location and use the location from this header instead. |

SAMPLES

In this sample we find the 7 day weather forecast for Madrid, Spain:

```
from("weather:foo?location=Madrid,Spain&period=7
days").to("jms:queue:weather");
```

To just find the current weather for your current location you can use this:

```
from("weather:foo").to("jms:queue:weather");
```

And to find the weather using the producer we do:

```
from("direct:start")
.to("weather:foo?location=Madrid,Spain");
```

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", String.class);
```

And to get the weather at the current location, then:

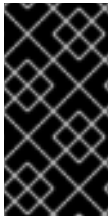
```
String json = template.requestBodyAndHeader("direct:start", "",
"CamelWeatherLocation", "current", String.class);
```

CHAPTER 136. WEBSOCKET

WEBSOCKET COMPONENT

Available as of Camel 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

As Camel 2.10 uses Jetty 7.5.4.v20111024, only the D00 to [D13](#) IETF implementations are available. Camel 2.11 uses Jetty 7.6.7.

URI FORMAT

```
websocket://hostname[:port][/resourceUri][?options]
```

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

COMPONENT OPTIONS

The `WebsocketComponent` can be configured prior to use, to setup host, to act as a websocket server.

| Option | Default | Description |
|----------------------|--------------|---|
| host | 0.0.0.0 | The hostname. |
| port | 9292 | The port number. |
| staticResources | null | Path for static resources such as index.html files etc. If this option has been configured, then a server is started on the given hostname and port, to service the static resources, eg such as an index.html file. If this option has not been configured, then no server is started. |
| sslContextParameters | | Reference to a org.apache.camel.util.jsse.SSLContextParameters in the Registry . This reference overrides any configured <code>SSLContextParameters</code> at the component level. See Using the JSSE Configuration Utility . |
| enableJmx | false | If this option is true, Jetty JMX support will be enabled for this endpoint. |
| sslKeyPassword | null | Consumer only: The password for the keystore when using SSL. |
| sslPassword | null | Consumer only: The password when using SSL. |
| sslKeystore | null | Consumer only: The path to the keystore. |
| minThreads | null | Consumer only: To set a value for minimum number of threads in server thread pool. |
| maxThreads | null | Consumer only: To set a value for maximum number of threads in server thread pool. |

| | | |
|-------------------------|-------------------|---|
| <code>threadPool</code> | <code>null</code> | Consumer only: To use a custom thread pool for the server. |
|-------------------------|-------------------|---|

ENDPOINT OPTIONS

The `WebSocketEndpoint` can be configured prior to use

| Option | Default | Description |
|--------------------------------------|-------------------|---|
| <code>sslContextParametersRef</code> | | Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . This reference overrides any configured <code>SSLContextParameters</code> at the component level. See Using the JSE Configuration Utility. Use the <code>sslContextParameters</code> option instead |
| <code>sslContextParameters</code> | | Camel 2.11.1: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . This reference overrides any configured <code>SSLContextParameters</code> at the component level. See Using the JSE Configuration Utility. |
| <code>sendToAll</code> | <code>null</code> | Producer only: To send to all websocket subscribers. Can be used to configure on endpoint level, instead of having to use the <code>WebSocketConstants.SEND_TO_ALL</code> header on the message. |

| | | |
|-----------------|-------------|--|
| staticResources | null | 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. |
|-----------------|-------------|--|

MESSAGE HEADERS

The websocket component uses 2 headers to indicate to either send messages back to a single/current client, or to all clients.

| Key | Description |
|--|---|
| <code>WebsocketConstants.SEND_TO_ALL</code> | Sends the message to all clients which are currently connected. You can use the sendToAll option on the endpoint instead of using this header. |
| <code>WebsocketConstants.CONNECTION_KEY</code> | Sends the message to the client with the given connection key. |

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.

```
// expose a echo websocket client, that sends back an echo
from("websocket://echo")
    .log(">>> Message received from WebSocket Client : ${body}")
    .transform().simple("${body}${body}")
    // send back to the client, by sending the message to the same
endpoint
    // this is needed as by default messages is InOnly
    // and we will by default send back to the current client using the
provided connection key
    .to("websocket://echo");
```


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");
```

SETTING UP SSL FOR WEBSOCKET COMPONENT

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 cometdComponent = getContext().getComponent("cometds",
CometdComponent.class);
cometdComponent.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);
    }
  };
...
}
```

- [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 137. XMLRPC

XMLRPC COMPONENT

Available as of Camel 2.11

This component provides a dataformat for xml, which allows serialization and deserialization of request messages and response message using Apache XmlRpc's bindary dataformat. You can also invoke the XMLRPC Service through the camel-xmlrpc 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-xmlrpc</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

XMLRPC OVERVIEW

It's a [spec](#) and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet.

It's remote procedure calling using HTTP as the transport and XML as the encoding. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

An example of a typical XML-RPC request would be:

```
<?xml version="1.0"?>
```

```

<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>40</i4></value>
    </param>
  </params>
</methodCall>

```

An example of a typical XML-RPC response would be:

```

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>

```

A typical XML-RPC fault would be:

```

<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>

```

URI FORMAT

```
xmlrpc://serverUri[?options]
```

OPTIONS

| Property | Default | Description |
|------------------------------|--------------|---|
| basicEncoding | null | Sets the encoding for basic authentication, null means UTF-8 is chosen. |
| basicUserName | null | The user name for basic authentication. |
| basicPassword | null | The password for basic authentication. |
| clientConfigurer | null | The reference id of the XmlRpcClient configurer which implement the interface of XmlRpcClientConfigurer to setup the XmlRpcClient as user wants. The value should be start with "#" such as "#myConfigurer" |
| connectionTimeout | 0 | Set the connection timeout in milliseconds, 0 is to disable it |
| contentLengthOptional | false | whether a "Content-Length" header may be omitted. The XML-RPC specification demands, that such a header be present. |
| enabledForExceptions | false | whether the response should contain a "faultCause" element in case of errors. The "faultCause" is an exception, which the server has trapped and written into a byte stream as a serializable object. |
| enabledForExtensions | false | whether extensions are enabled. By default, the client or server is strictly compliant to the XML-RPC specification and extensions are disabled. |

| | | |
|------------------------|--------------|--|
| encoding | null | Sets the requests encoding, null means UTF-8 is chosen. |
| gzipCompressing | false | Whether gzip compression is being used for transmitting the request. |
| gzipRequesting | false | Whether gzip compression is being used for transmitting the request. |
| replyTimeout | 0 | Set the reply timeout in milliseconds, 0 is to disable it. |
| userAgent | null | The http user agent header to set when doing xmlrpc requests |

MESSAGE HEADERS

Camel XmlRpc uses these headers.

| Header | Description |
|------------------------------|--|
| CamelXmlRpcMethodName | The XmlRpc method name which will be use for invoking the XmlRpc server. |

USING THE XMLRPC DATA FORMAT

As the XmlRpc message could be request or response, when you use the `XmlRpcDataFormat`, you need to specify the dataformat is for request or not.

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

```

<!-- we define the xml rpc data formats to be used -->
<dataFormats>
  <xmlrpc id="xmlrpcRequest" request="true"/>
  <xmlrpc id="xmlrpcResponse" request="false"/>
</dataFormats>

<route>
  <from uri="direct:request"/>
  <marshal ref="xmlrpcRequest"/>
  <unmarshal>
    <xmlrpc request="true"/>
  </unmarshal>
  <to uri="mock:request" />
</route>

<route>
  <from uri="direct:response"/>
  <marshal>
    <xmlrpc request="false"/>
  </marshal>
  <unmarshal ref="xmlrpcResponse"/>
  <to uri="mock:response" />
</route>
</camelContext>

```

INVOKE XMLRPC SERVICE FROM CLIENT

To invoke the XmlRpc service, you need to specify the `methodName` on the message header and put the parameters into the message body like below code, then you can get the result message as you want. If the fault message is return, you should get an exception which cause if `XmlRpcException`.

```
String response = template.requestBodyAndHeader(xmlRpcServiceAddress,
new Object[]{"me"}, XmlRpcConstants.METHOD_NAME, "hello", String.class);
```

HOW TO CONFIGURE THE XMLRPCCLIENT WITH JAVA CODE

`camel-xmlrpc` provides a pluggable strategy for configuring the `XmlRpcClient` used by the component, user just to implement the `XmlRpcClientConfigurer` interface and can configure the `XmlRpcClient` as he wants. The `clientConfigure` instance reference can be set through the uri option `clientConfigure`.


```
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public class MyClientConfigurer implements XmlRpcClientConfigurer {

    @Override
    public void configureXmlRpcClient(XmlRpcClient client) {
        // get the configure first
        XmlRpcClientConfigImpl clientConfig =
(XmlRpcClientConfigImpl)client.getClientConfig();
        // change the value of clientConfig
        clientConfig.setEnabledForExtensions(true);
        // set the option on the XmlRpcClient
        client.setMaxThreads(10);
    }
}
```

CHAPTER 138. XML SECURITY COMPONENT

XML SECURITY COMPONENT

Available as of Camel 2.12.0

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.

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>
```

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. Detached XML signature are not supported in the current implementation.

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 also 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="generate_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 -->
    <!-- Further Object elements possible, see option 'properties' below
-->
    </Signature>

```

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 or enveloping XML signature will be set to the body of the out-message.

-

With the verifier endpoint, you can validate a enveloped or enveloping XML signature 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 one camel context.

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.

COMMON SIGNING AND VERIFYING OPTIONS

There are options which can be used for both endpoints, signer and verifier.

| Name | Type | Default | Description |
|-----------------|---|----------------|---|
| uriDereferencer | javax.xml.crypto.URI Dereferencer | null | URI dereferencer. You can specify here your own URI dereferencer, if you want to restrict the dereferencing or have special requirements for dereferencing. |
| baseUri | String | null | Base URI used in the URI dereferencer. Relative URIs are concatenated with the base URI. |

| | | | |
|-------------------------|-------------------------------|---------------|---|
| cryptoContextProperties | Map<String, ? extends Object> | null | <p>Crypto context properties. See javax.xml.crypto.XMLCryptoContext.setProperty(String, Object).</p> <p>The properties can depend on the provider. For example, the JDK provider "XMLDSig" has the property "org.jcp.xml.dsig.validateManifests" for enabling manifest validation. The following properties are set by default to the value Boolean.TRUE for the XML validation: "org.jcp.xml.dsig.validateManifests", "javax.xml.crypto.dsig.cacheReference". If you want to switch these features off you must set the property values to Boolean.FALSE.</p> |
| disallowDoctypeDecl | Boolean | Boolean.TRUE | Indicator whether DTD DOCTYPE declarations shall be disallowed in the incoming XML message. |
| omitXmlDeclaration | Boolean | Boolean.FALSE | Indicator whether the XML declaration header shall be omitted in the output XML message. |
| clearHeaders | Boolean | Boolean.TRUE | Indicator whether the XML signature message headers defined in XmlSignatureConstants shall be deleted at the end of the signer or verifier processing. |

SIGNING OPTIONS

The signer endpoint has the following options.

| Name | Type | Default | Description |
|---------------------|-----------------------------|--|--|
| keyAccessor | KeyAccessor | null | Provides the signing key and the KeyInfo instance. There is an example implementation which uses a keystore, see DefaultKeyAccessor |
| addKeyInfoReference | Boolean | Boolean.TRUE | Indicator whether the KeyInfo element provided by the key accessor should be added to the XML signature. |
| signatureAlgorithm | String | http://www.w3.org/2000/09/xmldsig#rsa-sha1 | signature algorithm consisting of a digest and encryption algorithm. The digest algorithm is used to calculate the digest of the SignedInfo element and the encryption algorithm is used to sign this digest. Which algorithm are supported depends on the JSR 105 provider (see option provider). |

| | | | |
|-----------------|--------|-----------------|--|
| digestAlgorithm | String | see description | Digest algorithm for calculating the digest of the in-message body. If not specified then the digest algorithm of the signature algorithm is used (e.g. http://www.w3.org/2001/04/xmlenc#sha256). Which digest algorithm can be used depends on the JSR 105 provider (see option provider). |
| parentLocalName | String | null | Local name of the parent of the Signature element. The Signature element will be added at the end of the children of the parent. Necessary for enveloped XML signature. If this option is null, then an enveloping XML signature is created. See also option parentNamespace . |
| parentNamespace | String | null | Namespace of the parent of the Signature element. See option parentLocalName |

| | | | |
|--------------------------------|--|-----------------|--|
| canonicalizationMethod | javax.xml.crypto.AlgorithmMethod | C14n | 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<String> inclusiveNamespacePrefixes) to create a canonicalization method. |
| transformMethods | List< javax.xml.crypto.AlgorithmMethod > | see description | 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/xmldsig#enveloped-signature is added at position 0 of the list. Use methods in XmlSignatureHelper to create the transform methods. |
| prefixForXmlSignatureNamespace | String | ds | Prefix for the XML signature namespace. If null is specified or an empty string then no prefix is used for the signature namespace. |

| | | | |
|----------------------|--------|-----------------|--|
| contentReferenceUri | String | see description | The URI of the reference to the signed content (in-message body). If null and we are in the enveloped XML signature case then the URI is set to "". If null and we are in the enveloping XML signature case then the URI is set to "#object_id" (where "object_id" is the value of the option contentObjectId which means that the reference points to the Object element containing the in-message body. You can use this option to reference a specific part in your in-message body if you do not want to sign the complete in-message body. This value can be overwritten by the header "CamelXmlSignatureContentReferenceUri". |
| contentReferenceType | String | null | Value of the type attribute of the content reference. This value can be overwritten by the header "CamelXmlSignatureContentReferenceType". |

| | | | |
|-------------------|--|---------------|--|
| plainText | Boolean | Boolean.FALSE | Indicator whether the in-message body contains plain text. Normally, the signature generator treats the incoming message body as XML. If the message body is plain text, then you must set this option to true . The value can be overwritten by the header "CamelXmlSignature MessageIsPlainText". |
| plainTextEncoding | String | null | Only used when the option plainText is set to true . Then you can specify the encoding of the plain text. If null then UTF-8 is used. The value can be overwritten by the header "CamelXmlSignature MessageIsPlainTextEncoding". |
| properties | XmlSignatureProperties | null | For adding additional References and Objects to the XML signature which contain additional properties, you can provide a bean which implements the XmlSignatureProperties interface. |

VERIFYING OPTIONS

The verifier endpoint has the following options.

| Name | Type | Default | Description |
|-------------------------|---|--|--|
| keySelector | javax.xml.crypto.Key Selector | null | Provides the key for validating the XML signature. There is an example implementation which uses a keystore, see DefaultKeySelector . |
| xmlSignatureChecker | XmlSignatureChecker | null | 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/xmlsig-bestpractices/#check-what-is-signed |
| validationFailedHandler | ValidationFailedHandler | DefaultValidationFailedHandler | 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 . If a reference validation fails, a XmlSignatureInvalidContentHashException is thrown. For more detailed information, see the JavaDoc. |
| | | | |

| | | | |
|----------------------|--------------------------------------|---|--|
| xmlSignature2Message | XmlSignature2Message | DefaultXmlSignature2Message | <p>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</p> |
|----------------------|--------------------------------------|---|--|

| | | | |
|----------------------|--------|-----------|--|
| | | | <p>is a reference with URI="" and transform "http://www.w3.org/2000/09/xmldsig#enveloped-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".</p> |
| outputNodeSearchType | String | "Default" | <p>Determines the type of the search of the output node. See option xmlSignature2Message. The default implementation DefaultXmlSignature2Message supports the three search types "Default", "ElementName", and "XPath".</p> |

| | | | |
|------------------|--------|------|---|
| outputNodeSearch | Object | null | <p>Search value of the output node search. The type depends on the search type. For the default search implementation DefaultXmlSignature2Message the following values can be supplied. If the search type is "Default", then the search value is not used. If the search type is "ElementName", then the search value contains the namespace and local name of the output element. The namespace must be embraced in brackets. If the search type is "XPath", the search value contains an instance of javax.xml.crypto.dsig.spec.XPathFilterParametersSpec which represents an XPath. You can create such an instance via the method XmlSignatureHelper.getXpathFilter(String xpath, Map<String, String> namespaceMap). The XPath determines the output node which can be of type Element, TextNode, or Document.</p> |
|------------------|--------|------|---|

| | | | |
|-------------------------|---------|---------------|---|
| removeSignatureElements | Boolean | Boolean.FALSE | Indicator for removing Signature elements in the output message in the enveloped XML signature case. Used in the XmlSignature2Message instance. The default implementation does use this indicator for the two search types "ElementName" and "XPath". |
| secureValidation | Boolean | Boolean.TRUE | Enables secure validation. If true then secure validation is enabled - see here for more information. |

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>
```

SEE ALSO

-

[Best Practices](#)

CHAPTER 139. XMPP

XMPP COMPONENT

The `xmpp: component` implements an XMPP (Jabber) transport.

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.

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

OPTIONS

| Name | Description |
|----------------------------|--|
| <code>room</code> | 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 <code>superman@jabber.org</code> and want to join the <code>krypton</code> room, then the room URL is <code>krypton@conference.jabber.org</code> . Note the <code>conference</code> part. |
| <code>user</code> | User name (without server name). If not specified, anonymous login will be attempted. |
| <code>password</code> | Password. |
| <code>resource</code> | XMPP resource. The default is <code>Camel</code> . |
| <code>createAccount</code> | If <code>true</code> , an attempt to create an account will be made. Default is <code>false</code> . |
| <code>participant</code> | JID (Jabber ID) of person to receive messages. <code>room</code> parameter has precedence over <code>participant</code> . |

| | |
|--------------------------------|---|
| nickname | Use nickname when joining room. If room is specified and nickname is not, user will be used for the nickname. |
| serviceName | The name of the service you are connecting to. For Google Talk, this would be gmail.com . |
| testConnectionOnStartup | *Camel 2.11* 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 . |
| connectionPollDelay | *Camel 2.11* The amount of time in seconds between polls 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 . |

HEADERS AND SETTING SUBJECT OR LANGUAGE

Apache Camel sets the message IN headers as properties on the XMPP message. You can configure a `HeaderFilterStrategy` 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.

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:

```
// send a message from fromuser@gmail.com to touser@gmail.com
from("direct:start").
    to("xmpp://talk.google.com:5222/touser@gmail.com?
serviceName=gmail.com&user=fromuser&password=secret").
    to("mock:result");
```

CHAPTER 140. XQUERY ENDPOINT

XQUERY

The `xquery:` component allows you to process a message using an [XQuery](#) template. This can be ideal when using [Templating](#) to generate responses for requests.

URI FORMAT

```
xquery:templateName
```

Where `templateName` is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

For example you could use something like this:

```
from("activemq:My.Queue").  
  to("xquery:com/acme/mytransform.xquery");
```

To use an XQuery template to formulate a response to a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use `InOnly`, consume the message, and send it to another destination, you could use the following route:

```
from("activemq:My.Queue").  
  to("xquery:com/acme/mytransform.xquery").  
  to("activemq:Another.Queue");
```

CHAPTER 141. XSLT

XSLT

The `xslt:` component allows you to process a message using an [XSLT](#) template. This can be ideal when using [Templating](#) to generate responses for requests.

URI FORMAT

```
xslt:templateName[?options]
```

Where `templateName` is the classpath-local URI of the template to invoke; or the complete URL of the remote template. Refer to the [Spring Documentation for more detail of the URI syntax](#)

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

Here are some example URIs

| URI | Description |
|--|---|
| <code>xslt:com/acme/mytransform.xml</code> | Refers to the file, <code>com/acme/mytransform.xml</code> , on the classpath. |
| <code>xslt:file:///foo/bar.xml</code> | Refers to the file, <code>/foo/bar.xml</code> . |
| <code>xslt:http://acme.com/cheese/foo.xml</code> | Refers to the remote HTTP resource. |

From Camel 2.9 onwards the [XSLT](#) component is provided directly in the `camel-core`.

OPTIONS

| Name | Default Value | Description |
|------|---------------|-------------|
|------|---------------|-------------|

| | | |
|--------------------------------|-------------|--|
| converter | null | Option to override default XmlConverter . Will lookup for the converter in the Registry. The provided converted must be of type <code>org.apache.camel.converter.jaxp.XmlConverter</code> . |
| transformerFactory | null | Option to override default TransformerFactory . Will lookup for the transformerFactory in the Registry. The provided transformer factory must be of type <code>javax.xml.transform.TransformerFactory</code> . |
| transformerFactoryClass | null | Option to override default TransformerFactory . Will create a TransformerFactoryClass instance and set it to the converter. |
| uriResolver | null | Camel 2.3: Allows you to use a custom <code>javax.xml.transform.URIResolver</code> . Camel will by default use its own implementation <code>org.apache.camel.builder.xml.XsltUriResolver</code> which is capable of loading from classpath. |
| resultHandlerFactory | null | Camel 2.3: Allows you to use a custom <code>org.apache.camel.builder.xml.ResultHandlerFactory</code> which is capable of using custom <code>org.apache.camel.builder.xml.ResultHandler</code> types. |
| failOnNullBody | true | Camel 2.3: Whether or not to throw an exception if the input body is null. |

| | | |
|-----------------------------------|---------------------|---|
| <code>deleteOutputFile</code> | <code>false</code> | Camel 2.6: If you have <code>output=file</code> 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. |
| <code>output</code> | <code>string</code> | Camel 2.3: Option to specify which output type to use. Possible values are: <code>string</code> , <code>bytes</code> , <code>DOM</code> , <code>file</code> . The first three options are all in memory based, where as <code>file</code> is streamed directly to a <code>java.io.File</code> . For <code>file</code> you must specify the filename in the IN header with the key <code>Exchange.XSLT_FILE_NAME</code> which is also <code>CamelXsltFileName</code> . Also any paths leading to the filename must be created beforehand, otherwise an exception is thrown at runtime. |
| <code>contentCache</code> | <code>true</code> | Camel 2.6: Cache for the resource content (the stylesheet file) when it is loaded. If set to <code>false</code> Camel will reload the stylesheet file on each message processing. This is good for development. |
| <code>allowStAX</code> | <code>false</code> | Camel 2.8.3/2.9: Whether to allow using StAX as the <code>javax.xml.transform.Source</code> . |
| <code>transformerCacheSize</code> | <code>0</code> | Camel 2.9.3/2.10.1: The number of <code>javax.xml.transform.Transformer</code> object that are cached for reuse to avoid calls to <code>Template.newTransformer()</code> . |

| | | |
|-------|-------|---|
| saxon | false | Camel 2.11: Whether to use Saxon as the transformerFactoryClass . If enabled then the class net.sf.saxon.Transforme rFactoryImpl . You would need to add Saxon to the classpath. |
|-------|-------|---|

USING XSLT ENDPOINTS

For example you could use something like

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xml");
```

To use an XSLT 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 the following route:

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xml").
  to("activemq:Another.Queue");
```

GETTING PARAMETERS INTO THE XSLT TO WORK WITH

By default, all headers are added as parameters which are available in the XSLT. To do this you will need to declare the parameter so it is then *useable*.

```
<setHeader headerName="myParam"><constant>42</constant></setHeader>
<to uri="xslt:MyTransform.xml"/>
```

And the XSLT just needs to declare it at the top level for it to be available:

```
<xsl: ..... >
  <xsl:param name="myParam"/>
```

```
<xsl:template ...>
```

SPRING XML VERSIONS

To use the above examples in Spring XML you would use something like

```
<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>
```

There is a [test case](#) along with [its Spring XML](#) if you want a concrete example.

USING XSL:INCLUDE

Camel provides its own implementation of `URIResolver` which allows Camel to load included files from the classpath and more intelligent than before.

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

Will now be located relative from the starting endpoint, which for example could be:

```
.to("xslt:org/apache/camel/component/xslt/staff_include_relative.xsl")
```

Which means Camel will locate the file in the classpath as `org/apache/camel/component/xslt/staff_template.xsl`. This allows you to use xsl include and have xsl files located in the same folder such as we do in the example `org/apache/camel/component/xslt`.

You can use the following two prefixes `classpath:` or `file:` to instruct Camel to look either in classpath or file system. If you omit the prefix then

Camel uses the prefix from the endpoint configuration. If that neither has one, then classpath is assumed.

You can also refer back in the paths such as

```
<xsl:include href="../../staff_other_template.xml"/>
```

Which then will resolve the xml file under org/apache/camel/component.

USING XSL:INCLUDE AND DEFAULT PREFIX

When using xsl:include such as:

```
<xsl:include href="staff_template.xml"/>
```

Then in Camel 2.10.3 and older, then Camel will use "classpath:" as the default prefix, and load the resource from the classpath. This works for most cases, but if you configure the starting resource to load from file,

```
.to("xslt:file:etc/xslt/staff_include_relative.xml")
```

.. then you would have to prefix all your includes with "file:" as well.

```
<xsl:include href="file:staff_template.xml"/>
```

From Camel 2.10.4 onwards we have made this easier as Camel will use the prefix from the endpoint configuration as the default prefix. So from Camel 2.10.4 onwards you can do:

```
<xsl:include href="staff_template.xml"/>
```

Which will load the staff_template.xml resource from the file system, as the endpoint was configured with "file:" as prefix. You can still though explicit configure a prefix, and then mix and match. And have both file and

classpath loading. But that would be unusual, as most people either use file or classpath based resources.

DYNAMIC STYLESHEETS

Available as of Camel 2.9 Camel provides the `CamelXsltResourceUri` header which you can use to define a stylesheet to use instead of what is configured on the endpoint URI. This allows you to provide a dynamic stylesheet at runtime.

CHAPTER 142. YAMMER

YAMMER

Available as of Camel 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>
```

URI FORMAT

```
yammer:[function]?[options]
```

YAMMERCOMPONENT

The `yammer` component can be configured with the Yammer account settings which are mandatory to configure before using. You can also configure these options directly in the endpoint.

| Option | Description |
|-----------------------------|---------------------|
| <code>consumerKey</code> | The consumer key |
| <code>consumerSecret</code> | The consumer secret |
| <code>accessToken</code> | The access token |

CONSUMING MESSAGES

The `camel-yammer` component provides several endpoints for consuming messages:

| URI | Description |
|---------------------------------------|---|
| <code>yammer:messages?options</code> | 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. |
| <code>yammer:my_feed?options</code> | The user's feed, based on the selection they have made between "Following" and "Top" conversations. |
| <code>yammer:algo?options</code> | 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. |
| <code>yammer:following?options</code> | The "Following" feed which is conversations involving people, groups and topics that the user is following. |
| <code>yammer:sent?options</code> | 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 |

URI OPTIONS FOR CONSUMING MESSAGES

| Name | Default Value | Description |
|----------------|---------------|--|
| usejson | false | Set to true if you want to use raw JSON rather than converting to POJOs. |
| delay | 5000 | in milliseconds |
| consumerKey | null | Consumer Key. Can also be configured on the YammerComponent level instead. |
| consumerSecret | null | Consumer Secret. Can also be configured on the YammerComponent level instead. |
| accessToken | null | Access Token. Can also be configured on the YammerComponent level instead. |
| limit | -1 | Return only the specified number of messages. Works for threaded=true and threaded=extended. |

| | | |
|-----------|-------------|---|
| threaded | null | <p>threaded=true will only return the first message in each thread. This parameter is intended for apps which display message threads collapsed.</p> <p>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.</p> |
| olderThan | -1 | <p>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.</p> |
| newerThan | -1 | <p>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.</p> |

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 & producing endpoints, a Messages object is returned. Take for example a route like:

```
from("yammer:messages?
consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAc
cessToken").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.

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=aAc
cessToken").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());
```

RETRIEVING USER RELATIONSHIPS

The `camel-yammer` component can retrieve user relationships:

```
yammer:relationships?[options]
```

URI OPTIONS FOR RETRIEVING RELATIONSHIPS

| Name | Default Value | Description |
|-----------------------------|---------------------|--|
| <code>usejson</code> | false | Set to true if you want to use raw JSON rather than converting to POJOs. |
| <code>delay</code> | 5000 | in milliseconds |
| <code>consumerKey</code> | null | Consumer Key. Can also be configured on the YammerComponent level instead. |
| <code>consumerSecret</code> | null | Consumer Secret. Can also be configured on the YammerComponent level instead. |
| <code>accessToken</code> | null | Access Token. Can also be configured on the YammerComponent level instead. |
| <code>userId</code> | current user | To view the relationships for a user other than the current user. |

RETRIEVING USERS

The `camel-yammer` component provides several endpoints for retrieving users:

| URI | Description |
|-------------------------------------|--|
| <code>yammer:users?options</code> | Retrieve users in the current user's Yammer network. |
| <code>yammer:current?options</code> | View data about the current user. |

URI OPTIONS FOR RETRIEVING USERS

| Name | Default Value | Description |
|-----------------------------|--------------------|--|
| <code>usejson</code> | <code>false</code> | Set to true if you want to use raw JSON rather than converting to POJOs. |
| <code>delay</code> | <code>5000</code> | in milliseconds |
| <code>consumerKey</code> | <code>null</code> | Consumer Key. Can also be configured on the YammerComponent level instead. |
| <code>consumerSecret</code> | <code>null</code> | Consumer Secret. Can also be configured on the YammerComponent level instead. |

| | | |
|-------------|-------------|---|
| accessToken | null | Access Token. Can also be configured on the YammerComponent level instead. |
|-------------|-------------|---|

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=aAc
cessToken").to("mock:result");
```

This will go out and fetch the current user's User object and set it as the Camel message body.

CHAPTER 143. ZOOKEEPER

ZOOKEEPER

Available as of Camel 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.
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>
```

URI FORMAT



```
zookeeper://zookeeper-server[:port][/path][?options]
```

The path from the uri specifies the node in the ZooKeeper server (aka znode) that will be the target of the endpoint.

OPTIONS

| Name | Default Value | Description |
|--------------------------|---------------|--|
| sessionId | null | The session id used to identify a connection to the cluster |
| password | null | The password to use when making a connection |
| listChildren | false | whether the children of the node should be listed |
| repeat | false | should changes to the znode be 'watched' and repeatedly processed. |
| backoff | 5000 | the time interval to backoff for after an error before retrying. |
| timeout | 5000 | the time interval to wait on connection before timing out. |
| create | false | should the endpoint create the node if it does not currently exist. |
| createMode | EPHEMERAL | the create mode that should be used for the newly created node (see below). |
| sendEmptyMessageOnDelete | true | Camel 2.10: upon the delete of a znode, should an empty message be send to the consumer |

USE CASES

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");
```

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.

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'`

```
Exchange e = createExchangeWithBody(testPayload);
template.sendBodyAndHeader("direct:write-to-znode", e,
"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

- EPOCHEMERAL_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'

```
Exchange e = createExchangeWithBody(testPayload);
template.sendBodyAndHeader("direct:create-and-write-to-persistent-znode",
e, "CamelZooKeeperCreateMode", "PERSISTENT");
```

ZOOKEEPER ENABLED ROUTE POLICY.

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");
```