



Red Hat JBoss Fuse 6.0

Implementing Enterprise Integration Patterns

Using Apache Camel's to connect applications

Red Hat JBoss Fuse 6.0 Implementing Enterprise Integration Patterns

Using Apache Camel's to connect applications

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

This guide describes how to build routes using Apache Camel. It covers the basic building blocks and EIP components.

Table of Contents

CHAPTER 1. BUILDING BLOCKS FOR ROUTE DEFINITIONS	4
1.1. IMPLEMENTING A ROUTEBUILDER CLASS	4
1.2. BASIC JAVA DSL SYNTAX	5
1.3. ROUTER SCHEMA IN A SPRING XML FILE	8
1.4. ENDPOINTS	10
1.5. PROCESSORS	14
CHAPTER 2. BASIC PRINCIPLES OF ROUTE BUILDING	23
2.1. PIPELINE PROCESSING	23
2.2. MULTIPLE INPUTS	26
2.3. EXCEPTION HANDLING	29
2.4. BEAN INTEGRATION	44
2.5. ASPECT ORIENTED PROGRAMMING	54
2.6. TRANSFORMING MESSAGE CONTENT	56
2.7. PROPERTY PLACEHOLDERS	63
2.8. THREADING MODEL	72
2.9. CONTROLLING START-UP AND SHUTDOWN OF ROUTES	79
2.10. SCHEDULED ROUTE POLICY	83
2.11. JMX NAMING	92
CHAPTER 3. INTRODUCING ENTERPRISE INTEGRATION PATTERNS	95
3.1. OVERVIEW OF THE PATTERNS	95
CHAPTER 4. MESSAGING SYSTEMS	102
4.1. MESSAGE	102
4.2. MESSAGE CHANNEL	103
4.3. MESSAGE ENDPOINT	105
4.4. PIPES AND FILTERS	106
4.5. MESSAGE ROUTER	108
4.6. MESSAGE TRANSLATOR	110
CHAPTER 5. MESSAGING CHANNELS	112
5.1. POINT-TO-POINT CHANNEL	112
5.2. PUBLISH-SUBSCRIBE CHANNEL	113
5.3. DEAD LETTER CHANNEL	115
5.4. GUARANTEED DELIVERY	123
5.5. MESSAGE BUS	125
CHAPTER 6. MESSAGE CONSTRUCTION	127
6.1. CORRELATION IDENTIFIER	127
6.2. EVENT MESSAGE	127
6.3. RETURN ADDRESS	129
CHAPTER 7. MESSAGE ROUTING	131
7.1. CONTENT-BASED ROUTER	131
7.2. MESSAGE FILTER	132
7.3. RECIPIENT LIST	134
7.4. SPLITTER	142
7.5. AGGREGATOR	151
7.6. RESEQUENCER	168
7.7. ROUTING SLIP	171
7.8. THROTTLER	173
7.9. DELAYER	175

7.10. LOAD BALANCER	178
7.11. MULTICAST	185
7.12. COMPOSED MESSAGE PROCESSOR	192
7.13. SCATTER-GATHER	194
7.14. LOOP	198
7.15. SAMPLING	200
7.16. DYNAMIC ROUTER	202
CHAPTER 8. MESSAGE TRANSFORMATION	205
8.1. CONTENT ENRICHER	205
8.2. CONTENT FILTER	209
8.3. NORMALIZER	210
8.4. CLAIM CHECK	212
8.5. SORT	214
8.6. VALIDATE	215
CHAPTER 9. MESSAGING ENDPOINTS	217
9.1. MESSAGING MAPPER	217
9.2. EVENT DRIVEN CONSUMER	218
9.3. POLLING CONSUMER	218
9.4. COMPETING CONSUMERS	219
9.5. MESSAGE DISPATCHER	221
9.6. SELECTIVE CONSUMER	223
9.7. DURABLE SUBSCRIBER	225
9.8. IDEMPOTENT CONSUMER	228
9.9. TRANSACTIONAL CLIENT	233
9.10. MESSAGING GATEWAY	234
9.11. SERVICE ACTIVATOR	235
CHAPTER 10. SYSTEM MANAGEMENT	238
10.1. DETOUR	238
10.2. LOGEIP	239
10.3. WIRE TAP	240
APPENDIX A. MIGRATING FROM SERVICEMIX EIP	246
A.1. MIGRATING ENDPOINTS	246
A.2. COMMON ELEMENTS	248
A.3. SERVICEMIX EIP PATTERNS	249
A.4. CONTENT-BASED ROUTER	250
A.5. CONTENT ENRICHER	252
A.6. MESSAGE FILTER	253
A.7. PIPELINE	255
A.8. RESEQUENCER	256
A.9. STATIC RECIPIENT LIST	258
A.10. STATIC ROUTING SLIP	259
A.11. WIRE TAP	260
A.12. XPATH SPLITTER	261
INDEX	263

CHAPTER 1. BUILDING BLOCKS FOR ROUTE DEFINITIONS

Abstract

Apache Camel supports two alternative *Domain Specific Languages* (DSL) for defining routes: a Java DSL and a Spring XML DSL. The basic building blocks for defining routes are *endpoints* and *processors*, where the behavior of a processor is typically modified by *expressions* or logical *predicates*. Apache Camel enables you to define expressions and predicates using a variety of different languages.

1.1. IMPLEMENTING A ROUTEBUILDER CLASS

Overview

To use the *Domain Specific Language* (DSL), you extend the **RouteBuilder** class and override its **configure()** method (where you define your routing rules).

You can define as many **RouteBuilder** classes as necessary. Each class is instantiated once and is registered with the **CamelContext** object. Normally, the lifecycle of each **RouteBuilder** object is managed automatically by the container in which you deploy the router.

RouteBuilder classes

As a router developer, your core task is to implement one or more **RouteBuilder** classes. There are two alternative **RouteBuilder** classes that you can inherit from:

- **org.apache.camel.builder.RouteBuilder**—this is the generic **RouteBuilder** base class that is suitable for deploying into *any* container type. It is provided in the **camel-core** artifact.
- **org.apache.camel.spring.SpringRouteBuilder**—this base class is specially adapted to the Spring container. In particular, it provides extra support for the following Spring specific features: looking up beans in the Spring registry (using the **beanRef()** Java DSL command) and transactions (see the *Transactions Guide* for details). It is provided in the **camel-spring** artifact.

The **RouteBuilder** class defines methods used to initiate your routing rules (for example, **from()**, **intercept()**, and **exception()**).

Implementing a RouteBuilder

[Example 1.1, “Implementation of a RouteBuilder Class”](#) shows a minimal **RouteBuilder** implementation. The **configure()** method body contains a routing rule; each rule is a single Java statement.

Example 1.1. Implementation of a RouteBuilder Class

```
import org.apache.camel.builder.RouteBuilder;

public class MyRouteBuilder extends RouteBuilder {

    public void configure() {
        // Define routing rules here:
    }
}
```



```

    from("file:src/data?noop=true").to("file:target/messages");

    // More rules can be included, in you like.
    // ...
}
}

```

The form of the rule **from(URL1).to(URL2)** instructs the router to read files from the directory **src/data** and send them to the directory **target/messages**. The option **?noop=true** instructs the router to retain (not delete) the source files in the **src/data** directory.

1.2. BASIC JAVA DSL SYNTAX

What is a DSL?

A Domain Specific Language (DSL) is a mini-language designed for a special purpose. A DSL does not have to be logically complete but needs enough expressive power to describe problems adequately in the chosen domain. Typically, a DSL does not require a dedicated parser, interpreter, or compiler. A DSL can piggyback on top of an existing object-oriented host language, provided DSL constructs map cleanly to constructs in the host language API.

Consider the following sequence of commands in a hypothetical DSL:

```

command01;
command02;
command03;

```

You can map these commands to Java method invocations, as follows:

```

command01().command02().command03()

```

You can even map blocks to Java method invocations. For example:

```

command01().startBlock().command02().command03().endBlock()

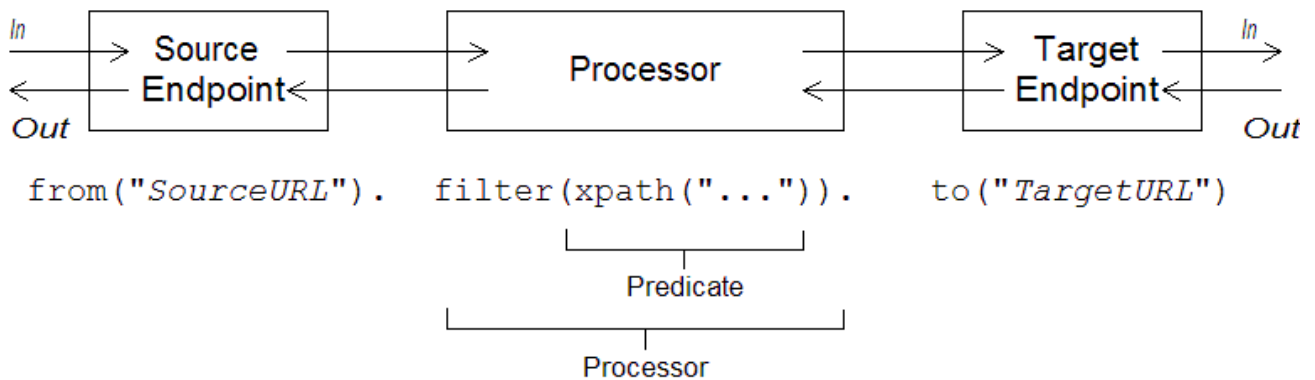
```

The DSL syntax is implicitly defined by the data types of the host language API. For example, the return type of a Java method determines which methods you can legally invoke next (equivalent to the next command in the DSL).

Router rule syntax

Apache Camel defines a *router DSL* for defining routing rules. You can use this DSL to define rules in the body of a **RouteBuilder.configure()** implementation. [Figure 1.1, “Local Routing Rules”](#) shows an overview of the basic syntax for defining local routing rules.

Figure 1.1. Local Routing Rules



A local rule always starts with a `from("EndpointURL")` method, which specifies the source of messages (*consumer endpoint*) for the routing rule. You can then add an arbitrarily long chain of processors to the rule (for example, `filter()`). You typically finish off the rule with a `to("EndpointURL")` method, which specifies the target (*producer endpoint*) for the messages that pass through the rule. However, it is not always necessary to end a rule with `to()`. There are alternative ways of specifying the message target in a rule.



NOTE

You can also define a global routing rule, by starting the rule with a special processor type (such as `intercept()`, `exception()`, or `errorHandler()`). Global rules are outside the scope of this guide.

Consumers and producers

A local rule always starts by defining a consumer endpoint, using `from("EndpointURL")`, and typically (but not always) ends by defining a producer endpoint, using `to("EndpointURL")`. The endpoint URLs, *EndpointURL*, can use any of the components configured at deploy time. For example, you could use a file endpoint, `file:MyMessageDirectory`, an Apache CXF endpoint, `cx:MyServiceName`, or an Apache ActiveMQ endpoint, `activemq:queue:MyQName`. For a complete list of component types, see ["EIP Component Reference"](#).

Exchanges

An *exchange* object consists of a message, augmented by metadata. Exchanges are of central importance in Apache Camel, because the exchange is the standard form in which messages are propagated through routing rules. The main constituents of an exchange are, as follows:

- *In* message—is the current message encapsulated by the exchange. As the exchange progresses through a route, this message may be modified. So the *In* message at the start of a route is typically *not* the same as the *In* message at the end of the route. The `org.apache.camel.Message` type provides a generic model of a message, with the following parts:
 - Body.
 - Headers.
 - Attachments.

It is important to realize that this is a *generic* model of a message. Apache Camel supports a

large variety of protocols and endpoint types. Hence, it is *not* possible to standardize the format of the message body or the message headers. For example, the body of a JMS message would have a completely different format to the body of a HTTP message or a Web services message. For this reason, the body and the headers are declared to be of **Object** type. The original content of the body and the headers is then determined by the endpoint that created the exchange instance (that is, the endpoint appearing in the **from()** command).

- *Out* message—is a temporary holding area for a reply message or for a transformed message. Certain processing nodes (in particular, the **to()** command) can modify the current message by treating the *In* message as a request, sending it to a producer endpoint, and then receiving a reply from that endpoint. The reply message is then inserted into the *Out* message slot in the exchange.

Normally, if an *Out* message has been set by the current node, Apache Camel modifies the exchange as follows before passing it to the next node in the route: the old *In* message is discarded and the *Out* message is moved to the *In* message slot. Thus, the reply becomes the new current message. For a more detailed discussion of how Apache Camel connects nodes together in a route, see [Section 2.1, “Pipeline Processing”](#).

There is one special case where an *Out* message is treated differently, however. If the consumer endpoint at the start of a route is expecting a reply message, the *Out* message at the very end of the route is taken to be the consumer endpoint's reply message (and, what is more, in this case the final node *must* create an *Out* message or the consumer endpoint would hang) .

- Message exchange pattern (MEP)—affects the interaction between the exchange and endpoints in the route, as follows:
 - *Consumer endpoint*—the consumer endpoint that creates the original exchange sets the initial value of the MEP. The initial value indicates whether the consumer endpoint expects to receive a reply (for example, the *InOut* MEP) or not (for example, the *InOnly* MEP).
 - *Producer endpoints*—the MEP affects the producer endpoints that the exchange encounters along the route (for example, when an exchange passes through a **to()** node). For example, if the current MEP is *InOnly*, a **to()** node would not expect to receive a reply from the endpoint. Sometimes you need to change the current MEP in order to customize the exchange's interaction with a producer endpoint. For more details, see [Section 1.4, “Endpoints”](#).
- Exchange properties—a list of named properties containing metadata for the current message.

Message exchange patterns

Using an **Exchange** object makes it easy to generalize message processing to different *message exchange patterns*. For example, an asynchronous protocol might define an MEP that consists of a single message that flows from the consumer endpoint to the producer endpoint (an *InOnly* MEP). An RPC protocol, on the other hand, might define an MEP that consists of a request message and a reply message (an *InOut* MEP). Currently, Apache Camel supports the following MEPs:

- **InOnly**
- **RobustInOnly**
- **InOut**
- **InOptionalOut**

- `OutOnly`
- `RobustOutOnly`
- `OutIn`
- `OutOptionalIn`

Where these message exchange patterns are represented by constants in the enumeration type, `org.apache.camel.ExchangePattern`.

Grouped exchanges

Sometimes it is useful to have a single exchange that encapsulates multiple exchange instances. For this purpose, you can use a *grouped exchange*. A grouped exchange is essentially an exchange instance that contains a `java.util.List` of `Exchange` objects stored in the `Exchange.GROUPED_EXCHANGE` exchange property. For an example of how to use grouped exchanges, see [Section 7.5, “Aggregator”](#).

Processors

A *processor* is a node in a route that can access and modify the stream of exchanges passing through the route. Processors can take *expression* or *predicate* arguments, that modify their behavior. For example, the rule shown in [Figure 1.1, “Local Routing Rules”](#) includes a `filter()` processor that takes an `xpath()` predicate as its argument.

Expressions and predicates

Expressions (evaluating to strings or other data types) and predicates (evaluating to true or false) occur frequently as arguments to the built-in processor types. For example, the following filter rule propagates In messages, only if the `foo` header is equal to the value `bar`:

```
from("seda:a").filter(header("foo").isEqualTo("bar")).to("seda:b");
```

Where the filter is qualified by the predicate, `header("foo").isEqualTo("bar")`. To construct more sophisticated predicates and expressions, based on the message content, you can use one of the expression and predicate languages (see [Expression and Predicate Languages](#)).

1.3. ROUTER SCHEMA IN A SPRING XML FILE

Namespace

The router schema—which defines the XML DSL—belongs to the following XML schema namespace:

```
http://camel.apache.org/schema/spring
```

Specifying the schema location

The location of the router schema is normally specified to be `http://camel.apache.org/schema/spring/camel-spring.xsd`, which references the latest version of the schema on the Apache Web site. For example, the root `beans` element of an Apache

Camel Spring file is normally configured as shown in [Example 1.2, “Specifying the Router Schema Location”](#).

Example 1.2. Specifying the Router Schema Location

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:camel="http://camel.apache.org/schema/spring"
      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 id="camel"
    xmlns="http://camel.apache.org/schema/spring">
    <!-- Define your routing rules here -->
  </camelContext>
</beans>
```

Runtime schema location

At run time, Apache Camel does *not* download the router schema from schema location specified in the Spring file. Instead, Apache Camel automatically picks up a copy of the schema from the root directory of the **camel-spring** JAR file. This ensures that the version of the schema used to parse the Spring file always matches the current runtime version. This is important, because the latest version of the schema posted up on the Apache Web site might not match the version of the runtime you are currently using.

Using an XML editor

Generally, it is recommended that you edit your Spring files using a full-feature XML editor. An XML editor's auto-completion features make it much easier to author XML that complies with the router schema and the editor can warn you instantly, if the XML is badly-formed.

XML editors generally *do* rely on downloading the schema from the location that you specify in the **xsi:schemaLocation** attribute. In order to be sure you are using the correct schema version whilst editing, it is usually a good idea to select a specific version of the **camel-spring.xsd** file. For example, to edit a Spring file for the 2.3 version of Apache Camel, you could modify the beans element as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:camel="http://camel.apache.org/schema/spring"
      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-2.3.0.xsd">
  ...
```

Change back to the default, **camel-spring.xsd**, when you are finished editing. To see which schema versions are currently available for download, navigate to the Web page, <http://camel.apache.org/schema/spring>.

1.4. ENDPOINTS

Overview

Apache Camel endpoints are the sources and sinks of messages in a route. An endpoint is a very general sort of building block: the only requirement it must satisfy is that it acts either as a source of messages (a consumer endpoint) or as a sink of messages (a producer endpoint). Hence, there are a great variety of different endpoint types supported in Apache Camel, ranging from protocol supporting endpoints, such as HTTP, to simple timer endpoints, such as Quartz, that generate dummy messages at regular time intervals. One of the major strengths of Apache Camel is that it is relatively easy to add a custom component that implements a new endpoint type.

Endpoint URIs

Endpoints are identified by *endpoint URIs*, which have the following general form:

```
scheme:contextPath[?queryOptions]
```

The URI *scheme* identifies a protocol, such as **http**, and the *contextPath* provides URI details that are interpreted by the protocol. In addition, most schemes allow you to define query options, *queryOptions*, which are specified in the following format:

```
?option01=value01&option02=value02&...
```

For example, the following HTTP URI can be used to connect to the Google search engine page:

```
http://www.google.com
```

The following File URI can be used to read all of the files appearing under the **C:\temp\src\data** directory:

```
file://C:/temp/src/data
```

Not every *scheme* represents a protocol. Sometimes a *scheme* just provides access to a useful utility, such as a timer. For example, the following Timer endpoint URI generates an exchange every second (=1000 milliseconds). You could use this to schedule activity in a route.

```
timer://tickTock?period=1000
```

Apache Camel components

Each URI *scheme* maps to a *Apache Camel component*, where a Apache Camel component is essentially an endpoint factory. In other words, to use a particular type of endpoint, you must deploy the corresponding Apache Camel component in your runtime container. For example, to use JMS endpoints, you would deploy the JMS component in your container.

Apache Camel provides a large variety of different components that enable you to integrate your application with various transport protocols and third-party products. For example, some of the more commonly used components are: File, JMS, CXF (Web services), HTTP, Jetty, Direct, and Mock. For the full list of supported components, see the [Apache Camel component documentation](#).

Most of the Apache Camel components are packaged separately to the Camel core. If you use Maven to

build your application, you can easily add a component (and its third-party dependencies) to your application simply by adding a dependency on the relevant component artifact. For example, to include the HTTP component, you would add the following Maven dependency to your project POM file:

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.10.0.redhat-60024</camel-version>
  ...
</properties>

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

The following components are built-in to the Camel core (in the **camel-core** artifact), so they are always available:

- Bean
- Browse
- Dataset
- Direct
- File
- Log
- Mock
- Properties
- Ref
- SEDA
- Timer
- VM

Consumer endpoints

A *consumer endpoint* is an endpoint that appears at the *start* of a route (that is, in a **from()** DSL command). In other words, the consumer endpoint is responsible for initiating processing in a route: it creates a new exchange instance (typically, based on some message that it has received or obtained), and provides a thread to process the exchange in the rest of the route.

For example, the following JMS consumer endpoint pulls messages off the **payments** queue and processes them in the route:

```
from("jms:queue:payments")
  .process(SomeProcessor)
  .to("TargetURI");
```

Or equivalently, in Spring XML:

```
<camelContext id="CamelContextID"
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:queue:payments"/>
    <process ref="someProcessorId"/>
    <to uri="TargetURI"/>
  </route>
</camelContext>
```

Some components are *consumer only*—that is, they can only be used to define consumer endpoints. For example, the Quartz component is used exclusively to define consumer endpoints. The following Quartz endpoint generates an event every second (1000 milliseconds):

```
from("quartz://secondTimer?trigger.repeatInterval=1000")
  .process(SomeProcessor)
  .to("TargetURI");
```

If you like, you can specify the endpoint URI as a formatted string, using the **fromF()** Java DSL command. For example, to substitute the username and password into the URI for an FTP endpoint, you could write the route in Java, as follows:

```
fromF("ftp:%s@fusesource.com?password=%s", username, password)
  .process(SomeProcessor)
  .to("TargetURI");
```

Where the first occurrence of **%s** is replaced by the value of the **username** string and the second occurrence of **%s** is replaced by the **password** string. This string formatting mechanism is implemented by **String.format()** and is similar to the formatting provided by the C **printf()** function. For details, see java.util.Formatter.

Producer endpoints

A *producer endpoint* is an endpoint that appears in the *middle* or at the *end* of a route (for example, in a **to()** DSL command). In other words, the producer endpoint receives an existing exchange object and sends the contents of the exchange to the specified endpoint.

For example, the following JMS producer endpoint pushes the contents of the current exchange onto the specified JMS queue:

```
from("SourceURI")
  .process(SomeProcessor)
  .to("jms:queue:orderForms");
```

Or equivalently in Spring XML:

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



```

<route>
  <from uri="SourceURI"/>
  <process ref="someProcessorId"/>
  <to uri="jms:queue:orderForms"/>
</route>
</camelContext>

```

Some components are *producer only*—that is, they can only be used to define producer endpoints. For example, the HTTP endpoint is used exclusively to define producer endpoints.

```

from("SourceURI")
  .process(SomeProcessor)
  .to("http://www.google.com/search?hl=en&q=camel+router");

```

If you like, you can specify the endpoint URI as a formatted string, using the `toF()` Java DSL command. For example, to substitute a custom Google query into the HTTP URI, you could write the route in Java, as follows:

```

from("SourceURI")
  .process(SomeProcessor)
  .toF("http://www.google.com/search?hl=en&q=%s", myGoogleQuery);

```

Where the occurrence of `%s` is replaced by your custom query string, `myGoogleQuery`. For details, see [java.util.Formatter](#).

Specifying time periods in a URI

Many of the Apache Camel components have options whose value is a time period (for example, for specifying timeout values and so on). By default, such time period options are normally specified as a pure number, which is interpreted as a millisecond time period. But Apache Camel also supports a more readable syntax for time periods, which enables you to express the period in hours, minutes, and seconds. Formally, the human-readable time period is a string that conforms to the following syntax:

```
[NHour(h|hour)][NMin(m|minute)][NSec(s|second)]
```

Where each term in square brackets, `[]`, is optional and the notation, `(A|B)`, indicates that `A` and `B` are alternatives.

For example, you can configure `timer` endpoint with a 45 minute period as follows:

```

from("timer:foo?period=45m")
  .to("log:foo");

```

You can also use arbitrary combinations of the hour, minute, and second units, as follows:

```

from("timer:foo?period=1h15m")
  .to("log:foo");
from("timer:bar?period=2h30s")
  .to("log:bar");
from("timer:bar?period=3h45m58s")
  .to("log:bar");

```

1.5. PROCESSORS

Overview

To enable the router to do something more interesting than simply connecting a consumer endpoint to a producer endpoint, you can add *processors* to your route. A processor is a command you can insert into a routing rule to perform arbitrary processing of messages that flow through the rule. Apache Camel provides a wide variety of different processors, as shown in [Table 1.1, “Apache Camel Processors”](#).

Table 1.1. Apache Camel Processors

Java DSL	XML DSL	Description
<code>aggregate()</code>	<code>aggregate</code>	Aggregator EIP : Creates an aggregator, which combines multiple incoming exchanges into a single exchange.
<code>aop()</code>	<code>aop</code>	Use Aspect Oriented Programming (AOP) to do work before and after a specified sub-route. See Section 2.5, “Aspect Oriented Programming” .
<code>bean()</code> , <code>beanRef()</code>	<code>bean</code>	Process the current exchange by invoking a method on a Java object (or bean). See Section 2.4, “Bean Integration” .
<code>choice()</code>	<code>choice</code>	Content Based Router EIP : Selects a particular sub-route based on the exchange content, using when and otherwise clauses.
<code>convertBodyTo()</code>	<code>convertBodyTo</code>	Converts the <i>In</i> message body to the specified type.
<code>delay()</code>	<code>delay</code>	Delayer EIP : Delays the propagation of the exchange to the latter part of the route.
<code>doTry()</code>	<code>doTry</code>	Creates a try/catch block for handling exceptions, using doCatch , doFinally , and end clauses.
<code>end()</code>	<i>N/A</i>	Ends the current command block.
<code>enrich()</code> , <code>enrichRef()</code>	<code>enrich</code>	Content Enricher EIP : Combines the current exchange with data requested from a specified <i>producer</i> endpoint URI.

Java DSL	XML DSL	Description
<code>filter()</code>	<code>filter</code>	Message Filter EIP : Uses a predicate expression to filter incoming exchanges.
<code>idempotentConsumer()</code>	<code>idempotentConsumer</code>	Idempotent Consumer EIP : Implements a strategy to suppress duplicate messages.
<code>inheritErrorHandler()</code>	<code>@inheritErrorHandler</code>	Boolean option that can be used to disable the inherited error handler on a particular route node (defined as a sub-clause in the Java DSL and as an attribute in the XML DSL).
<code>inOnly()</code>	<code>inOnly</code>	Either sets the current exchange's MEP to <i>InOnly</i> (if no arguments) or sends the exchange as an <i>InOnly</i> to the specified endpoint(s).
<code>inOut()</code>	<code>inOut</code>	Either sets the current exchange's MEP to <i>InOut</i> (if no arguments) or sends the exchange as an <i>InOut</i> to the specified endpoint(s).
<code>loadBalance()</code>	<code>loadBalance</code>	Load Balancer EIP : Implements load balancing over a collection of endpoints.
<code>log()</code>	<code>log</code>	Logs a message to the console.
<code>loop()</code>	<code>loop</code>	Loop EIP : Repeatedly resends each exchange to the latter part of the route.
<code>markRollbackOnly()</code>	<code>@markRollbackOnly</code>	<i>(Transactions)</i> Marks the current transaction for rollback only (no exception is raised). In the XML DSL, this option is set as a boolean attribute on the rollback element. See " EIP Transaction Guide ".

Java DSL	XML DSL	Description
<code>markRollbackOnlyLast()</code>	<code>@markRollbackOnlyLast</code>	<i>(Transactions)</i> If one or more transactions have previously been associated with this thread and then suspended, this command marks the latest transaction for rollback only (no exception is raised). In the XML DSL, this option is set as a boolean attribute on the rollback element. See "EIP Transaction Guide" .
<code>marshal()</code>	<code>marshal</code>	Transforms into a low-level or binary format using the specified data format, in preparation for sending over a particular transport protocol. See the section called "Marshalling and unmarshalling" .
<code>multicast()</code>	<code>multicast</code>	Multicast EIP : Multicasts the current exchange to multiple destinations, where each destination gets its own copy of the exchange.
<code>onCompletion()</code>	<code>onCompletion</code>	Defines a sub-route (terminated by end() in the Java DSL) that gets executed after the main route has completed. For conditional execution, use the onwhen sub-clause. Can also be defined on its own line (not in a route).
<code>onException()</code>	<code>onException</code>	Defines a sub-route (terminated by end() in the Java DSL) that gets executed whenever the specified exception occurs. Usually defined on its own line (not in a route).
<code>pipeline()</code>	<code>pipeline</code>	Pipes and Filters EIP : Sends the exchange to a series of endpoints, where the output of one endpoint becomes the input of the next endpoint. See also Section 2.1, "Pipeline Processing" .
<code>policy()</code>	<code>policy</code>	Apply a policy to the current route (currently only used for transactional policies—see "EIP Transaction Guide").
<code>pollEnrich(),pollEnrichRef()</code>	<code>pollEnrich</code>	Content Enricher EIP : Combines the current exchange with data polled from a specified <i>consumer</i> endpoint URI.

Java DSL	XML DSL	Description
<code>process(),processRef</code>	<code>process</code>	Execute a custom processor on the current exchange. See the section called “Custom processor” and “Programming EIP Components” .
<code>recipientList()</code>	<code>recipientList</code>	Recipient List EIP : Sends the exchange to a list of recipients that is calculated at runtime (for example, based on the contents of a header).
<code>removeHeader()</code>	<code>removeHeader</code>	Removes the specified header from the exchange's <i>In</i> message.
<code>removeHeaders()</code>	<code>removeHeaders</code>	Removes the headers matching the specified pattern from the exchange's <i>In</i> message. The pattern can have the form, prefix* —in which case it matches every name starting with prefix—otherwise, it is interpreted as a regular expression.
<code>removeProperty()</code>	<code>removeProperty</code>	Removes the specified exchange property from the exchange.
<code>resequence()</code>	<code>resequence</code>	Resequencer EIP : Re-orders incoming exchanges on the basis of a specified comparator operation. Supports a <i>batch</i> mode and a <i>stream</i> mode.
<code>rollback()</code>	<code>rollback</code>	<i>(Transactions)</i> Marks the current transaction for rollback only (also raising an exception, by default). See “EIP Transaction Guide” .
<code>routingSlip()</code>	<code>routingSlip</code>	Routing Slip EIP : Routes the exchange through a pipeline that is constructed dynamically, based on the list of endpoint URIs extracted from a slip header.
<code>sample()</code>	<code>sample</code>	Creates a sampling throttler, allowing you to extract a sample of exchanges from the traffic on a route.
<code>setBody()</code>	<code>setBody</code>	Sets the message body of the exchange's <i>In</i> message.

Java DSL	XML DSL	Description
<code>setExchangePattern()</code>	<code>setExchangePattern</code>	Sets the current exchange's MEP to the specified value. See the section called "Message exchange patterns" .
<code>setHeader()</code>	<code>setHeader</code>	Sets the specified header in the exchange's <i>In</i> message.
<code>setOutHeader()</code>	<code>setOutHeader</code>	Sets the specified header in the exchange's <i>Out</i> message.
<code>setProperty()</code>	<code>setProperty()</code>	Sets the specified exchange property.
<code>sort()</code>	<code>sort</code>	Sorts the contents of the <i>In</i> message body (where a custom comparator can optionally be specified).
<code>split()</code>	<code>split</code>	Splitter EIP : Splits the current exchange into a sequence of exchanges, where each split exchange contains a fragment of the original message body.
<code>stop()</code>	<code>stop</code>	Stops routing the current exchange and marks it as completed.
<code>threads()</code>	<code>threads</code>	Creates a thread pool for concurrent processing of the latter part of the route.
<code>throttle()</code>	<code>throttle</code>	Throttler EIP : Limit the flow rate to the specified level (exchanges per second).
<code>throwException()</code>	<code>throwException</code>	Throw the specified Java exception.
<code>to()</code>	<code>to</code>	Send the exchange to one or more endpoints. See Section 2.1, "Pipeline Processing" .
<code>toF()</code>	<i>N/A</i>	Send the exchange to an endpoint, using string formatting. That is, the endpoint URI string can embed substitutions in the style of the C <code>printf()</code> function.

Java DSL	XML DSL	Description
<code>transacted()</code>	<code>transacted</code>	Create a Spring transaction scope that encloses the latter part of the route. See "EIP Transaction Guide" .
<code>transform()</code>	<code>transform</code>	Message Translator EIP : Copy the <i>In</i> message headers to the <i>Out</i> message headers and set the <i>Out</i> message body to the specified value.
<code>unmarshal()</code>	<code>unmarshal</code>	Transforms the <i>In</i> message body from a low-level or binary format to a high-level format, using the specified data format. See the section called "Marshalling and unmarshalling" .
<code>validate()</code>	<code>validate</code>	Takes a predicate expression to test whether the current message is valid. If the predicate returns false , throws a PredicateValidationException exception.
<code>wireTap()</code>	<code>wireTap</code>	Wire Tap EIP : Sends a copy of the current exchange to the specified wire tap URI, using the ExchangePattern.InOnly MEP.

Some sample processors

To get some idea of how to use processors in a route, see the following examples:

- [the section called "Choice"](#).
- [the section called "Filter"](#).
- [the section called "Throttler"](#).
- [the section called "Custom processor"](#).

Choice

The `choice()` processor is a conditional statement that is used to route incoming messages to alternative producer endpoints. Each alternative producer endpoint is preceded by a `when()` method, which takes a predicate argument. If the predicate is true, the following target is selected, otherwise processing proceeds to the next `when()` method in the rule. For example, the following `choice()` processor directs incoming messages to either *Target1*, *Target2*, or *Target3*, depending on the values of *Predicate1* and *Predicate2*:

```
from("SourceURL")
```

```

.choice()
    .when(Predicate1).to("Target1")
    .when(Predicate2).to("Target2")
    .otherwise().to("Target3");

```

Or equivalently in Spring XML:

```

<camelContext id="buildSimpleRouteWithChoice"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <choice>
      <when>
        <!-- First predicate -->
        <simple>header.foo = 'bar'</simple>
        <to uri="Target1"/>
      </when>
      <when>
        <!-- Second predicate -->
        <simple>header.foo = 'manchu'</simple>
        <to uri="Target2"/>
      </when>
      <otherwise>
        <to uri="Target3"/>
      </otherwise>
    </choice>
  </route>
</camelContext>

```

In the Java DSL, there is a special case where you might need to use the **endChoice()** command. Some of the standard Apache Camel processors enable you to specify extra parameters using special sub-clauses, effectively opening an extra level of nesting which is usually terminated by the **end()** command. For example, you could specify a load balancer clause as **loadBalance().roundRobin().to("mock:foo").to("mock:bar").end()**, which load balances messages between the **mock:foo** and **mock:bar** endpoints. If the load balancer clause is embedded in a choice condition, however, it is necessary to terminate the clause using the **endChoice()** command, as follows:

```

from("direct:start")
    .choice()
        .when(body().contains("Camel"))

        .loadBalance().roundRobin().to("mock:foo").to("mock:bar").endChoice()
        .otherwise()
            .to("mock:result");

```

Filter

The **filter()** processor can be used to prevent uninteresting messages from reaching the producer endpoint. It takes a single predicate argument: if the predicate is true, the message exchange is allowed through to the producer; if the predicate is false, the message exchange is blocked. For example, the following filter blocks a message exchange, unless the incoming message contains a header, **foo**, with value equal to **bar**:


```
from("SourceURL").filter(header("foo").isEqualTo("bar")).to("TargetURL");
```

Or equivalently in Spring XML:

```
<camelContext id="filterRoute"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <filter>
      <simple>header.foo = 'bar'</simple>
      <to uri="TargetURL"/>
    </filter>
  </route>
</camelContext>
```

Throttler

The **throttle()** processor ensures that a producer endpoint does not get overloaded. The throttler works by limiting the number of messages that can pass through per second. If the incoming messages exceed the specified rate, the throttler accumulates excess messages in a buffer and transmits them more slowly to the producer endpoint. For example, to limit the rate of throughput to 100 messages per second, you can define the following rule:

```
from("SourceURL").throttle(100).to("TargetURL");
```

Or equivalently in Spring XML:

```
<camelContext id="throttleRoute"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <throttle maximumRequestsPerPeriod="100" timePeriodMillis="1000">
      <to uri="TargetURL"/>
    </throttle>
  </route>
</camelContext>
```

Custom processor

If none of the standard processors described here provide the functionality you need, you can always define your own custom processor. To create a custom processor, define a class that implements the **org.apache.camel.Processor** interface and overrides the **process()** method. The following custom processor, **MyProcessor**, removes the header named **foo** from incoming messages:

Example 1.3. Implementing a Custom Processor Class

```
public class MyProcessor implements org.apache.camel.Processor {
  public void process(org.apache.camel.Exchange exchange) {
    inMessage = exchange.getIn();
    if (inMessage != null) {
      inMessage.removeHeader("foo");
    }
  }
}
```

```
    }  
  }  
};
```

To insert the custom processor into a router rule, invoke the **process()** method, which provides a generic mechanism for inserting processors into rules. For example, the following rule invokes the processor defined in [Example 1.3, “Implementing a Custom Processor Class”](#):

```
org.apache.camel.Processor myProc = new MyProcessor();  
from("SourceURL").process(myProc).to("TargetURL");
```

CHAPTER 2. BASIC PRINCIPLES OF ROUTE BUILDING

Abstract

Apache Camel provides several processors and components that you can link together in a route. This chapter provides a basic orientation by explaining the principles of building a route using the provided building blocks.

2.1. PIPELINE PROCESSING

Overview

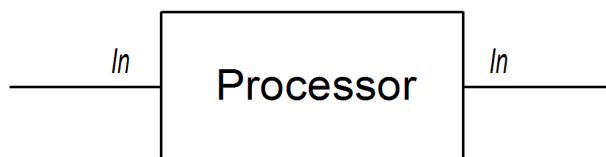
In Apache Camel, pipelining is the dominant paradigm for connecting nodes in a route definition. The pipeline concept is probably most familiar to users of the UNIX operating system, where it is used to join operating system commands. For example, `ls | more` is an example of a command that pipes a directory listing, `ls`, to the page-scrolling utility, `more`. The basic idea of a pipeline is that the *output* of one command is fed into the *input* of the next. The natural analogy in the case of a route is for the *Out* message from one processor to be copied to the *In* message of the next processor.

Processor nodes

Every node in a route, except for the initial endpoint, is a *processor*, in the sense that they inherit from the `org.apache.camel.Processor` interface. In other words, processors make up the basic building blocks of a DSL route. For example, DSL commands such as `filter()`, `delayer()`, `setBody()`, `setHeader()`, and `to()` all represent processors. When considering how processors connect together to build up a route, it is important to distinguish two different processing approaches.

The first approach is where the processor simply modifies the exchange's *In* message, as shown in [Figure 2.1, "Processor Modifying an In Message"](#). The exchange's *Out* message remains `null` in this case.

Figure 2.1. Processor Modifying an In Message



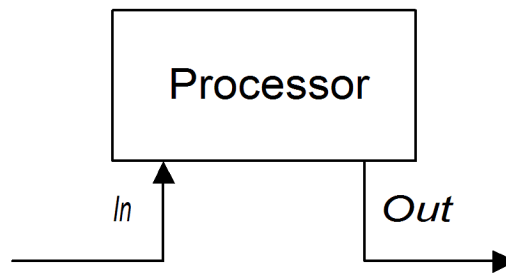
The following route shows a `setHeader()` command that modifies the current *In* message by adding (or modifying) the `BillingSystem` heading:

```

from("activemq:orderQueue")
    .setHeader("BillingSystem", xpath("/order/billingSystem"))
    .to("activemq:billingQueue");
  
```

The second approach is where the processor creates an *Out* message to represent the result of the processing, as shown in [Figure 2.2, "Processor Creating an Out Message"](#).

Figure 2.2. Processor Creating an Out Message



The following route shows a `transform()` command that creates an *Out* message with a message body containing the string, `DummyBody`:

```

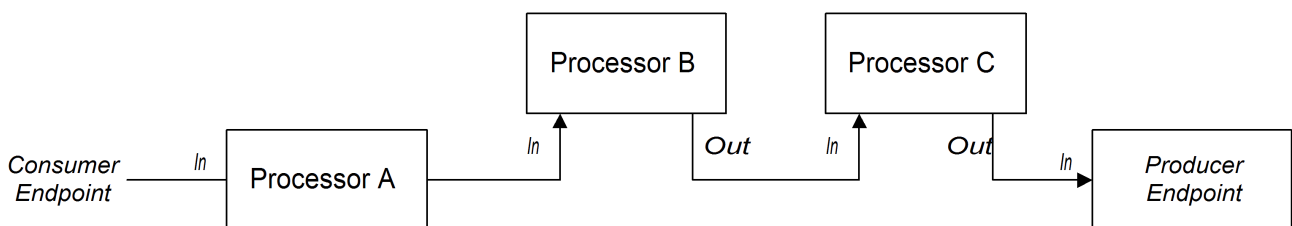
from("activemq:orderQueue")
  .transform(constant("DummyBody"))
  .to("activemq:billingQueue");
  
```

where `constant("DummyBody")` represents a constant expression. You cannot pass the string, `DummyBody`, directly, because the argument to `transform()` must be an expression type.

Pipeline for InOnly exchanges

Figure 2.3, “Sample Pipeline for InOnly Exchanges” shows an example of a processor pipeline for *InOnly* exchanges. Processor A acts by modifying the *In* message, while processors B and C create an *Out* message. The route builder links the processors together as shown. In particular, processors B and C are linked together in the form of a *pipeline*: that is, processor B's *Out* message is moved to the *In* message before feeding the exchange into processor C, and processor C's *Out* message is moved to the *In* message before feeding the exchange into the producer endpoint. Thus the processors' outputs and inputs are joined into a continuous pipeline, as shown in Figure 2.3, “Sample Pipeline for InOnly Exchanges”.

Figure 2.3. Sample Pipeline for InOnly Exchanges



Apache Camel employs the pipeline pattern by default, so you do not need to use any special syntax to create a pipeline in your routes. For example, the following route pulls messages from a `userdataQueue` queue, pipes the message through a Velocity template (to produce a customer address in text format), and then sends the resulting text address to the queue, `envelopeAddressQueue`:

```

from("activemq:userdataQueue")
  .to(ExchangePattern.InOut, "velocity:file:AdresTemplate.vm")
  .to("activemq:envelopeAddresses");
  
```

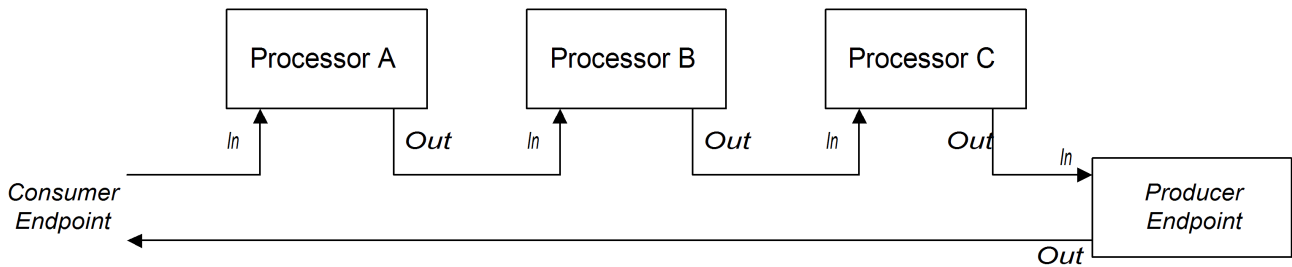
Where the Velocity endpoint, `velocity:file:AdresTemplate.vm`, specifies the location of a Velocity template file, `file:AdresTemplate.vm`, in the file system. The `to()` command changes the exchange pattern to *InOut* before sending the exchange to the Velocity endpoint and then changes it

back to *InOnly* afterwards. For more details of the Velocity endpoint, see [chapter "Velocity" in "EIP Component Reference"](#).

Pipeline for InOut exchanges

Figure 2.4, “Sample Pipeline for InOut Exchanges” shows an example of a processor pipeline for *InOut* exchanges, which you typically use to support remote procedure call (RPC) semantics. Processors A, B, and C are linked together in the form of a pipeline, with the output of each processor being fed into the input of the next. The final *Out* message produced by the producer endpoint is sent all the way back to the consumer endpoint, where it provides the reply to the original request.

Figure 2.4. Sample Pipeline for InOut Exchanges



Note that in order to support the *InOut* exchange pattern, it is *essential* that the last node in the route (whether it is a producer endpoint or some other kind of processor) creates an *Out* message. Otherwise, any client that connects to the consumer endpoint would hang and wait indefinitely for a reply message. You should be aware that not all producer endpoints create *Out* messages.

Consider the following route that processes payment requests, by processing incoming HTTP requests:

```

from("jetty:http://localhost:8080/foo")
  .to("cxf:bean:addAccountDetails")
  .to("cxf:bean:getCreditRating")
  .to("cxf:bean:processTransaction");
  
```

Where the incoming payment request is processed by passing it through a pipeline of Web services, **cxf:bean:addAccountDetails**, **cxf:bean:getCreditRating**, and **cxf:bean:processTransaction**. The final Web service, **processTransaction**, generates a response (*Out* message) that is sent back through the JETTY endpoint.

When the pipeline consists of just a sequence of endpoints, it is also possible to use the following alternative syntax:

```

from("jetty:http://localhost:8080/foo")
  .pipeline("cxf:bean:addAccountDetails", "cxf:bean:getCreditRating",
    "cxf:bean:processTransaction");
  
```

Pipeline for InOptionalOut exchanges

The pipeline for *InOptionalOut* exchanges is essentially the same as the pipeline in Figure 2.4, “Sample Pipeline for InOut Exchanges”. The difference between *InOut* and *InOptionalOut* is that an exchange with the *InOptionalOut* exchange pattern is allowed to have a null *Out* message as a reply. That is, in the case of an *InOptionalOut* exchange, a **null** *Out* message is copied to the *In* message of the next node in the pipeline. By contrast, in the case of an *InOut* exchange, a **null** *Out* message is discarded and the original *In* message from the current node would be copied to the *In* message of the next node instead.

2.2. MULTIPLE INPUTS

Overview

A standard route takes its input from just a single endpoint, using the `from(EndpointURL)` syntax in the Java DSL. But what if you need to define multiple inputs for your route? Apache Camel provides several alternatives for specifying multiple inputs to a route. The approach to take depends on whether you want the exchanges to be processed independently of each other or whether you want the exchanges from different inputs to be combined in some way (in which case, you should use the [the section called “Content enricher pattern”](#)).

Multiple independent inputs

The simplest way to specify multiple inputs is using the multi-argument form of the `from()` DSL command, for example:

```
from("URI1", "URI2", "URI3").to("DestinationUri");
```

Or you can use the following equivalent syntax:

```
from("URI1").from("URI2").from("URI3").to("DestinationUri");
```

In both of these examples, exchanges from each of the input endpoints, `URI1`, `URI2`, and `URI3`, are processed independently of each other and in separate threads. In fact, you can think of the preceding route as being equivalent to the following three separate routes:

```
from("URI1").to("DestinationUri");
from("URI2").to("DestinationUri");
from("URI3").to("DestinationUri");
```

Segmented routes

For example, you might want to merge incoming messages from two different messaging systems and process them using the same route. In most cases, you can deal with multiple inputs by dividing your route into segments, as shown in [Figure 2.5, “Processing Multiple Inputs with Segmented Routes”](#).

Figure 2.5. Processing Multiple Inputs with Segmented Routes

```
from("activemq:Nyse").to(InternalUrl)
                                ↓
                                from(InternalUrl).to("activemq:USTxn")
                                ↑
from("activemq:Nasdaq").to(InternalUrl)
```

The initial segments of the route take their inputs from some external queues—for example, `activemq:Nyse` and `activemq:Nasdaq`—and send the incoming exchanges to an internal endpoint, `InternalUrl`. The second route segment merges the incoming exchanges, taking them from the internal endpoint and sending them to the destination queue, `activemq:USTxn`. The `InternalUrl` is the URL for an endpoint that is intended only for use *within* a router application. The following types of endpoints are suitable for internal use:

- [the section called “Direct endpoints”](#).

- the section called “SEDA endpoints”.
- the section called “VM endpoints”.

The main purpose of these endpoints is to enable you to glue together different segments of a route. They all provide an effective way of merging multiple inputs into a single route.

Direct endpoints

The direct component provides the simplest mechanism for linking together routes. The event model for the direct component is *synchronous*, so that subsequent segments of the route run in the same thread as the first segment. The general format of a direct URL is **direct:EndpointID**, where the endpoint ID, *EndpointID*, is simply a unique alphanumeric string that identifies the endpoint instance.

For example, if you want to take the input from two message queues, **activemq:Nyse** and **activemq:Nasdaq**, and merge them into a single message queue, **activemq:USTxn**, you can do this by defining the following set of routes:

```
from("activemq:Nyse").to("direct:mergeTxns");
from("activemq:Nasdaq").to("direct:mergeTxns");

from("direct:mergeTxns").to("activemq:USTxn");
```

Where the first two routes take the input from the message queues, **Nyse** and **Nasdaq**, and send them to the endpoint, **direct:mergeTxns**. The last queue combines the inputs from the previous two queues and sends the combined message stream to the **activemq:USTxn** queue.

The implementation of the direct endpoint behaves as follows: whenever an exchange arrives at a producer endpoint (for example, **to("direct:mergeTxns")**), the direct endpoint passes the exchange directly to all of the consumers endpoints that have the same endpoint ID (for example, **from("direct:mergeTxns")**). Direct endpoints can only be used to communicate between routes that belong to the same **CamelContext** in the same Java virtual machine (JVM) instance.

SEDA endpoints

The SEDA component provides an alternative mechanism for linking together routes. You can use it in a similar way to the direct component, but it has a different underlying event and threading model, as follows:

- Processing of a SEDA endpoint is *not* synchronous. That is, when you send an exchange to a SEDA producer endpoint, control immediately returns to the preceding processor in the route.
- SEDA endpoints contain a queue buffer (of **java.util.concurrent.BlockingQueue** type), which stores all of the incoming exchanges prior to processing by the next route segment.
- Each SEDA consumer endpoint creates a thread pool (the default size is 5) to process exchange objects from the blocking queue.
- The SEDA component supports the *competing consumers* pattern, which guarantees that each incoming exchange is processed only once, even if there are multiple consumers attached to a specific endpoint.

One of the main advantages of using a SEDA endpoint is that the routes can be more responsive, owing to the built-in consumer thread pool. The stock transactions example can be re-written to use SEDA endpoints instead of direct endpoints, as follows:

```

from("activemq:Nyse").to("seda:mergeTxns");
from("activemq:Nasdaq").to("seda:mergeTxns");

from("seda:mergeTxns").to("activemq:USTxn");

```

The main difference between this example and the direct example is that when using SEDA, the second route segment (from **seda:mergeTxns** to **activemq:USTxn**) is processed by a pool of five threads.



NOTE

There is more to SEDA than simply pasting together route segments. The staged event-driven architecture (SEDA) encompasses a design philosophy for building more manageable multi-threaded applications. The purpose of the SEDA component in Apache Camel is simply to enable you to apply this design philosophy to your applications. For more details about SEDA, see <http://www.eecs.harvard.edu/~mdw/proj/seda/>.

VM endpoints

The VM component is very similar to the SEDA endpoint. The only difference is that, whereas the SEDA component is limited to linking together route segments from within the same **CamelContext**, the VM component enables you to link together routes from distinct Apache Camel applications, as long as they are running within the same Java virtual machine.

The stock transactions example can be re-written to use VM endpoints instead of SEDA endpoints, as follows:

```

from("activemq:Nyse").to("vm:mergeTxns");
from("activemq:Nasdaq").to("vm:mergeTxns");

```

And in a separate router application (running in the same Java VM), you can define the second segment of the route as follows:

```

from("vm:mergeTxns").to("activemq:USTxn");

```

Content enricher pattern

The content enricher pattern defines a fundamentally different way of dealing with multiple inputs to a route. When an exchange enters the enricher processor, the enricher contacts an external resource to retrieve information, which is then added to the original message. In this pattern, the external resource effectively represents a second input to the message.

For example, suppose you are writing an application that processes credit requests. Before processing a credit request, you need to augment it with the data that assigns a credit rating to the customer, where the ratings data is stored in a file in the directory, **src/data/ratings**. You can combine the incoming credit request with data from the ratings file using the **pollEnrich()** pattern and a **GroupedExchangeAggregationStrategy** aggregation strategy, as follows:

```

from("jms:queue:creditRequests")
    .pollEnrich("file:src/data/ratings?noop=true", new
GroupedExchangeAggregationStrategy())
    .bean(new MergeCreditRequestAndRatings(), "merge")
    .to("jms:queue:reformattedRequests");

```


Where the **GroupedExchangeAggregationStrategy** class is a standard aggregation strategy from the **org.apache.camel.processor.aggregate** package that adds each new exchange to a **java.util.List** instance and stores the resulting list in the **Exchange.GROUPED_EXCHANGE** exchange property. In this case, the list contains two elements: the original exchange (from the **creditRequests** JMS queue); and the enricher exchange (from the file endpoint).

To access the grouped exchange, you can use code like the following:

```
public class MergeCreditRequestAndRatings {
    public void merge(Exchange ex) {
        // Obtain the grouped exchange
        List<Exchange> list = ex.getProperty(Exchange.GROUPED_EXCHANGE,
List.class);

        // Get the exchanges from the grouped exchange
        Exchange originalEx = list.get(0);
        Exchange ratingsEx = list.get(1);

        // Merge the exchanges
        ...
    }
}
```

An alternative approach to this application would be to put the merge code directly into the implementation of the custom aggregation strategy class.

For more details about the content enricher pattern, see [Section 8.1, “Content Enricher”](#).

2.3. EXCEPTION HANDLING

Abstract

Apache Camel provides several different mechanisms, which let you handle exceptions at different levels of granularity: you can handle exceptions within a route using **doTry**, **doCatch**, and **doFinally**; or you can specify what action to take for each exception type and apply this rule to all routes in a **RouteBuilder** using **onException**; or you can specify what action to take for *all* exception types and apply this rule to all routes in a **RouteBuilder** using **errorHandler**.

For more details about exception handling, see [Section 5.3, “Dead Letter Channel”](#).

2.3.1. onException Clause

Overview

The **onException** clause is a powerful mechanism for trapping exceptions that occur in one or more routes: it is type-specific, enabling you to define distinct actions to handle different exception types; it allows you to define actions using essentially the same (actually, slightly extended) syntax as a route, giving you considerable flexibility in the way you handle exceptions; and it is based on a trapping model, which enables a single **onException** clause to deal with exceptions occurring at any node in any route.

Trapping exceptions using onException

The **onException** clause is a mechanism for *trapping*, rather than catching exceptions. That is, once you define an **onException** clause, it traps exceptions that occur at any point in a route. This contrasts with the Java try/catch mechanism, where an exception is caught, only if a particular code fragment is *explicitly* enclosed in a try block.

What really happens when you define an **onException** clause is that the Apache Camel runtime implicitly encloses each route node in a try block. This is why the **onException** clause is able to trap exceptions at any point in the route. But this wrapping is done for you automatically; it is not visible in the route definitions.

Java DSL example

In the following Java DSL example, the **onException** clause applies to all of the routes defined in the **RouteBuilder** class. If a **ValidationException** exception occurs while processing either of the routes (**from("seda:inputA")** or **from("seda:inputB")**), the **onException** clause traps the exception and redirects the current exchange to the **validationFailed** JMS queue (which serves as a deadletter queue).

```
// Java
public class MyRouteBuilder extends RouteBuilder {

    public void configure() {
        onException(ValidationException.class)
            .to("activemq:validationFailed");

        from("seda:inputA")
            .to("validation:foo/bar.xsd", "activemq:someQueue");

        from("seda:inputB").to("direct:foo")
            .to("rnc:mySchema.rnc", "activemq:anotherQueue");
    }
}
```

XML DSL example

The preceding example can also be expressed in the XML DSL, using the **onException** element to define the exception clause, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:camel="http://camel.apache.org/schema/spring"
    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">
        <onException>
            <exception>com.mycompany.ValidationException</exception>
            <to uri="activemq:validationFailed"/>
        </onException>
        <route>
            <from uri="seda:inputA"/>
```

```

        <to uri="validation:foo/bar.xsd"/>
        <to uri="activemq:someQueue"/>
    </route>
    <route>
        <from uri="seda:inputB"/>
        <to uri="rnc:mySchema.rnc"/>
        <to uri="activemq:anotherQueue"/>
    </route>
</camelContext>

</beans>

```

Trapping multiple exceptions

You can define multiple **onException** clauses to trap exceptions in a **RouteBuilder** scope. This enables you to take different actions in response to different exceptions. For example, the following series of **onException** clauses defined in the Java DSL define different deadletter destinations for **ValidationException**, **ValidationException**, and **Exception**:

```

onException(ValidationException.class).to("activemq:validationFailed");
onException(java.io.IOException.class).to("activemq:ioExceptions");
onException(Exception.class).to("activemq:exceptions");

```

You can define the same series of **onException** clauses in the XML DSL as follows:

```

<onException>
    <exception>com.mycompany.ValidationException</exception>
    <to uri="activemq:validationFailed"/>
</onException>
<onException>
    <exception>java.io.IOException</exception>
    <to uri="activemq:ioExceptions"/>
</onException>
<onException>
    <exception>java.lang.Exception</exception>
    <to uri="activemq:exceptions"/>
</onException>

```

You can also group multiple exceptions together to be trapped by the same **onException** clause. In the Java DSL, you can group multiple exceptions as follows:

```

onException(ValidationException.class, BuesinessException.class)
    .to("activemq:validationFailed");

```

In the XML DSL, you can group multiple exceptions together by defining more than one **exception** element inside the **onException** element, as follows:

```

<onException>
    <exception>com.mycompany.ValidationException</exception>
    <exception>com.mycompany.BuesinessException</exception>
    <to uri="activemq:validationFailed"/>
</onException>

```

When trapping multiple exceptions, the order of the **onException** clauses is significant. Apache Camel initially attempts to match the thrown exception against the *first* clause. If the first clause fails to match, the next **onException** clause is tried, and so on until a match is found. Each matching attempt is governed by the following algorithm:

1. If the thrown exception is a [chained exception](#) (that is, where an exception has been caught and rethrown as a different exception), the most nested exception type serves initially as the basis for matching. This exception is tested as follows:
 - a. If the exception-to-test has exactly the type specified in the **onException** clause (tested using **instanceof**), a match is triggered.
 - b. If the exception-to-test is a sub-type of the type specified in the **onException** clause, a match is triggered.
2. If the most nested exception fails to yield a match, the next exception in the chain (the wrapping exception) is tested instead. The testing continues up the chain until either a match is triggered or the chain is exhausted.

Deadletter channel

The basic examples of **onException** usage have so far all exploited the *deadletter channel* pattern. That is, when an **onException** clause traps an exception, the current exchange is routed to a special destination (the deadletter channel). The deadletter channel serves as a holding area for failed messages that have *not* been processed. An administrator can inspect the messages at a later time and decide what action needs to be taken.

For more details about the deadletter channel pattern, see [Section 5.3, “Dead Letter Channel”](#).

Use original message

By the time an exception is raised in the middle of a route, the message in the exchange could have been modified considerably (and might not even be readable by a human). Often, it is easier for an administrator to decide what corrective actions to take, if the messages visible in the deadletter queue are the *original* messages, as received at the start of the route.

In the Java DSL, you can replace the message in the exchange by the original message, using the **useOriginalMessage()** DSL command, as follows:

```
onException(ValidationException.class)
    .useOriginalMessage()
    .to("activemq:validationFailed");
```

In the XML DSL, you can retrieve the original message by setting the **useOriginalMessage** attribute on the **onException** element, as follows:

```
<onException useOriginalMessage="true">
  <exception>com.mycompany.ValidationException</exception>
  <to uri="activemq:validationFailed"/>
</onException>
```

Redelivery policy

Instead of interrupting the processing of a message and giving up as soon as an exception is raised,

Apache Camel gives you the option of attempting to *redeliver* the message at the point where the exception occurred. In networked systems, where timeouts can occur and temporary faults arise, it is often possible for failed messages to be processed successfully, if they are redelivered shortly after the original exception was raised.

The Apache Camel redelivery supports various strategies for redelivering messages after an exception occurs. Some of the most important options for configuring redelivery are as follows:

maximumRedeliveries()

Specifies the maximum number of times redelivery can be attempted (default is **0**). A negative value means redelivery is always attempted (equivalent to an infinite value).

retryWhile()

Specifies a predicate (of **Predicate** type), which determines whether Apache Camel ought to continue redelivering. If the predicate evaluates to **true** on the current exchange, redelivery is attempted; otherwise, redelivery is stopped and no further redelivery attempts are made.

This option takes precedence over the **maximumRedeliveries()** option.

In the Java DSL, redelivery policy options are specified using DSL commands in the **onException** clause. For example, you can specify a maximum of six redeliveries, after which the exchange is sent to the **validationFailed** deadletter queue, as follows:

```
onException(ValidationException.class)
    .maximumRedeliveries(6)
    .retryAttemptedLogLevel(org.apache.camel.LogginLevel.WARN)
    .to("activemq:validationFailed");
```

In the XML DSL, redelivery policy options are specified by setting attributes on the **redeliveryPolicy** element. For example, the preceding route can be expressed in XML DSL as follows:

```
<onException useOriginalMessage="true">
    <exception>com.mycompany.ValidationException</exception>
    <redeliveryPolicy maximumRedeliveries="6"/>
    <to uri="activemq:validationFailed"/>
</onException>
```

The latter part of the route—after the redelivery options are set—is not processed until after the last redelivery attempt has failed. For detailed descriptions of all the redelivery options, see [Section 5.3, “Dead Letter Channel”](#).

Alternatively, you can specify redelivery policy options in a **redeliveryPolicyProfile** instance. You can then reference the **redeliveryPolicyProfile** instance using the **onException** element's **redeliverPolicyRef** attribute. For example, the preceding route can be expressed as follows:

```
<redeliveryPolicyProfile id="redelivPolicy" maximumRedeliveries="6"
    retryAttemptedLogLevel="WARN"/>

<onException useOriginalMessage="true"
    redeliverPolicyRef="redelivPolicy">
    <exception>com.mycompany.ValidationException</exception>
    <to uri="activemq:validationFailed"/>
</onException>
```

**NOTE**

The approach using **redeliveryPolicyProfile** is useful, if you want to re-use the same redelivery policy in multiple **onException** clauses.

Conditional trapping

Exception trapping with **onException** can be made conditional by specifying the **onWhen** option. If you specify the **onWhen** option in an **onException** clause, a match is triggered only when the thrown exception matches the clause *and* the **onWhen** predicate evaluates to **true** on the current exchange.

For example, in the following Java DSL fragment, the first **onException** clause triggers, only if the thrown exception matches **MyUserException** and the **user** header is non-null in the current exchange:

```
// Java

// Here we define onException() to catch MyUserException when
// there is a header[user] on the exchange that is not null
onException(MyUserException.class)
    .onWhen(header("user").isNotNull())
    .maximumRedeliveries(2)
    .to(ERROR_USER_QUEUE);

// Here we define onException to catch MyUserException as a kind
// of fallback when the above did not match.
// Notice: The order how we have defined these onException is
// important as Camel will resolve in the same order as they
// have been defined
onException(MyUserException.class)
    .maximumRedeliveries(2)
    .to(ERROR_QUEUE);
```

The preceding **onException** clauses can be expressed in the XML DSL as follows:

```
<redeliveryPolicyProfile id="twoRedeliveries" maximumRedeliveries="2"/>

<onException redeliveryPolicyRef="twoRedeliveries">
  <exception>com.mycompany.MyUserException</exception>
  <onWhen>
    <simple>${header.user} != null</simple>
  </onWhen>
  <to uri="activemq:error_user_queue"/>
</onException>

<onException redeliveryPolicyRef="twoRedeliveries">
  <exception>com.mycompany.MyUserException</exception>
  <to uri="activemq:error_queue"/>
</onException>
```

Handling exceptions

By default, when an exception is raised in the middle of a route, processing of the current exchange is

interrupted and the thrown exception is propagated back to the consumer endpoint at the start of the route. When an **onException** clause is triggered, the behavior is essentially the same, except that the **onException** clause performs some processing before the thrown exception is propagated back.

But this default behavior is *not* the only way to handle an exception. The **onException** provides various options to modify the exception handling behavior, as follows:

- the section called “[Suppressing exception rethrow](#)”—you have the option of suppressing the rethrown exception after the **onException** clause has completed. In other words, in this case the exception does *not* propagate back to the consumer endpoint at the start of the route.
- the section called “[Continuing processing](#)”—you have the option of resuming normal processing of the exchange from the point where the exception originally occurred. Implicitly, this approach also suppresses the rethrown exception.
- the section called “[Sending a response](#)”—in the special case where the consumer endpoint at the start of the route expects a reply (that is, having an *InOut* MEP), you might prefer to construct a custom fault reply message, rather than propagating the exception back to the consumer endpoint.

Suppressing exception rethrow

To prevent the current exception from being rethrown and propagated back to the consumer endpoint, you can set the **handled()** option to **true** in the Java DSL, as follows:

```
onException(ValidationException.class)
    .handled(true)
    .to("activemq:validationFailed");
```

In the Java DSL, the argument to the **handled()** option can be of boolean type, of **Predicate** type, or of **Expression** type (where any non-boolean expression is interpreted as **true**, if it evaluates to a non-null value).

The same route can be configured to suppress the rethrown exception in the XML DSL, using the **handled** element, as follows:

```
<onException>
  <exception>com.mycompany.ValidationException</exception>
  <handled>
    <constant>true</constant>
  </handled>
  <to uri="activemq:validationFailed"/>
</onException>
```

Continuing processing

To continue processing the current message from the point in the route where the exception was originally thrown, you can set the **continued** option to **true** in the Java DSL, as follows:

```
onException(ValidationException.class)
    .continued(true);
```

In the Java DSL, the argument to the `continued()` option can be of boolean type, of `Predicate` type, or of `Expression` type (where any non-boolean expression is interpreted as `true`, if it evaluates to a non-null value).

The same route can be configured in the XML DSL, using the `continued` element, as follows:

```
<onException>
  <exception>com.mycompany.ValidationException</exception>
  <continued>
    <constant>true</constant>
  </continued>
</onException>
```

Sending a response

When the consumer endpoint that starts a route expects a reply, you might prefer to construct a custom fault reply message, instead of simply letting the thrown exception propagate back to the consumer. There are two essential steps you need to follow in this case: suppress the rethrown exception using the `handled` option; and populate the exchange's `Out` message slot with a custom fault message.

For example, the following Java DSL fragment shows how to send a reply message containing the text string, `Sorry`, whenever the `MyFunctionalException` exception occurs:

```
// we catch MyFunctionalException and want to mark it as handled (= no
// failure returned to client)
// but we want to return a fixed text response, so we transform OUT body
// as Sorry.
onException(MyFunctionalException.class)
  .handled(true)
  .transform().constant("Sorry");
```

If you are sending a fault response to the client, you will often want to incorporate the text of the exception message in the response. You can access the text of the current exception message using the `exceptionMessage()` builder method. For example, you can send a reply containing just the text of the exception message whenever the `MyFunctionalException` exception occurs, as follows:

```
// we catch MyFunctionalException and want to mark it as handled (= no
// failure returned to client)
// but we want to return a fixed text response, so we transform OUT body
// and return the exception message
onException(MyFunctionalException.class)
  .handled(true)
  .transform(exceptionMessage());
```

The exception message text is also accessible from the Simple language, through the `exception.message` variable. For example, you could embed the current exception text in a reply message, as follows:

```
// we catch MyFunctionalException and want to mark it as handled (= no
// failure returned to client)
// but we want to return a fixed text response, so we transform OUT body
// and return a nice message
// using the simple language where we want insert the exception message
onException(MyFunctionalException.class)
```



```

        .handled(true)
        .transform().simple("Error reported: ${exception.message} - cannot
process this message.");

```

The preceding **onException** clause can be expressed in XML DSL as follows:

```

<onException>
  <exception>com.mycompany.MyFunctionalException</exception>
  <handled>
    <constant>true</constant>
  </handled>
  <transform>
    <simple>Error reported: ${exception.message} - cannot process this
message.</simple>
  </transform>
</onException>

```

Exception thrown while handling an exception

An exception that gets thrown while handling an existing exception (in other words, one that gets thrown in the middle of processing an **onException** clause) is handled in a special way. Such an exception is handled by the special fallback exception handler, which handles the exception as follows:

- All existing exception handlers are ignored and processing fails immediately.
- The new exception is logged.
- The new exception is set on the exchange object.

The simple strategy avoids complex failure scenarios which could otherwise end up with an **onException** clause getting locked into an infinite loop.

Scopes

The **onException** clauses can be effective in either of the following scopes:

- *RouteBuilder scope*—**onException** clauses defined as standalone statements inside a **RouteBuilder.configure()** method affect all of the routes defined in that **RouteBuilder** instance. On the other hand, these **onException** clauses *have no effect whatsoever* on routes defined inside any other **RouteBuilder** instance. The **onException** clauses *must* appear before the route definitions.

All of the examples up to this point are defined using the **RouteBuilder** scope.

- *Route scope*—**onException** clauses can also be embedded directly within a route. These **onException** clauses affect *only* the route in which they are defined.

Route scope

You can embed an **onException** clause anywhere inside a route definition, but you must terminate the embedded **onException** clause using the **end()** DSL command.

For example, you can define an embedded **onException** clause in the Java DSL, as follows:

```
// Java
from("direct:start")
  .onException(OrderFailedException.class)
    .maximumRedeliveries(1)
    .handled(true)
    .beanRef("orderService", "orderFailed")
    .to("mock:error")
  .end()
  .beanRef("orderService", "handleOrder")
  .to("mock:result");
```

You can define an embedded **onException** clause in the XML DSL, as follows:

```
<route errorHandlerRef="deadLetter">
  <from uri="direct:start"/>
  <onException>
    <exception>com.mycompany.OrderFailedException</exception>
    <redeliveryPolicy maximumRedeliveries="1"/>
    <handled>
      <constant>true</constant>
    </handled>
    <bean ref="orderService" method="orderFailed"/>
    <to uri="mock:error"/>
  </onException>
  <bean ref="orderService" method="handleOrder"/>
  <to uri="mock:result"/>
</route>
```

2.3.2. Error Handler

Overview

The **errorHandler()** clause provides similar features to the **onException** clause, except that this mechanism is *not* able to discriminate between different exception types. The **errorHandler()** clause is the original exception handling mechanism provided by Apache Camel and was available before the **onException** clause was implemented.

Java DSL example

The **errorHandler()** clause is defined in a **RouteBuilder** class and applies to all of the routes in that **RouteBuilder** class. It is triggered whenever an exception of *any kind* occurs in one of the applicable routes. For example, to define an error handler that routes all failed exchanges to the ActiveMQ **deadLetter** queue, you can define a **RouteBuilder** as follows:

```
public class MyRouteBuilder extends RouteBuilder {

  public void configure() {
    errorHandler(deadLetterChannel("activemq:deadLetter"));

    // The preceding error handler applies
    // to all of the following routes:
    from("activemq:orderQueue")
      .to("pop3://fulfillment@acme.com");
  }
}
```

```

        from("file:src/data?noop=true")
          .to("file:target/messages");
        // ...
    }
}

```

Redirection to the dead letter channel will not occur, however, until all attempts at redelivery have been exhausted.

XML DSL example

In the XML DSL, you define an error handler within a **camelContext** scope using the **errorHandler** element. For example, to define an error handler that routes all failed exchanges to the ActiveMQ **deadLetter** queue, you can define an **errorHandler** element as follows:

```

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:camel="http://camel.apache.org/schema/spring"
  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">
    <errorHandler type="DeadLetterChannel"
      deadLetterUri="activemq:deadLetter"/>

    <route>
      <from uri="activemq:orderQueue"/>
      <to uri="pop3://fulfillment@acme.com"/>
    </route>
    <route>
      <from uri="file:src/data?noop=true"/>
      <to uri="file:target/messages"/>
    </route>
  </camelContext>

</beans>

```

Types of error handler

Table 2.1, “Error Handler Types” provides an overview of the different types of error handler you can define.

Table 2.1. Error Handler Types

Java DSL Builder	XML DSL Type Attribute	Description
<code>defaultErrorHandler()</code>	<code>DefaultErrorHandler</code>	Propagates exceptions back to the caller and supports the redelivery policy, but it does not support a dead letter queue.

Java DSL Builder	XML DSL Type Attribute	Description
<code>deadLetterChannel()</code>	<code>DeadLetterChannel</code>	Supports the same features as the default error handler and, in addition, supports a dead letter queue.
<code>loggingErrorHandler()</code>	<code>LoggingErrorHandler</code>	Logs the exception text whenever an exception occurs.
<code>noErrorHandler()</code>	<code>NoErrorHandler</code>	Dummy handler implementation that can be used to disable the error handler.
	<code>TransactionErrorHandler</code>	An error handler for transacted routes. A default transaction error handler instance is automatically used for a route that is marked as transacted.

2.3.3. doTry, doCatch, and doFinally

Overview

To handle exceptions within a route, you can use a combination of the **doTry**, **doCatch**, and **doFinally** clauses, which handle exceptions in a similar way to Java's **try**, **catch**, and **finally** blocks.

Similarities between doCatch and Java catch

In general, the **doCatch()** clause in a route definition behaves in an analogous way to the **catch()** statement in Java code. In particular, the following features are supported by the **doCatch()** clause:

- *Multiple doCatch clauses*—you can have multiple **doCatch** clauses within a single **doTry** block. The **doCatch** clauses are tested in the order they appear, just like Java **catch()** statements. Apache Camel executes the first **doCatch** clause that matches the thrown exception.



NOTE

This algorithm is different from the exception matching algorithm used by the **onException** clause—see [Section 2.3.1, “onException Clause”](#) for details.

- *Rethrowing exceptions*—you can rethrow the current exception from within a **doCatch** clause using the **handled** sub-clause (see [the section called “Rethrowing exceptions in doCatch”](#)).

Special features of doCatch

There are some special features of the **doCatch()** clause, however, that have no analogue in the Java **catch()** statement. The following features are specific to **doCatch()**:

- *Catching multiple exceptions*—the **doCatch** clause allows you to specify a list of exceptions to catch, in contrast to the Java **catch()** statement, which catches only one exception (see [the section called “Example”](#)).
- *Conditional catching*—you can catch an exception conditionally, by appending an **onWhen** sub-clause to the **doCatch** clause (see [the section called “Conditional exception catching using onWhen”](#)).

Example

The following example shows how to write a **doTry** block in the Java DSL, where the **doCatch()** clause will be executed, if either the **IOException** exception or the **IllegalStateException** exception are raised, and the **doFinally()** clause is *always* executed, irrespective of whether an exception is raised or not.

```
from("direct:start")
  .doTry()
    .process(new ProcessorFail())
    .to("mock:result")
  .doCatch(IOException.class, IllegalStateException.class)
    .to("mock:catch")
  .doFinally()
    .to("mock:finally")
  .end();
```

Or equivalently, in Spring XML:

```
<route>
  <from uri="direct:start"/>
  <!-- here the try starts. its a try .. catch .. finally just as
regular java code -->
  <doTry>
    <process ref="processorFail"/>
    <to uri="mock:result"/>
    <doCatch>
      <!-- catch multiple exceptions -->
      <exception>java.io.IOException</exception>
      <exception>java.lang.IllegalStateException</exception>
      <to uri="mock:catch"/>
    </doCatch>
    <doFinally>
      <to uri="mock:finally"/>
    </doFinally>
  </doTry>
</route>
```

Rethrowing exceptions in doCatch

It is possible to rethrow an exception in a **doCatch()** clause by calling the **handled()** sub-clause with its argument set to **false**, as follows:

```
from("direct:start")
  .doTry()
    .process(new ProcessorFail())
```

```

        .to("mock:result")
    .doCatch(IOException.class)
        // mark this as NOT handled, eg the caller will also get the
exception
        .handled(false)
        .to("mock:io")
    .doCatch(Exception.class)
        // and catch all other exceptions
        .to("mock:error")
    .end();

```

In the preceding example, if the **IOException** is caught by **doCatch()**, the current exchange is sent to the **mock:io** endpoint, and then the **IOException** is rethrown. This gives the consumer endpoint at the start of the route (in the **from()** command) an opportunity to handle the exception as well.

The following example shows how to define the same route in Spring XML:

```

<route>
  <from uri="direct:start"/>
  <doTry>
    <process ref="processorFail"/>
    <to uri="mock:result"/>
    <doCatch>
      <exception>java.io.IOException</exception>
      <!-- mark this as NOT handled, eg the caller will also get the
exception -->
      <handled>
        <constant>>false</constant>
      </handled>
      <to uri="mock:io"/>
    </doCatch>
    <doCatch>
      <!-- and catch all other exceptions they are handled by
default (ie handled = true) -->
      <exception>java.lang.Exception</exception>
      <to uri="mock:error"/>
    </doCatch>
  </doTry>
</route>

```

Conditional exception catching using onWhen

A special feature of the Apache Camel **doCatch()** clause is that you can conditionalize the catching of exceptions based on an expression that is evaluated at run time. In other words, if you catch an exception using a clause of the form, **doCatch(ExceptionList).doWhen(Expression)**, an exception will only be caught, if the predicate expression, *Expression*, evaluates to **true** at run time.

For example, the following **doTry** block will catch the exceptions, **IOException** and **IllegalStateException**, only if the exception message contains the word, **Severe**:

```

from("direct:start")
  .doTry()
    .process(new ProcessorFail())
    .to("mock:result")
  .doCatch(IOException.class, IllegalStateException.class)

```

```

        .onWhen(exceptionMessage().contains("Severe"))
        .to("mock:catch")
    .doCatch(CamelExchangeException.class)
        .to("mock:catchCamel")
    .doFinally()
        .to("mock:finally")
    .end();

```

Or equivalently, in Spring XML:

```

<route>
  <from uri="direct:start"/>
  <doTry>
    <process ref="processorFail"/>
    <to uri="mock:result"/>
    <doCatch>
      <exception>java.io.IOException</exception>
      <exception>java.lang.IllegalStateException</exception>
      <onWhen>
        <simple>${exception.message} contains 'Severe'</simple>
      </onWhen>
      <to uri="mock:catch"/>
    </doCatch>
    <doCatch>
      <exception>org.apache.camel.CamelExchangeException</exception>
      <to uri="mock:catchCamel"/>
    </doCatch>
    <doFinally>
      <to uri="mock:finally"/>
    </doFinally>
  </doTry>
</route>

```

2.3.4. Propagating SOAP Exceptions

Overview

The Camel CXF component provides an integration with Apache CXF, enabling you to send and receive SOAP messages from Apache Camel endpoints. You can easily define Apache Camel endpoints in XML, which can then be referenced in a route using the endpoint's bean ID. For more details, see [CXF](#).

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.

2.4. BEAN INTEGRATION

Overview

Bean integration provides a general purpose mechanism for processing messages using arbitrary Java objects. By inserting a bean reference into a route, you can call an arbitrary method on a Java object, which can then access and modify the incoming exchange. The mechanism that maps an exchange's contents to the parameters and return values of a bean method is known as *parameter binding*. Parameter binding can use any combination of the following approaches in order to initialize a method's parameters:

- *Conventional method signatures* — If the method signature conforms to certain conventions, the parameter binding can use Java reflection to determine what parameters to pass.
- *Annotations and dependency injection* — For a more flexible binding mechanism, employ Java annotations to specify what to inject into the method's arguments. This dependency injection

mechanism relies on Spring 2.5 component scanning. Normally, if you are deploying your Apache Camel application into a Spring container, the dependency injection mechanism will work automatically.

- *Explicitly specified parameters* — You can specify parameters explicitly (either as constants or using the Simple language), at the point where the bean is invoked.

Bean registry

Beans are made accessible through a *bean registry*, which is a service that enables you to look up beans using either the class name or the bean ID as a key. The way that you create an entry in the bean registry depends on the underlying framework—for example, plain Java, Spring, Guice, or Blueprint. Registry entries are usually created implicitly (for example, when you instantiate a Spring bean in a Spring XML file).

Registry plug-in strategy

Apache Camel implements a plug-in strategy for the bean registry, defining an integration layer for accessing beans which makes the underlying registry implementation transparent. Hence, it is possible to integrate Apache Camel applications with a variety of different bean registries, as shown in [Table 2.2, “Registry Plug-Ins”](#).

Table 2.2. Registry Plug-Ins

Registry Implementation	Camel Component with Registry Plug-In
Spring bean registry	camel-spring
Guice bean registry	camel-guice
Blueprint bean registry	camel-blueprint
OSGi service registry	deployed in <i>OSGi container</i>

Normally, you do not have to worry about configuring bean registries, because the relevant bean registry is automatically installed for you. For example, if you are using the Spring framework to define your routes, the Spring **ApplicationContextRegistry** plug-in is automatically installed in the current **CamelContext** instance.

Deployment in an OSGi container is a special case. When an Apache Camel route is deployed into the OSGi container, the **CamelContext** automatically sets up a registry chain for resolving bean instances: the registry chain consists of the OSGi registry, followed by the Blueprint (or Spring) registry.

Accessing a bean created in Java

To process exchange objects using a Java bean (which is a plain old Java object or POJO), use the **bean()** processor, which binds the inbound exchange to a method on the Java object. For example, to process inbound exchanges using the class, **MyBeanProcessor**, define a route like the following:

```
from("file:data/inbound")
    .bean(MyBeanProcessor.class, "processBody")
    .to("file:data/outbound");
```

Where the `bean()` processor creates an instance of `MyBeanProcessor` type and invokes the `processBody()` method to process inbound exchanges. This approach is adequate if you only want to access the `MyBeanProcessor` instance from a single route. However, if you want to access the same `MyBeanProcessor` instance from multiple routes, use the variant of `bean()` that takes the `Object` type as its first argument. For example:

```
MyBeanProcessor myBean = new MyBeanProcessor();

from("file:data/inbound")
    .bean(myBean, "processBody")
    .to("file:data/outbound");
from("activemq:inboundData")
    .bean(myBean, "processBody")
    .to("activemq:outboundData");
```

Accessing overloaded bean methods

If a bean defines overloaded methods, you can choose which of the overloaded methods to invoke by specifying the method name along with its parameter types. For example, if the `MyBeanProcessor` class has two overloaded methods, `processBody(String)` and `processBody(String, String)`, you can invoke the latter overloaded method as follows:

```
from("file:data/inbound")
    .bean(MyBeanProcessor.class, "processBody(String,String)")
    .to("file:data/outbound");
```

Alternatively, if you want to identify a method by the number of parameters it takes, rather than specifying the type of each parameter explicitly, you can use the wildcard character, `*`. For example, to invoke a method named `processBody` that takes two parameters, irrespective of the exact type of the parameters, invoke the `bean()` processor as follows:

```
from("file:data/inbound")
    .bean(MyBeanProcessor.class, "processBody(*,*)")
    .to("file:data/outbound");
```

When specifying the method, you can use either a simple unqualified type name—for example, `processBody(Exchange)`—or a fully qualified type name—for example, `processBody(org.apache.camel.Exchange)`.



NOTE

In the current implementation, the specified type name must be an exact match of the parameter type. Type inheritance is not taken into account.

Specify parameters explicitly

You can specify parameter values explicitly, when you call the bean method. The following simple type values can be passed:

- Boolean: `true` or `false`.
- Numeric: `123`, `7`, and so on.

- String: **'In single quotes'** or **"In double quotes"**.
- Null object: **null**.

The following example shows how you can mix explicit parameter values with type specifiers in the same method invocation:

```
from("file:data/inbound")
    .bean(MyBeanProcessor.class, "processBody(String, 'Sample string value',
true, 7)")
    .to("file:data/outbound");
```

In the preceding example, the value of the first parameter would presumably be determined by a parameter binding annotation (see [the section called “Basic annotations”](#)).

In addition to the simple type values, you can also specify parameter values using the Simple language ([chapter “The Simple Language” in “Routing Expression and Predicate Languages”](#)). This means that the *full power of the Simple language is available* when specifying parameter values. For example, to pass the message body and the value of the **title** header to a bean method:

```
from("file:data/inbound")
    .bean(MyBeanProcessor.class,
"processBodyAndHeader(${body},${header.title})")
    .to("file:data/outbound");
```

You can also pass the entire header hash map as a parameter. For example, in the following example, the second method parameter must be declared to be of type **java.util.Map**:

```
from("file:data/inbound")
    .bean(MyBeanProcessor.class,
"processBodyAndAllHeaders(${body},${header})")
    .to("file:data/outbound");
```

Basic method signatures

To bind exchanges to a bean method, you can define a method signature that conforms to certain conventions. In particular, there are two basic conventions for method signatures:

- [the section called “Method signature for processing message bodies”](#).
- [the section called “Method signature for processing exchanges”](#).

Method signature for processing message bodies

If you want to implement a bean method that accesses or modifies the incoming message body, you must define a method signature that takes a single **String** argument and returns a **String** value. For example:

```
// Java
package com.acme;

public class MyBeanProcessor {
    public String processBody(String body) {
        // Do whatever you like to 'body'...
```

```

        return newBody;
    }
}

```

Method signature for processing exchanges

For greater flexibility, you can implement a bean method that accesses the incoming exchange. This enables you to access or modify all headers, bodies, and exchange properties. For processing exchanges, the method signature takes a single **org.apache.camel.Exchange** parameter and returns **void**. For example:

```

// Java
package com.acme;

public class MyBeanProcessor {
    public void processExchange(Exchange exchange) {
        // Do whatever you like to 'exchange'...
        exchange.getIn().setBody("Here is a new message body!");
    }
}

```

Accessing a bean created in Spring XML

Instead of creating a bean instance in Java, you can create an instance using Spring XML. In fact, this is the only feasible approach if you are defining your routes in XML. To define a bean in XML, use the standard Spring **bean** element. The following example shows how to create an instance of **MyBeanProcessor**:

```

<beans ...>
    ...
    <bean id="myBeanId" class="com.acme.MyBeanProcessor"/>
</beans>

```

It is also possible to pass data to the bean's constructor arguments using Spring syntax. For full details of how to use the Spring **bean** element, see [The IoC Container](#) from the Spring reference guide.

When you create an object instance using the **bean** element, you can reference it later using the bean's ID (the value of the **bean** element's **id** attribute). For example, given the **bean** element with ID equal to **myBeanId**, you can reference the bean in a Java DSL route using the **beanRef()** processor, as follows:

```

from("file:data/inbound").beanRef("myBeanId",
    "processBody").to("file:data/outbound");

```

Where the **beanRef()** processor invokes the **MyBeanProcessor.processBody()** method on the specified bean instance. You can also invoke the bean from within a Spring XML route, using the Camel schema's **bean** element. For example:

```

<camelContext id="CamelContextID"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file:data/inbound"/>
        <bean ref="myBeanId" method="processBody"/>
    </route>
</camelContext>

```

```

    <to uri="file:data/outbound"/>
  </route>
</camelContext>

```

Parameter binding annotations

The basic parameter bindings described in [the section called “Basic method signatures”](#) might not always be convenient to use. For example, if you have a legacy Java class that performs some data manipulation, you might want to extract data from an inbound exchange and map it to the arguments of an existing method signature. For this kind of parameter binding, Apache Camel provides the following kinds of Java annotation:

- [the section called “Basic annotations”](#).
- [the section called “Expression language annotations”](#).
- [the section called “Inherited annotations”](#).

Basic annotations

[Table 2.3, “Basic Bean Annotations”](#) shows the annotations from the `org.apache.camel` Java package that you can use to inject message data into the arguments of a bean method.

Table 2.3. Basic Bean Annotations

Annotation	Meaning	Parameter?
<code>@Attachments</code>	Binds to a list of attachments.	
<code>@Body</code>	Binds to an inbound message body.	
<code>@Header</code>	Binds to an inbound message header.	String name of the header.
<code>@Headers</code>	Binds to a <code>java.util.Map</code> of the inbound message headers.	
<code>@OutHeaders</code>	Binds to a <code>java.util.Map</code> of the outbound message headers.	
<code>@Property</code>	Binds to a named exchange property.	String name of the property.
<code>@Properties</code>	Binds to a <code>java.util.Map</code> of the exchange properties.	

For example, the following class shows you how to use basic annotations to inject message data into the `processExchange()` method arguments.

```

// Java
import org.apache.camel.*;

```

```

public class MyBeanProcessor {
    public void processExchange(
        @Header(name="user") String user,
        @Body String body,
        Exchange exchange
    ) {
        // Do whatever you like to 'exchange'...
        exchange.getIn().setBody(body + "UserName = " + user);
    }
}

```

Notice how you are able to mix the annotations with the default conventions. As well as injecting the annotated arguments, the parameter binding also automatically injects the exchange object into the `org.apache.camel.Exchange` argument.

Expression language annotations

The expression language annotations provide a powerful mechanism for injecting message data into a bean method's arguments. Using these annotations, you can invoke an arbitrary script, written in the scripting language of your choice, to extract data from an inbound exchange and inject the data into a method argument. [Table 2.4, “Expression Language Annotations”](#) shows the annotations from the `org.apache.camel.language` package (and sub-packages, for the non-core annotations) that you can use to inject message data into the arguments of a bean method.

Table 2.4. Expression Language Annotations

Annotation	Description
<code>@Bean</code>	Injects a Bean expression.
<code>@Constant</code>	Injects a Constant expression
<code>@EL</code>	Injects an EL expression.
<code>@Groovy</code>	Injects a Groovy expression.
<code>@Header</code>	Injects a Header expression.
<code>@JavaScript</code>	Injects a JavaScript expression.
<code>@OGNL</code>	Injects an OGNL expression.
<code>@PHP</code>	Injects a PHP expression.
<code>@Python</code>	Injects a Python expression.
<code>@Ruby</code>	Injects a Ruby expression.
<code>@Simple</code>	Injects a Simple expression.

Annotation	Description
@XPath	Injects an XPath expression.
@XQuery	Injects an XQuery expression.

For example, the following class shows you how to use the **@XPath** annotation to extract a username and a password from the body of an incoming message in XML format:

```
// Java
import org.apache.camel.language.*;

public class MyBeanProcessor {
    public void checkCredentials(
        @XPath("/credentials/username/text()") String user,
        @XPath("/credentials/password/text()") String pass
    ) {
        // Check the user/pass credentials...
        ...
    }
}
```

The **@Bean** annotation is a special case, because it enables you to inject the result of invoking a registered bean. For example, to inject a correlation ID into a method argument, you can use the **@Bean** annotation to invoke an ID generator class, as follows:

```
// Java
import org.apache.camel.language.*;

public class MyBeanProcessor {
    public void processCorrelatedMsg(
        @Bean("myCorrIdGenerator") String corrId,
        @Body String body
    ) {
        // Check the user/pass credentials...
        ...
    }
}
```

Where the string, **myCorrIdGenerator**, is the bean ID of the ID generator instance. The ID generator class can be instantiated using the spring **bean** element, as follows:

```
<beans ...>
    ...
    <bean id="myCorrIdGenerator" class="com.acme.MyIdGenerator"/>
</beans>
```

Where the **MySimpleIdGenerator** class could be defined as follows:

```
// Java
package com.acme;
```

```

public class MyIdGenerator {

    private UserManager userManager;

    public String generate(
        @Header(name = "user") String user,
        @Body String payload
    ) throws Exception {
        User user = userManager.lookupUser(user);
        String userId = user.getPrimaryId();
        String id = userId + generateHashCodeForPayload(payload);
        return id;
    }
}

```

Notice that you can also use annotations in the referenced bean class, **MyIdGenerator**. The only restriction on the **generate()** method signature is that it must return the correct type to inject into the argument annotated by **@Bean**. Because the **@Bean** annotation does not let you specify a method name, the injection mechanism simply invokes the first method in the referenced bean that has the matching return type.



NOTE

Some of the language annotations are available in the core component (**@Bean**, **@Constant**, **@Simple**, and **@XPath**). For non-core components, however, you will have to make sure that you load the relevant component. For example, to use the OGNL script, you must load the **camel-ognl** component.

Inherited annotations

Parameter binding annotations can be inherited from an interface or from a superclass. For example, if you define a Java interface with a **Header** annotation and a **Body** annotation, as follows:

```

// Java
import org.apache.camel.*;

public interface MyBeanProcessorIntf {
    void processExchange(
        @Header(name="user") String user,
        @Body String body,
        Exchange exchange
    );
}

```

The overloaded methods defined in the implementation class, **MyBeanProcessor**, now inherit the annotations defined in the base interface, as follows:

```

// Java
import org.apache.camel.*;

public class MyBeanProcessor implements MyBeanProcessorIntf {
    public void processExchange(
        String user, // Inherits Header annotation
        String body, // Inherits Body annotation

```



```

        Exchange exchange
    ) {
        ...
    }
}

```

Interface implementations

The class that implements a Java interface is often **protected**, **private** or in **package-only** scope. If you try to invoke a method on an implementation class that is restricted in this way, the bean binding falls back to invoking the corresponding interface method, which is publicly accessible.

For example, consider the following public **BeanIntf** interface:

```

// Java
public interface BeanIntf {
    void processBodyAndHeader(String body, String title);
}

```

Where the **BeanIntf** interface is implemented by the following protected **BeanIntfImpl** class:

```

// Java
protected class BeanIntfImpl implements BeanIntf {
    void processBodyAndHeader(String body, String title) {
        ...
    }
}

```

The following bean invocation would fall back to invoking the public **BeanIntf.processBodyAndHeader** method:

```

from("file:data/inbound")
    .bean(BeanIntfImpl.class, "processBodyAndHeader(${body},
${header.title}")
    .to("file:data/outbound");

```

Invoking static methods

Bean integration has the capability to invoke static methods *without* creating an instance of the associated class. For example, consider the following Java class that defines the static method, **changeSomething()**:

```

// Java
...
public final class MyStaticClass {
    private MyStaticClass() {
    }

    public static String changeSomething(String s) {
        if ("Hello World".equals(s)) {
            return "Bye World";
        }
        return null;
    }
}

```

```

    }

    public void doSomething() {
        // noop
    }
}

```

You can use bean integration to invoke the static `changeSomething` method, as follows:

```

from("direct:a")
  .bean(MyStaticClass.class, "changeSomething")
  .to("mock:a");

```

Note that, although this syntax looks identical to the invocation of an ordinary function, bean integration exploits Java reflection to identify the method as static and proceeds to invoke the method *without* instantiating `MyStaticClass`.

Invoking an OSGi service

In the special case where a route is deployed into a Red Hat JBoss Fuse container, it is possible to invoke an OSGi service directly using bean integration. For example, assuming that one of the bundles in the OSGi container has exported the service,

`org.fusesource.example.HelloWorldOsgiService`, you could invoke the `sayHello` method using the following bean integration code:

```

from("file:data/inbound")
  .bean(org.fusesource.example.HelloWorldOsgiService.class, "sayHello")
  .to("file:data/outbound");

```

You could also invoke the OSGi service from within a Spring or blueprint XML file, using the bean component, as follows:

```

<to uri="bean:org.fusesource.example.HelloWorldOsgiService?
method=sayHello"/>

```

The way this works is that Apache Camel sets up a chain of registries when it is deployed in the OSGi container. First of all, it looks up the specified class name in the OSGi service registry; if this lookup fails, it then falls back to the local Spring DM or blueprint registry.

2.5. ASPECT ORIENTED PROGRAMMING

Overview

The aspect oriented programming (AOP) feature in Apache Camel enables you to apply *before* and *after* processing to a specified portion of a route. As a matter of fact, AOP does *not* provide anything that you could not do with the regular route syntax. The advantage of the AOP syntax, however, is that it enables you to specify before and after processing at a *single point* in the route. In some cases, this gives a more readable syntax. The typical use case for AOP is the application of a symmetrical pair of operations before and after a route fragment is processed. For example, typical pairs of operations that you might want to apply using AOP are: encrypt and decrypt; begin transaction and commit transaction; allocate resources and deallocate resources; and so on.

Java DSL example

In Java DSL, the route fragment to which you apply before and after processing is bracketed between **aop()** and **end()**. For example, the following route performs AOP processing around the route fragment that calls the bean methods:

```
from("jms:queue:inbox")
    .aop().around("log:before", "log:after")
        .to("bean:order?method=validate")
        .to("bean:order?method=handle")
    .end()
    .to("jms:queue:order");
```

Where the **around()** subclause specifies an endpoint, **log:before**, where the exchange is routed *before* processing the route fragment and an endpoint, **log:after**, where the exchange is routed *after* processing the route fragment.

AOP options in the Java DSL

Starting an AOP block with **aop().around()** is probably the most common use case, but the AOP block supports other subclauses, as follows:

- **around()**—specifies *before* and *after* endpoints.
- **begin()**—specifies *before* endpoint only.
- **after()**—specifies *after* endpoint only.
- **aroundFinally()**—specifies a *before* endpoint, and an *after* endpoint that is always called, even when an exception occurs in the enclosed route fragment.
- **afterFinally()**—specifies an *after* endpoint that is always called, even when an exception occurs in the enclosed route fragment.

Spring XML example

In the XML DSL, the route fragment to which you apply *before* and *after* processing is enclosed in the **aop** element. For example, the following Spring XML route performs AOP processing around the route fragment that calls the bean methods:

```
<route>
  <from uri="jms:queue:inbox"/>
  <aop beforeUri="log:before" afterUri="log:after">
    <to uri="bean:order?method=validate"/>
    <to uri="bean:order?method=handle"/>
  </aop>
  <to uri="jms:queue:order"/>
</route>
```

Where the **beforeUri** attribute specifies the endpoint where the exchange is routed *before* processing the route fragment, and the **afterUri** attribute specifies the endpoint where the exchange is routed *after* processing the route fragment.

AOP options in the Spring XML

The **aop** element supports the following optional attributes:

- `beforeUri`
- `afterUri`
- `afterFinallyUri`

The various use cases described for the Java DSL can be obtained in Spring XML using the appropriate combinations of these attributes. For example, the `aroundFinally()` Java DSL subclass is equivalent to the combination of `beforeUri` and `afterFinallyUri` in Spring XML.

2.6. TRANSFORMING MESSAGE CONTENT

Overview

Apache Camel supports a variety of approaches to transforming message content. In addition to a simple native API for modifying message content, Apache Camel supports integration with several different third-party libraries and transformation standards. The following kinds of transformations are discussed in this section:

- [the section called “Simple transformations”](#).
- [the section called “Marshalling and unmarshalling”](#).

Simple transformations

The Java DSL has a built-in API that enables you to perform simple transformations on incoming and outgoing messages. For example, the rule shown in [Example 2.1, “Simple Transformation of Incoming Messages”](#) appends the text, `world!`, to the end of the incoming message body.

Example 2.1. Simple Transformation of Incoming Messages

```
from("SourceURL").setBody(body().append(" world!")).to("TargetURL");
```

Where the `setBody()` command replaces the content of the incoming message's body. You can use the following API classes to perform simple transformations of the message content in a router rule:

- `org.apache.camel.model.ProcessorDefinition`
- `org.apache.camel.builder.Builder`
- `org.apache.camel.builder.ValueBuilder`

ProcessorDefinition class

The `org.apache.camel.model.ProcessorDefinition` class defines the DSL commands you can insert directly into a router rule—for example, the `setBody()` command in [Example 2.1, “Simple Transformation of Incoming Messages”](#). [Table 2.5, “Transformation Methods from the ProcessorDefinition Class”](#) shows the `ProcessorDefinition` methods that are relevant to transforming message content:

Table 2.5. Transformation Methods from the ProcessorDefinition Class

Method	Description
Type <code>convertBodyTo(Class type)</code>	Converts the IN message body to the specified type.
Type <code>removeFaultHeader(String name)</code>	Adds a processor which removes the header on the FAULT message.
Type <code>removeHeader(String name)</code>	Adds a processor which removes the header on the IN message.
Type <code>removeProperty(String name)</code>	Adds a processor which removes the exchange property.
ExpressionClause<ProcessorDefinition <Type>> <code>setBody()</code>	Adds a processor which sets the body on the IN message.
Type <code>setFaultBody(Expression expression)</code>	Adds a processor which sets the body on the FAULT message.
Type <code>setFaultHeader(String name, Expression expression)</code>	Adds a processor which sets the header on the FAULT message.
ExpressionClause<ProcessorDefinition <Type>> <code>setHeader(String name)</code>	Adds a processor which sets the header on the IN message.
Type <code>setHeader(String name, Expression expression)</code>	Adds a processor which sets the header on the IN message.
ExpressionClause<ProcessorDefinition <Type>> <code>setOutHeader(String name)</code>	Adds a processor which sets the header on the OUT message.
Type <code>setOutHeader(String name, Expression expression)</code>	Adds a processor which sets the header on the OUT message.
ExpressionClause<ProcessorDefinition <Type>> <code>setProperty(String name)</code>	Adds a processor which sets the exchange property.
Type <code>setProperty(String name, Expression expression)</code>	Adds a processor which sets the exchange property.
ExpressionClause<ProcessorDefinition <Type>> <code>transform()</code>	Adds a processor which sets the body on the OUT message.
Type <code>transform(Expression expression)</code>	Adds a processor which sets the body on the OUT message.

Builder class

The `org.apache.camel.builder.Builder` class provides access to message content in contexts

where expressions or predicates are expected. In other words, **Builder** methods are typically invoked in the *arguments* of DSL commands—for example, the **body()** command in [Example 2.1, “Simple Transformation of Incoming Messages”](#). [Table 2.6, “Methods from the Builder Class”](#) summarizes the static methods available in the **Builder** class.

Table 2.6. Methods from the Builder Class

Method	Description
<code>static <E extends Exchange> ValueBuilder<E> body()</code>	Returns a predicate and value builder for the inbound body on an exchange.
<code>static <E extends Exchange, T> ValueBuilder<E> bodyAs(Class<T> type)</code>	Returns a predicate and value builder for the inbound message body as a specific type.
<code>static <E extends Exchange> ValueBuilder<E> constant(Object value)</code>	Returns a constant expression.
<code>static <E extends Exchange> ValueBuilder<E> faultBody()</code>	Returns a predicate and value builder for the fault body on an exchange.
<code>static <E extends Exchange, T> ValueBuilder<E> faultBodyAs(Class<T> type)</code>	Returns a predicate and value builder for the fault message body as a specific type.
<code>static <E extends Exchange> ValueBuilder<E> header(String name)</code>	Returns a predicate and value builder for headers on an exchange.
<code>static <E extends Exchange> ValueBuilder<E> outBody()</code>	Returns a predicate and value builder for the outbound body on an exchange.
<code>static <E extends Exchange> ValueBuilder<E> outBodyAs(Class<T> type)</code>	Returns a predicate and value builder for the outbound message body as a specific type.
<code>static ValueBuilder property(String name)</code>	Returns a predicate and value builder for properties on an exchange.
<code>static ValueBuilder regexReplaceAll(Expression content, String regex, Expression replacement)</code>	Returns an expression that replaces all occurrences of the regular expression with the given replacement.
<code>static ValueBuilder regexReplaceAll(Expression content, String regex, String replacement)</code>	Returns an expression that replaces all occurrences of the regular expression with the given replacement.
<code>static ValueBuilder sendTo(String uri)</code>	Returns an expression processing the exchange to the given endpoint uri.

Method	Description
<code>static <E extends Exchange> ValueBuilder<E> systemProperty(String name)</code>	Returns an expression for the given system property.
<code>static <E extends Exchange> ValueBuilder<E> systemProperty(String name, String defaultValue)</code>	Returns an expression for the given system property.

ValueBuilder class

The `org.apache.camel.builder.ValueBuilder` class enables you to modify values returned by the `Builder` methods. In other words, the methods in `ValueBuilder` provide a simple way of modifying message content. [Table 2.7, “Modifier Methods from the ValueBuilder Class”](#) summarizes the methods available in the `ValueBuilder` class. That is, the table shows only the methods that are used to modify the value they are invoked on (for full details, see the *API Reference* documentation).

Table 2.7. Modifier Methods from the ValueBuilder Class

Method	Description
<code>ValueBuilder<E> append(Object value)</code>	Appends the string evaluation of this expression with the given value.
<code>Predicate contains(Object value)</code>	Create a predicate that the left hand expression contains the value of the right hand expression.
<code>ValueBuilder<E> convertTo(Class type)</code>	Converts the current value to the given type using the registered type converters.
<code>ValueBuilder<E> convertToString()</code>	Converts the current value a String using the registered type converters.
<code>Predicate endsWith(Object value)</code>	
<code><T> T evaluate(Exchange exchange, Class<T> type)</code>	
<code>Predicate in(Object... values)</code>	
<code>Predicate in(Predicate... predicates)</code>	
<code>Predicate isEqualTo(Object value)</code>	Returns true, if the current value is equal to the given value argument.

Method	Description
Predicate isGreaterThan(Object value)	Returns true, if the current value is greater than the given value argument.
Predicate isGreaterThanOrEqualTo(Object value)	Returns true, if the current value is greater than or equal to the given value argument.
Predicate isInstanceOf(Class type)	Returns true, if the current value is an instance of the given type.
Predicate isLessThan(Object value)	Returns true, if the current value is less than the given value argument.
Predicate isLessThanOrEqualTo(Object value)	Returns true, if the current value is less than or equal to the given value argument.
Predicate isNotEqualTo(Object value)	Returns true, if the current value is not equal to the given value argument.
Predicate isNotNull()	Returns true, if the current value is not null .
Predicate isNull()	Returns true, if the current value is null .
Predicate matches(Expression expression)	
Predicate not(Predicate predicate)	Negates the predicate argument.
ValueBuilder prepend(Object value)	Prepends the string evaluation of this expression to the given value.
Predicate regex(String regex)	
ValueBuilder<E> regexReplaceAll(String regex, Expression<E> replacement)	Replaces all occurrences of the regular expression with the given replacement.
ValueBuilder<E> regexReplaceAll(String regex, String replacement)	Replaces all occurrences of the regular expression with the given replacement.
ValueBuilder<E> regexTokenize(String regex)	Tokenizes the string conversion of this expression using the given regular expression.
ValueBuilder sort(Comparator comparator)	Sorts the current value using the given comparator.

Method	Description
Predicate startsWith(Object value)	Returns true, if the current value matches the string value of the value argument.
ValueBuilder<E> tokenize()	Tokenizes the string conversion of this expression using the comma token separator.
ValueBuilder<E> tokenize(String token)	Tokenizes the string conversion of this expression using the given token separator.

Marshalling and unmarshalling

You can convert between low-level and high-level message formats using the following commands:

- **marshal()**— Converts a high-level data format to a low-level data format.
- **unmarshal()** — Converts a low-level data format to a high-level data format.

Apache Camel supports marshalling and unmarshalling of the following data formats:

- *Java serialization*— Enables you to convert a Java object to a blob of binary data. For this data format, unmarshalling converts a binary blob to a Java object, and marshalling converts a Java object to a binary blob. For example, to read a serialized Java object from an endpoint, *SourceURL*, and convert it to a Java object, you use a rule like the following:

```
from("SourceURL").unmarshal().serialization()
.<FurtherProcessing>.to("TargetURL");
```

Or alternatively, in Spring XML:

```
<camelContext id="serialization"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <serialization/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

- *JAXB*— Provides a mapping between XML schema types and Java types (see <https://jaxb.dev.java.net/>). For JAXB, unmarshalling converts an XML data type to a Java object, and marshalling converts a Java object to an XML data type. Before you can use JAXB data formats, you must compile your XML schema using a JAXB compiler to generate the Java classes that represent the XML data types in the schema. This is called *binding* the schema. After the schema is bound, you define a rule to unmarshal XML data to a Java object, using code like the following:

```
org.apache.camel.spi.DataFormat jaxb = new
org.apache.camel.model.dataformat.JaxbDataFormat("GeneratedPackageNa
```

```
me");
from("SourceURL").unmarshal(jaxb)
.<FurtherProcessing>.to("TargetURL");
```

where *GeneratedPackagename* is the name of the Java package generated by the JAXB compiler, which contains the Java classes representing your XML schema.

Or alternatively, in Spring XML:

```
<camelContext id="jaxb"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <jaxb prettyPrint="true" contextPath="GeneratedPackageName"/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

- *XMLBeans* — Provides an alternative mapping between XML schema types and Java types (see <http://xmlbeans.apache.org/>). For XMLBeans, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XMLBeans, you use code like the following:

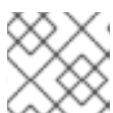
```
from("SourceURL").unmarshal().xmlBeans()
.<FurtherProcessing>.to("TargetURL");
```

Or alternatively, in Spring XML:

```
<camelContext id="xmlBeans"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <xmlBeans prettyPrint="true"/>
    </unmarshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

- *XStream* — Provides another mapping between XML types and Java types (see <http://xstream.codehaus.org/>). XStream is a serialization library (like Java serialization), enabling you to convert any Java object to XML. For XStream, unmarshalling converts an XML data type to a Java object, and marshalling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XStream, you use code like the following:

```
from("SourceURL").unmarshal().xstream()
.<FurtherProcessing>.to("TargetURL");
```



NOTE

The XStream data format is currently *not* supported in Spring XML.

2.7. PROPERTY PLACEHOLDERS

Overview

The property placeholders feature can be used to substitute strings into various contexts (such as endpoint URIs and attributes in XML DSL elements), where the placeholder settings are stored in Java properties files. This feature can be useful, if you want to share settings between different Apache Camel applications or if you want to centralize certain configuration settings.

For example, the following route sends requests to a Web server, whose host and port are substituted by the placeholders, `{{remote.host}}` and `{{remote.port}}`:

```
from("direct:start").to("http://{{remote.host}}:{{remote.port}}");
```

The placeholder values are defined in a Java properties file, as follows:

```
# Java properties file
remote.host=myserver.com
remote.port=8080
```

Property files

Property settings are stored in one or more Java properties files and must conform to the standard Java properties file format. Each property setting appears on its own line, in the format **Key=Value**. Lines with `#` or `!` as the first non-blank character are treated as comments.

For example, a property file could have content as shown in [Example 2.2, “Sample Property File”](#).

Example 2.2. Sample Property File

```
# Property placeholder settings
# (in Java properties file format)
cool.end=mock:result
cool.result=result
cool.concat=mock:{{cool.result}}
cool.start=direct:cool
cool.showid=true

cheese.end=mock:cheese
cheese.quote=Camel rocks
cheese.type=Gouda

bean.foo=foo
bean.bar=bar
```

Resolving properties

The properties component must be configured with the locations of one or more property files before you can start using it in route definitions. You must provide the property values using one of the following resolvers:

```
classpath:PathName,PathName,...
```

(Default) Specifies locations on the classpath, where *PathName* is a file pathname delimited using forward slashes.

file:*PathName, PathName, ...*

Specifies locations on the file system, where *PathName* is a file pathname delimited using forward slashes.

ref:*BeanID*

Specifies the ID of a `java.util.Properties` object in the registry.

blueprint:*BeanID*

Specifies the ID of a `cm:property-placeholder` bean, which is used in the context of an OSGi blueprint file to access properties defined in the *OSGi Configuration Admin* service. For details, see [the section called “Integration with OSGi blueprint property placeholders”](#).

For example, to specify the `com/fusesource/cheese.properties` property file and the `com/fusesource/bar.properties` property file, both located on the classpath, you would use the following location string:

```
classpath:com/fusesource/cheese.properties,com/fusesource/bar.properties
```



NOTE

You can omit the `classpath:` prefix in this example, because the classpath resolver is used by default.

Specifying locations using system properties and environment variables

You can embed Java system properties and O/S environment variables in a location *PathName*.

Java system properties can be embedded in a location resolver using the syntax, `${PropertyName}`. For example, if the root directory of Red Hat JBoss Fuse is stored in the Java system property, `karaf.home`, you could embed that directory value in a file location, as follows:

```
file:${karaf.home}/etc/foo.properties
```

O/S environment variables can be embedded in a location resolver using the syntax, `${env:VarName}`. For example, if the root directory of JBoss Fuse is stored in the environment variable, `SMX_HOME`, you could embed that directory value in a file location, as follows:

```
file:${env:SMX_HOME}/etc/foo.properties
```

Configuring the properties component

Before you can start using property placeholders, you must configure the properties component, specifying the locations of one or more property files.

In the Java DSL, you can configure the properties component with the property file locations, as follows:

```
// Java
```

```
import org.apache.camel.component.properties.PropertiesComponent;
...
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation("com/fusesource/cheese.properties,com/fusesource/bar.properties");
context.addComponent("properties", pc);
```

As shown in the `addComponent()` call, the name of the properties component *must* be set to **properties**.

In the XML DSL, you can configure the properties component using the dedicated **propertyPlaceholder** element, as follows:

```
<camelContext ...>
  <propertyPlaceholder
    id="properties"

    location="com/fusesource/cheese.properties,com/fusesource/bar.properties"
  />
</camelContext>
```

If you want the properties component to ignore any missing **.properties** files when it is being initialized, you can set the **ignoreMissingLocation** option to **true** (normally, a missing **.properties** file would result in an error being raised).

Placeholder syntax

After it is configured, the property component automatically substitutes placeholders (in the appropriate contexts). The syntax of a placeholder depends on the context, as follows:

- *In endpoint URIs and in Spring XML files*—the placeholder is specified as **{{Key}}**.
- *When setting XML DSL attributes*—**xs:string** attributes are set using the following syntax:

```
AttributeName="{{Key}}"
```

Other attribute types (for example, **xs:int** or **xs:boolean**) must be set using the following syntax:

```
prop:AttributeName="Key"
```

Where **prop** is associated with the **http://camel.apache.org/schema/placeholder** namespace.

- *When setting Java DSL EIP options*—to set an option on an Enterprise Integration Pattern (EIP) command in the Java DSL, add a **placeholder()** clause like the following to the fluent DSL:

```
.placeholder("OptionName", "Key")
```

- *In Simple language expressions*—the placeholder is specified as **\${properties:Key}**.

Substitution in endpoint URIs

Wherever an endpoint URI string appears in a route, the first step in parsing the endpoint URI is to apply the property placeholder parser. The placeholder parser automatically substitutes any property names appearing between double braces, `{{Key}}`. For example, given the property settings shown in [Example 2.2, "Sample Property File"](#), you could define a route as follows:

```
from("{{cool.start}}")
    .to("log:{{cool.start}}?showBodyType=false&showExchangeId={{cool.showid}}")
    .to("mock:{{cool.result}}");
```

By default, the placeholder parser looks up the **properties** bean ID in the registry to find the property component. If you prefer, you can explicitly specify the scheme in the endpoint URIs. For example, by prefixing **properties:** to each of the endpoint URIs, you can define the following equivalent route:

```
from("properties:{{cool.start}}")
    .to("properties:log:{{cool.start}}?showBodyType=false&showExchangeId={{cool.showid}}")
    .to("properties:mock:{{cool.result}}");
```

When specifying the scheme explicitly, you also have the option of specifying options to the properties component. For example, to override the property file location, you could set the **location** option as follows:

```
from("direct:start").to("properties:{{bar.end}}?
location=com/mycompany/bar.properties");
```

Substitution in Spring XML files

You can also use property placeholders in the XML DSL, for setting various attributes of the DSL elements. In this context, the placeholder syntax also uses double braces, `{{Key}}`. For example, you could define a **jmxAgent** element using property placeholders, as follows:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <propertyPlaceholder id="properties"
location="org/apache/camel/spring/jmx.properties"/>

  <!-- we can use property placeholders when we define the JMX agent -->
  <jmxAgent id="agent" registryPort="{{myjmx.port}}"
    usePlatformMBeanServer="{{myjmx.usePlatform}}"
    createConnector="true"
    statisticsLevel="RoutesOnly"
  />

  <route>
    <from uri="seda:start"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

Substitution of XML DSL attribute values

You can use the regular placeholder syntax for specifying attribute values of **xs:string** type—for example, `<jmxAgent registryPort="{{myjmx.port}}"` ...>. But for attributes of any other

type (for example, `xs:int` or `xs:boolean`), you must use the special syntax, `prop:AttributeName="Key"`.

For example, given that a property file defines the `stop.flag` property to have the value, `true`, you can use this property to set the `stopOnException` boolean attribute, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:prop="http://camel.apache.org/schema/placeholder"
      ... >

  <bean id="illegal" class="java.lang.IllegalArgumentException">
    <constructor-arg index="0" value="Good grief!"/>
  </bean>

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

    <propertyPlaceholder id="properties"

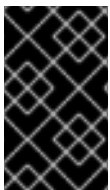
location="classpath:org/apache/camel/component/properties/myprop.properties"
    >

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

    <route>
      <from uri="direct:start"/>
      <multicast prop:stopOnException="stop.flag">
        <to uri="mock:a"/>
        <throwException ref="damn"/>
        <to uri="mock:b"/>
      </multicast>
    </route>

  </camelContext>

</beans>
```



IMPORTANT

The `prop` prefix must be explicitly assigned to the `http://camel.apache.org/schema/placeholder` namespace in your Spring file, as shown in the `beans` element of the preceding example.

Substitution of Java DSL EIP options

When invoking an EIP command in the Java DSL, you can set any EIP option using the value of a property placeholder, by adding a sub-clause of the form, `placeholder("OptionName", "Key")`.

For example, given that a property file defines the `stop.flag` property to have the value, `true`, you can use this property to set the `stopOnException` option of the multicast EIP, as follows:

```
from("direct:start")
  .multicast().placeholder("stopOnException", "stop.flag")
  .to("mock:a").throwException(new
```

```
IllegalAccessException("Damn")).to("mock:b");
```

Substitution in Simple language expressions

You can also substitute property placeholders in Simple language expressions, but in this case the syntax of the placeholder is `${properties:Key}`. For example, you can substitute the `cheese.quote` placeholder inside a Simple expression, as follows:

```
from("direct:start")
    .transform().simple("Hi ${body} do you think
    ${properties:cheese.quote}?");
```

It is also possible to override the location of the property file using the syntax, `${properties:Location:Key}`. For example, to substitute the `bar.quote` placeholder using the settings from the `com/mycompany/bar.properties` property file, you can define a Simple expression as follows:

```
from("direct:start")
    .transform().simple("Hi ${body}.
    ${properties:com/mycompany/bar.properties:bar.quote}.");
```

Integration with OSGi blueprint property placeholders

If you deploy your route into the Red Hat JBoss Fuse OSGi container, you can integrate the Apache Camel property placeholder mechanism with JBoss Fuse's blueprint property placeholder mechanism (in fact, the integration is enabled by default). There are two basic approaches to setting up the integration, as follows:

- [the section called "Implicit blueprint integration"](#).
- [the section called "Explicit blueprint integration"](#).

Implicit blueprint integration

If you define a `camelContext` element inside an OSGi blueprint file, the Apache Camel property placeholder mechanism automatically integrates with the blueprint property placeholder mechanism. That is, placeholders obeying the Apache Camel syntax (for example, `{{cool.end}}`) that appear within the scope of `camelContext` are implicitly resolved by looking up the *blueprint property placeholder* mechanism.

For example, consider the following route defined in an OSGi blueprint file, where the last endpoint in the route is defined by the property placeholder, `{{result}}`:

```
<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="myblueprint.placeholder" persistent-
```



```

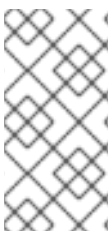
id="camel.blueprint">
  <!-- list some properties for this test -->
  <cm:default-properties>
    <cm:property name="result" value="mock:result"/>
  </cm:default-properties>
</cm:property-placeholder>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <!-- in the route we can use {{ }} placeholders which will look up
in blueprint,
    as Camel will auto detect the OSGi blueprint property
placeholder and use it -->
    <route>
      <from uri="direct:start"/>
      <to uri="mock:foo"/>
      <to uri="{{result}}"/>
    </route>
  </camelContext>
</blueprint>

```

The blueprint property placeholder mechanism is initialized by creating a **cm:property-placeholder** bean. In the preceding example, the **cm:property-placeholder** bean is associated with the **camel.blueprint** persistent ID, where a persistent ID is the standard way of referencing a group of related properties from the *OSGi Configuration Admin* service. In other words, the **cm:property-placeholder** bean provides access to all of the properties defined under the **camel.blueprint** persistent ID. It is also possible to specify default values for some of the properties (using the nested **cm:property** elements).

In the context of blueprint, the Apache Camel placeholder mechanism searches for an instance of **cm:property-placeholder** in the bean registry. If it finds such an instance, it automatically integrates the Apache Camel placeholder mechanism, so that placeholders like, **{{result}}**, are resolved by looking up the key in the blueprint property placeholder mechanism (in this example, through the **myblueprint.placeholder** bean).



NOTE

The default blueprint placeholder syntax (accessing the blueprint properties directly) is **\${Key}**. Hence, *outside* the scope of a **camelContext** element, the placeholder syntax you must use is **\${Key}**. Whereas, *inside* the scope of a **camelContext** element, the placeholder syntax you must use is **{{Key}}**.

Explicit blueprint integration

If you want to have more control over where the Apache Camel property placeholder mechanism finds its properties, you can define a **propertyPlaceholder** element and specify the resolver locations explicitly.

For example, consider the following blueprint configuration, which differs from the previous example in that it creates an explicit **propertyPlaceholder** instance:

```

<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="myblueprint.placeholder" persistent-
id="camel.blueprint">
        <!-- list some properties for this test -->
        <cm:default-properties>
            <cm:property name="result" value="mock:result"/>
        </cm:default-properties>
    </cm:property-placeholder>

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

        <!-- using Camel properties component and refer to the blueprint
property placeholder by its id -->
        <propertyPlaceholder id="properties"
location="blueprint:myblueprint.placeholder"/>

        <!-- in the route we can use {{ }} placeholders which will lookup
in blueprint -->
        <route>
            <from uri="direct:start"/>
            <to uri="mock:foo"/>
            <to uri="{{result}}"/>
        </route>

    </camelContext>

</blueprint>

```

In the preceding example, the **propertyPlaceholder** element specifies explicitly which **cm:property-placeholder** bean to use by setting the location to **blueprint:myblueprint.placeholder**. That is, the **blueprint:** resolver explicitly references the ID, **myblueprint.placeholder**, of the **cm:property-placeholder** bean.

This style of configuration is useful, if there is more than one **cm:property-placeholder** bean defined in the blueprint file and you need to specify which one to use. It also makes it possible to source properties from multiple locations, by specifying a comma-separated list of locations. For example, if you wanted to look up properties both from the **cm:property-placeholder** bean and from the properties file, **myproperties.properties**, on the classpath, you could define the **propertyPlaceholder** element as follows:

```

<propertyPlaceholder id="properties"

location="blueprint:myblueprint.placeholder,classpath:myproperties.properties"/>

```

Integration with Spring property placeholders

If you define your Apache Camel application using XML DSL in a Spring XML file, you can integrate the Apache Camel property placeholder mechanism with Spring property placeholder mechanism by declaring a Spring bean of type,

org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer.

Define a **BridgePropertyPlaceholderConfigurer**, which replaces both Apache Camel's **propertyPlaceholder** element and Spring's **ctx:property-placeholder** element in the Spring XML file. You can then refer to the configured properties using either the Spring **#{PropName}** syntax or the Apache Camel **{{PropName}}** syntax.

For example, defining a bridge property placeholder that reads its property settings from the **cheese.properties** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<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-compendium"
  xmlns:ctx="http://www.springframework.org/schema/context"
  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://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/osgi-compendium
http://www.springframework.org/schema/osgi-compendium/spring-osgi-
compendium.xsd
">

  <!-- Bridge Spring property placeholder with Camel -->
  <!-- Do not use <ctx:property-placeholder ... > at the same time -->
  <bean id="bridgePropertyPlaceholder"

class="org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer">
  <property name="location"

value="classpath:org/apache/camel/component/properties/cheese.properties"/
>
  </bean>

  <!-- A bean that uses Spring property placeholder -->
  <!-- The ${hi} is a spring property placeholder -->
  <bean id="hello"
class="org.apache.camel.component.properties.HelloBean">
  <property name="greeting" value="${hi}"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
  <!-- Use Camel's property placeholder {{ }} style -->
  <route>
    <from uri="direct:{{cool.bar}}"/>
    <bean ref="hello"/>
    <to uri="{{cool.end}}"/>
  </route>
  </camelContext>

</beans>
```

2.8. THREADING MODEL

Java thread pool API

The Apache Camel threading model is based on the powerful Java concurrency API, [java.util.concurrent](#), that first became available in Sun's JDK 1.5. The key interface in this API is the **ExecutorService** interface, which represents a thread pool. Using the concurrency API, you can create many different kinds of thread pool, covering a wide range of scenarios.

Apache Camel thread pool API

The Apache Camel thread pool API builds on the Java concurrency API by providing a central factory (of **org.apache.camel.spi.ExecutorServiceManager** type) for all of the thread pools in your Apache Camel application. Centralising the creation of thread pools in this way provides several advantages, including:

- Simplified creation of thread pools, using utility classes.
- Integrating thread pools with graceful shutdown.
- Threads automatically given informative names, which is beneficial for logging and management.

Component threading model

Some Apache Camel components—such as SEDA, JMS, and Jetty—are inherently multi-threaded. These components have all been implemented using the Apache Camel threading model and thread pool API.

If you are planning to implement your own Apache Camel component, it is recommended that you integrate your threading code with the Apache Camel threading model. For example, if your component needs a thread pool, it is recommended that you create it using the CamelContext's **ExecutorServiceManager** object.

Processor threading model

Some of the standard processors in Apache Camel create their own thread pool by default. These threading-aware processors are also integrated with the Apache Camel threading model and they provide various options that enable you to customize customize the thread pools that they use.

[Table 2.8, “Processor Threading Options”](#) shows the various options for controlling and setting thread pools on the threading-aware processors built-in to Apache Camel.

Table 2.8. Processor Threading Options

Processor	Java DSL	XML DSL
aggregate	<pre>parallelProcessing() executorService() executorServiceRef()</pre>	<pre>@parallelProcessing @executorServiceRef</pre>

Processor	Java DSL	XML DSL
multicast	<pre>parallelProcessing() executorService() executorServiceRef()</pre>	<pre>@parallelProcessing @executorServiceRef</pre>
recipientList	<pre>parallelProcessing() executorService() executorServiceRef()</pre>	<pre>@parallelProcessing @executorServiceRef</pre>
split	<pre>parallelProcessing() executorService() executorServiceRef()</pre>	<pre>@parallelProcessing @executorServiceRef</pre>
threads	<pre>executorService() executorServiceRef() poolSize() maxPoolSize() keepAliveTime() timeUnit() maxQueueSize() rejectedPolicy()</pre>	<pre>@executorServiceRef @poolSize @maxPoolSize @keepAliveTime @timeUnit @maxQueueSize @rejectedPolicy</pre>
wireTap	<pre>wireTap(String uri, ExecutorService executorService) wireTap(String uri, String executorServiceRef)</pre>	<pre>@executorServiceRef</pre>

Creating a default thread pool

To create a default thread pool for one of the threading-aware processors, enable the **parallelProcessing** option, using the **parallelProcessing()** sub-clause, in the Java DSL, or the **parallelProcessing** attribute, in the XML DSL.

For example, in the Java DSL, you can invoke the multicast processor with a default thread pool (where the thread pool is used to process the multicast destinations concurrently) as follows:

```
from("direct:start")
  .multicast().parallelProcessing()
    .to("mock:first")
    .to("mock:second")
    .to("mock:third");
```

You can define the same route in XML DSL as follows

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <multicast parallelProcessing="true">
      <to uri="mock:first"/>
      <to uri="mock:second"/>
      <to uri="mock:third"/>
    </multicast>
  </route>
</camelContext>
```

Default thread pool profile settings

The default thread pools are automatically created by a thread factory that takes its settings from the *default thread pool profile*. The default thread pool profile has the settings shown in [Table 2.9, “Default Thread Pool Profile Settings”](#) (assuming that these settings have not been modified by the application code).

Table 2.9. Default Thread Pool Profile Settings

Thread Option	Default Value
maxQueueSize	1000
poolSize	10
maxPoolSize	20
keepAliveTime	60 (seconds)
rejectedPolicy	CallerRuns

Changing the default thread pool profile

It is possible to change the default thread pool profile settings, so that all subsequent default thread pools will be created with the custom settings. You can change the profile either in Java or in Spring XML.

For example, in the Java DSL, you can customize the **poolSize** option and the **maxQueueSize** option in the default thread pool profile, as follows:

```
// Java
import org.apache.camel.spi.ExecutorServiceManager;
import org.apache.camel.spi.ThreadPoolProfile;
...
ExecutorServiceManager manager = context.getExecutorServiceManager();
ThreadPoolProfile defaultProfile = manager.getDefaultThreadPoolProfile();

// Now, customize the profile settings.
```

```
defaultProfile.setPoolSize(3);
defaultProfile.setMaxQueueSize(100);
...
```

In the XML DSL, you can customize the default thread pool profile, as follows:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <threadPoolProfile
    id="changedProfile"
    defaultProfile="true"
    poolSize="3"
    maxQueueSize="100"/>
  ...
</camelContext>
```

Note that it is essential to set the **defaultProfile** attribute to **true** in the preceding XML DSL example, otherwise the thread pool profile would be treated like a custom thread pool profile (see [the section called “Creating a custom thread pool profile”](#)), instead of replacing the default thread pool profile.

Customizing a processor's thread pool

It is also possible to specify the thread pool for a threading-aware processor more directly, using either the **executorService** or **executorServiceRef** options (where these options are used instead of the **parallelProcessing** option). There are two approaches you can use to customize a processor's thread pool, as follows:

- *Specify a custom thread pool*—explicitly create an **ExecutorService** (thread pool) instance and pass it to the **executorService** option.
- *Specify a custom thread pool profile*—create and register a custom thread pool factory. When you reference this factory using the **executorServiceRef** option, the processor automatically uses the factory to create a custom thread pool instance.

When you pass a bean ID to the **executorServiceRef** option, the threading-aware processor first tries to find a custom thread pool with that ID in the registry. If no thread pool is registered with that ID, the processor then attempts to look up a custom thread pool profile in the registry and uses the custom thread pool profile to instantiate a custom thread pool.

Creating a custom thread pool

A custom thread pool can be any thread pool of [java.util.concurrent.ExecutorService](#) type. The following approaches to creating a thread pool instance are recommended in Apache Camel:

- Use the **org.apache.camel.builder.ThreadPoolBuilder** utility to build the thread pool class.
- Use the **org.apache.camel.spi.ExecutorServiceManager** instance from the current **CamelContext** to create the thread pool class.

Ultimately, there is not much difference between the two approaches, because the **ThreadPoolBuilder** is actually defined using the **ExecutorServiceManager** instance. Normally, the **ThreadPoolBuilder** is preferred, because it offers a simpler approach. But there is at least one

kind of thread (the **ScheduledExecutorService**) that can only be created by accessing the **ExecutorServiceManager** instance directory.

Table 2.10, “Thread Pool Builder Options” shows the options supported by the **ThreadPoolBuilder** class, which you can set when defining a new custom thread pool.

Table 2.10. Thread Pool Builder Options

Builder Option	Description
maxQueueSize()	Sets the maximum number of pending tasks that this thread pool can store in its incoming task queue. A value of -1 specifies an unbounded queue. Default value is taken from default thread pool profile.
poolSize()	Sets the minimum number of threads in the pool (this is also the initial pool size). Default value is taken from default thread pool profile.
maxPoolSize()	Sets the maximum number of threads that can be in the pool. Default value is taken from default thread pool profile.
keepAliveTime()	If any threads are idle for longer than this period of time (specified in seconds), they are terminated. This allows the thread pool to shrink when the load is light. Default value is taken from default thread pool profile.
rejectedPolicy()	<p>Specifies what course of action to take, if the incoming task queue is full. You can specify four possible values:</p> <p>CallerRuns <i>(Default value)</i> Gets the caller thread to run the latest incoming task. As a side effect, this option prevents the caller thread from receiving any more tasks until it has finished processing the latest incoming task.</p> <p>Abort Aborts the latest incoming task by throwing an exception.</p> <p>Discard Quietly discards the latest incoming task.</p> <p>DiscardOldest Discards the oldest unhandled task and then attempts to enqueue the latest incoming task in the task queue.</p>

Builder Option	Description
build()	Finishes building the custom thread pool and registers the new thread pool under the ID specified as the argument to build() .

In Java DSL, you can define a custom thread pool using the **ThreadPoolBuilder**, as follows:

```
// Java
import org.apache.camel.builder.ThreadPoolBuilder;
import java.util.concurrent.ExecutorService;
...
ThreadPoolBuilder poolBuilder = new ThreadPoolBuilder(context);
ExecutorService customPool =
poolBuilder.poolSize(5).maxPoolSize(5).maxQueueSize(100).build("customPool
");
...

from("direct:start")
    .multicast().executorService(customPool)
        .to("mock:first")
        .to("mock:second")
        .to("mock:third");
```

Instead of passing the object reference, **customPool**, directly to the **executorService()** option, you can look up the thread pool in the registry, by passing its bean ID to the **executorServiceRef()** option, as follows:

```
// Java
from("direct:start")
    .multicast().executorServiceRef("customPool")
        .to("mock:first")
        .to("mock:second")
        .to("mock:third");
```

In XML DSL, you access the **ThreadPoolBuilder** using the **threadPool** element. You can then reference the custom thread pool using the **executorServiceRef** attribute to look up the thread pool by ID in the Spring registry, as follows:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <threadPool id="customPool"
        poolSize="5"
        maxPoolSize="5"
        maxQueueSize="100" />

    <route>
        <from uri="direct:start"/>
        <multicast executorServiceRef="customPool">
            <to uri="mock:first"/>
            <to uri="mock:second"/>
            <to uri="mock:third"/>
        </multicast>
    </route>
</camelContext>
```

```

        </multicast>
    </route>
</camelContext>

```

Creating a custom thread pool profile

If you have many custom thread pool instances to create, you might find it more convenient to define a custom thread pool profile, which acts as a factory for thread pools. Whenever you reference a thread pool profile from a threading-aware processor, the processor automatically uses the profile to create a new thread pool instance. You can define a custom thread pool profile either in Java DSL or in XML DSL.

For example, in Java DSL you can create a custom thread pool profile with the bean ID, **customProfile**, and reference it from within a route, as follows:

```

// Java
import org.apache.camel.spi.ThreadPoolProfile;
import org.apache.camel.impl.ThreadPoolProfileSupport;
...
// Create the custom thread pool profile
ThreadPoolProfile customProfile = new
ThreadPoolProfileSupport("customProfile");
customProfile.setPoolSize(5);
customProfile.setMaxPoolSize(5);
customProfile.setMaxQueueSize(100);
context.getExecutorServiceManager().registerThreadPoolProfile(customProfile);
...
// Reference the custom thread pool profile in a route
from("direct:start")
    .multicast().executorServiceRef("customProfile")
        .to("mock:first")
        .to("mock:second")
        .to("mock:third");

```

In XML DSL, use the **threadPoolProfile** element to create a custom pool profile (where you let the **defaultProfile** option default to **false**, because this is *not* a default thread pool profile). You can create a custom thread pool profile with the bean ID, **customProfile**, and reference it from within a route, as follows:

```

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <threadPoolProfile
        id="customProfile"
        poolSize="5"
        maxPoolSize="5"
        maxQueueSize="100" />

    <route>
        <from uri="direct:start"/>
        <multicast executorServiceRef="customProfile">
            <to uri="mock:first"/>
            <to uri="mock:second"/>
            <to uri="mock:third"/>
        </multicast>
    </route>
</camelContext>

```

```

        </multicast>
    </route>
</camelContext>

```

Sharing a thread pool between components

Some of the standard poll-based components—such as File and FTP—allow you to specify the thread pool to use. This makes it possible for different components to share the same thread pool, reducing the overall number of threads in the JVM.

For example, the [File component](#) and the [FTP component](#) both expose the `scheduledExecutorService` property, which you can use to specify the component's `ExecutorService` object.

2.9. CONTROLLING START-UP AND SHUTDOWN OF ROUTES

Overview

By default, routes are automatically started when your Apache Camel application (as represented by the `CamelContext` instance) starts up and routes are automatically shut down when your Apache Camel application shuts down. For non-critical deployments, the details of the shutdown sequence are usually not very important. But in a production environment, it is often crucial that existing tasks should run to completion during shutdown, in order to avoid data loss. You typically also want to control the order in which routes shut down, so that dependencies are not violated (which would prevent existing tasks from running to completion).

For this reason, Apache Camel provides a set of features to support *graceful shutdown* of applications. Graceful shutdown gives you full control over the stopping and starting of routes, enabling you to control the shutdown order of routes and enabling current tasks to run to completion.

Setting the route ID

It is good practice to assign a route ID to each of your routes. As well as making logging messages and management features more informative, the use of route IDs enables you to apply greater control over the stopping and starting of routes.

For example, in the Java DSL, you can assign the route ID, `myCustomerRouteId`, to a route by invoking the `routeId()` command as follows:

```
from("SourceURI").routeId("myCustomRouteId").process(...).to(TargetURI);
```

In the XML DSL, set the `route` element's `id` attribute, as follows:

```

<camelContext id="CamelContextID"
  xmlns="http://camel.apache.org/schema/spring">
  <route id="myCustomRouteId" >
    <from uri="SourceURI"/>
    <process ref="someProcessorId"/>
    <to uri="TargetURI"/>
  </route>
</camelContext>

```

Disabling automatic start-up of routes

By default, all of the routes that the CamelContext knows about at start time will be started automatically. If you want to control the start-up of a particular route manually, however, you might prefer to disable automatic start-up for that route.

To control whether a Java DSL route starts up automatically, invoke the **autoStartup** command, either with a **boolean** argument (**true** or **false**) or a **String** argument (**true** or **false**). For example, you can disable automatic start-up of a route in the Java DSL, as follows:

```
from("SourceURI")
    .routeId("nonAuto")
    .autoStartup(false)
    .to(TargetURI);
```

You can disable automatic start-up of a route in the XML DSL by setting the **autoStartup** attribute to **false** on the **route** element, as follows:

```
<camelContext id="CamelContextID"
  xmlns="http://camel.apache.org/schema/spring">
  <route id="nonAuto" autoStartup="false">
    <from uri="SourceURI"/>
    <to uri="TargetURI"/>
  </route>
</camelContext>
```

Manually starting and stopping routes

You can manually start or stop a route at any time in Java by invoking the **startRoute()** and **stopRoute()** methods on the **CamelContext** instance. For example, to start the route having the route ID, **nonAuto**, invoke the **startRoute()** method on the **CamelContext** instance, **context**, as follows:

```
// Java
context.startRoute("nonAuto");
```

To stop the route having the route ID, **nonAuto**, invoke the **stopRoute()** method on the **CamelContext** instance, **context**, as follows:

```
// Java
context.stopRoute("nonAuto");
```

Startup order of routes

By default, Apache Camel starts up routes in a non-deterministic order. In some applications, however, it can be important to control the startup order. To control the startup order in the Java DSL, use the **startupOrder()** command, which takes a positive integer value as its argument. The route with the lowest integer value starts first, followed by the routes with successively higher startup order values.

For example, the first two routes in the following example are linked together through the **seda:buffer** endpoint. You can ensure that the first route segment starts *after* the second route segment by assigning startup orders (2 and 1 respectively), as follows:

■

Example 2.3. Startup Order in Java DSL

```

from("jetty:http://fooserver:8080")
    .routeId("first")
    .startupOrder(2)
    .to("seda:buffer");

from("seda:buffer")
    .routeId("second")
    .startupOrder(1)
    .to("mock:result");

// This route's startup order is unspecified
from("jms:queue:foo").to("jms:queue:bar");

```

Or in Spring XML, you can achieve the same effect by setting the **route** element's **startupOrder** attribute, as follows:

Example 2.4. Startup Order in XML DSL

```

<route id="first" startupOrder="2">
  <from uri="jetty:http://fooserver:8080"/>
  <to uri="seda:buffer"/>
</route>

<route id="second" startupOrder="1">
  <from uri="seda:buffer"/>
  <to uri="mock:result"/>
</route>

<!-- This route's startup order is unspecified -->
<route>
  <from uri="jms:queue:foo"/>
  <to uri="jms:queue:bar"/>
</route>

```

Each route must be assigned a *unique* startup order value. You can choose any positive integer value that is less than 1000. Values of 1000 and over are reserved for Apache Camel, which automatically assigns these values to routes without an explicit startup value. For example, the last route in the preceding example would automatically be assigned the startup value, 1000 (so it starts up after the first two routes).

Shutdown sequence

When a **CamelContext** instance is shutting down, Apache Camel controls the shutdown sequence using a pluggable *shutdown strategy*. The default shutdown strategy implements the following shutdown sequence:

1. Routes are shut down in the *reverse* of the start-up order.
2. Normally, the shutdown strategy waits until the currently active exchanges have finished processing. The treatment of running tasks is configurable, however.

- Overall, the shutdown sequence is bound by a timeout (default, 300 seconds). If the shutdown sequence exceeds this timeout, the shutdown strategy will force shutdown to occur, even if some tasks are still running.

Shutdown order of routes

Routes are shut down in the reverse of the start-up order. That is, when a start-up order is defined using the `startupOrder()` command (in Java DSL) or `startupOrder` attribute (in XML DSL), the first route to shut down is the route with the *highest* integer value assigned by the start-up order and the last route to shut down is the route with the *lowest* integer value assigned by the start-up order.

For example, in [Example 2.3, “Startup Order in Java DSL”](#), the first route segment to be shut down is the route with the ID, **first**, and the second route segment to be shut down is the route with the ID, **second**. This example illustrates a general rule, which you should observe when shutting down routes: *the routes that expose externally-accessible consumer endpoints should be shut down first*, because this helps to throttle the flow of messages through the rest of the route graph.



NOTE

Apache Camel also provides the option `shutdownRoute(Defer)`, which enables you to specify that a route must be amongst the last routes to shut down (overriding the start-up order value). But you should rarely ever need this option. This option was mainly needed as a workaround for earlier versions of Apache Camel (prior to 2.3), for which routes would shut down in the *same* order as the start-up order.

Shutting down running tasks in a route

If a route is still processing messages when the shutdown starts, the shutdown strategy normally waits until the currently active exchange has finished processing before shutting down the route. This behavior can be configured on each route using the `shutdownRunningTask` option, which can take either of the following values:

`ShutdownRunningTask.CompleteCurrentTaskOnly`

(Default) Usually, a route operates on just a single message at a time, so you can safely shut down the route after the current task has completed.

`ShutdownRunningTask.CompleteAllTasks`

Specify this option in order to shut down *batch consumers* gracefully. Some consumer endpoints (for example, File, FTP, Mail, iBATIS, and JPA) operate on a batch of messages at a time. For these endpoints, it is more appropriate to wait until all of the messages in the current batch have completed.

For example, to shut down a File consumer endpoint gracefully, you should specify the `CompleteAllTasks` option, as shown in the following Java DSL fragment:

```
// Java
public void configure() throws Exception {
    from("file:target/pending")
        .routeId("first").startupOrder(2)
        .shutdownRunningTask(ShutdownRunningTask.CompleteAllTasks)
        .delay(1000).to("seda:foo");

    from("seda:foo")
```

```

        .routeId("second").startupOrder(1)
        .to("mock:bar");
    }

```

The same route can be defined in the XML DSL as follows:

```

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <!-- let this route complete all its pending messages when asked to
  shut down -->
  <route id="first"
    startupOrder="2"
    shutdownRunningTask="CompleteAllTasks">
    <from uri="file:target/pending"/>
    <delay><constant>1000</constant></delay>
    <to uri="seda:foo"/>
  </route>

  <route id="second" startupOrder="1">
    <from uri="seda:foo"/>
    <to uri="mock:bar"/>
  </route>
</camelContext>

```

Shutdown timeout

The shutdown timeout has a default value of 300 seconds. You can change the value of the timeout by invoking the `setTimeout()` method on the shutdown strategy. For example, you can change the timeout value to 600 seconds, as follows:

```

// Java
// context = CamelContext instance
context.getShutdownStrategy().setTimeout(600);

```

Integration with custom components

If you are implementing a custom Apache Camel component (which also inherits from the `org.apache.camel.Service` interface), you can ensure that your custom code receives a shutdown notification by implementing the `org.apache.camel.spi.ShutdownPrepared` interface. This gives the component an opportunity execute custom code in preparation for shutdown.

2.10. SCHEDULED ROUTE POLICY

2.10.1. Overview of Scheduled Route Policies

Overview

A scheduled route policy can be used to trigger events that affect a route at runtime. In particular, the implementations that are currently available enable you to start, stop, suspend, or resume a route at any time (or times) specified by the policy.

Scheduling tasks

The scheduled route policies are capable of triggering the following kinds of event:

- *Start a route*—start the route at the time (or times) specified. This event only has an effect, if the route is currently in a stopped state, awaiting activation.
- *Stop a route*—stop the route at the time (or times) specified. This event only has an effect, if the route is currently active.
- *Suspend a route*—temporarily de-activate the consumer endpoint at the start of the route (as specified in `from()`). The rest of the route is still active, but clients will not be able to send new messages into the route.
- *Resume a route*—re-activate the consumer endpoint at the start of the route, returning the route to a fully active state.

Quartz component

The Quartz component is a timer component based on Terracotta's [Quartz](#), which is an open source implementation of a job scheduler. The Quartz component provides the underlying implementation for both the simple scheduled route policy and the cron scheduled route policy.

2.10.2. Simple Scheduled Route Policy

Overview

The simple scheduled route policy is a route policy that enables you to start, stop, suspend, and resume routes, where the timing of these events is defined by providing the time and date of an initial event and (optionally) by specifying a certain number of subsequent repetitions. To define a simple scheduled route policy, create an instance of the following class:

```
org.apache.camel.routepolicy.quartz.SimpleScheduledRoutePolicy
```

Dependency

The simple scheduled route policy depends on the Quartz component, `camel-quartz`. For example, if you are using Maven as your build system, you would need to add a dependency on the `camel-quartz` artifact.

Java DSL example

[Example 2.5, “Java DSL Example of Simple Scheduled Route”](#) shows how to schedule a route to start up using the Java DSL. The initial start time, `startTime`, is defined to be 3 seconds after the current time. The policy is also configured to start the route a *second* time, 3 seconds after the initial start time, which is configured by setting `routeStartRepeatCount` to 1 and `routeStartRepeatInterval` to 3000 milliseconds.

In Java DSL, you attach the route policy to the route by calling the `routePolicy()` DSL command in the route.

Example 2.5. Java DSL Example of Simple Scheduled Route

```
// Java
SimpleScheduledRoutePolicy policy = new SimpleScheduledRoutePolicy();
```



```

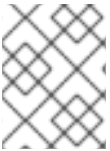
long startTime = System.currentTimeMillis() + 3000L;
policy.setRouteStartDate(new Date(startTime));
policy.setRouteStartRepeatCount(1);
policy.setRouteStartRepeatInterval(3000);

```

```

from("direct:start")
    .routeId("test")
    .routePolicy(policy)
    .to("mock:success");

```



NOTE

You can specify multiple policies on the route by calling `routePolicy()` with multiple arguments.

XML DSL example

Example 2.6, “XML DSL Example of Simple Scheduled Route” shows how to schedule a route to start up using the XML DSL.

In XML DSL, you attach the route policy to the route by setting the `routePolicyRef` attribute on the `route` element.

Example 2.6. XML DSL Example of Simple Scheduled Route

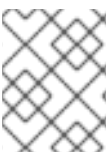
```

<bean id="date" class="java.util.Date"/>

<bean id="startPolicy"
class="org.apache.camel.routePolicy.quartz.SimpleScheduledRoutePolicy">
    <property name="routeStartDate" ref="date"/>
    <property name="routeStartRepeatCount" value="1"/>
    <property name="routeStartRepeatInterval" value="3000"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="myroute" routePolicyRef="startPolicy">
        <from uri="direct:start"/>
        <to uri="mock:success"/>
    </route>
</camelContext>

```



NOTE

You can specify multiple policies on the route by setting the value of `routePolicyRef` as a comma-separated list of bean IDs.

Defining dates and times

The initial times of the triggers used in the simple scheduled route policy are specified using the `java.util.Date` type. The most flexible way to define a `Date` instance is through the `java.util.GregorianCalendar` class. Use the convenient constructors and methods of the

GregorianCalendar class to define a date and then obtain a **Date** instance by calling **GregorianCalendar.getTime()**.

For example, to define the time and date for January 1, 2011 at noon, call a **GregorianCalendar** constructor as follows:

```
// Java
import java.util.GregorianCalendar;
import java.util.Calendar;
...
GregorianCalendar gc = new GregorianCalendar(
    2011,
    Calendar.JANUARY,
    1,
    12, // hourOfDay
    0, // minutes
    0 // seconds
);

java.util.Date triggerDate = gc.getTime();
```

The **GregorianCalendar** class also supports the definition of times in different time zones. By default, it uses the local time zone on your computer.

Graceful shutdown

When you configure a simple scheduled route policy to stop a route, the route stopping algorithm is automatically integrated with the graceful shutdown procedure (see [Section 2.9, “Controlling Start-Up and Shutdown of Routes”](#)). This means that the task waits until the current exchange has finished processing before shutting down the route. You can set a timeout, however, that forces the route to stop after the specified time, irrespective of whether or not the route has finished processing the exchange.

Scheduling tasks

You can use a simple scheduled route policy to define one or more of the following scheduling tasks:

- [the section called “Starting a route”](#).
- [the section called “Stopping a route”](#).
- [the section called “Suspending a route”](#).
- [the section called “Resuming a route”](#).

Starting a route

The following table lists the parameters for scheduling one or more route starts.

Parameter	Type	Default	Description
routeStartDate	java.util.Date	<i>None</i>	Specifies the date and time when the route is started for the first time.

Parameter	Type	Default	Description
routeStartRepeatCount	int	0	When set to a non-zero value, specifies how many times the route should be started.
routeStartRepeatInterval	long	0	Specifies the time interval between starts, in units of milliseconds.

Stopping a route

The following table lists the parameters for scheduling one or more route stops.

Parameter	Type	Default	Description
routeStopDate	java.util.Date	<i>None</i>	Specifies the date and time when the route is stopped for the first time.
routeStopRepeatCount	int	0	When set to a non-zero value, specifies how many times the route should be stopped.
routeStopRepeatInterval	long	0	Specifies the time interval between stops, in units of milliseconds.
routeStopGracePeriod	int	10000	Specifies how long to wait for the current exchange to finish processing (grace period) before forcibly stopping the route. Set to 0 for an infinite grace period.
routeStopTimeUnit	long	TimeUnit.MILLISECONDS	Specifies the time unit of the grace period.

Suspending a route

The following table lists the parameters for scheduling the suspension of a route one or more times.

Parameter	Type	Default	Description
-----------	------	---------	-------------

Parameter	Type	Default	Description
routeSuspendDate	java.util.Date	<i>None</i>	Specifies the date and time when the route is suspended for the first time.
routeSuspendRepeatCount	int	0	When set to a non-zero value, specifies how many times the route should be suspended.
routeSuspendRepeatInterval	long	0	Specifies the time interval between suspends, in units of milliseconds.

Resuming a route

The following table lists the parameters for scheduling the resumption of a route one or more times.

Parameter	Type	Default	Description
routeResumeDate	java.util.Date	<i>None</i>	Specifies the date and time when the route is resumed for the first time.
routeResumeRepeatCount	int	0	When set to a non-zero value, specifies how many times the route should be resumed.
routeResumeRepeatInterval	long	0	Specifies the time interval between resumes, in units of milliseconds.

2.10.3. Cron Scheduled Route Policy

Overview

The cron scheduled route policy is a route policy that enables you to start, stop, suspend, and resume routes, where the timing of these events is specified using cron expressions. To define a cron scheduled route policy, create an instance of the following class:

```
org.apache.camel.routepolicy.quartz.CronScheduledRoutePolicy
```

Dependency

The simple scheduled route policy depends on the Quartz component, **camel-quartz**. For example, if you are using Maven as your build system, you would need to add a dependency on the **camel-quartz** artifact.

Java DSL example

[Example 2.7, “Java DSL Example of a Cron Scheduled Route”](#) shows how to schedule a route to start up using the Java DSL. The policy is configured with the cron expression, `*/3 * * * * ?`, which triggers a start event every 3 seconds.

In Java DSL, you attach the route policy to the route by calling the **routePolicy()** DSL command in the route.

Example 2.7. Java DSL Example of a Cron Scheduled Route

```
// Java
CronScheduledRoutePolicy policy = new CronScheduledRoutePolicy();
policy.setRouteStartTime("*/3 * * * * ?");

from("direct:start")
    .routeId("test")
    .routePolicy(policy)
    .to("mock:success");;
```



NOTE

You can specify multiple policies on the route by calling **routePolicy()** with multiple arguments.

XML DSL example

[Example 2.8, “XML DSL Example of a Cron Scheduled Route”](#) shows how to schedule a route to start up using the XML DSL.

In XML DSL, you attach the route policy to the route by setting the **routePolicyRef** attribute on the **route** element.

Example 2.8. XML DSL Example of a Cron Scheduled Route

```
<bean id="date" class="org.apache.camel.routepolicy.quartz.SimpleDate"/>

<bean id="startPolicy"
class="org.apache.camel.routepolicy.quartz.CronScheduledRoutePolicy">
    <property name="routeStartTime" value="*/3 * * * * ?"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="testRoute" routePolicyRef="startPolicy">
        <from uri="direct:start"/>
        <to uri="mock:success"/>
    </route>
</camelContext>
```



NOTE

You can specify multiple policies on the route by setting the value of **routePolicyRef** as a comma-separated list of bean IDs.

Defining cron expressions

The *cron expression* syntax has its origins in the UNIX **cron** utility, which schedules jobs to run in the background on a UNIX system. A cron expression is effectively a syntax for wildcarding dates and times that enables you to specify either a single event or multiple events that recur periodically.

A cron expression consists of 6 or 7 fields in the following order:

```
Seconds Minutes Hours DayOfMonth Month DayOfWeek [Year]
```

The **Year** field is optional and usually omitted, unless you want to define an event that occurs once and once only. Each field consists of a mixture of literals and special characters. For example, the following cron expression specifies an event that fires once every day at midnight:

```
0 0 24 * * ?
```

The ***** character is a wildcard that matches every value of a field. Hence, the preceding expression matches every day of every month. The **?** character is a dummy placeholder that means *ignore this field*. It always appears either in the **DayOfMonth** field *or* in the **DayOfWeek** field, because it is not logically consistent to specify both of these fields at the same time. For example, if you want to schedule an event that fires once a day, but only from Monday to Friday, use the following cron expression:

```
0 0 24 ? * MON-FRI
```

Where the hyphen character specifies a range, **MON-FRI**. You can also use the forward slash character, **/**, to specify increments. For example, to specify that an event fires every 5 minutes, use the following cron expression:

```
0 0/5 * * * ?
```

For a full explanation of the cron expression syntax, see the Wikipedia article on [CRON expressions](#).

Scheduling tasks

You can use a cron scheduled route policy to define one or more of the following scheduling tasks:

- [the section called “Starting a route”](#).
- [the section called “Stopping a route”](#).
- [the section called “Suspending a route”](#).
- [the section called “Resuming a route”](#).

Starting a route

The following table lists the parameters for scheduling one or more route starts.

Parameter	Type	Default	Description
<code>routeStartString</code>	String	<i>None</i>	Specifies a cron expression that triggers one or more route start events.

Stopping a route

The following table lists the parameters for scheduling one or more route stops.

Parameter	Type	Default	Description
<code>routeStopTime</code>	String	<i>None</i>	Specifies a cron expression that triggers one or more route stop events.
<code>routeStopGracePeriod</code>	int	10000	Specifies how long to wait for the current exchange to finish processing (grace period) before forcibly stopping the route. Set to 0 for an infinite grace period.
<code>routeStopTimeUnit</code>	long	TimeUnit.MILLISECONDS	Specifies the time unit of the grace period.

Suspending a route

The following table lists the parameters for scheduling the suspension of a route one or more times.

Parameter	Type	Default	Description
<code>routeSuspendTime</code>	String	<i>None</i>	Specifies a cron expression that triggers one or more route suspend events.

Resuming a route

The following table lists the parameters for scheduling the resumption of a route one or more times.

Parameter	Type	Default	Description
<code>routeResumeTime</code>	String	None	Specifies a cron expression that triggers one or more route resume events.

2.11. JMX NAMING

Overview

Apache Camel allows you to customise the name of a **CamelContext** bean as it appears in JMX, by defining a *management name pattern* for it. For example, you can customise the name pattern of an XML **CamelContext** instance, as follows:

```
<camelContext id="myCamel" managementNamePattern="#name#">
    ...
</camelContext>
```

If you do not explicitly set a name pattern for the **CamelContext** bean, Apache Camel reverts to a default naming strategy.

Default naming strategy

By default, the JMX name of a **CamelContext** bean is equal to the value of the bean's **id** attribute, prefixed by the current bundle ID. For example, if the **id** attribute on a **camelContext** element is **myCamel** and the current bundle ID is 250, the JMX name would be **250-myCamel**. In cases where there is more than one **CamelContext** instance with the same **id** in the bundle, the JMX name is disambiguated by adding a counter value as a suffix. For example, if there are multiple instances of **myCamel** in the bundle, the corresponding JMX MBeans are named as follows:

```
250-myCamel-1
250-myCamel-2
250-myCamel-3
...
```

Customising the JMX naming strategy

One drawback of the default naming strategy is that you cannot guarantee that a given **CamelContext** bean will have the same JMX name between runs. If you want to have greater consistency between runs, you can control the JMX name more precisely by defining a *JMX name pattern* for the **CamelContext** instances.

Specifying a name pattern in Java

To specify a name pattern on a **CamelContext** in Java, call the **setNamePattern** method, as follows:

```
// Java
context.getManagementNameStrategy().setNamePattern("#name#");
```


Specifying a name pattern in XML

To specify a name pattern on a **CamelContext** in XML, set the **managementNamePattern** attribute on the **camelContext** element, as follows:

```
<camelContext id="myCamel" managementNamePattern="#name#">
```

Name pattern tokens

You can construct a JMX name pattern by mixing literal text with any of the following tokens:

Table 2.11. JMX Name Pattern Tokens

Token	Description
#camelId#	Value of the id attribute on the CamelContext bean.
#name#	Same as #camelId# .
#counter#	An incrementing counter (starting at 1).
#bundleId#	The OSGi bundle ID of the deployed bundle (<i>OSGi only</i>).
#symbolicName#	The OSGi symbolic name (<i>OSGi only</i>).
#version#	The OSGi bundle version (<i>OSGi only</i>).

Examples

Here are some examples of JMX name patterns you could define using the supported tokens:

```
<camelContext id="fooContext" managementNamePattern="FooApplication-
#name#">
    ...
</camelContext>
<camelContext id="myCamel" managementNamePattern="#bundleID#-
#symbolicName#-#name#">
    ...
</camelContext>
```

Ambiguous names

Because the customised naming pattern overrides the default naming strategy, it is possible to define ambiguous JMX MBean names using this approach. For example:

```
<camelContext id="foo" managementNamePattern="SameOldSameOld"> ...
</camelContext>
...
<camelContext id="bar" managementNamePattern="SameOldSameOld"> ...
</camelContext>
```

In this case, Apache Camel would fail on start-up and report an *MBean already exists* exception. You should, therefore, take extra care to ensure that you do not define ambiguous name patterns.

CHAPTER 3. INTRODUCING ENTERPRISE INTEGRATION PATTERNS

Abstract

The Apache Camel's *Enterprise Integration Patterns* are inspired by a book of the same name written by Gregor Hohpe and Bobby Woolf. The patterns described by these authors provide an excellent toolbox for developing enterprise integration projects. In addition to providing a common language for discussing integration architectures, many of the patterns can be implemented directly using Apache Camel's programming interfaces and XML configuration.

3.1. OVERVIEW OF THE PATTERNS



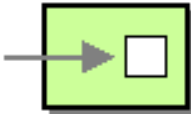
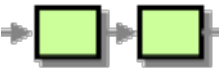
Enterprise Integration Patterns book

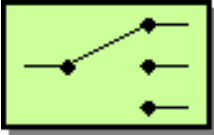
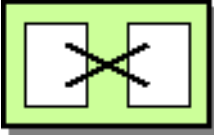
Apache Camel supports most of the patterns from the book, [Enterprise Integration Patterns](#) by Gregor Hohpe and Bobby Woolf.

Messaging systems

The messaging systems patterns, shown in [Table 3.1](#), “Messaging Systems”, introduce the fundamental concepts and components that make up a messaging system.

Table 3.1. Messaging Systems



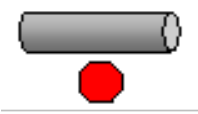

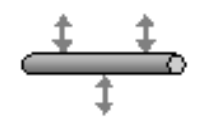
Icon	Name	Use Case
	Message	How can two applications connected by a message channel exchange a piece of information?
	Message Channel	How does one application communicate with another application using messaging?
	Message Endpoint	How does an application connect to a messaging channel to send and receive messages?
	Pipes and Filters	How can we perform complex processing on a message while still maintaining independence and flexibility?

Icon	Name	Use Case
	Message Router	How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of defined conditions?
	Message Translator	How do systems using different data formats communicate with each other using messaging?

Messaging channels

A messaging channel is the basic component used for connecting the participants in a messaging system. The patterns in [Table 3.2, “Messaging Channels”](#) describe the different kinds of messaging channels available.

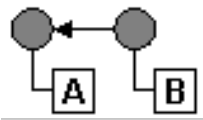
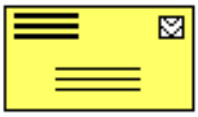
Table 3.2. Messaging Channels

Icon	Name	Use Case
	Point to Point Channel	How can the caller be sure that exactly one receiver will receive the document or will perform the call?
	Publish Subscribe Channel	How can the sender broadcast an event to all interested receivers?
	Dead Letter Channel	What will the messaging system do with a message it cannot deliver?
	Guaranteed Delivery	How does the sender make sure that a message will be delivered, even if the messaging system fails?
	Message Bus	What is an architecture that enables separate, decoupled applications to work together, such that one or more of the applications can be added or removed without affecting the others?

Message construction

The message construction patterns, shown in [Table 3.3, “Message Construction”](#), describe the various forms and functions of the messages that pass through the system.

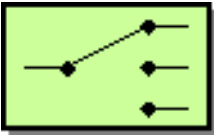
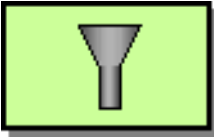
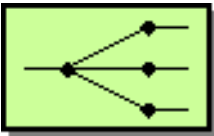
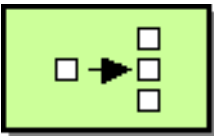
Table 3.3. Message Construction

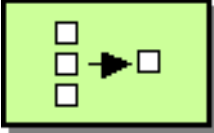
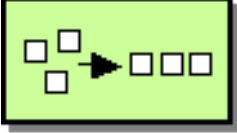
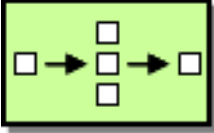
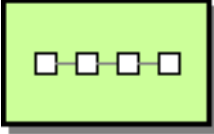
Icon	Name	Use Case
	Correlation Identifier	How does a requestor identify the request that generated the received reply?
	Return Address	How does a replier know where to send the reply?

Message routing

The message routing patterns, shown in [Table 3.4, “Message Routing”](#), describe various ways of linking message channels together, including various algorithms that can be applied to the message stream (without modifying the body of the message).

Table 3.4. Message Routing

Icon	Name	Use Case
	Content Based Router	How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?
	Message Filter	How does a component avoid receiving uninteresting messages?
	Recipient List	How do we route a message to a list of dynamically specified recipients?
	Splitter	How can we process a message if it contains multiple elements, each of which might have to be processed in a different way?

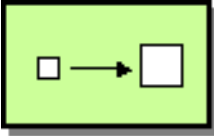
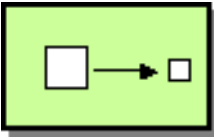
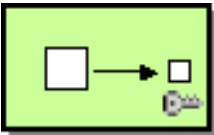
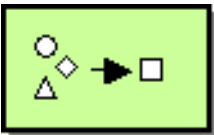
Icon	Name	Use Case
	Aggregator	How do we combine the results of individual, but related messages so that they can be processed as a whole?
	Resequencer	How can we get a stream of related, but out-of-sequence, messages back into the correct order?
	Composed Message Processor	How can you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?
	Scatter-Gather	How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?
	Routing Slip	How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design-time, and might vary for each message?
	Throttler	How can I throttle messages to ensure that a specific endpoint does not get overloaded, or that we don't exceed an agreed SLA with some external service?
	Delayer	How can I delay the sending of a message?
	Load Balancer	How can I balance load across a number of endpoints?
	Multicast	How can I route a message to a number of endpoints at the same time?
	Loop	How can I repeat processing a message in a loop?

Icon	Name	Use Case
	Sampling	How can I sample one message out of many in a given period to avoid downstream route does not get overloaded?

Message transformation

The message transformation patterns, shown in [Table 3.5, “Message Transformation”](#), describe how to modify the contents of messages for various purposes.

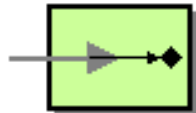
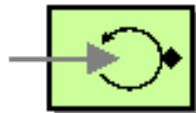
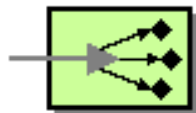
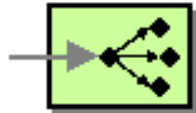
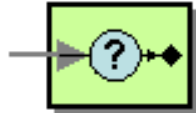
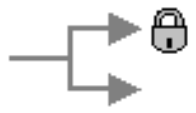
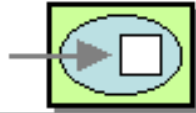
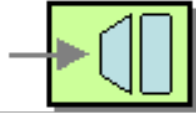
Table 3.5. Message Transformation

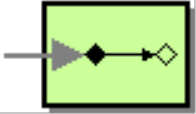
Icon	Name	Use Case
	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?
	Content Filter	How do you simplify dealing with a large message, when you are interested in only a few data items?
	Claim Check	How can we reduce the data volume of message sent across the system without sacrificing information content?
	Normalizer	How do you process messages that are semantically equivalent, but arrive in a different format?
	Sort	How can I sort the body of a message?

Messaging endpoints

A messaging endpoint denotes the point of contact between a messaging channel and an application. The messaging endpoint patterns, shown in [Table 3.6, “Messaging Endpoints”](#), describe various features and qualities of service that can be configured on an endpoint.

Table 3.6. Messaging Endpoints

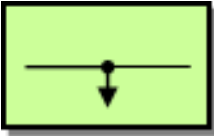
Icon	Name	Use Case
	Messaging Mapper	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	Event Driven Consumer	How can an application automatically consume messages as they become available?
	Polling Consumer	How can an application consume a message when the application is ready?
	Competing Consumers	How can a messaging client process multiple messages concurrently?
	Message Dispatcher	How can multiple consumers on a single channel coordinate their message processing?
	Selective Consumer	How can a message consumer select which messages it wants to receive?
	Durable Subscriber	How can a subscriber avoid missing messages when it's not listening for them?
	Idempotent Consumer	How can a message receiver deal with duplicate messages?
	Transactional Client	How can a client control its transactions with the messaging system?
	Messaging Gateway	How do you encapsulate access to the messaging system from the rest of the application?

Icon	Name	Use Case
	Service Activator	How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

System management

The system management patterns, shown in [Table 3.7, “System Management”](#), describe how to monitor, test, and administer a messaging system.

Table 3.7. System Management

Icon	Name	Use Case
	Wire Tap	How do you inspect messages that travel on a point-to-point channel?

CHAPTER 4. MESSAGING SYSTEMS

Abstract

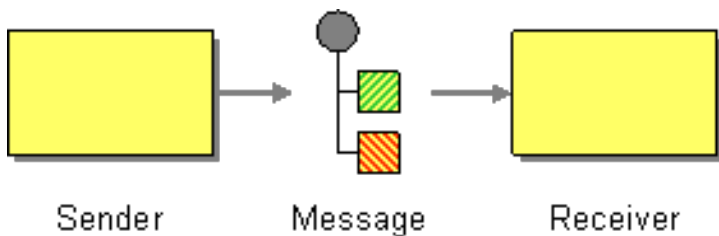
This chapter introduces the fundamental building blocks of a messaging system, such as endpoints, messaging channels, and message routers.

4.1. MESSAGE

Overview

A *message* is the smallest unit for transmitting data in a messaging system (represented by the grey dot in the figure below). The message itself might have some internal structure—for example, a message containing multiple parts—which is represented by geometrical figures attached to the grey dot in [Figure 4.1, “Message Pattern”](#).

Figure 4.1. Message Pattern



Types of message

Apache Camel defines the following distinct message types:

- *In* message — A message that travels through a route from a consumer endpoint to a producer endpoint (typically, initiating a message exchange).
- *Out* message — A message that travels through a route from a producer endpoint back to a consumer endpoint (usually, in response to an *In* message).

All of these message types are represented internally by the `org.apache.camel.Message` interface.

Message structure

By default, Apache Camel applies the following structure to all message types:

- *Headers* — Contains metadata or header data extracted from the message.
- *Body* — Usually contains the entire message in its original form.
- *Attachments* — Message attachments (required for integrating with certain messaging systems, such as [JBI](#)).

It is important to remember that this division into headers, body, and attachments is an abstract model of the message. Apache Camel supports many different components, that generate a wide variety of message formats. Ultimately, it is the underlying component implementation that decides what gets placed into the headers and body of a message.

Correlating messages

Internally, Apache Camel remembers the message IDs, which are used to correlate individual messages. In practice, however, the most important way that Apache Camel correlates messages is through *exchange* objects.

Exchange objects

An exchange object is an entity that encapsulates related messages, where the collection of related messages is referred to as a *message exchange* and the rules governing the sequence of messages are referred to as an *exchange pattern*. For example, two common exchange patterns are: one-way event messages (consisting of an *In* message), and request-reply exchanges (consisting of an *In* message, followed by an *Out* message).

Accessing messages

When defining a routing rule in the Java DSL, you can access the headers and body of a message using the following DSL builder methods:

- **header(String name), body()** — Returns the named header and the body of the current *In* message.
- **outBody()** — Returns the body of the current *Out* message.

For example, to populate the *In* message's **username** header, you can use the following Java DSL route:

```
from(SourceURL).setHeader("username", "John.Doe").to(TargetURL);
```

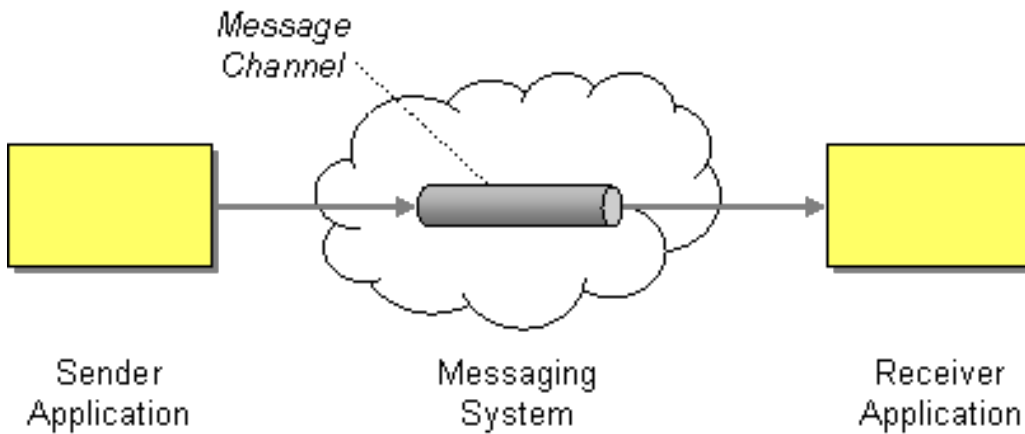
4.2. MESSAGE CHANNEL

Overview

A *message channel* is a logical channel in a messaging system. That is, sending messages to different message channels provides an elementary way of sorting messages into different message types. Message queues and message topics are examples of message channels. You should remember that a logical channel is *not* the same as a physical channel. There can be several different ways of physically realizing a logical channel.

In Apache Camel, a message channel is represented by an endpoint URI of a message-oriented component as shown in [Figure 4.2, “Message Channel Pattern”](#).

Figure 4.2. Message Channel Pattern



Message-oriented components

The following message-oriented components in Apache Camel support the notion of a message channel:

- [ActiveMQ](#)
- [JMS](#)
- [AMQP](#)

ActiveMQ

In ActiveMQ, message channels are represented by *queues* or *topics*. The endpoint URI for a specific queue, *QueueName*, has the following format:

```
activemq:QueueName
```

The endpoint URI for a specific topic, *TopicName*, has the following format:

```
activemq:topic:TopicName
```

For example, to send messages to the queue, **Foo.Bar**, use the following endpoint URI:

```
activemq:Foo.Bar
```

See [chapter "ActiveMQ" in "EIP Component Reference"](#) for more details and instructions on setting up the ActiveMQ component.

JMS

The Java Messaging Service (JMS) is a generic wrapper layer that is used to access many different kinds of message systems (for example, you can use it to wrap ActiveMQ, MQSeries, Tibco, BEA, Sonic, and others). In JMS, message channels are represented by queues, or topics. The endpoint URI for a specific queue, *QueueName*, has the following format:

```
jms:QueueName
```

The endpoint URI for a specific topic, *TopicName*, has the following format:

```
jms:topic:TopicName
```

See [chapter "JMS" in "EIP Component Reference"](#) for more details and instructions on setting up the JMS component.

AMQP

In AMQP, message channels are represented by queues, or topics. The endpoint URI for a specific queue, *QueueName*, has the following format:

```
amqp:QueueName
```

The endpoint URI for a specific topic, *TopicName*, has the following format:

```
amqp:topic:TopicName
```

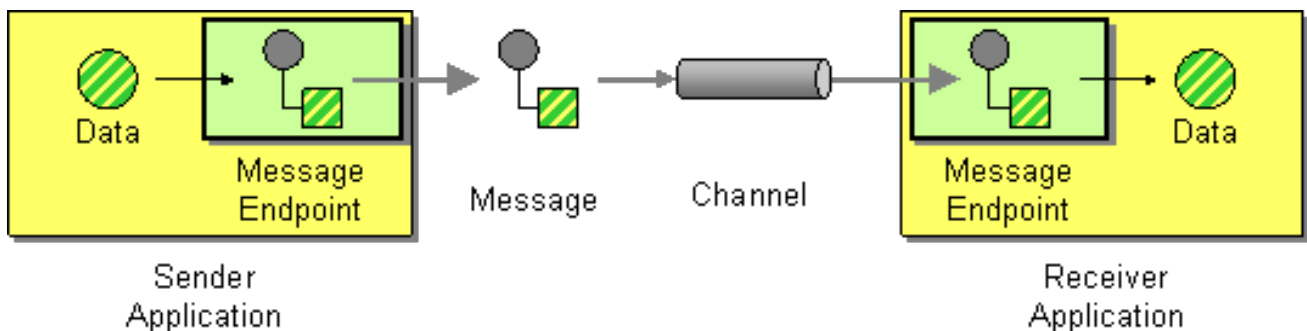
See [chapter "AMQP" in "EIP Component Reference"](#) for more details and instructions on setting up the AMQP component.

4.3. MESSAGE ENDPOINT

Overview

A *message endpoint* is the interface between an application and a messaging system. As shown in [Figure 4.3, "Message Endpoint Pattern"](#), you can have a sender endpoint, sometimes called a proxy or a service consumer, which is responsible for sending *In* messages, and a receiver endpoint, sometimes called an endpoint or a service, which is responsible for receiving *In* messages.

Figure 4.3. Message Endpoint Pattern



Types of endpoint

Apache Camel defines two basic types of endpoint:

- *Consumer endpoint* — Appears at the start of a Apache Camel route and reads *In* messages from an incoming channel (equivalent to a *receiver* endpoint).
- *Producer endpoint* — Appears at the end of a Apache Camel route and writes *In* messages to an outgoing channel (equivalent to a *sender* endpoint). It is possible to define a route with multiple producer endpoints.

Endpoint URIs

In Apache Camel, an endpoint is represented by an *endpoint URI*, which typically encapsulates the following kinds of data:

- *Endpoint URI for a consumer endpoint*— Advertises a specific location (for example, to expose a service to which senders can connect). Alternatively, the URI can specify a message source, such as a message queue. The endpoint URI can include settings to configure the endpoint.
- *Endpoint URI for a producer endpoint*— Contains details of where to send messages and includes the settings to configure the endpoint. In some cases, the URI specifies the location of a remote receiver endpoint; in other cases, the destination can have an abstract form, such as a queue name.

An endpoint URI in Apache Camel has the following general form:

```
ComponentPrefix:ComponentSpecificURI
```

Where *ComponentPrefix* is a URI prefix that identifies a particular Apache Camel component (see "[EIP Component Reference](#)" for details of all the supported components). The remaining part of the URI, *ComponentSpecificURI*, has a syntax defined by the particular component. For example, to connect to the JMS queue, **Foo.Bar**, you can define an endpoint URI like the following:

```
jms:Foo.Bar
```

To define a route that connects the consumer endpoint, **file://local/router/messages/foo**, directly to the producer endpoint, **jms:Foo.Bar**, you can use the following Java DSL fragment:

```
from("file://local/router/messages/foo").to("jms:Foo.Bar");
```

Alternatively, you can define the same route in XML, as follows:

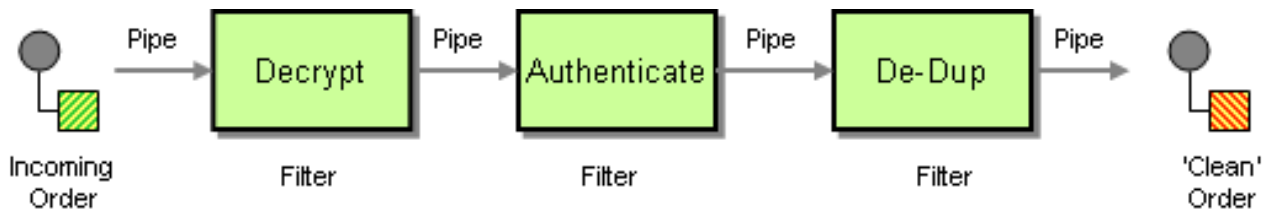
```
<camelContext id="CamelContextID"
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://local/router/messages/foo"/>
    <to uri="jms:Foo.Bar"/>
  </route>
</camelContext>
```

4.4. PIPES AND FILTERS

Overview

The *pipes and filters* pattern, shown in [Figure 4.4, “Pipes and Filters Pattern”](#), describes a way of constructing a route by creating a chain of filters, where the output of one filter is fed into the input of the next filter in the pipeline (analogous to the UNIX **pipe** command). The advantage of the pipeline approach is that it enables you to compose services (some of which can be external to the Apache Camel application) to create more complex forms of message processing.

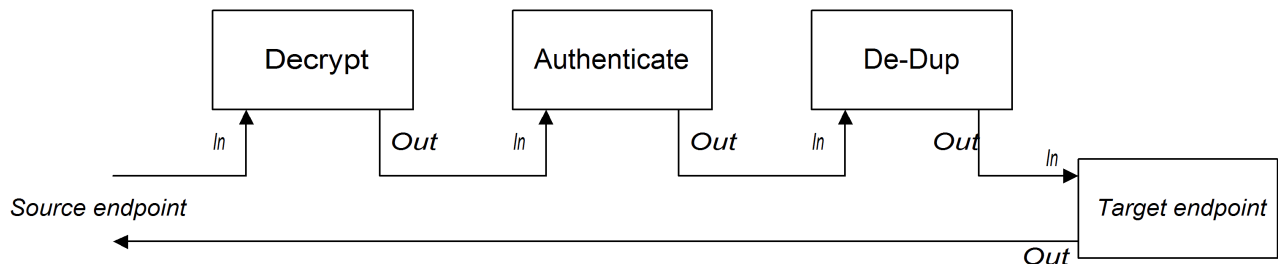
Figure 4.4. Pipes and Filters Pattern



Pipeline for the InOut exchange pattern

Normally, all of the endpoints in a pipeline have an input (*In* message) and an output (*Out* message), which implies that they are compatible with the *InOut* message exchange pattern. A typical message flow through an *InOut* pipeline is shown in Figure 4.5, “Pipeline for InOut Exchanges”.

Figure 4.5. Pipeline for InOut Exchanges



Where the pipeline connects the output of each endpoint to the input of the next one. The *Out* message from the final endpoint gets sent back to the original caller. You can define a route for this pipeline, as follows:

```
from("jms:RawOrders").pipeline("cxf:bean:decrypt",
    "cxf:bean:authenticate", "cxf:bean:dedup", "jms:CleanOrders");
```

The same route can be configured in XML, as follows:

```
<camelContext id="buildPipeline"
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="jms:RawOrders"/>
    <to uri="cxf:bean:decrypt"/>
    <to uri="cxf:bean:authenticate"/>
    <to uri="cxf:bean:dedup"/>
    <to uri="jms:CleanOrders"/>
  </route>
</camelContext>
```

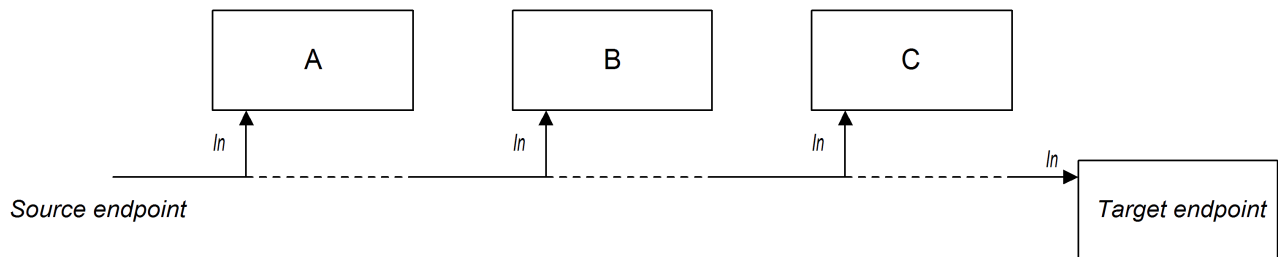
There is no dedicated pipeline element in XML. The preceding combination of **from** and **to** elements is semantically equivalent to a pipeline. See the section called “Comparison of pipeline() and to() DSL commands”.

Pipeline for the InOnly and RobustInOnly exchange patterns

When there are no *Out* messages available from the endpoints in the pipeline (as is the case for the **InOnly** and **RobustInOnly** exchange patterns), a pipeline cannot be connected in the normal way. In this special case, the pipeline is constructed by passing a copy of the original *In* message to each of the

endpoints in the pipeline, as shown in [Figure 4.6, “Pipeline for InOnly Exchanges”](#). This type of pipeline is equivalent to a recipient list with fixed destinations(see [Section 7.3, “Recipient List”](#)).

Figure 4.6. Pipeline for InOnly Exchanges



The route for this pipeline is defined using the same syntax as an *InOut* pipeline (either in Java DSL or in XML).

Comparison of pipeline() and to() DSL commands

In the Java DSL, you can define a pipeline route using either of the following syntaxes:

- *Using the pipeline() processor command*— Use the pipeline processor to construct a pipeline route as follows:

```
from(SourceURI).pipeline(FilterA, FilterB, TargetURI);
```

- *Using the to() command*— Use the **to()** command to construct a pipeline route as follows:

```
from(SourceURI).to(FilterA, FilterB, TargetURI);
```

Alternatively, you can use the equivalent syntax:

```
from(SourceURI).to(FilterA).to(FilterB).to(TargetURI);
```

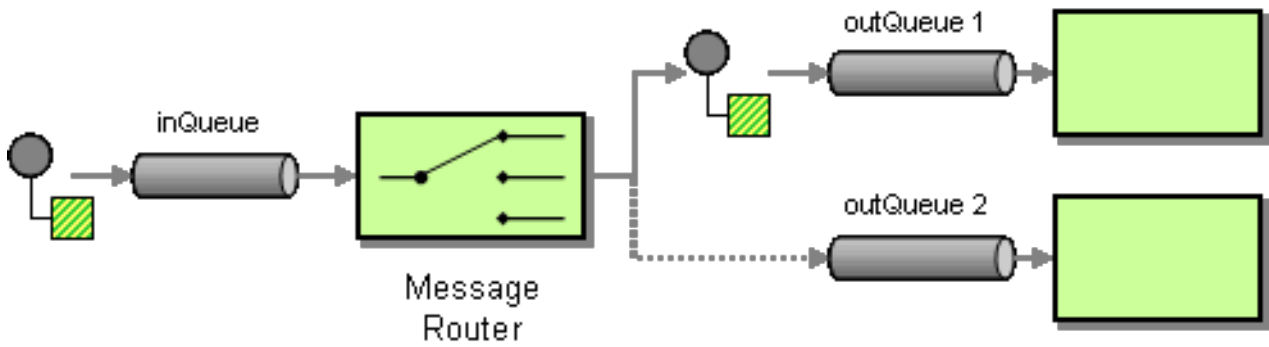
Exercise caution when using the **to()** command syntax, because it is *not* always equivalent to a pipeline processor. In Java DSL, the meaning of **to()** can be modified by the preceding command in the route. For example, when the **multicast()** command precedes the **to()** command, it binds the listed endpoints into a multicast pattern, instead of a pipeline pattern(see [Section 7.11, “Multicast”](#)).

4.5. MESSAGE ROUTER

Overview

A *message router*, shown in [Figure 4.7, “Message Router Pattern”](#), is a type of filter that consumes messages from a single consumer endpoint and redirects them to the appropriate target endpoint, based on a particular decision criterion. A message router is concerned only with redirecting messages; it does not modify the message content.

Figure 4.7. Message Router Pattern



A message router can easily be implemented in Apache Camel using the **choice()** processor, where each of the alternative target endpoints can be selected using a **when()** subclause (for details of the choice processor, see [Section 1.5, “Processors”](#)).

Java DSL example

The following Java DSL example shows how to route messages to three alternative destinations (either **seda:a**, **seda:b**, or **seda:c**) depending on the contents of the **foo** header:

```
from("seda:a").choice()
    .when(header("foo").isEqualTo("bar")).to("seda:b")
    .when(header("foo").isEqualTo("cheese")).to("seda:c")
    .otherwise().to("seda:d");
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildSimpleRouteWithChoice"
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

Choice without otherwise

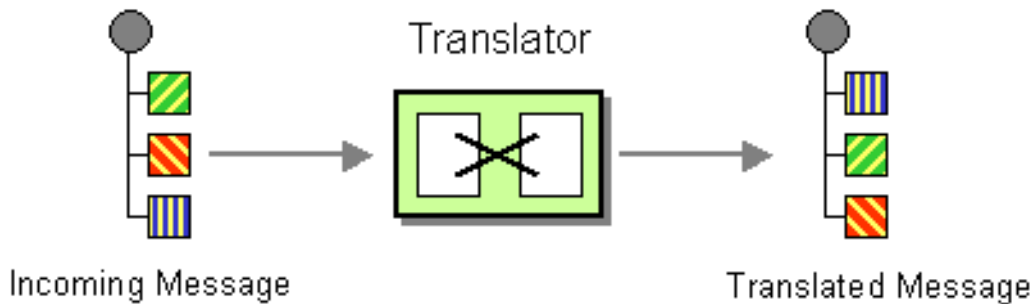
If you use `choice()` without an `otherwise()` clause, any unmatched exchanges are dropped by default.

4.6. MESSAGE TRANSLATOR

Overview

The *message translator* pattern, shown in [Figure 4.8, “Message Translator Pattern”](#) describes a component that modifies the contents of a message, translating it to a different format. You can use Apache Camel's bean integration feature to perform the message translation.

Figure 4.8. Message Translator Pattern



Bean integration

You can transform a message using bean integration, which enables you to call a method on any registered bean. For example, to call the method, `myMethodName()`, on the bean with ID, `myTransformerBean`:

```
from("activemq:SomeQueue")
    .beanRef("myTransformerBean", "myMethodName")
    .to("mqseries:AnotherQueue");
```

Where the `myTransformerBean` bean is defined in either a Spring XML file or in JNDI. If, you omit the method name parameter from `beanRef()`, the bean integration will try to deduce the method name to invoke by examining the message exchange.

You can also add your own explicit **Processor** instance to perform the transformation, as follows:

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

Or, you can use the DSL to explicitly configure the transformation, as follows:

```
from("direct:start").setBody(body().append(" World!")).to("mock:result");
```

You can also use *templating* to consume a message from one destination, transform it with something like [Velocity](#) or XQuery and then send it on to another destination. For example, using the *InOnly* exchange pattern (one-way messaging) :

-

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

If you want to use *InOut* (request-reply) semantics to process requests on the **My.Queue** queue on [ActiveMQ](#) with a template generated response, then you could use a route like the following to send responses back to the **JMSReplyTo** destination:

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

CHAPTER 5. MESSAGING CHANNELS

Abstract

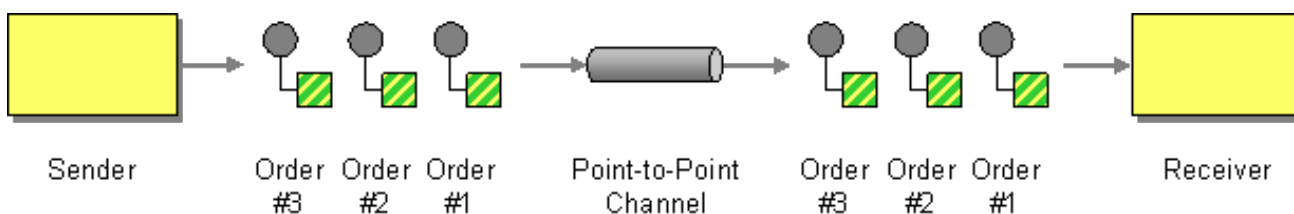
Messaging channels provide the plumbing for a messaging application. This chapter describes the different kinds of messaging channels available in a messaging system, and the roles that they play.

5.1. POINT-TO-POINT CHANNEL

Overview

A *point-to-point channel*, shown in [Figure 5.1, “Point to Point Channel Pattern”](#) is a [message channel](#) that guarantees that only one receiver consumes any given message. This is in contrast with a [publish-subscribe channel](#), which allows multiple receivers to consume the same message. In particular, with a point-to-point channel, it is possible for multiple receivers to subscribe to the same channel. If more than one receiver competes to consume a message, it is up to the message channel to ensure that only one receiver actually consumes the message.

Figure 5.1. Point to Point Channel Pattern



Components that support point-to-point channel

The following Apache Camel components support the point-to-point channel pattern:

- [JMS](#)
- [ActiveMQ](#)
- [SEDA](#)
- [JPA](#)
- [XMPP](#)

JMS

In JMS, a point-to-point channel is represented by a *queue*. For example, you can specify the endpoint URI for a JMS queue called **Foo.Bar** as follows:

```
jms:queue:Foo.Bar
```

The qualifier, **queue:**, is optional, because the JMS component creates a queue endpoint by default. Therefore, you can also specify the following equivalent endpoint URI:

```
jms:Foo.Bar
```

See [chapter "JMS" in "EIP Component Reference"](#) for more details.

ActiveMQ

In ActiveMQ, a point-to-point channel is represented by a queue. For example, you can specify the endpoint URI for an ActiveMQ queue called **Foo.Bar** as follows:

```
activemq:queue:Foo.Bar
```

See [chapter "ActiveMQ" in "EIP Component Reference"](#) for more details.

SEDA

The Apache Camel Staged Event-Driven Architecture (SEDA) component is implemented using a blocking queue. Use the SEDA component if you want to create a lightweight point-to-point channel that is *internal* to the Apache Camel application. For example, you can specify an endpoint URI for a SEDA queue called **SedaQueue** as follows:

```
seda:SedaQueue
```

JPA

The Java Persistence API (JPA) component is an EJB 3 persistence standard that is used to write entity beans out to a database. See [chapter "JPA" in "EIP Component Reference"](#) for more details.

XMPP

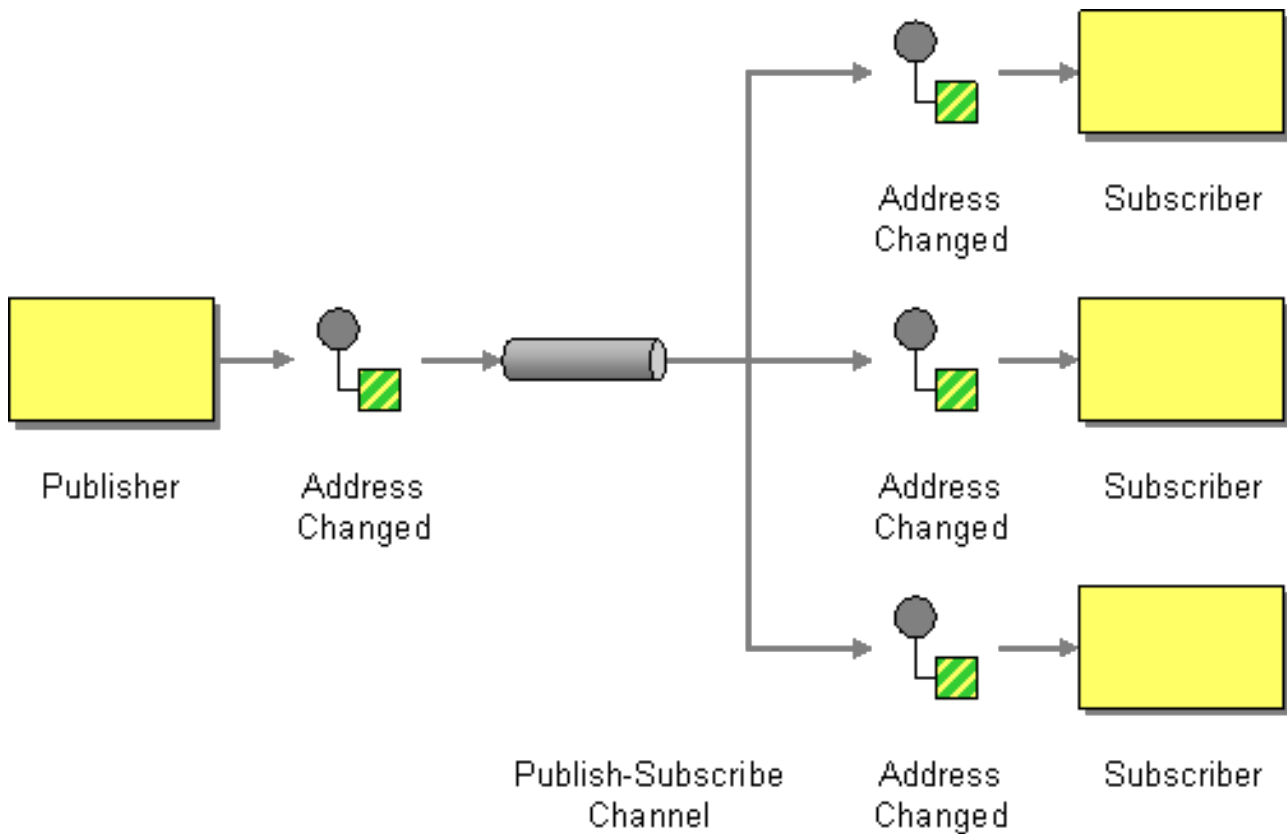
The XMPP (Jabber) component supports the point-to-point channel pattern when it is used in the person-to-person mode of communication. See [chapter "XMPP" in "EIP Component Reference"](#) for more details.

5.2. PUBLISH-SUBSCRIBE CHANNEL

Overview

A *publish-subscribe channel*, shown in [Figure 5.2, "Publish Subscribe Channel Pattern"](#), is a [message channel](#) that enables multiple subscribers to consume any given message. This is in contrast with a [point-to-point channel](#). Publish-subscribe channels are frequently used as a means of broadcasting events or notifications to multiple subscribers.

Figure 5.2. Publish Subscribe Channel Pattern



Components that support publish-subscribe channel

The following Apache Camel components support the publish-subscribe channel pattern:

- [JMS](#)
- [ActiveMQ](#)
- [XMPP](#)
- [SEDA](#) for working with SEDA in the same [CamelContext](#) which can work in pub-sub, but allowing multiple consumers.
- [VM](#) as SEDA, but for use within the same JVM.

JMS

In JMS, a publish-subscribe channel is represented by a *topic*. For example, you can specify the endpoint URI for a JMS topic called **StockQuotes** as follows:

```
jms:topic:StockQuotes
```

See [chapter "JMS" in "EIP Component Reference"](#) for more details.

ActiveMQ

In ActiveMQ, a publish-subscribe channel is represented by a topic. For example, you can specify the endpoint URI for an ActiveMQ topic called **StockQuotes**, as follows:

■

```
activemq:topic:StockQuotes
```

See [chapter "ActiveMQ" in "EIP Component Reference"](#) for more details.

XMPP

The XMPP (Jabber) component supports the publish-subscribe channel pattern when it is used in the group communication mode. See [chapter "XMPP" in "EIP Component Reference"](#) for more details.

Static subscription lists

If you prefer, you can also implement publish-subscribe logic within the Apache Camel application itself. A simple approach is to define a *static subscription list*, where the target endpoints are all explicitly listed at the end of the route. However, this approach is not as flexible as a JMS or ActiveMQ topic.

Java DSL example

The following Java DSL example shows how to simulate a publish-subscribe channel with a single publisher, `seda:a`, and three subscribers, `seda:b`, `seda:c`, and `seda:d`:

```
from("seda:a").to("seda:b", "seda:c", "seda:d");
```



NOTE

This only works for the *InOnly* message exchange pattern.

XML configuration example

The following example shows how to configure the same route in XML:

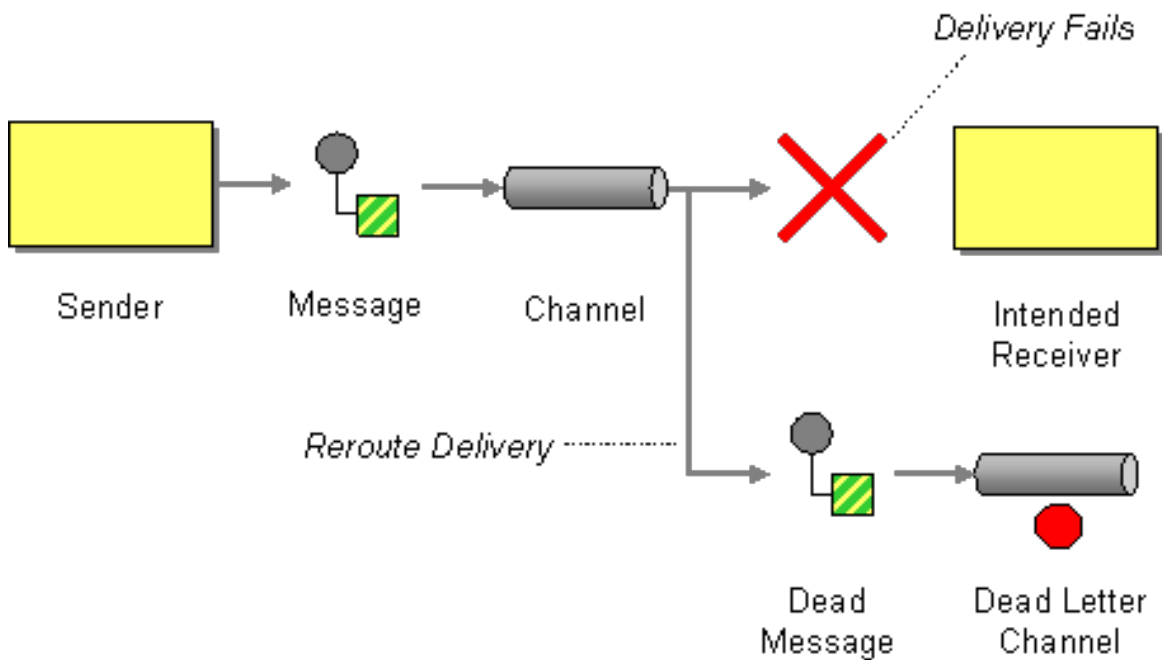
```
<camelContext id="buildStaticRecipientList"
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

5.3. DEAD LETTER CHANNEL

Overview

The *dead letter channel* pattern, shown in [Figure 5.3, "Dead Letter Channel Pattern"](#), describes the actions to take when the messaging system fails to deliver a message to the intended recipient. This includes such features as retrying delivery and, if delivery ultimately fails, sending the message to a dead letter channel, which archives the undelivered messages.

Figure 5.3. Dead Letter Channel Pattern



Creating a dead letter channel in Java DSL

The following example shows how to create a dead letter channel using Java DSL:

```
errorHandler(deadLetterChannel("seda:errors"));
from("seda:a").to("seda:b");
```

Where the `errorHandler()` method is a Java DSL interceptor, which implies that *all* of the routes defined in the current route builder are affected by this setting. The `deadLetterChannel()` method is a Java DSL command that creates a new dead letter channel with the specified destination endpoint, `seda:errors`.

The `errorHandler()` interceptor provides a catch-all mechanism for handling *all* error types. If you want to apply a more fine-grained approach to exception handling, you can use the `onException` clauses instead (see [the section called "onException clause"](#)).

XML DSL example

You can define a dead letter channel in the XML DSL, as follows:

```
<route errorHandlerRef="myDeadLetterErrorHandler">
  ...
</route>

<bean id="myDeadLetterErrorHandler"
class="org.apache.camel.builder.DeadLetterChannelBuilder">
  <property name="deadLetterUri" value="jms:queue:dead"/>
  <property name="redeliveryPolicy" ref="myRedeliveryPolicyConfig"/>
</bean>

<bean id="myRedeliveryPolicyConfig"
class="org.apache.camel.processor.RedeliveryPolicy">
```



```

    <property name="maximumRedeliveries" value="3"/>
    <property name="redeliveryDelay" value="5000"/>
</bean>

```

Redelivery policy

Normally, you do not send a message straight to the dead letter channel, if a delivery attempt fails. Instead, you re-attempt delivery up to some maximum limit, and after all redelivery attempts fail you would send the message to the dead letter channel. To customize message redelivery, you can configure the dead letter channel to have a *redelivery policy*. For example, to specify a maximum of two redelivery attempts, and to apply an exponential backoff algorithm to the time delay between delivery attempts, you can configure the dead letter channel as follows:

```

errorHandler(deadLetterChannel("seda:errors").maximumRedeliveries(2).useExponentialBackOff());
from("seda:a").to("seda:b");

```

Where you set the redelivery options on the dead letter channel by invoking the relevant methods in a chain (each method in the chain returns a reference to the current **RedeliveryPolicy** object).

[Table 5.1, “Redelivery Policy Settings”](#) summarizes the methods that you can use to set redelivery policies.

Table 5.1. Redelivery Policy Settings

Method Signature	Default	Description
backOffMultiplier(double multiplier)	2	If exponential backoff is enabled, let m be the backoff multiplier and let d be the initial delay. The sequence of redelivery attempts are then timed as follows: <pre> d, m*d, m*m*d, m*m*m*d, ... </pre>
collisionAvoidancePercent(double collisionAvoidancePercent)	15	If collision avoidance is enabled, let p be the collision avoidance percent. The collision avoidance policy then tweaks the next delay by a random amount, up to plus/minus p% of its current value.
delayPattern(String delayPattern)	<i>None</i>	Apache Camel 2.0:
disableRedelivery()	true	Apache Camel 2.0: Disables the redelivery feature. To enable redelivery, set maximumRedeliveries() to a positive integer value.

Method Signature	Default	Description
<code>handled(boolean handled)</code>	<code>true</code>	Apache Camel 2.0: If <code>true</code> , the current exception is cleared when the message is moved to the dead letter channel; if <code>false</code> , the exception is propagated back to the client.
<code>initialRedeliveryDelay(long initialRedeliveryDelay)</code>	<code>1000</code>	Specifies the delay (in milliseconds) before attempting the first redelivery.
<code>logStackTrace(boolean logStackTrace)</code>	<code>false</code>	Apache Camel 2.0: If <code>true</code> , the JVM stack trace is included in the error logs.
<code>maximumRedeliveries(int maximumRedeliveries)</code>	<code>0</code>	Apache Camel 2.0: Maximum number of delivery attempts.
<code>maximumRedeliveryDelay(long maxDelay)</code>	<code>60000</code>	Apache Camel 2.0: When using an exponential backoff strategy (see <code>useExponentialBackOff()</code>), it is theoretically possible for the redelivery delay to increase without limit. This property imposes an upper limit on the redelivery delay (in milliseconds)
<code>onRedelivery(Processor processor)</code>	<code>None</code>	Apache Camel 2.0: Configures a processor that gets called before every redelivery attempt.
<code>redeliveryDelay(long int)</code>	<code>0</code>	Apache Camel 2.0: Specifies the delay (in milliseconds) between redelivery attempts.
<code>retriesExhaustedLogLevel(LoggingLevel logLevel)</code>	<code>LogLevel.ERROR</code>	Apache Camel 2.0: Specifies the logging level at which to log delivery failure (specified as an <code>org.apache.camel.LoggingLevel</code> constant).
<code>retryAttemptedLogLevel(LoggingLevel logLevel)</code>	<code>LogLevel.DEBUG</code>	Apache Camel 2.0: Specifies the logging level at which to redelivery attempts (specified as an <code>org.apache.camel.LoggingLevel</code> constant).

Method Signature	Default	Description
<code>useCollisionAvoidance()</code>	<code>false</code>	Enables collision avoidance, which adds some randomization to the backoff timings to reduce contention probability.
<code>useOriginalMessage()</code>	<code>false</code>	Apache Camel 2.0: If this feature is enabled, the message sent to the dead letter channel is a copy of the <i>original</i> message exchange, as it existed at the beginning of the route (in the from() node).
<code>useExponentialBackOff()</code>	<code>false</code>	Enables exponential backoff.

Redelivery headers

If Apache Camel attempts to redeliver a message, it automatically sets the headers described in [Table 5.2, “Dead Letter Redelivery Headers”](#) on the *In* message.

Table 5.2. Dead Letter Redelivery Headers

Header Name	Type	Description
<code>CamelRedeliveryCounter</code>	<code>Integer</code>	Apache Camel 2.0: Counts the number of unsuccessful delivery attempts. This value is also set in Exchange.REDELIVERY_COUNTER .
<code>CamelRedelivered</code>	<code>Boolean</code>	Apache Camel 2.0: True, if one or more redelivery attempts have been made. This value is also set in Exchange.REDELIVERED .
<code>CamelRedeliveryMaxCounter</code>	<code>Integer</code>	Apache Camel 2.6: Holds the maximum redelivery setting (also set in the Exchange.REDELIVERY_MAX_COUNTER exchange property). This header is absent if you use retryWhile or have unlimited maximum redelivery configured.

Using the original message

Available as of Apache Camel 2.0 Because an exchange object is subject to modification as it passes through the route, the exchange that is current when an exception is raised is not necessarily the copy that you would want to store in the dead letter channel. In many cases, it is preferable to log the

message that arrived at the start of the route, before it was subject to any kind of transformation by the route. For example, consider the following route:

```
from("jms:queue:order:input")
    .to("bean:validateOrder");
    .to("bean:transformOrder")
    .to("bean:handleOrder");
```

The preceding route listen for incoming JMS messages and then processes the messages using the sequence of beans: **validateOrder**, **transformOrder**, and **handleOrder**. But when an error occurs, we do not know in which state the message is in. Did the error happen before the **transformOrder** bean or after? We can ensure that the original message from **jms:queue:order:input** is logged to the dead letter channel by enabling the **useOriginalMessage** option as follows:

```
// will use original body
errorHandler(deadLetterChannel("jms:queue:dead")
    .useOriginalMessage().maximumRedeliveries(5).redeliveryDelay(5000));
```

Redeliver delay pattern

Available as of Apache Camel 2.0 The **delayPattern** option is used to specify delays for particular ranges of the redelivery count. The delay pattern has the following syntax:

limit1:delay1;limit2:delay2;limit3:delay3;..., where each *delayN* is applied to redeliveries in the range **limitN <= redeliveryCount < limitN+1**

For example, consider the pattern, **5:1000;10:5000;20:20000**, which defines three groups and results in the following redelivery delays:

- Attempt number 1–4 = 0 milliseconds (as the first group starts with 5).
- Attempt number 5–9 = 1000 milliseconds (the first group).
- Attempt number 10–19 = 5000 milliseconds (the second group).
- Attempt number 20– = 20000 milliseconds (the last group).

You can start a group with limit 1 to define a starting delay. For example, **1:1000;5:5000** results in the following redelivery delays:

- Attempt number 1–4 = 1000 millis (the first group)
- Attempt number 5– = 5000 millis (the last group)

There is no requirement that the next delay should be higher than the previous and you can use any delay value you like. For example, the delay pattern, **1:5000;3:1000**, starts with a 5 second delay and then reduces the delay to 1 second.

Which endpoint failed?

When Apache Camel routes messages, it updates an **Exchange** property that contains the *last* endpoint the **Exchange** was sent to. Hence, you can obtain the URI for the current exchange's most recent destination using the following code:

```
// Java
String lastEndpointUri = exchange.getProperty(Exchange.TO_ENDPOINT,
String.class);
```

Where **Exchange.TO_ENDPOINT** is a string constant equal to **CamelToEndpoint**. This property is updated whenever Camel sends a message to *any* endpoint.

If an error occurs during routing and the exchange is moved into the dead letter queue, Apache Camel will additionally set a property named **CamelFailureEndpoint**, which identifies the last destination the exchange was sent to before the error occurred. Hence, you can access the failure endpoint from within a dead letter queue using the following code:

```
// Java
String failedEndpointUri = exchange.getProperty(Exchange.FAILURE_ENDPOINT,
String.class);
```

Where **Exchange.FAILURE_ENDPOINT** is a string constant equal to **CamelFailureEndpoint**.



NOTE

These properties remain set in the current exchange, even if the failure occurs *after* the given destination endpoint has finished processing. For example, consider the following route:

```
from("activemq:queue:foo")
.to("http://someserver/somepath")
.beanRef("foo");
```

Now suppose that a failure happens in the **foo** bean. In this case the **Exchange.TO_ENDPOINT** property and the **Exchange.FAILURE_ENDPOINT** property still contain the value, <http://someserver/somepath>.

onRedelivery processor

When a dead letter channel is performing redeliveries, it is possible to configure a **Processor** that is executed just *before* every redelivery attempt. This can be used for situations where you need to alter the message before it is redelivered.

For example, the following dead letter channel is configured to call the **MyRedeliverProcessor** before redelivering exchanges:

```
// we configure our Dead Letter Channel to invoke
// MyRedeliveryProcessor before a redelivery is
// attempted. This allows us to alter the message before
errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(5)
.onRedelivery(new MyRedeliverProcessor())
// setting delay to zero is just to make unit testing faster
.redeliveryDelay(0L));
```

Where the **MyRedeliverProcessor** process is implemented as follows:

```
// This is our processor that is executed before every redelivery attempt
// here we can do what we want in the java code, such as altering the
```

```

message
public class MyRedeliverProcessor implements Processor {

    public void process(Exchange exchange) throws Exception {
        // the message is being redelivered so we can alter it

        // we just append the redelivery counter to the body
        // you can of course do all kind of stuff instead
        String body = exchange.getIn().getBody(String.class);
        int count =
exchange.getIn().getHeader(Exchange.REDELIVERY_COUNTER, Integer.class);

        exchange.getIn().setBody(body + count);

        // the maximum redelivery was set to 5
        int max =
exchange.getIn().getHeader(Exchange.REDELIVERY_MAX_COUNTER,
Integer.class);
        assertEquals(5, max);
    }
}

```

onException clause

Instead of using the `errorHandler()` interceptor in your route builder, you can define a series of `onException()` clauses that define different redelivery policies and different dead letter channels for various exception types. For example, to define distinct behavior for each of the `NullPointerException`, `IOException`, and `Exception` types, you can define the following rules in your route builder using Java DSL:

```

onException(NullPointerException.class)
    .maximumRedeliveries(1)
    .setHeader("messageInfo", "Oh dear! An NPE.")
    .to("mock:npe_error");

onException(IOException.class)
    .initialRedeliveryDelay(5000L)
    .maximumRedeliveries(3)
    .backOffMultiplier(1.0)
    .useExponentialBackOff()
    .setHeader("messageInfo", "Oh dear! Some kind of I/O exception.")
    .to("mock:io_error");

onException(Exception.class)
    .initialRedeliveryDelay(1000L)
    .maximumRedeliveries(2)
    .setHeader("messageInfo", "Oh dear! An exception.")
    .to("mock:error");

from("seda:a").to("seda:b");

```

Where the redelivery options are specified by chaining the redelivery policy methods (as listed in [Table 5.1, “Redelivery Policy Settings”](#)), and you specify the dead letter channel's endpoint using the `to()` DSL command. You can also call other Java DSL commands in the `onException()` clauses. For

example, the preceding example calls `setHeader()` to record some error details in a message header named, `messageInfo`.

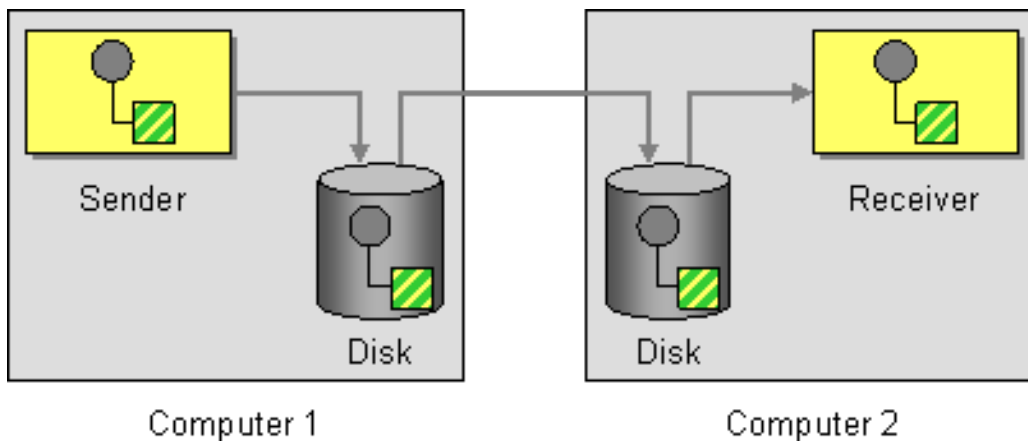
In this example, the `NullPointerException` and the `IOException` exception types are configured specially. All other exception types are handled by the generic `Exception` exception interceptor. By default, Apache Camel applies the exception interceptor that most closely matches the thrown exception. If it fails to find an exact match, it tries to match the closest base type, and so on. Finally, if no other interceptor matches, the interceptor for the `Exception` type matches all remaining exceptions.

5.4. GUARANTEED DELIVERY

Overview

Guaranteed delivery means that once a message is placed into a message channel, the messaging system guarantees that the message will reach its destination, even if parts of the application should fail. In general, messaging systems implement the guaranteed delivery pattern, shown in [Figure 5.4](#), “[Guaranteed Delivery Pattern](#)”, by writing messages to persistent storage before attempting to deliver them to their destination.

Figure 5.4. Guaranteed Delivery Pattern



Components that support guaranteed delivery

The following Apache Camel components support the guaranteed delivery pattern:

- [JMS](#)
- [ActiveMQ](#)
- [ActiveMQ Journal](#)
- [File Component](#)

JMS

In JMS, the `deliveryPersistent` query option indicates whether or not persistent storage of messages is enabled. Usually it is unnecessary to set this option, because the default behavior is to enable persistent delivery. To configure all the details of guaranteed delivery, it is necessary to set configuration options on the JMS provider. These details vary, depending on what JMS provider you are using. For example, MQSeries, TibCo, BEA, Sonic, and others, all provide various qualities of service to support guaranteed delivery.

See [chapter "JMS" in "EIP Component Reference"](#) for more details.

ActiveMQ

In ActiveMQ, message persistence is enabled by default. From version 5 onwards, ActiveMQ uses the AMQ message store as the default persistence mechanism. There are several different approaches you can use to enable message persistence in ActiveMQ.

The simplest option (different from [Figure 5.4, "Guaranteed Delivery Pattern"](#)) is to enable persistence in a central broker and then connect to that broker using a reliable protocol. After a message is sent to the central broker, delivery to consumers is guaranteed. For example, in the Apache Camel configuration file, `META-INF/spring/camel-context.xml`, you can configure the ActiveMQ component to connect to the central broker using the OpenWire/TCP protocol as follows:

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

If you prefer to implement an architecture where messages are stored locally before being sent to a remote endpoint (similar to [Figure 5.4, "Guaranteed Delivery Pattern"](#)), you do this by instantiating an embedded broker in your Apache Camel application. A simple way to achieve this is to use the ActiveMQ Peer-to-Peer protocol, which implicitly creates an embedded broker to communicate with other peer endpoints. For example, in the `camel-context.xml` configuration file, you can configure the ActiveMQ component to connect to all of the peers in group, **GroupA**, as follows:

```
<beans ... >
  ...
  <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="peer://GroupA/broker1"/>
  </bean>
  ...
</beans>
```

Where **broker1** is the broker name of the embedded broker (other peers in the group should use different broker names). One limiting feature of the Peer-to-Peer protocol is that it relies on IP multicast to locate the other peers in its group. This makes it unsuitable for use in wide area networks (and in some local area networks that do not have IP multicast enabled).

A more flexible way to create an embedded broker in the ActiveMQ component is to exploit ActiveMQ's VM protocol, which connects to an embedded broker instance. If a broker of the required name does not already exist, the VM protocol automatically creates one. You can use this mechanism to create an embedded broker with custom configuration. For example:

```
<beans ... >
  ...
  <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="vm://broker1?
brokerConfig=xbean:activemq.xml"/>
  </bean>
  ...
</beans>
```



```

</bean>
...
</beans>

```

Where `activemq.xml` is an ActiveMQ file which configures the embedded broker instance. Within the ActiveMQ configuration file, you can choose to enable one of the following persistence mechanisms:

- *AMQ persistence (the default)* — A fast and reliable message store that is native to ActiveMQ. For details, see [amqpPersistenceAdapter](#) and [AMQ Message Store](#).
- *JDBC persistence* — Uses JDBC to store messages in any JDBC-compatible database. For details, see [jdbcPersistenceAdapter](#) and [ActiveMQ Persistence](#).
- *Journal persistence* — A fast persistence mechanism that stores messages in a rolling log file. For details, see [journalPersistenceAdapter](#) and [ActiveMQ Persistence](#).
- *Kaha persistence* — A persistence mechanism developed specifically for ActiveMQ. For details, see [kahaPersistenceAdapter](#) and [ActiveMQ Persistence](#).

See [chapter "ActiveMQ" in "EIP Component Reference"](#) for more details.

ActiveMQ Journal

The ActiveMQ Journal component is optimized for a special use case where multiple, concurrent producers write messages to queues, but there is only one active consumer. Messages are stored in rolling log files and concurrent writes are aggregated to boost efficiency.

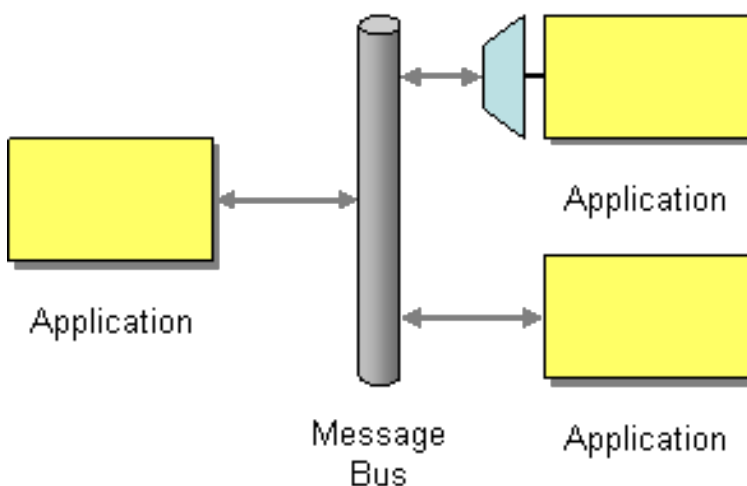
See [for more details](#).

5.5. MESSAGE BUS

Overview

Message bus refers to a messaging architecture, shown in [Figure 5.5, "Message Bus Pattern"](#), that enables you to connect diverse applications running on diverse computing platforms. In effect, the Apache Camel and its components constitute a message bus.

Figure 5.5. Message Bus Pattern



The following features of the message bus pattern are reflected in Apache Camel:

- *Common communication infrastructure* — The router itself provides the core of the common communication infrastructure in Apache Camel. However, in contrast to some message bus architectures, Apache Camel provides a heterogeneous infrastructure: messages can be sent into the bus using a wide variety of different transports and using a wide variety of different message formats.
- *Adapters* — Where necessary, Apache Camel can translate message formats and propagate messages using different transports. In effect, Apache Camel is capable of behaving like an adapter, so that external applications can hook into the message bus without refactoring their messaging protocols.

In some cases, it is also possible to integrate an adapter directly into an external application. For example, if you develop an application using Apache CXF, where the service is implemented using JAX-WS and JAXB mappings, it is possible to bind a variety of different transports to the service. These transport bindings function as adapters.

CHAPTER 6. MESSAGE CONSTRUCTION

Abstract

The message construction patterns describe the various forms and functions of the messages that pass through the system.

6.1. CORRELATION IDENTIFIER

Overview

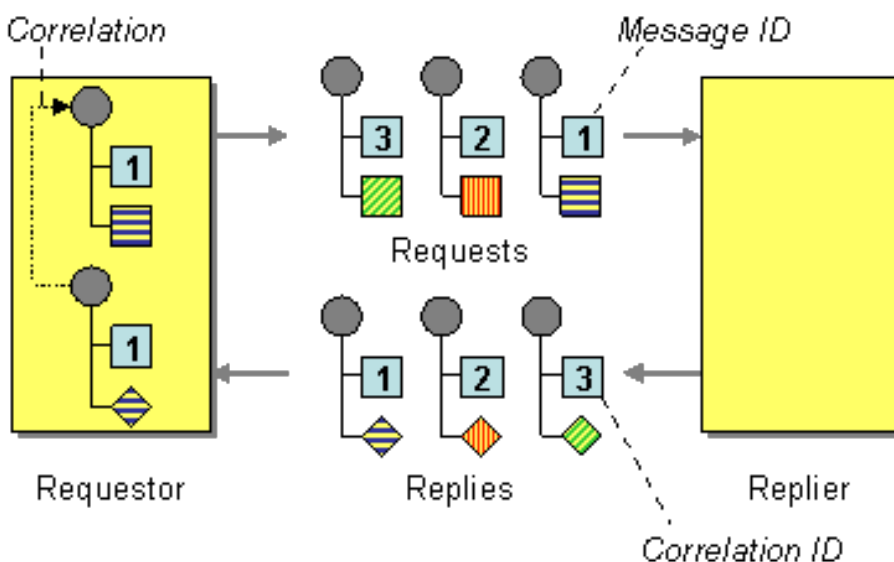
The *correlation identifier* pattern, shown in [Figure 6.1, “Correlation Identifier Pattern”](#), describes how to match reply messages with request messages, given that an asynchronous messaging system is used to implement a request-reply protocol. The essence of this idea is that request messages should be generated with a unique token, the *request ID*, that identifies the request message and reply messages should include a token, the *correlation ID*, that contains the matching request ID.

Apache Camel supports the Correlation Identifier from the EIP patterns by getting or setting a header on a Message.

When working with the [ActiveMQ](#) or [JMS](#) components, the correlation identifier header is called `JMSCorrelationID`. You can add your own correlation identifier to any message exchange to help correlate messages together in a single conversation (or business process). A correlation identifier is usually stored in a Apache Camel message header.

Some [EIP](#) patterns spin off a sub message and, in those cases, Apache Camel adds a correlation ID to the [Exchange](#) as a property with the key, `Exchange.CORRELATION_ID`, which links back to the source [Exchange](#). For example, the [Splitter](#), [Multicast](#), [Recipient List](#), and [Wire Tap](#) EIPs do this.

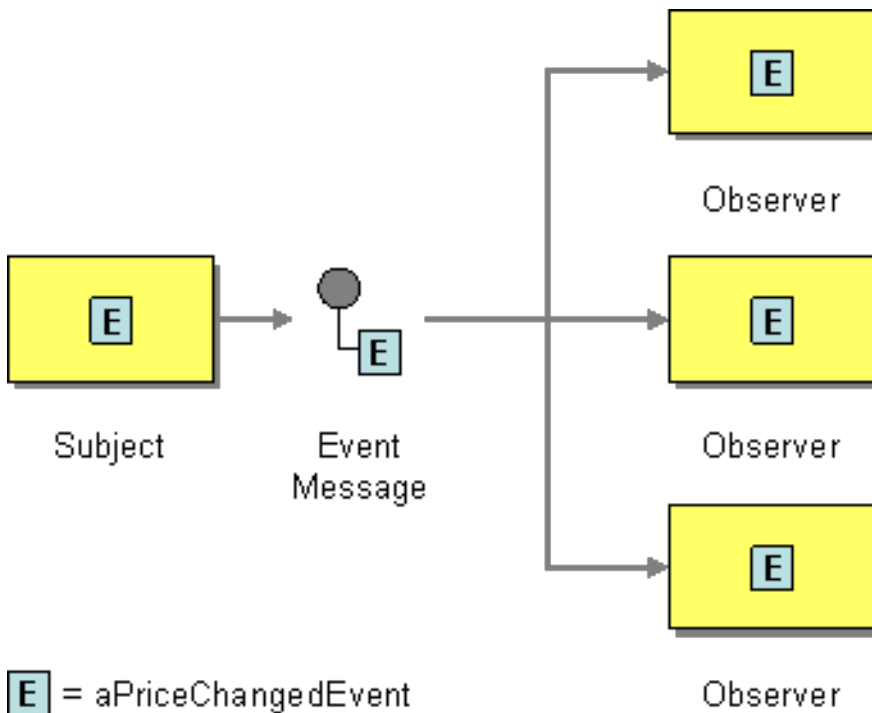
Figure 6.1. Correlation Identifier Pattern



6.2. EVENT MESSAGE

Event Message

Camel supports the [Event Message](#) from the [Introducing Enterprise Integration Patterns](#) by supporting the [Exchange Pattern](#) on a [Message](#) which can be set to **InOnly** to indicate a oneway event message. Camel [Components](#) then implement this pattern using the underlying transport or protocols.



The default behaviour of many [Components](#) is InOnly such as for [JMS](#), [File](#) or [SEDA](#)

Explicitly specifying InOnly

If you are using a component which defaults to InOut you can override the [Exchange Pattern](#) for an endpoint using the pattern property.

```
foo:bar?exchangePattern=InOnly
```

From 2.0 onwards on Camel you can specify the [Exchange Pattern](#) using the dsl.

Using the Fluent Builders

```
from("mq:someQueue").
  inOnly().
  bean(Foo.class);
```

or you can invoke an endpoint with an explicit pattern

```
from("mq:someQueue").
  inOnly("mq:anotherQueue");
```

Using the Spring XML Extensions

```
<route>
  <from uri="mq:someQueue"/>
  <inOnly uri="bean:foo"/>
</route>
```

```

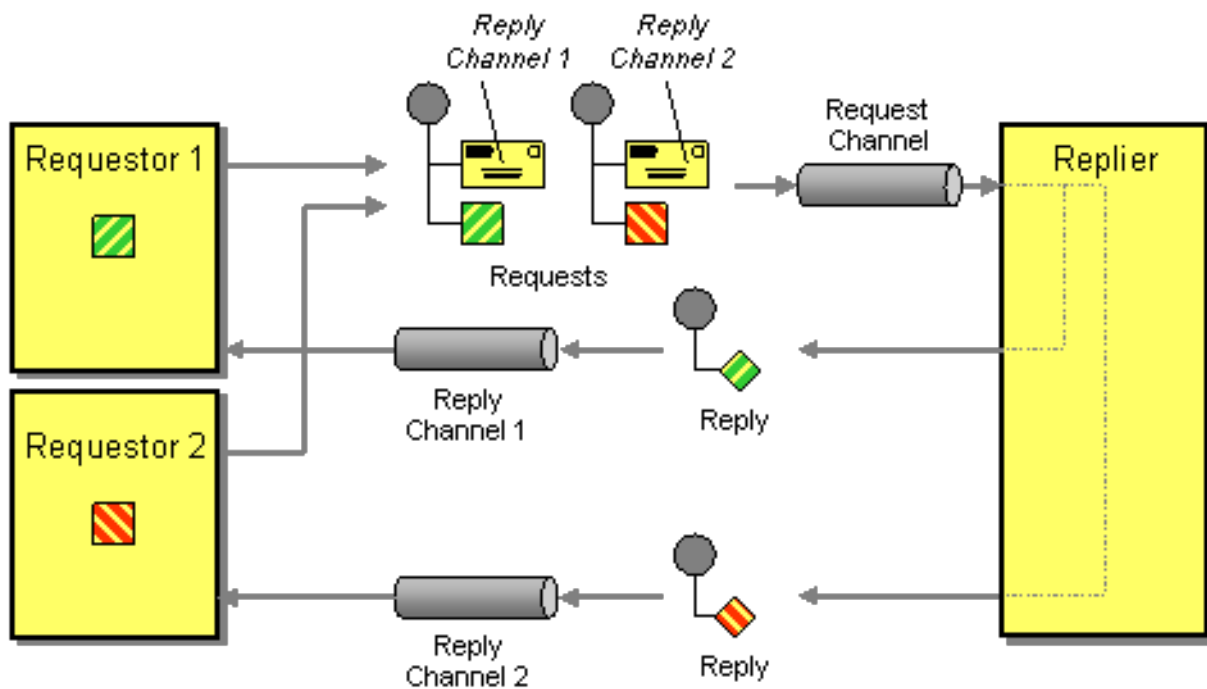
<route>
  <from uri="mq:someQueue"/>
  <inOnly uri="mq:anotherQueue"/>
</route>

```

6.3. RETURN ADDRESS

Return Address

Apache Camel supports the [Return Address](#) from the [Introducing Enterprise Integration Patterns](#) using the `JMSReplyTo` header.



For example when using [JMS](#) with *InOut*, the component will by default be returned to the address given in `JMSReplyTo`.

Example

Requestor Code

```

getMockEndpoint("mock:bar").expectedBodiesReceived("Bye World");
template.sendBodyAndHeader("direct:start", "World", "JMSReplyTo",
"queue:bar");

```

Route Using the [Fluent Builders](#)

```

from("direct:start").to("activemq:queue:foo?preserveMessageQos=true");
from("activemq:queue:foo").transform(body().prepend("Bye "));
from("activemq:queue:bar?disableReplyTo=true").to("mock:bar");

```

Route Using the [Spring XML Extensions](#)

```

<route>

```

```
<from uri="direct:start"/>
  <to uri="activemq:queue:foo?preserveMessageQos=true"/>
</route>

<route>
  <from uri="activemq:queue:foo"/>
  <transform>
    <simple>Bye ${in.body}</simple>
  </transform>
</route>

<route>
  <from uri="activemq:queue:bar?disableReplyTo=true"/>
  <to uri="mock:bar"/>
</route>
```

For a complete example of this pattern, see this [junit test case](#)

CHAPTER 7. MESSAGE ROUTING

Abstract

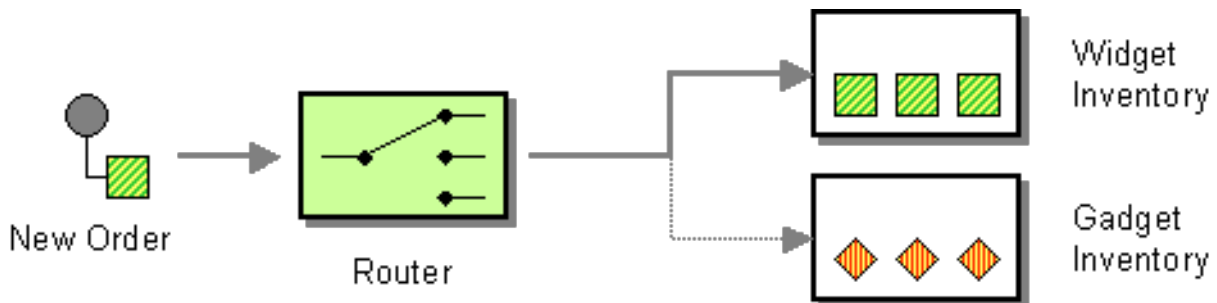
The message routing patterns describe various ways of linking message channels together. This includes various algorithms that can be applied to the message stream (without modifying the body of the message).

7.1. CONTENT-BASED ROUTER

Overview

A *content-based router*, shown in [Figure 7.1](#), “Content-Based Router Pattern”, enables you to route messages to the appropriate destination based on the message contents.

Figure 7.1. Content-Based Router Pattern



Java DSL example

The following example shows how to route a request from an input, **seda:a**, endpoint to either **seda:b**, **queue:c**, or **seda:d** depending on the evaluation of various predicate expressions:

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").choice()
            .when(header("foo").isEqualTo("bar")).to("seda:b")
            .when(header("foo").isEqualTo("cheese")).to("seda:c")
            .otherwise().to("seda:d");
    }
};
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildSimpleRouteWithChoice"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
    </choice>
  </route>
</camelContext>
```

```

    </when>
    <when>
      <xpath>$foo = 'cheese'</xpath>
      <to uri="seda:c"/>
    </when>
    <otherwise>
      <to uri="seda:d"/>
    </otherwise>
  </choice>
</route>
</camelContext>

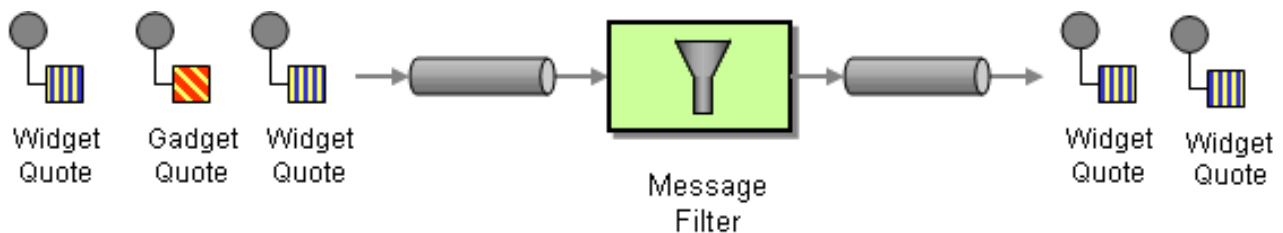
```

7.2. MESSAGE FILTER

Overview

A *message filter* is a processor that eliminates undesired messages based on specific criteria. In Apache Camel, the message filter pattern, shown in [Figure 7.2, “Message Filter Pattern”](#), is implemented by the `filter()` Java DSL command. The `filter()` command takes a single predicate argument, which controls the filter. When the predicate is **true**, the incoming message is allowed to proceed, and when the predicate is **false**, the incoming message is blocked.

Figure 7.2. Message Filter Pattern



Java DSL example

The following example shows how to create a route from endpoint, `seda:a`, to endpoint, `seda:b`, that blocks all messages except for those messages whose `foo` header have the value, `bar`:

```

RouteBuilder builder = new RouteBuilder() {
    public void configure() {

from("seda:a").filter(header("foo").isEqualTo("bar")).to("seda:b");
    }
};

```

To evaluate more complex filter predicates, you can invoke one of the supported scripting languages, such as XPath, XQuery, or SQL (see [Expression and Predicate Languages](#)). The following example defines a route that blocks all messages except for those containing a `person` element whose `name` attribute is equal to `James`:

```

from("direct:start").
    filter().xpath("/person[@name='James']").
    to("mock:result");

```


XML configuration example

The following example shows how to configure the route with an XPath predicate in XML (see [Expression and Predicate Languages](#)):

```
<camelContext id="simpleFilterRoute"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$foo = 'bar'</xpath>
      <to uri="seda:b"/>
    </filter>
  </route>
</camelContext>
```



FILTERED ENDPOINT REQUIRED INSIDE </FILTER> TAG

Make sure you put the endpoint you want to filter (for example, `<to uri="seda:b"/>`) before the closing `</filter>` tag or the filter will not be applied (in 2.8+, omitting this will result in an error).

Filtering with beans

Here is an example of using a bean to define the filter behavior:

```
from("direct:start")
  .filter().method(MyBean.class,
"isGoldCustomer").to("mock:result").end()
  .to("mock:end");

public static class MyBean {
  public boolean isGoldCustomer(@Header("level") String level) {
    return level.equals("gold");
  }
}
```

Using stop()

Available as of Camel 2.0

Stop is a special type of filter that filters out *all* messages. Stop is convenient to use in a [Content-Based Router](#) when you need to stop further processing in one of the predicates.

In the following example, we do not want messages with the word **Bye** in the message body to propagate any further in the route. We prevent this in the `when()` predicate using `.stop()`.

```
from("direct:start")
  .choice()
    .when(body().contains("Hello")).to("mock:hello")
    .when(body().contains("Bye")).to("mock:bye").stop()
    .otherwise().to("mock:other")
  .end()
  .to("mock:result");
```

Knowing if **Exchange** was filtered or not

Available as of Camel 2.5

The **Message Filter** EIP will add a property on the **Exchange** which states if it was filtered or not.

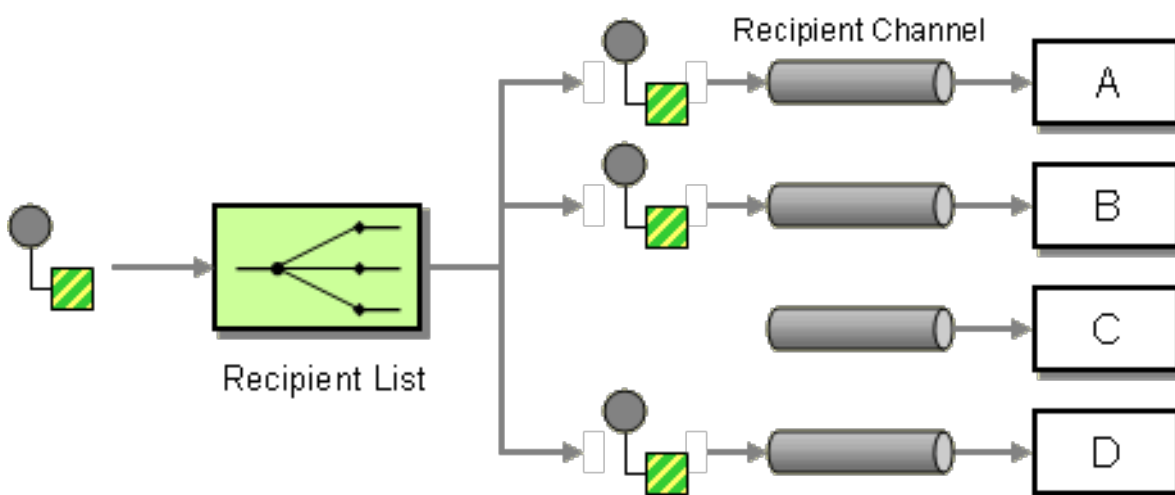
The property has the key **Exchange.FILTER_MATCHED** which has the String value of **CamelFilterMatched**. Its value is a boolean indicating **true** or **false**. If the value is **true** then the **Exchange** was routed in the filter block.

7.3. RECIPIENT LIST

Overview

A *recipient list*, shown in **Figure 7.3, “Recipient List Pattern”**, is a type of router that sends each incoming message to multiple different destinations. In addition, a recipient list typically requires that the list of recipients be calculated at run time.

Figure 7.3. Recipient List Pattern



Recipient list with fixed destinations

The simplest kind of recipient list is where the list of destinations is fixed and known in advance, and the exchange pattern is *InOnly*. In this case, you can hardwire the list of destinations into the **to()** Java DSL command.



NOTE

The examples given here, for the recipient list with fixed destinations, work *only* with the *InOnly* exchange pattern (similar to a [pipeline](#)). If you want to create a recipient list for exchange patterns with *Out* messages, use the [multicast](#) pattern instead.

Java DSL example

The following example shows how to route an *InOnly* exchange from a consumer endpoint, **queue:a**, to a fixed list of destinations:

```
from("seda:a").to("seda:b", "seda:c", "seda:d");
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildStaticRecipientList"
  xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

Recipient list calculated at run time

In most cases, when you use the recipient list pattern, the list of recipients should be calculated at runtime. To do this use the `recipientList()` processor, which takes a list of destinations as its sole argument. Because Apache Camel applies a type converter to the list argument, it should be possible to use most standard Java list types (for example, a collection, a list, or an array). For more details about type converters, see [section "Built-In Type Converters" in "Programming EIP Components"](#).

The recipients receive a copy of the *same* exchange instance and Apache Camel executes them sequentially.

Java DSL example

The following example shows how to extract the list of destinations from a message header called `recipientListHeader`, where the header value is a comma-separated list of endpoint URIs:

```
from("direct:a").recipientList(header("recipientListHeader").tokenize(",")
);
```

In some cases, if the header value is a list type, you might be able to use it directly as the argument to `recipientList()`. For example:

```
from("seda:a").recipientList(header("recipientListHeader"));
```

However, this example is entirely dependent on how the underlying component parses this particular header. If the component parses the header as a simple string, this example will *not* work. The header must be parsed into some type of Java list.

XML configuration example

The following example shows how to configure the preceding route in XML, where the header value is a comma-separated list of endpoint URIs:

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

```

    <from uri="seda:a"/>
    <recipientList delimiter=", ">
      <header>recipientListHeader</header>
    </recipientList>
  </route>
</camelContext>

```

Sending to multiple recipients in parallel

Available as of Camel 2.2

The [Recipient List](#) supports **parallelProcessing**, which is similar to the corresponding feature in [Splitter](#). Use the parallel processing feature to send the exchange to multiple recipients concurrently—for example:

```
from("direct:a").recipientList(header("myHeader")).parallelProcessing();
```

In Spring XML, the parallel processing feature is implemented as an attribute on the **recipientList** tag—for example:

```

<route>
  <from uri="direct:a"/>
  <recipientList parallelProcessing="true">
    <header>myHeader</header>
  </recipientList>
</route>

```

Stop on exception

Available as of Camel 2.2

The [Recipient List](#) supports the **stopOnException** feature, which you can use to stop sending to any further recipients, if any recipient fails.

```
from("direct:a").recipientList(header("myHeader")).stopOnException();
```

And in Spring XML its an attribute on the recipient list tag.

In Spring XML, the stop on exception feature is implemented as an attribute on the **recipientList** tag—for example:

```

<route>
  <from uri="direct:a"/>
  <recipientList stopOnException="true">
    <header>myHeader</header>
  </recipientList>
</route>

```



NOTE

You can combine **parallelProcessing** and **stopOnException** in the same route.

Ignore invalid endpoints

Available as of Camel 2.3

The [Recipient List](#) supports the `ignoreInvalidEndpoints` option, which enables the recipient list to skip invalid endpoints ([Routing Slip](#) also supports this option). For example:

```
from("direct:a").recipientList(header("myHeader")).ignoreInvalidEndpoints(
);
```

And in Spring XML, you can enable this option by setting the `ignoreInvalidEndpoints` attribute on the `recipientList` tag, as follows

```
<route>
  <from uri="direct:a"/>
  <recipientList ignoreInvalidEndpoints="true">
    <header>myHeader</header>
  </recipientList>
</route>
```

Consider the case where `myHeader` contains the two endpoints, `direct:foo,xxx:bar`. The first endpoint is valid and works. The second is invalid and, therefore, ignored. Apache Camel logs at **INFO** level whenever an invalid endpoint is encountered.

Using custom AggregationStrategy

Available as of Camel 2.2

You can use a custom `AggregationStrategy` with the [Recipient List](#), which is useful for aggregating replies from the recipients in the list. By default, Apache Camel uses the `UseLatestAggregationStrategy` aggregation strategy, which keeps just the last received reply. For a more sophisticated aggregation strategy, you can define your own implementation of the `AggregationStrategy` interface—see [Aggregator](#) EIP for details. For example, to apply the custom aggregation strategy, `MyOwnAggregationStrategy`, to the reply messages, you can define a Java DSL route as follows:

```
from("direct:a")
  .recipientList(header("myHeader")).aggregationStrategy(new
MyOwnAggregationStrategy())
  .to("direct:b");
```

In Spring XML, you can specify the custom aggregation strategy as an attribute on the `recipientList` tag, as follows:

```
<route>
  <from uri="direct:a"/>
  <recipientList strategyRef="myStrategy">
    <header>myHeader</header>
  </recipientList>
  <to uri="direct:b"/>
</route>

<bean id="myStrategy" class="com.mycompany.MyOwnAggregationStrategy"/>
```

Using custom thread pool

Available as of Camel 2.2

This is only needed when you use **parallelProcessing**. By default Camel uses a thread pool with 10 threads. Notice this is subject to change when we overhaul thread pool management and configuration later (hopefully in Camel 2.2).

You configure this just as you would with the custom aggregation strategy.

Using method call as recipient list

You can use a [Bean](#) to provide the recipients, for example:

```
from("activemq:queue:test").recipientList().method(MessageRouter.class,
"routeTo");
```

Where the **MessageRouter** bean is defined as follows:

```
public class MessageRouter {
    public String routeTo() {
        String queueName = "activemq:queue:test2";
        return queueName;
    }
}
```

Bean as recipient list

You can make a bean behave as a recipient list by adding the **@RecipientList** annotation to a methods that returns a list of recipients. For example:

```
public class MessageRouter {
    @RecipientList
    public String routeTo() {
        String queueList = "activemq:queue:test1,activemq:queue:test2";
        return queueList;
    }
}
```

In this case, do *not* include the **recipientList** DSL command in the route. Define the route as follows:

```
from("activemq:queue:test").bean(MessageRouter.class, "routeTo");
```

Using timeout

Available as of Camel 2.5

If you use **parallelProcessing**, you can configure a total **timeout** value in milliseconds. Camel will then process the messages in parallel until the timeout is hit. This allows you to continue processing if one message is slow.

In the example below, the **recipientList** header has the value, **direct:a,direct:b,direct:c**, so that the message is sent to three recipients. We have a timeout of 250 milliseconds, which means only the last two messages can be completed within the timeframe. The aggregation therefore yields the string result, **BC**.

```

from("direct:start")
    .recipientList(header("recipients"), ",")
    .aggregationStrategy(new AggregationStrategy() {
        public Exchange aggregate(Exchange oldExchange, Exchange
newExchange) {
            if (oldExchange == null) {
                return newExchange;
            }

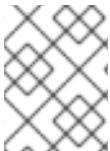
            String body = oldExchange.getIn().getBody(String.class);
            oldExchange.getIn().setBody(body +
newExchange.getIn().getBody(String.class));
            return oldExchange;
        }
    })
    .parallelProcessing().timeout(250)
    // use end to indicate end of recipientList clause
    .end()
    .to("mock:result");

from("direct:a").delay(500).to("mock:A").setBody(constant("A"));

from("direct:b").to("mock:B").setBody(constant("B"));

from("direct:c").to("mock:C").setBody(constant("C"));

```



NOTE

This **timeout** feature is also supported by **splitter** and both **multicast** and **recipientList**.

By default if a timeout occurs the **AggregationStrategy** is not invoked. However you can implement a specialized version

```

// Java
public interface TimeoutAwareAggregationStrategy extends
AggregationStrategy {

    /**
     * A timeout occurred
     *
     * @param oldExchange the oldest exchange (is <tt>null</tt> on
first aggregation as we only have the new exchange)
     * @param index       the index
     * @param total       the total
     * @param timeout     the timeout value in millis
     */
    void timeout(Exchange oldExchange, int index, int total, long
timeout);

```

This allows you to deal with the timeout in the **AggregationStrategy** if you really need to.



TIMEOUT IS TOTAL

The timeout is total, which means that after X time, Camel will aggregate the messages which has completed within the timeframe. The remainders will be cancelled. Camel will also only invoke the **timeout** method in the **TimeoutAwareAggregationStrategy** once, for the first index which caused the timeout.

Apply custom processing to the outgoing messages

Before **recipientList** sends a message to one of the recipient endpoints, it creates a message replica, which is a shallow copy of the original message. If you want to perform some custom processing on each message replica before the replica is sent to its endpoint, you can invoke the **onPrepare** DSL command in the **recipientList** clause. The **onPrepare** command inserts a custom processor just *after* the message has been shallow-copied and just *before* the message is dispatched to its endpoint. For example, in the following route, the **CustomProc** processor is invoked on the message replica for *each recipient endpoint*:

```
from("direct:start")
    .recipientList().onPrepare(new CustomProc());
```

A common use case for the **onPrepare** DSL command is to perform a deep copy of some or all elements of a message. This allows each message replica to be modified independently of the others. For example, the following **CustomProc** processor class performs a deep copy of the message body, where the message body is presumed to be of type, **BodyType**, and the deep copy is performed by the method, **BodyType.deepCopy()**.

```
// Java
import org.apache.camel.*;
...
public class CustomProc implements Processor {

    public void process(Exchange exchange) throws Exception {
        BodyType body = exchange.getIn().getBody(BodyType.class);

        // Make a _deep_ copy of of the body object
        BodyType clone = BodyType.deepCopy();
        exchange.getIn().setBody(clone);

        // Headers and attachments have already been
        // shallow-copied. If you need deep copies,
        // add some more code here.
    }
}
```

Options

The **recipientList** DSL command supports the following options:

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

delimiter	,	Delimiter used if the Expression returned multiple endpoints.
strategyRef		Refers to an AggregationStrategy to be used to assemble the replies from the recipients, into a single outgoing message from the Recipient List . By default Camel will use the last reply as the outgoing message.
parallelProcessing	false	Camel 2.2: If enables then sending messages to the recipients occurs concurrently. Note the caller thread will still wait until all messages has been fully processed, before it continues. Its only the sending and processing the replies from the recipients which happens concurrently.
executorServiceRef		Camel 2.2: Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatic implied, and you do not have to enable that option as well.
stopOnException	false	Camel 2.2: Whether or not to stop continue processing immediately when an exception occurred. If disable, then Camel will send the message to all recipients regardless if one of them failed. You can deal with exceptions in the AggregationStrategy class where you have full control how to handle that.
ignoreInvalidEndpoints	false	Camel 2.3: If an endpoint uri could not be resolved, should it be ignored. Otherwise Camel will thrown an exception stating the endpoint uri is not valid.
streaming	false	Camel 2.5: If enabled then Camel will process replies out-of-order, eg in the order they come back. If disabled, Camel will process replies in the same order as the Expression specified.

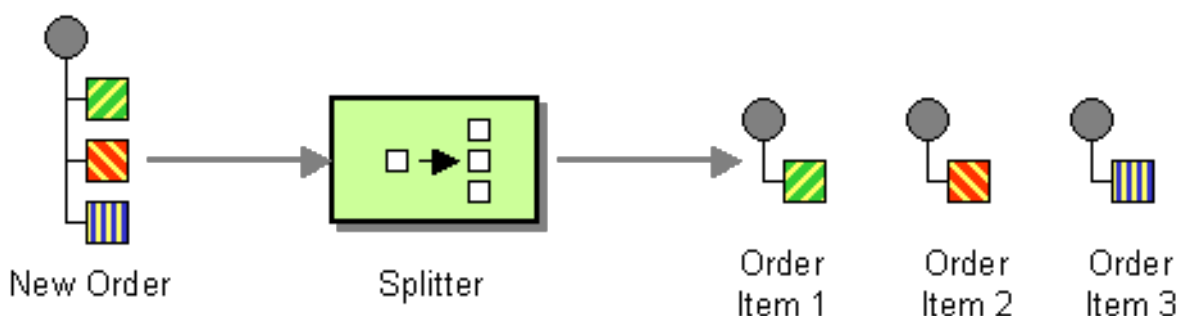
<code>timeout</code>		Camel 2.5: Sets a total timeout specified in millis. If the Recipient List hasn't been able to send and process all replies within the given timeframe, then the timeout triggers and the Recipient List breaks out and continues. Notice if you provide a TimeoutAwareAggregationStrategy then the <code>timeout</code> method is invoked before breaking out.
<code>onPrepareRef</code>		Camel 2.8: Refers to a custom Processor to prepare the copy of the Exchange each recipient will receive. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.
<code>shareUnitOfWork</code>	<code>false</code>	Camel 2.8: Whether the unit of work should be shared. See the same option on Splitter for more details.

7.4. SPLITTER

Overview

A *splitter* is a type of router that splits an incoming message into a series of outgoing messages. Each of the outgoing messages contains a piece of the original message. In Apache Camel, the splitter pattern, shown in [Figure 7.4, “Splitter Pattern”](#), is implemented by the `split()` Java DSL command.

Figure 7.4. Splitter Pattern



The Apache Camel splitter actually supports two patterns, as follows:

- *Simple splitter*—implements the splitter pattern on its own.
- *Splitter/aggregator*—combines the splitter pattern with the aggregator pattern, such that the pieces of the message are recombined after they have been processed.

Java DSL example

The following example defines a route from **seda:a** to **seda:b** that splits messages by converting each line of an incoming message into a separate outgoing message:

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a")
            .split(bodyAs(String.class).tokenize("\n"))
            .to("seda:b");
    }
};
```

The splitter can use any expression language, so you can split messages using any of the supported scripting languages, such as XPath, XQuery, or SQL (see ["Routing Expression and Predicate Languages"](#)). The following example extracts **bar** elements from an incoming message and insert them into separate outgoing messages:

```
from("activemq:my.queue")
    .split(xpath("//foo/bar"))
    .to("file://some/directory")
```

XML configuration example

The following example shows how to configure a splitter route in XML, using the XPath scripting language:

```
<camelContext id="buildSplitter"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <split>
            <xpath>//foo/bar</xpath>
            <to uri="seda:b"/>
        </split>
    </route>
</camelContext>
```

You can use the `tokenize` expression in the XML DSL to split bodies or headers using a token, where the `tokenize` expression is defined using the **tokenize** element. In the following example, the message body is tokenized using the `\n` separator character. To use a regular expression pattern, set **regex=true** in the **tokenize** element.

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split>
            <tokenize token="\n"/>
            <to uri="mock:result"/>
        </split>
    </route>
</camelContext>
```

Splitting into groups of lines

To split a big file into chunks of 1000 lines, you can define a splitter route as follows in the Java DSL:

```
from("file:inbox")
    .split().tokenize("\n", 1000).streaming()
    .to("activemq:queue:order");
```

The second argument to **tokenize** specifies the number of lines that should be grouped into a single chunk. The **streaming()** clause directs the splitter not to read the whole file at once (resulting in much better performance if the file is large).

The same route can be defined in XML DSL as follows:

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="\n" group="1000"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

The output when using the **group** option is always of **java.lang.String** type.

Splitter reply

If the exchange that enters the splitter has the *InOut* message-exchange pattern (that is, a reply is expected), the splitter returns a copy of the original input message as the reply message in the *Out* message slot. You can override this default behavior by implementing your own [aggregation strategy](#).

Parallel execution

If you want to execute the resulting pieces of the message in parallel, you can enable the parallel processing option, which instantiates a thread pool to process the message pieces. For example:

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
from("activemq:my.queue").split(xPathBuilder).parallelProcessing().to("act
ivemq:my.parts");
```

You can customize the underlying **ThreadPoolExecutor** used in the parallel splitter. For example, you can specify a custom executor in the Java DSL as follows:

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(8, 16, 0L,
    TimeUnit.MILLISECONDS, new LinkedBlockingQueue());
from("activemq:my.queue")
    .split(xPathBuilder)
    .parallelProcessing()
    .executorService(threadPoolExecutor)
    .to("activemq:my.parts");
```

You can specify a custom executor in the XML DSL as follows:

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

```

    <from uri="direct:parallel-custom-pool"/>
    <split executorServiceRef="threadPoolExecutor">
      <xpath>/invoice/lineItems</xpath>
      <to uri="mock:result"/>
    </split>
  </route>
</camelContext>

<bean id="threadPoolExecutor"
class="java.util.concurrent.ThreadPoolExecutor">
  <constructor-arg index="0" value="8"/>
  <constructor-arg index="1" value="16"/>
  <constructor-arg index="2" value="0"/>
  <constructor-arg index="3" value="MILLISECONDS"/>
  <constructor-arg index="4"><bean
class="java.util.concurrent.LinkedBlockingQueue"/></constructor-arg>
</bean>

```

Using a bean to perform splitting

As the splitter can use *any* expression to do the splitting, we can use a bean to perform splitting, by invoking the `method()` expression. The bean should return an iterable value such as:

`java.util.Collection`, `java.util.Iterator`, or an array.

The following route defines a `method()` expression that calls a method on the `mySplitterBean` bean instance:

```

from("direct:body")
  // here we use a POJO bean mySplitterBean to do the split of the
  payload
  .split()
  .method("mySplitterBean", "splitBody")
  .to("mock:result");
from("direct:message")
  // here we use a POJO bean mySplitterBean to do the split of the
  message
  // with a certain header value
  .split()
  .method("mySplitterBean", "splitMessage")
  .to("mock:result");

```

Where `mySplitterBean` is an instance of the `MySplitterBean` class, which is defined as follows:

```

public class MySplitterBean {

  /**
   * The split body method returns something that is iterable such as
   a java.util.List.
   *
   * @param body the payload of the incoming message
   * @return a list containing each part split
   */
  public List<String> splitBody(String body) {
    // since this is based on an unit test you can of course
    // use different logic for splitting as Apache Camel have out

```

```

    // of the box support for splitting a String based on comma
    // but this is for show and tell, since this is java code
    // you have the full power how you like to split your messages
    List<String> answer = new ArrayList<String>();
    String[] parts = body.split(",");
    for (String part : parts) {
        answer.add(part);
    }
    return answer;
}

/**
 * The split message method returns something that is iterable such
 * as a java.util.List.
 *
 * @param header the header of the incoming message with the name user
 * @param body the payload of the incoming message
 * @return a list containing each part split
 */
public List<Message> splitMessage(@Header(value = "user") String
header, @Body String body) {
    // we can leverage the Parameter Binding Annotations
    // http://camel.apache.org/parameter-binding-annotations.html
    // to access the message header and body at same time,
    // then create the message that we want, splitter will
    // take care rest of them.
    // *NOTE* this feature requires Apache Camel version >= 1.6.1
    List<Message> answer = new ArrayList<Message>();
    String[] parts = header.split(",");
    for (String part : parts) {
        DefaultMessage message = new DefaultMessage();
        message.setHeader("user", part);
        message.setBody(body);
        answer.add(message);
    }
    return answer;
}
}

```

Exchange properties

The following properties are set on each split exchange:

header	type	description
CamelSplitIndex	int	Apache Camel 2.0: A split counter that increases for each Exchange being split. The counter starts from 0.
CamelSplitSize	int	Apache Camel 2.0: The total number of Exchanges that was split. This header is not applied for stream based splitting.

header	type	description
CamelSplitComplete	boolean	Apache Camel 2.4: Whether or not this Exchange is the last.

Splitter/aggregator pattern

It is a common pattern for the message pieces to be aggregated back into a single exchange, after processing of the individual pieces has completed. To support this pattern, the `split()` DSL command lets you provide an **AggregationStrategy** object as the second argument.

Java DSL example

The following example shows how to use a custom aggregation strategy to recombine a split message after all of the message pieces have been processed:

```
from("direct:start")
    .split(body().tokenize("@"), new MyOrderStrategy())
    // each split message is then send to this bean where we can
process it
    .to("bean:MyOrderService?method=handleOrder")
    // this is important to end the splitter route as we do not want
to do more routing
    // on each split message
    .end()
    // after we have split and handled each message we want to send a
single combined
    // response back to the original caller, so we let this bean build it
for us
    // this bean will receive the result of the aggregate strategy:
MyOrderStrategy
    .to("bean:MyOrderService?method=buildCombinedResponse")
```

AggregationStrategy implementation

The custom aggregation strategy, **MyOrderStrategy**, used in the preceding route is implemented as follows:

```
/**
 * This is our own order aggregation strategy where we can control
 * how each split message should be combined. As we do not want to
 * lose any message, we copy from the new to the old to preserve the
 * order lines as long we process them
 */
public static class MyOrderStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
    {
        // put order together in old exchange by adding the order from new
exchange

        if (oldExchange == null) {
```

```

        // the first time we aggregate we only have the new exchange,
        // so we just return it
        return newExchange;
    }

    String orders = oldExchange.getIn().getBody(String.class);
    String newLine = newExchange.getIn().getBody(String.class);

    LOG.debug("Aggregate old orders: " + orders);
    LOG.debug("Aggregate new order: " + newLine);

    // put orders together separating by semi colon
    orders = orders + ";" + newLine;
    // put combined order back on old to preserve it
    oldExchange.getIn().setBody(orders);

    // return old as this is the one that has all the orders gathered
until now
    return oldExchange;
}
}

```

Stream based processing

When parallel processing is enabled, it is theoretically possible for a later message piece to be ready for aggregation before an earlier piece. In other words, the message pieces might arrive at the aggregator out of order. By default, this does not happen, because the splitter implementation rearranges the message pieces back into their original order before passing them into the aggregator.

If you would prefer to aggregate the message pieces as soon as they are ready (and possibly out of order), you can enable the streaming option, as follows:

```

from("direct:streaming")
    .split(body().tokenize(", "), new MyOrderStrategy())
    .parallelProcessing()
    .streaming()
    .to("activemq:my.parts")
    .end()
    .to("activemq:all.parts");

```

You can also supply a custom iterator to use with streaming, as follows:

```

// Java
import static org.apache.camel.builder.ExpressionBuilder.beanExpression;
...
from("direct:streaming")
    .split(beanExpression(new MyCustomIteratorFactory(), "iterator"))
    .streaming().to("activemq:my.parts")

```



STREAMING AND XPATH

You cannot use streaming mode in conjunction with XPath. XPath requires the complete DOM XML document in memory.

Stream based processing with XML

If an incoming messages is a very large XML file, you can process the message most efficiently using the **tokenizeXML** sub-command in streaming mode.

For example, given a large XML file that contains a sequence of **order** elements, you can split the file into **order** elements using a route like the following:

```
from("file:inbox")
  .split().tokenizeXML("order").streaming()
  .to("activemq:queue:order");
```

You can do the same thing in XML, by defining a route like the following:

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="order" xml="true"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

It is often the case that you need access to namespaces that are defined in one of the enclosing (ancestor) elements of the token elements. You can copy namespace definitions from one of the ancestor elements into the token element, by specifying which element you want to inherit namespace definitions from.

In the Java DSL, you specify the ancestor element as the second argument of **tokenizeXML**. For example, to inherit namespace definitions from the enclosing **orders** element:

```
from("file:inbox")
  .split().tokenizeXML("order", "orders").streaming()
  .to("activemq:queue:order");
```

In the XML DSL, you specify the ancestor element using the **inheritNamespaceTagName** attribute. For example:

```
<route>
  <from uri="file:inbox"/>
  <split streaming="true">
    <tokenize token="order"
      xml="true"
      inheritNamespaceTagName="orders"/>
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

Options

The **split** DSL command supports the following options:

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

strategyRef		Refers to an AggregationStrategy to be used to assemble the replies from the sub-messages, into a single outgoing message from the Splitter . See the section titled <i>What does the splitter return</i> below for whats used by default.
parallelProcessing	false	If enables then processing the sub-messages occurs concurrently. Note the caller thread will still wait until all sub-messages has been fully processed, before it continues.
executorServiceRef		Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatic implied, and you do not have to enable that option as well.
stopOnException	false	Camel 2.2: Whether or not to stop continue processing immediately when an exception occurred. If disable, then Camel continue splitting and process the sub-messages regardless if one of them failed. You can deal with exceptions in the AggregationStrategy class where you have full control how to handle that.
streaming	false	If enabled then Camel will split in a streaming fashion, which means it will split the input message in chunks. This reduces the memory overhead. For example if you split big messages its recommended to enable streaming. If streaming is enabled then the sub-message replies will be aggregated out-of-order, eg in the order they come back. If disabled, Camel will process sub-message replies in the same order as they where splitted.

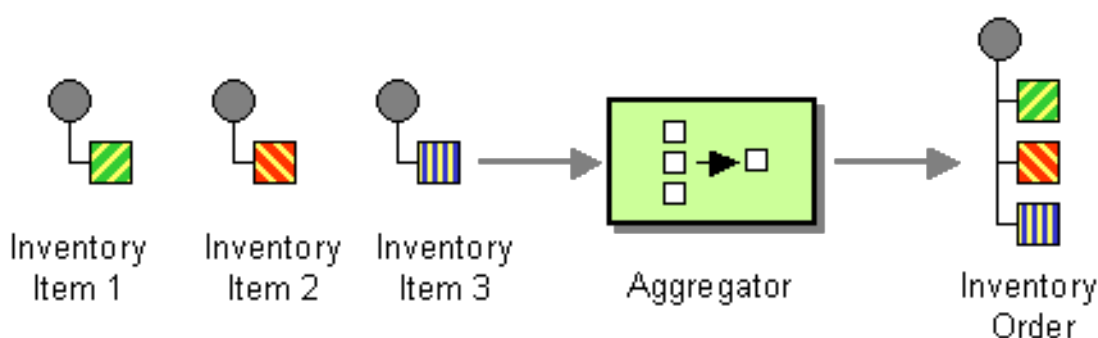
<code>timeout</code>		Camel 2.5: Sets a total timeout specified in millis. If the Recipient List hasn't been able to split and process all replies within the given timeframe, then the timeout triggers and the Splitter breaks out and continues. Notice if you provide a TimeoutAwareAggregationStrategy then the <code>timeout</code> method is invoked before breaking out.
<code>onPrepareRef</code>		Camel 2.8: Refers to a custom Processor to prepare the sub-message of the Exchange , before its processed. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.
<code>shareUnitOfWork</code>	<code>false</code>	Camel 2.8: Whether the unit of work should be shared. See further below for more details.

7.5. AGGREGATOR

Overview

The *aggregator* pattern, shown in [Figure 7.5, “Aggregator Pattern”](#), enables you to combine a batch of related messages into a single message.

Figure 7.5. Aggregator Pattern



To control the aggregator's behavior, Apache Camel allows you to specify the properties described in *Enterprise Integration Patterns*, as follows:

- *Correlation expression* — Determines which messages should be aggregated together. The correlation expression is evaluated on each incoming message to produce a *correlation key*. Incoming messages with the same correlation key are then grouped into the same batch. For example, if you want to aggregate *all* incoming messages into a single message, you can use a constant expression.

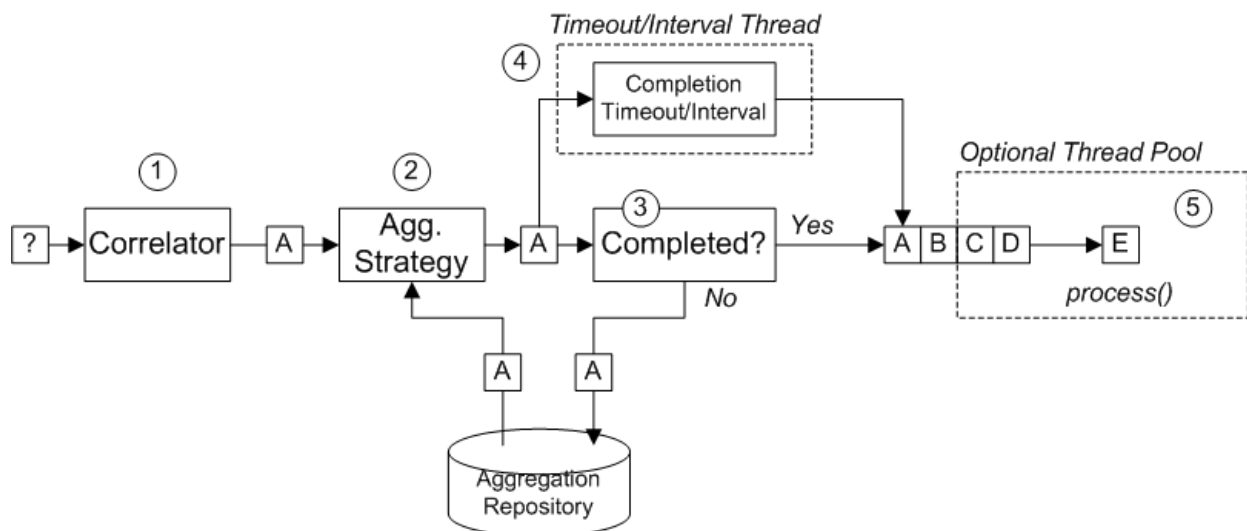
- *Completeness condition* — Determines when a batch of messages is complete. You can specify this either as a simple size limit or, more generally, you can specify a predicate condition that flags when the batch is complete.
- *Aggregation algorithm* — Combines the message exchanges for a single correlation key into a single message exchange.

For example, consider a stock market data system that receives 30,000 messages per second. You might want to throttle down the message flow if your GUI tool cannot cope with such a massive update rate. The incoming stock quotes can be aggregated together simply by choosing the latest quote and discarding the older prices. (You can apply a delta processing algorithm, if you prefer to capture some of the history.)

How the aggregator works

Figure 7.6, “Aggregator Implementation” shows an overview of how the aggregator works, assuming it is fed with a stream of exchanges that have correlation keys such as A, B, C, or D.

Figure 7.6. Aggregator Implementation



The incoming stream of exchanges shown in Figure 7.6, “Aggregator Implementation” is processed as follows:

1. The *correlator* is responsible for sorting exchanges based on the correlation key. For each incoming exchange, the correlation expression is evaluated, yielding the correlation key. For example, for the exchange shown in Figure 7.6, “Aggregator Implementation”, the correlation key evaluates to A.
2. The *aggregation strategy* is responsible for merging exchanges with the same correlation key. When a new exchange, A, comes in, the aggregator looks up the corresponding *aggregate exchange*, A', in the aggregation repository and combines it with the new exchange.

Until a particular aggregation cycle is completed, incoming exchanges are continuously aggregated with the corresponding aggregate exchange. An aggregation cycle lasts until terminated by one of the completion mechanisms.

3. If a completion predicate is specified on the aggregator, the aggregate exchange is tested to determine whether it is ready to be sent to the next processor in the route. Processing continues as follows:
 - If complete, the aggregate exchange is processed by the latter part of the route. There are

two alternative models for this: *synchronous* (the default), which causes the calling thread to block, or *asynchronous* (if parallel processing is enabled), where the aggregate exchange is submitted to an executor thread pool (as shown in [Figure 7.6, “Aggregator Implementation”](#)).

- If not complete, the aggregate exchange is saved back to the aggregation repository.
4. In parallel with the synchronous completion tests, it is possible to enable an asynchronous completion test by enabling *either* the **completionTimeout** option or the **completionInterval** option. These completion tests run in a separate thread and, whenever the completion test is satisfied, the corresponding exchange is marked as complete and starts to be processed by the latter part of the route (either synchronously or asynchronously, depending on whether parallel processing is enabled or not).
 5. If parallel processing is enabled, a thread pool is responsible for processing exchanges in the latter part of the route. By default, this thread pool contains ten threads, but you have the option of customizing the pool ([the section called “Threading options”](#)).

Java DSL example

The following example aggregates exchanges with the same **StockSymbol** header value, using the **UseLatestAggregationStrategy** aggregation strategy. For a given **StockSymbol** value, if more than three seconds elapse since the last exchange with that correlation key was received, the aggregated exchange is deemed to be complete and is sent to the **mock** endpoint.

```
from("direct:start")
  .aggregate(header("id"), new UseLatestAggregationStrategy())
    .completionTimeout(3000)
  .to("mock:aggregated");
```

XML DSL example

The following example shows how to configure the same route in XML:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregate strategyRef="aggregatorStrategy"
      completionTimeout="3000">
      <correlationExpression>
        <simple>header.StockSymbol</simple>
      </correlationExpression>
      <to uri="mock:aggregated"/>
    </aggregate>
  </route>
</camelContext>

<bean id="aggregatorStrategy"
  class="org.apache.camel.processor.aggregate.UseLatestAggregationStrategy"/>
```

Specifying the correlation expression

In the Java DSL, the correlation expression is always passed as the first argument to the **aggregate()**

DSL command. You are not limited to using the Simple expression language here. You can specify a correlation expression using any of the expression languages or scripting languages, such as XPath, XQuery, SQL, and so on.

For example, to correlate exchanges using an XPath expression, you could use the following Java DSL route:

```
from("direct:start")
    .aggregate(xpath("/stockQuote/@symbol"), new
    UseLatestAggregationStrategy())
    .completionTimeout(3000)
    .to("mock:aggregated");
```

If the correlation expression cannot be evaluated on a particular incoming exchange, the aggregator throws a **CamelExchangeException** by default. You can suppress this exception by setting the **ignoreInvalidCorrelationKeys** option. For example, in the Java DSL:

```
from(...).aggregate(...).ignoreInvalidCorrelationKeys()
```

In the XML DSL, you can set the **ignoreInvalidCorrelationKeys** option is set as an attribute, as follows:

```
<aggregate strategyRef="aggregatorStrategy"
    ignoreInvalidCorrelationKeys="true"
    ...>
    ...
</aggregate>
```

Specifying the aggregation strategy

In Java DSL, you can either pass the aggregation strategy as the second argument to the **aggregate()** DSL command or specify it using the **aggregationStrategy()** clause. For example, you can use the **aggregationStrategy()** clause as follows:

```
from("direct:start")
    .aggregate(header("id"))
    .aggregationStrategy(new UseLatestAggregationStrategy())
    .completionTimeout(3000)
    .to("mock:aggregated");
```

Apache Camel provides the following basic aggregation strategies (where the classes belong to the **org.apache.camel.processor.aggregate** Java package):

UseLatestAggregationStrategy

Return the last exchange for a given correlation key, discarding all earlier exchanges with this key. For example, this strategy could be useful for throttling the feed from a stock exchange, where you just want to know the latest price of a particular stock symbol.

UseOriginalAggregationStrategy

Return the first exchange for a given correlation key, discarding all later exchanges with this key. You must set the first exchange by calling **UseOriginalAggregationStrategy.setOriginal()** before you can use this strategy.

GroupedExchangeAggregationStrategy

Concatenates *all* of the exchanges for a given correlation key into a list, which is stored in the `Exchange.GROUPED_EXCHANGE` exchange property. See [the section called “Grouped exchanges”](#).

Implementing a custom aggregation strategy

If you want to apply a different aggregation strategy, you can implement one of the following aggregation strategy base interfaces:

`org.apache.camel.processor.aggregate.AggregationStrategy`

The basic aggregation strategy interface.

`org.apache.camel.processor.aggregate.TimeoutAwareAggregationStrategy`

Implement this interface, if you want your implementation to receive a notification when an aggregation cycle times out. The `timeout` notification method has the following signature:

```
void timeout(Exchange oldExchange, int index, int total, long timeout)
```

`org.apache.camel.processor.aggregate.CompletionAwareAggregationStrategy`

Implement this interface, if you want your implementation to receive a notification when an aggregation cycle completes normally. The notification method has the following signature:

```
void onCompletion(Exchange exchange)
```

For example, the following code shows two different custom aggregation strategies, `StringAggregationStrategy` and `ArrayListAggregationStrategy`:

```
//simply combines Exchange String body values using '+' as a delimiter
class StringAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
    {
        if (oldExchange == null) {
            return newExchange;
        }

        String oldBody = oldExchange.getIn().getBody(String.class);
        String newBody = newExchange.getIn().getBody(String.class);
        oldExchange.getIn().setBody(oldBody + "+" + newBody);
        return oldExchange;
    }
}

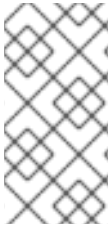
//simply combines Exchange body values into an ArrayList<Object>
class ArrayListAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
    {
        Object newBody = newExchange.getIn().getBody();
        ArrayList<Object> list = null;
```

```

        if (oldExchange == null) {
list = new ArrayList<Object>();
list.add(newBody);
newExchange.getIn().setBody(list);
return newExchange;
        } else {
list = oldExchange.getIn().getBody(ArrayList.class);
list.add(newBody);
return oldExchange;
        }
    }
}

```



NOTE

Since Apache Camel 2.0, the **AggregationStrategy.aggregate()** callback method is also invoked for the very first exchange. On the first invocation of the **aggregate** method, the **oldExchange** parameter is **null** and the **newExchange** parameter contains the first incoming exchange.

To aggregate messages using the custom strategy class, **ArrayListAggregationStrategy**, define a route like the following:

```

from("direct:start")
    .aggregate(header("StockSymbol"), new ArrayListAggregationStrategy())
    .completionTimeout(3000)
    .to("mock:result");

```

You can also configure a route with a custom aggregation strategy in XML, as follows:

```

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregate strategyRef="aggregatorStrategy"
      completionTimeout="3000">
      <correlationExpression>
        <simple>header.StockSymbol</simple>
      </correlationExpression>
      <to uri="mock:aggregated"/>
    </aggregate>
  </route>
</camelContext>

<bean id="aggregatorStrategy"
  class="com.my_package_name.ArrayListAggregationStrategy"/>

```

Exchange properties

The following properties are set on each aggregated exchange:

Table 7.1. Aggregated Exchange Properties

Header	Type	Description
<code>Exchange.AGGREGATED_SIZE</code>	<code>int</code>	The total number of exchanges aggregated into this exchange.
<code>Exchange.AGGREGATED_COMPLETED_BY</code>	<code>String</code>	Indicates the mechanism responsible for completing the aggregate exchange. Possible values are: predicate , size , timeout , interval , or consumer .

The following properties are set on exchanges redelivered by the HawtDB aggregation repository (see [the section called “Persistent aggregation repository”](#)):

Table 7.2. Redelivered Exchange Properties

Header	Type	Description
<code>Exchange.REDELIVERY_COUNTER</code>	<code>int</code>	Sequence number of the current redelivery attempt (starting at 1).

Specifying a completion condition

It is mandatory to specify *at least one* completion condition, which determines when an aggregate exchange leaves the aggregator and proceeds to the next node on the route. The following completion conditions can be specified:

`completionPredicate`

Evaluates a predicate after each exchange is aggregated in order to determine completeness. A value of **true** indicates that the aggregate exchange is complete.

`completionSize`

Completes the aggregate exchange after the specified number of incoming exchanges are aggregated.

`completionTimeout`

(*Incompatible with `completionInterval`*) Completes the aggregate exchange, if no incoming exchanges are aggregated within the specified timeout.

In other words, the timeout mechanism keeps track of a timeout for *each* correlation key value. The clock starts ticking after the latest exchange with a particular key value is received. If another exchange with the same key value is *not* received within the specified timeout, the corresponding aggregate exchange is marked complete and sent to the next node on the route.

`completionInterval`

(*Incompatible with `completionTimeout`*) Completes *all* outstanding aggregate exchanges, after each time interval (of specified length) has elapsed.

The time interval is *not* tailored to each aggregate exchange. This mechanism forces simultaneous completion of all outstanding aggregate exchanges. Hence, in some cases, this mechanism could complete an aggregate exchange immediately after it started aggregating.

completionFromBatchConsumer

When used in combination with a consumer endpoint that supports the *batch consumer* mechanism, this completion option automatically figures out when the current batch of exchanges is complete, based on information it receives from the consumer endpoint. See [the section called “Batch consumer”](#).

forceCompletionOnStop

When this option is enabled, it forces completion of all outstanding aggregate exchanges when the current route context is stopped.

The preceding completion conditions can be combined arbitrarily, except for the **completionTimeout** and **completionInterval** conditions, which cannot be simultaneously enabled. When conditions are used in combination, the general rule is that the first completion condition to trigger is the effective completion condition.

Specifying the completion predicate

You can specify an arbitrary predicate expression that determines when an aggregated exchange is complete. There are two possible ways of evaluating the predicate expression:

- *On the latest aggregate exchange*—this is the default behavior.
- *On the latest incoming exchange*—this behavior is selected when you enable the **eagerCheckCompletion** option.

For example, if you want to terminate a stream of stock quotes every time you receive an **ALERT** message (as indicated by the value of a **MsgType** header in the latest incoming exchange), you can define a route like the following:

```
from("direct:start")
  .aggregate(
    header("id"),
    new UseLatestAggregationStrategy()
  )
  .completionPredicate(
    header("MsgType").isEqualTo("ALERT")
  )
  .eagerCheckCompletion()
  .to("mock:result");
```

The following example shows how to configure the same route using XML:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregate strategyRef="aggregatorStrategy"
      eagerCheckCompletion="true">
      <correlationExpression>
        <simple>header.StockSymbol</simple>
```

```

        </correlationExpression>
        <completionPredicate>
            <simple>$MsgType = 'ALERT'</simple>
        </completionPredicate>
        <to uri="mock:result"/>
    </aggregate>
</route>
</camelContext>

<bean id="aggregatorStrategy"
class="org.apache.camel.processor.aggregate.UseLatestAggregationStrategy"/
>

```

Specifying a dynamic completion timeout

It is possible to specify a *dynamic completion timeout*, where the timeout value is recalculated for every incoming exchange. For example, to set the timeout value from the **timeout** header in each incoming exchange, you could define a route as follows:

```

from("direct:start")
    .aggregate(header("StockSymbol"), new UseLatestAggregationStrategy())
        .completionTimeout(header("timeout"))
    .to("mock:aggregated");

```

You can configure the same route in the XML DSL, as follows:

```

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <aggregate strategyRef="aggregatorStrategy">
            <correlationExpression>
                <simple>header.StockSymbol</simple>
            </correlationExpression>
            <completionTimeout>
                <header>timeout</header>
            </completionTimeout>
            <to uri="mock:aggregated"/>
        </aggregate>
    </route>
</camelContext>

<bean id="aggregatorStrategy"
class="org.apache.camel.processor.aggregate.UseLatestAggregationStrategy"/>

```



NOTE

You can also add a fixed timeout value and Apache Camel will fall back to use this value, if the dynamic value is **null** or **0**.

Specifying a dynamic completion size

It is possible to specify a *dynamic completion size*, where the completion size is recalculated for every incoming exchange. For example, to set the completion size from the **mySize** header in each incoming exchange, you could define a route as follows:

```
from("direct:start")
    .aggregate(header("StockSymbol"), new UseLatestAggregationStrategy())
        .completionSize(header("mySize"))
    .to("mock:aggregated");
```

And the same example using Spring XML:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregate strategyRef="aggregatorStrategy">
      <correlationExpression>
        <simple>header.StockSymbol</simple>
      </correlationExpression>
      <completionSize>
        <header>mySize</header>
      </completionSize>
      <to uri="mock:aggregated"/>
    </aggregate>
  </route>
</camelContext>

<bean id="aggregatorStrategy"
  class="org.apache.camel.processor.UseLatestAggregationStrategy"/>
```



NOTE

You can also add a fixed size value and Apache Camel will fall back to use this value, if the dynamic value is **null** or **0**.

Forcing completion with a special message

It is possible to force completion of all outstanding aggregate message by sending a special message to the route, with the **Exchange.AGGREGATION_COMPLETE_ALL_GROUPS** header set to **true**. This message acts like a signal to the aggregator: the remaining content of the message is ignored and the message is not processed any further.

Enforcing unique correlation keys

In some aggregation scenarios, you might want to enforce the condition that the correlation key is unique for each batch of exchanges. In other words, when the aggregate exchange for a particular correlation key completes, you want to make sure that no further aggregate exchanges with that correlation key are allowed to proceed. For example, you might want to enforce this condition, if the latter part of the route expects to process exchanges with unique correlation key values.

Depending on how the completion conditions are configured, there might be a risk of more than one aggregate exchange being generated with a particular correlation key. For example, although you might define a completion predicate that is designed to wait until *all* the exchanges with a particular correlation

key are received, you might also define a completion timeout, which could fire before all of the exchanges with that key have arrived. In this case, the late-arriving exchanges could give rise to a *second* aggregate exchange with the same correlation key value.

For such scenarios, you can configure the aggregator to suppress aggregate exchanges that duplicate previous correlation key values, by setting the **closeCorrelationKeyOnCompletion** option. In order to suppress duplicate correlation key values, it is necessary for the aggregator to record previous correlation key values in a cache. The size of this cache (the number of cached correlation keys) is specified as an argument to the **closeCorrelationKeyOnCompletion()** DSL command. To specify a cache of unlimited size, you can pass a value of zero or a negative integer. For example, to specify a cache size of **10000** key values:

```
from("direct:start")
    .aggregate(header("UniqueBatchID"), new MyConcatenateStrategy())
        .completionSize(header("mySize"))
        .closeCorrelationKeyOnCompletion(10000)
    .to("mock:aggregated");
```

If an aggregate exchange completes with a duplicate correlation key value, the aggregator throws a **ClosedCorrelationKeyException** exception.

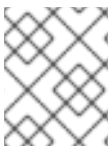
Grouped exchanges

You can combine all of the aggregated exchanges in an outgoing batch into a single **org.apache.camel.impl.GroupedExchange** holder class. To enable grouped exchanges, specify the **groupExchanges()** option, as shown in the following Java DSL route:

```
from("direct:start")
    .aggregate(header("StockSymbol"))
        .completionTimeout(3000)
        .groupExchanges()
    .to("mock:result");
```

The grouped exchange that is sent to **mock:result** contains the list of aggregated exchanges stored in the exchange property, **Exchange.GROUPED_EXCHANGE**. The following line of code shows how a subsequent processor can access the contents of the grouped exchange in the form of a list:

```
// Java
List<Exchange> grouped = ex.getProperty(Exchange.GROUPED_EXCHANGE,
List.class);
```



NOTE

When you enable the grouped exchanges feature, you *must not* configure an aggregation strategy (the grouped exchanges feature is itself an aggregation strategy).

Batch consumer

The aggregator can work together with the *batch consumer* pattern to aggregate the total number of messages reported by the batch consumer (a batch consumer endpoint sets the **CamelBatchSize**, **CamelBatchIndex**, and **CamelBatchComplete** properties on the incoming exchange). For example, to aggregate all of the files found by a File consumer endpoint, you could use a route like the following:

```

from("file://inbox")
    .aggregate(xpath("//order/@customerId"), new
AggregateCustomerOrderStrategy())
    .completionFromBatchConsumer()
    .to("bean:processOrder");

```

Currently, the following endpoints support the batch consumer mechanism: File, FTP, Mail, iBatis, and JPA.

Persistent aggregation repository

If you want pending aggregated exchanges to be stored persistently, you can use either the [HawtDB](#) component or the [SQL Component](#) for persistence support as a persistent aggregation repository. For example, if using HawtDB, you need to include a dependency on the `camel-hawtdb` component in your Maven POM. You can then configure a route to use the HawtDB aggregation repository as follows:

```

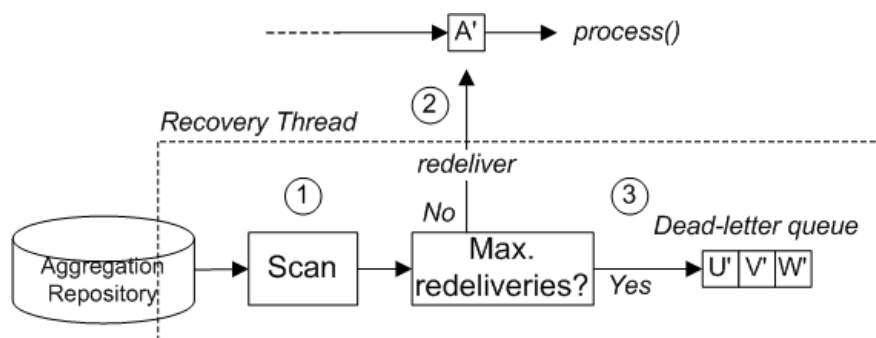
public void configure() throws Exception {
    HawtDBAggregationRepository repo = new AggregationRepository("repo1",
"target/data/hawtdb.dat");

    from("direct:start")
        .aggregate(header("id"), new UseLatestAggregationStrategy())
            .completionTimeout(3000)
            .aggregationRepository(repo)
            .to("mock:aggregated");
}

```

The HawtDB aggregation repository has a feature that enables it to recover and retry any failed exchanges (that is, any exchange that raised an exception while it was being processed by the latter part of the route). [Figure 7.7, “Recoverable Aggregation Repository”](#) shows an overview of the recovery mechanism.

Figure 7.7. Recoverable Aggregation Repository



The recovery mechanism works as follows:

1. The aggregator creates a dedicated recovery thread, which runs in the background, scanning the aggregation repository to find any failed exchanges.
2. Each failed exchange is checked to see whether its current redelivery count exceeds the maximum redelivery limit. If it is under the limit, the recovery task resubmits the exchange for processing in the latter part of the route.
3. If the current redelivery count is over the limit, the failed exchange is passed to the dead letter queue.

For more details about the HawtDB component, see [chapter "HawtDB" in "EIP Component Reference"](#).

Threading options

As shown in [Figure 7.6, "Aggregator Implementation"](#), the aggregator is decoupled from the latter part of the route, where the exchanges sent to the latter part of the route are processed by a dedicated thread pool. By default, this pool contains just a single thread. If you want to specify a pool with multiple threads, enable the `parallelProcessing` option, as follows:

```
from("direct:start")
    .aggregate(header("id"), new UseLatestAggregationStrategy())
      .completionTimeout(3000)
      .parallelProcessing()
    .to("mock:aggregated");
```

By default, this creates a pool with 10 worker threads.

If you want to exercise more control over the created thread pool, specify a custom `java.util.concurrent.ExecutorService` instance using the `executorService` option (in which case it is unnecessary to enable the `parallelProcessing` option).

Aggregator options

The aggregator supports the following options:

Table 7.3. Aggregator Options

Option	Default	Description
<code>correlationExpression</code>		Mandatory Expression which evaluates the correlation key to use for aggregation. The Exchange which has the same correlation key is aggregated together. If the correlation key could not be evaluated an Exception is thrown. You can disable this by using the <code>ignoreBadCorrelationKeys</code> option.

Option	Default	Description
aggregationStrategy		Mandatory AggregationStrategy which is used to <i>merge</i> the incoming Exchange with the existing already merged exchanges. At first call the oldExchange parameter is null . On subsequent invocations the oldExchange contains the merged exchanges and newExchange is of course the new incoming Exchange. From Camel 2.9.2 onwards, the strategy can optionally be a TimeoutAwareAggregationStrategy implementation, which supports a timeout callback
strategyRef		A reference to lookup the AggregationStrategy in the Registry .
completionSize		Number of messages aggregated before the aggregation is complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a size dynamically - will use Integer as result. If both are set Camel will fallback to use the fixed value if the Expression result was null or 0 .
completionTimeout		Time in millis that an aggregated exchange should be inactive before its complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a timeout dynamically - will use Long as result. If both are set Camel will fallback to use the fixed value if the Expression result was null or 0 . You cannot use this option together with <code>completionInterval</code> , only one of the two can be used.

Option	Default	Description
completionInterval		A repeating period in millis by which the aggregator will complete all current aggregated exchanges. Camel has a background task which is triggered every period. You cannot use this option together with <code>completionTimeout</code> , only one of them can be used.
completionPredicate		A Predicate to indicate when an aggregated exchange is complete.
completionFromBatchConsumer	false	This option is if the exchanges are coming from a Batch Consumer . Then when enabled the Aggregator will use the batch size determined by the Batch Consumer in the message header CamelBatchSize . See more details at Batch Consumer . This can be used to aggregate all files consumed from a File endpoint in that given poll.
eagerCheckCompletion	false	Whether or not to eager check for completion when a new incoming Exchange has been received. This option influences the behavior of the completionPredicate option as the Exchange being passed in changes accordingly. When false the Exchange passed in the Predicate is the <i>aggregated</i> Exchange which means any information you may store on the aggregated Exchange from the AggregationStrategy is available for the Predicate . When true the Exchange passed in the Predicate is the <i>incoming Exchange</i> , which means you can access data from the incoming Exchange.
forceCompletionOnStop	false	If true , complete all aggregated exchanges when the current route context is stopped.

Option	Default	Description
groupExchanges	false	If enabled then Camel will group all aggregated Exchanges into a single combined org.apache.camel.impl.GroupedExchange holder class that holds all the aggregated Exchanges. And as a result only one Exchange is being sent out from the aggregator. Can be used to combine many incoming Exchanges into a single output Exchange without coding a custom AggregationStrategy yourself.
ignoreInvalidCorrelationKeys	false	Whether or not to ignore correlation keys which could not be evaluated to a value. By default Camel will throw an Exception, but you can enable this option and ignore the situation instead.
closeCorrelationKeyOnCompletion		Whether or not <i>late</i> Exchanges should be accepted or not. You can enable this to indicate that if a correlation key has already been completed, then any new exchanges with the same correlation key be denied. Camel will then throw a closedCorrelationKeyException exception. When using this option you pass in a integer which is a number for a LRUcache which keeps that last X number of closed correlation keys. You can pass in 0 or a negative value to indicate a unbounded cache. By passing in a number you are ensured that cache wont grown too big if you use a log of different correlation keys.
discardOnCompletionTimeout	false	Camel 2.5: Whether or not exchanges which complete due to a timeout should be discarded. If enabled, then when a timeout occurs the aggregated message will not be sent out but dropped (discarded).

Option	Default	Description
aggregationRepository		Allows you to plug in your own implementation of org.apache.camel.spi.AggregationRepository which keeps track of the current inflight aggregated exchanges. Camel uses by default a memory based implementation.
aggregationRepositoryRef		Reference to lookup a aggregationRepository in the Registry .
parallelProcessing	false	When aggregated are completed they are being send out of the aggregator. This option indicates whether or not Camel should use a thread pool with multiple threads for concurrency. If no custom thread pool has been specified then Camel creates a default pool with 10 concurrent threads.
executorService		If using parallelProcessing you can specify a custom thread pool to be used. In fact also if you are not using parallelProcessing this custom thread pool is used to send out aggregated exchanges as well.
executorServiceRef		Reference to lookup a executorService in the Registry
timeoutCheckerExecutorsService		If using one of the completionTimeout , completionTimeoutExpression , or completionInterval options, a background thread is created to check for the completion for every aggregator. Set this option to provide a custom thread pool to be used rather than creating a new thread for every aggregator.

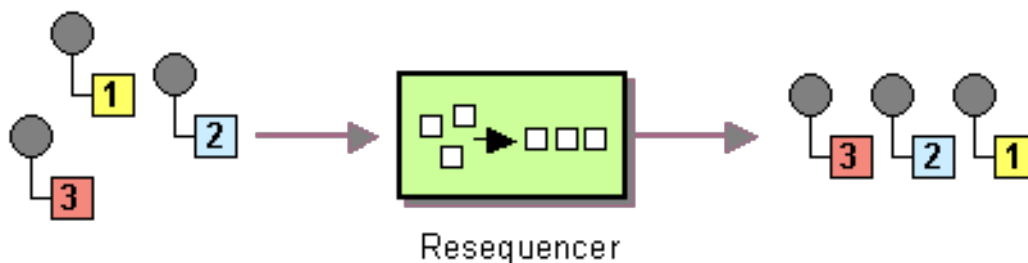
Option	Default	Description
<code>timeoutCheckerExecutorsServiceRef</code>		Reference to look up a <code>timeoutCheckerExecutorsService</code> in the registry.

7.6. RESEQUENCER

Overview

The *resequencer* pattern, shown in [Figure 7.8, “Resequencer Pattern”](#), enables you to resequence messages according to a sequencing expression. Messages that generate a low value for the sequencing expression are moved to the front of the batch and messages that generate a high value are moved to the back.

Figure 7.8. Resequencer Pattern



Apache Camel supports two resequencing algorithms:

- *Batch resequencing* — Collects messages into a batch, sorts the messages and sends them to their output.
- *Stream resequencing* — Re-orders (continuous) message streams based on the detection of gaps between messages.

By default the resequencer does not support duplicate messages and will only keep the last message, in cases where a message arrives with the same message expression. However, in batch mode you can enable the resequencer to allow duplicates.

Batch resequencing

The batch resequencing algorithm is enabled by default. For example, to resequence a batch of incoming messages based on the value of a timestamp contained in the `TimeStamp` header, you can define the following route in Java DSL:

```
from("direct:start").resequence(header("TimeStamp")).to("mock:result");
```

By default, the batch is obtained by collecting all of the incoming messages that arrive in a time interval of 1000 milliseconds (default *batch timeout*), up to a maximum of 100 messages (default *batch size*). You can customize the values of the batch timeout and the batch size by appending a `batch()` DSL command, which takes a `BatchResequencerConfig` instance as its sole argument. For example, to modify the preceding route so that the batch consists of messages collected in a 4000 millisecond time window, up to a maximum of 300 messages, you can define the Java DSL route as follows:

```
import org.apache.camel.model.config.BatchResequencerConfig;

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start").resequence(header("TimeStamp")).batch(new
BatchResequencerConfig(300,4000L)).to("mock:result");
    }
};
```

You can also specify a batch resequencer pattern using XML configuration. The following example defines a batch resequencer with a batch size of 300 and a batch timeout of 4000 milliseconds:

```
<camelContext id="resequencerBatch"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start" />
    <resequence>
      <!--
        batch-config can be omitted for default (batch) resequencer
settings
      -->
      <batch-config batchSize="300" batchTimeout="4000" />
      <simple>header.TimeStamp</simple>
      <to uri="mock:result" />
    </resequence>
  </route>
</camelContext>
```

Batch options

Table 7.4, “Batch Resequencer Options” shows the options that are available in batch mode only.

Table 7.4. Batch Resequencer Options

Java DSL	XML DSL	Default	Description
<code>allowDuplicates()</code>	<code>batch-config/@allowDuplicates</code>	false	If true , do not discard duplicate messages from the batch (where <i>duplicate</i> means that the message expression evaluates to the same value).
<code>reverse()</code>	<code>batch-config/@reverse</code>	false	If true , put the messages in reverse order (where the default ordering applied to a message expression is based on Java's string lexical ordering, as defined by String.compareTo()).

For example, if you want to resequence messages from JMS queues based on **JMSPriority**, you would need to combine the options, **allowDuplicates** and **reverse**, as follows:

```
from("jms:queue:foo")
    // sort by JMSPriority by allowing duplicates (message can have
    // same JMSPriority)
    // and use reverse ordering so 9 is first output (most important),
    // and 0 is last
    // use batch mode and fire every 3th second

.resequence(header("JMSPriority")).batch().timeout(3000).allowDuplicates()
.reverse()
.to("mock:result");
```

Stream resequencing

To enable the stream resequencing algorithm, you must append **stream()** to the **resequence()** DSL command. For example, to resequence incoming messages based on the value of a sequence number in the **seqnum** header, you define a DSL route as follows:

```
from("direct:start").resequence(header("seqnum")).stream().to("mock:result");
```

The stream-processing resequencer algorithm is based on the detection of gaps in a message stream, rather than on a fixed batch size. Gap detection, in combination with timeouts, removes the constraint of needing to know the number of messages of a sequence (that is, the batch size) in advance. Messages must contain a unique sequence number for which a predecessor and a successor is known. For example a message with the sequence number **3** has a predecessor message with the sequence number **2** and a successor message with the sequence number **4**. The message sequence **2, 3, 5** has a gap because the successor of **3** is missing. The resequencer therefore must retain message **5** until message **4** arrives (or a timeout occurs).

By default, the stream resequencer is configured with a timeout of 1000 milliseconds, and a maximum message capacity of 100. To customize the stream's timeout and message capacity, you can pass a **StreamResequencerConfig** object as an argument to **stream()**. For example, to configure a stream resequencer with a message capacity of 5000 and a timeout of 4000 milliseconds, you define a route as follows:

```
// Java
import org.apache.camel.model.config.StreamResequencerConfig;

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start").resequence(header("seqnum")).
            stream(new StreamResequencerConfig(5000, 4000L)).
            to("mock:result");
    }
};
```

If the maximum time delay between successive messages (that is, messages with adjacent sequence numbers) in a message stream is known, the resequencer's timeout parameter should be set to this value. In this case, you can guarantee that all messages in the stream are delivered in the correct order to the next processor. The lower the timeout value that is compared to the out-of-sequence time

difference, the more likely it is that the resequencer will deliver messages out of sequence. Large timeout values should be supported by sufficiently high capacity values, where the capacity parameter is used to prevent the resequencer from running out of memory.

If you want to use sequence numbers of some type other than **long**, you would must define a custom comparator, as follows:

```
// Java
ExpressionResultComparator<Exchange> comparator = new MyComparator();
StreamResequencerConfig config = new StreamResequencerConfig(5000, 4000L,
comparator);
from("direct:start").resequence(header("seqnum")).stream(config).to("mock:
result");
```

You can also specify a stream resequencer pattern using XML configuration. The following example defines a stream resequencer with a message capacity of 5000 and a timeout of 4000 milliseconds:

```
<camelContext id="resequencerStream"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <resequence>
      <stream-config capacity="5000" timeout="4000"/>
      <simple>header.seqnum</simple>
      <to uri="mock:result" />
    </resequence>
  </route>
</camelContext>
```

Ignore invalid exchanges

The resequencer EIP throws a **CamelExchangeException** exception, if the incoming exchange is not valid—that is, if the sequencing expression cannot be evaluated for some reason (for example, due to a missing header). You can use the **ignoreInvalidExchanges** option to ignore these exceptions, which means the resequencer will skip any invalid exchanges.

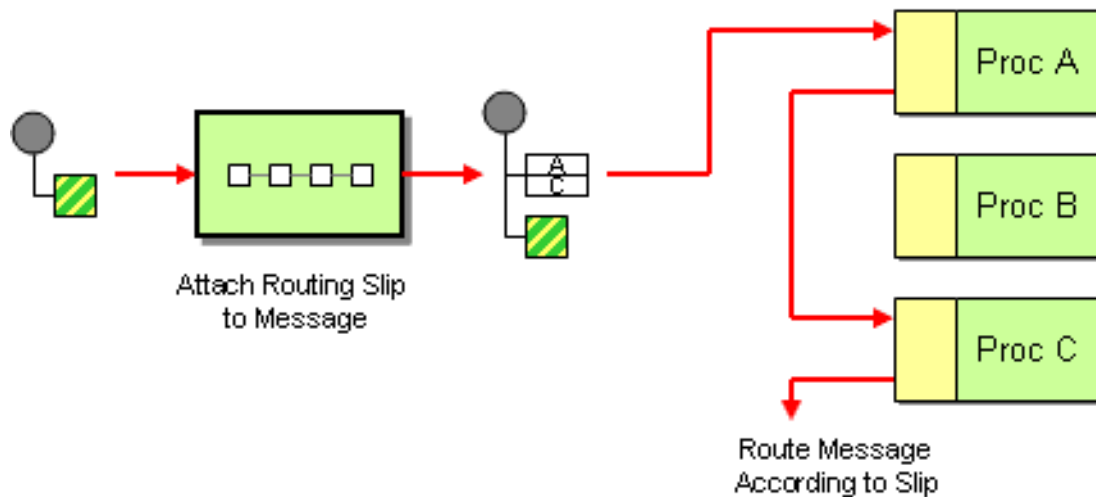
```
from("direct:start")
  .resequence(header("seqno")).batch().timeout(1000)
  // ignore invalid exchanges (they are discarded)
  .ignoreInvalidExchanges()
  .to("mock:result");
```

7.7. ROUTING SLIP

Overview

The *routing slip* pattern, shown in [Figure 7.9, “Routing Slip Pattern”](#), enables you to route a message consecutively through a series of processing steps, where the sequence of steps is not known at design time and can vary for each message. The list of endpoints through which the message should pass is stored in a header field (the *slip*), which Apache Camel reads at run time to construct a pipeline on the fly.

Figure 7.9. Routing Slip Pattern



The slip header

The routing slip appears in a user-defined header, where the header value is a comma-separated list of endpoint URIs. For example, a routing slip that specifies a sequence of security tasks—decrypting, authenticating, and de-duplicating a message—might look like the following:

```
cxf:bean:decrypt,cxf:bean:authenticate,cxf:bean:dedup
```

The current endpoint property

From Camel 2.5 the Routing Slip will set a property (**Exchange.SLIP_ENDPOINT**) on the exchange which contains the current endpoint as it advanced through the slip. This enables you to find out how far the exchange has progressed through the slip.

The [Routing Slip](#) will compute the slip *beforehand* which means, the slip is only computed once. If you need to compute the slip *on-the-fly* then use the [Dynamic Router](#) pattern instead.

Java DSL example

The following route takes messages from the **direct:a** endpoint and reads a routing slip from the **aRoutingSlipHeader** header:

```
from("direct:b").routingSlip("aRoutingSlipHeader");
```

You can specify the header name either as a string literal or as an expression.

You can also customize the URI delimiter using the two-argument form of **routingSlip()**. The following example defines a route that uses the **aRoutingSlipHeader** header key for the routing slip and uses the **#** character as the URI delimiter:

```
from("direct:c").routingSlip("aRoutingSlipHeader", "#");
```

XML configuration example

The following example shows how to configure the same route in XML:


```
<camelContext id="buildRoutingSlip"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:c"/>
    <routingSlip uriDelimiter="#">
      <headerName>aRoutingSlipHeader</headerName>
    </routingSlip>
  </route>
</camelContext>
```

Ignore invalid endpoints

The [Routing Slip](#) now supports **ignoreInvalidEndpoints**, which the [Recipient List](#) pattern also supports. You can use it to skip endpoints that are invalid. For example:

```
from("direct:a").routingSlip("myHeader").ignoreInvalidEndpoints();
```

In Spring XML, this feature is enabled by setting the **ignoreInvalidEndpoints** attribute on the **<routingSlip>** tag:

```
<route>
  <from uri="direct:a"/>
  <routingSlip ignoreInvalidEndpoints="true">
    <headerName>myHeader</headerName>
  </routingSlip>
</route>
```

Consider the case where **myHeader** contains the two endpoints, **direct:foo,xxx:bar**. The first endpoint is valid and works. The second is invalid and, therefore, ignored. Apache Camel logs at **INFO** level whenever an invalid endpoint is encountered.

Options

The **routingSlip** DSL command supports the following options:

Name	Default Value	Description
uriDelimiter	,	Delimiter used if the Expression returned multiple endpoints.
ignoreInvalidEndpoints	false	If an endpoint uri could not be resolved, should it be ignored. Otherwise Camel will throw an exception stating the endpoint uri is not valid.

7.8. THROTTLER

Overview

A *throttler* is a processor that limits the flow rate of incoming messages. You can use this pattern to protect a target endpoint from getting overloaded. In Apache Camel, you can implement the throttler pattern using the `throttle()` Java DSL command.

Java DSL example

To limit the flow rate to 100 messages per second, define a route as follows:

```
from("seda:a").throttle(100).to("seda:b");
```

If necessary, you can customize the time period that governs the flow rate using the `timePeriodMillis()` DSL command. For example, to limit the flow rate to 3 messages per 30000 milliseconds, define a route as follows:

```
from("seda:a").throttle(3).timePeriodMillis(30000).to("mock:result");
```

XML configuration example

The following example shows how to configure the preceding route in XML:

```
<camelContext id="throttleRoute"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <!-- throttle 3 messages per 30 sec -->
    <throttle timePeriodMillis="30000">
      <constant>3</constant>
      <to uri="mock:result"/>
    </throttle>
  </route>
</camelContext>
```

Dynamically changing maximum requests per period

Available as of Camel 2.8 Since we use an [Expression](#), you can adjust this value at runtime, for example you can provide a header with the value. At runtime Camel evaluates the expression and converts the result to a `java.lang.Long` type. In the example below we use a header from the message to determine the maximum requests per period. If the header is absent, then the [Throttler](#) uses the old value. So that allows you to only provide a header if the value is to be changed:

```
<camelContext id="throttleRoute"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:expressionHeader"/>
    <throttle timePeriodMillis="500">
      <!-- use a header to determine how many messages to throttle per 0.5
sec -->
      <header>throttleValue</header>
      <to uri="mock:result"/>
    </throttle>
  </route>
</camelContext>
```

Asynchronous delaying

The throttler can enable *non-blocking asynchronous delaying*, which means that Apache Camel schedules a task to be executed in the future. The task is responsible for processing the latter part of the route (after the throttler). This allows the caller thread to unblock and service further incoming messages. For example:

```
from("seda:a").throttle(100).asyncDelayed().to("seda:b");
```

Options

The **throttle** DSL command supports the following options:

Name	Default Value	Description
maximumRequestsPerPeriod		Maximum number of requests per period to throttle. This option must be provided and a positive number. Notice, in the XML DSL, from Camel 2.8 onwards this option is configured using an Expression instead of an attribute.
timePeriodMillis	1000	The time period in millis, in which the throttler will allow at most maximumRequestsPerPeriod number of messages.
asyncDelayed	false	Camel 2.4: If enabled then any messages which is delayed happens asynchronously using a scheduled thread pool.
executorServiceRef		Camel 2.4: Refers to a custom Thread Pool to be used if asyncDelay has been enabled.
callerRunsWhenRejected	true	Camel 2.4: Is used if asyncDelayed was enabled. This controls if the caller thread should execute the task if the thread pool rejected the task.

7.9. DELAYER

Overview

A *delayer* is a processor that enables you to apply a *relative* time delay to incoming messages.

Java DSL example

You can use the **delay()** command to add a *relative* time delay, in units of milliseconds, to incoming messages. For example, the following route delays all incoming messages by 2 seconds:

```
from("seda:a").delay(2000).to("mock:result");
```

Alternatively, you can specify the time delay using an expression:

```
from("seda:a").delay(header("MyDelay")).to("mock:result");
```

The DSL commands that follow **delay()** are interpreted as sub-clauses of **delay()**. Hence, in some contexts it is necessary to terminate the sub-clauses of **delay()** by inserting the **end()** command. For example, when **delay()** appears inside an **onException()** clause, you would terminate it as follows:

```
from("direct:start")
    .onException(Exception.class)
        .maximumRedeliveries(2)
        .backOffMultiplier(1.5)
        .handled(true)
        .delay(1000)
            .log("Halting for some time")
            .to("mock:halt")
        .end()
    .end()
    .to("mock:result");
```

XML configuration example

The following example demonstrates the delay in XML DSL:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <delay>
      <header>MyDelay</header>
    </delay>
    <to uri="mock:result"/>
  </route>
  <route>
    <from uri="seda:b"/>
    <delay>
      <constant>1000</constant>
    </delay>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

Creating a custom delay

You can use an expression combined with a bean to determine the delay as follows:

```
from("activemq:foo").
    delay().expression().method("someBean", "computeDelay").
    to("activemq:bar");
```

Where the bean class could be defined as follows:

```
public class SomeBean {
    public long computeDelay() {
        long delay = 0;
        // use java code to compute a delay value in millis
        return delay;
    }
}
```

Asynchronous delaying

You can let the delayer use *non-blocking asynchronous delaying*, which means that Apache Camel schedules a task to be executed in the future. The task is responsible for processing the latter part of the route (after the delayer). This allows the caller thread to unblock and service further incoming messages. For example:

```
from("activemq:queue:foo")
    .delay(1000)
    .asyncDelayed()
    .to("activemq:aDelayedQueue");
```

The same route can be written in the XML DSL, as follows:

```
<route>
  <from uri="activemq:queue:foo"/>
  <delay asyncDelayed="true">
    <constant>1000</constant>
  </delay>
  <to uri="activemq:aDealyedQueue"/>
</route>
```

Options

The delayer pattern supports the following options:

Name	Default Value	Description
asyncDelayed	false	Camel 2.4: If enabled then delayed messages happens asynchronously using a scheduled thread pool.
executorServiceRef		Camel 2.4: Refers to a custom Thread Pool to be used if asyncDeLay has been enabled.

<code>callerRunsWhenRejected</code>	<code>true</code>	Camel 2.4: Is used if asyncDelayed was enabled. This controls if the caller thread should execute the task if the thread pool rejected the task.
-------------------------------------	-------------------	--

7.10. LOAD BALANCER

Overview

The *load balancer* pattern allows you to delegate message processing to one of several endpoints, using a variety of different load-balancing policies.

Java DSL example

The following route distributes incoming messages between the target endpoints, `mock:x`, `mock:y`, `mock:z`, using a round robin load-balancing policy:

```
from("direct:start").loadBalance().roundRobin().to("mock:x", "mock:y",
"mock:z");
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <roundRobin/>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Load-balancing policies

The Apache Camel load balancer supports the following load-balancing policies:

- [Round robin](#)
- [Random](#)
- [Sticky](#)
- [Topic](#)
- [the section called "Failover"](#)
- [the section called "Weighted round robin and weighted random"](#)

- [the section called “Custom Load Balancer”](#)

Round robin

The round robin load-balancing policy cycles through all of the target endpoints, sending each incoming message to the next endpoint in the cycle. For example, if the list of target endpoints is, **mock:x**, **mock:y**, **mock:z**, then the incoming messages are sent to the following sequence of endpoints: **mock:x**, **mock:y**, **mock:z**, **mock:x**, **mock:y**, **mock:z**, and so on.

You can specify the round robin load-balancing policy in Java DSL, as follows:

```
from("direct:start").loadBalance().roundRobin().to("mock:x", "mock:y",
"mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <roundRobin/>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Random

The random load-balancing policy chooses the target endpoint randomly from the specified list.

You can specify the random load-balancing policy in Java DSL, as follows:

```
from("direct:start").loadBalance().random().to("mock:x", "mock:y",
"mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <random/>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Sticky

The sticky load-balancing policy directs the *In* message to an endpoint that is chosen by calculating a hash value from a specified expression. The advantage of this load-balancing policy is that expressions of the same value are always sent to the same server. For example, by calculating the hash value from a header that contains a username, you ensure that messages from a particular user are always sent to the same target endpoint. Another useful approach is to specify an expression that extracts the session ID from an incoming message. This ensures that all messages belonging to the same session are sent to the same target endpoint.

You can specify the sticky load-balancing policy in Java DSL, as follows:

```
from("direct:start").loadBalance().sticky(header("username")).to("mock:x",
"mock:y", "mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <sticky>
        <expression>
          <simple>header.username</simple>
        </expression>
      </sticky>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Topic

The topic load-balancing policy sends a copy of each *In* message to *all* of the listed destination endpoints (effectively broadcasting the message to all of the destinations, like a JMS topic).

You can use the Java DSL to specify the topic load-balancing policy, as follows:

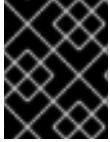
```
from("direct:start").loadBalance().topic().to("mock:x", "mock:y",
"mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <topic/>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```


Failover

Available as of Apache Camel 2.0 The **failover** load balancer is capable of trying the next processor in case an [Exchange](#) failed with an **exception** during processing. You can configure the **failover** with a list of specific exceptions that trigger failover. If you do not specify any exceptions, failover is triggered by any exception. The failover load balancer uses the same strategy for matching exceptions as the **onException** exception clause.



ENABLE STREAM CACHING IF USING STREAMS

If you use streaming, you should enable [Stream Caching](#) when using the failover load balancer. This is needed so the stream can be re-read when failing over.

The **failover** load balancer supports the following options:

Option	Type	Default	Description
inheritErrorHandler	boolean	true	<p>Camel 2.3: Specifies whether to use the errorHandler configured on the route. If you want to fail over immediately to the next endpoint, you should disable this option (value of false). If you enable this option, Apache Camel will first attempt to process the message using the errorHandler.</p> <p>For example, the errorHandler might be configured to redeliver messages and use delays between attempts. Apache Camel will initially try to redeliver to the <i>original</i> endpoint, and only fail over to the next endpoint when the errorHandler is exhausted.</p>
maximumFailoverAttempts	int	-1	<p>Camel 2.3: Specifies the maximum number of attempts to fail over to a new endpoint. The value, 0, implies that <i>no</i> failover attempts are made and the value, -1, implies an infinite number of failover attempts.</p>

roundRobin	boolean	false	Camel 2.3: Specifies whether the failover load balancer should operate in round robin mode or not. If not, it will <i>always</i> start from the first endpoint when a new message is to be processed. In other words it restarts from the top for every message. If round robin is enabled, it keeps state and continues with the next endpoint in a round robin fashion. When using round robin it will not <i>stick</i> to last known good endpoint, it will always pick the next endpoint to use.
-------------------	----------------	--------------	--

The following example is configured to fail over, only if an **IOException** exception is thrown:

```
from("direct:start")
    // here we will load balance if IOException was thrown
    // any other kind of exception will result in the Exchange as failed
    // to failover over any kind of exception we can just omit the
exception
    // in the failOver DSL
    .loadBalance().failover(IOException.class)
        .to("direct:x", "direct:y", "direct:z");
```

You can optionally specify multiple exceptions to fail over, as follows:

```
// enable redelivery so failover can react
errorHandler(defaultErrorHandler().maximumRedeliveries(5));

from("direct:foo")
    .loadBalance()
    .failover(IOException.class, MyOtherException.class)
    .to("direct:a", "direct:b");
```

You can configure the same route in XML, as follows:

```
<route errorHandlerRef="myErrorHandler">
  <from uri="direct:foo"/>
  <loadBalance>
    <failover>
      <exception>java.io.IOException</exception>
      <exception>com.mycompany.MyOtherException</exception>
    </failover>
    <to uri="direct:a"/>
    <to uri="direct:b"/>
  </loadBalance>
```

```
</route>
```

The following example shows how to fail over in round robin mode:

```
from("direct:start")
    // Use failover load balancer in stateful round robin mode,
    // which means it will fail over immediately in case of an exception
    // as it does NOT inherit error handler. It will also keep retrying,
as
    // it is configured to retry indefinitely.
    .loadBalance().failover(-1, false, true)
    .to("direct:bad", "direct:bad2", "direct:good", "direct:good2");
```

You can configure the same route in XML, as follows:

```
<route>
  <from uri="direct:start"/>
  <loadBalance>
    <!-- failover using stateful round robin,
    which will keep retrying the 4 endpoints indefinitely.
    You can set the maximumFailoverAttempt to break out after X
attempts -->
    <failover roundRobin="true"/>
    <to uri="direct:bad"/>
    <to uri="direct:bad2"/>
    <to uri="direct:good"/>
    <to uri="direct:good2"/>
  </loadBalance>
</route>
```

Weighted round robin and weighted random

In many enterprise environments, where server nodes of unequal processing power are hosting services, it is usually preferable to distribute the load in accordance with the individual server processing capacities. A *weighted round robin* algorithm or a *weighted random* algorithm can be used to address this problem.

The weighted load balancing policy allows you to specify a processing load *distribution ratio* for each server with respect to the others. You can specify this value as a positive processing weight for each server. A larger number indicates that the server can handle a larger load. The processing weight is used to determine the payload distribution ratio of each processing endpoint with respect to the others.

The parameters that can be used are

Table 7.5. Weighted Options

Option	Type	Default	Description
roundRobin	boolean	false	The default value for round-robin is false . In the absence of this setting or parameter, the load-balancing algorithm used is random.

distributionRatioDelimiter	String	,	The distributionRatioDelimiter is the delimiter used to specify the distributionRatio . If this attribute is not specified, comma , is the default delimiter.
-----------------------------------	---------------	---	---

The following Java DSL examples show how to define a weighted round-robin route and a weighted random route:

```
// Java
// round-robin
from("direct:start")
  .loadBalance().weighted(true, "4:2:1" distributionRatioDelimiter=":")
  .to("mock:x", "mock:y", "mock:z");

//random
from("direct:start")
  .loadBalance().weighted(false, "4,2,1")
  .to("mock:x", "mock:y", "mock:z");
```

You can configure the round-robin route in XML, as follows:

```
<!-- round-robin -->
<route>
  <from uri="direct:start"/>
  <loadBalance>
    <weighted roundRobin="true" distributionRatio="4:2:1"
distributionRatioDelimiter=":" />
    <to uri="mock:x"/>
    <to uri="mock:y"/>
    <to uri="mock:z"/>
  </loadBalance>
</route>
```

Custom Load Balancer

You can use a custom load balancer (eg your own implementation) also.

An example using Java DSL:

```
from("direct:start")
  // using our custom load balancer
  .loadBalance(new MyLoadBalancer())
  .to("mock:x", "mock:y", "mock:z");
```

And the same example using XML DSL:

```
<!-- this is the implementation of our custom load balancer -->
<bean id="myBalancer"
```

```

class="org.apache.camel.processor.CustomLoadBalanceTest$MyLoadBalancer"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <!-- refer to my custom load balancer -->
      <custom ref="myBalancer"/>
      <!-- these are the endpoints to balancer -->
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>

```

Notice in the XML DSL above we use `<custom>` which is only available in **Camel 2.8** onwards. In older releases you would have to do as follows instead:

```

<loadBalance ref="myBalancer">
  <!-- these are the endpoints to balancer -->
  <to uri="mock:x"/>
  <to uri="mock:y"/>
  <to uri="mock:z"/>
</loadBalance>

```

To implement a custom load balancer you can extend some support classes such as **LoadBalancerSupport** and **SimpleLoadBalancerSupport**. The former supports the asynchronous routing engine, and the latter does not. Here is an example:

```

public static class MyLoadBalancer extends LoadBalancerSupport {

    public boolean process(Exchange exchange, AsyncCallback callback) {
        String body = exchange.getIn().getBody(String.class);
        try {
            if ("x".equals(body)) {
                getProcessors().get(0).process(exchange);
            } else if ("y".equals(body)) {
                getProcessors().get(1).process(exchange);
            } else {
                getProcessors().get(2).process(exchange);
            }
        } catch (Throwable e) {
            exchange.setException(e);
        }
        callback.done(true);
        return true;
    }
}

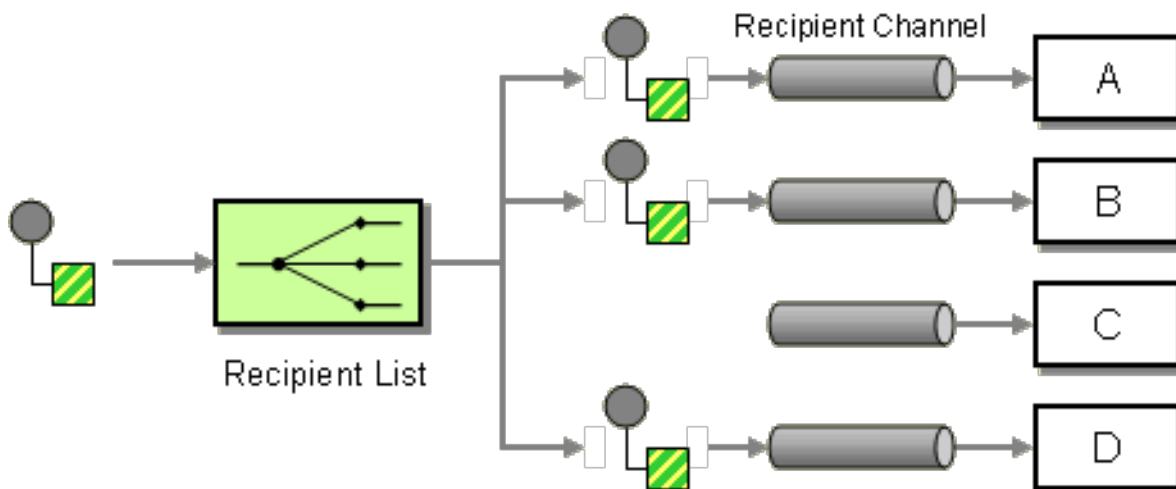
```

7.11. MULTICAST

Overview

The *multicast* pattern, shown in [Figure 7.10, "Multicast Pattern"](#), is a variation of the [recipient list](#) with a fixed destination pattern, which is compatible with the *InOut* message exchange pattern. This is in contrast to recipient list, which is only compatible with the *InOnly* exchange pattern.

Figure 7.10. Multicast Pattern



Multicast with a custom aggregation strategy

Whereas the multicast processor receives multiple *Out* messages in response to the original request (one from each of the recipients), the original caller is only expecting to receive a *single* reply. Thus, there is an inherent mismatch on the reply leg of the message exchange, and to overcome this mismatch, you must provide a custom *aggregation strategy* to the multicast processor. The aggregation strategy class is responsible for aggregating all of the *Out* messages into a single reply message.

Consider the example of an electronic auction service, where a seller offers an item for sale to a list of buyers. The buyers each put in a bid for the item, and the seller automatically selects the bid with the highest price. You can implement the logic for distributing an offer to a fixed list of buyers using the `multicast()` DSL command, as follows:

```
from("cxf:bean:offer").multicast(new HighestBidAggregationStrategy()).
    to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

Where the seller is represented by the endpoint, `cxf:bean:offer`, and the buyers are represented by the endpoints, `cxf:bean:Buyer1`, `cxf:bean:Buyer2`, `cxf:bean:Buyer3`. To consolidate the bids received from the various buyers, the multicast processor uses the aggregation strategy, `HighestBidAggregationStrategy`. You can implement the `HighestBidAggregationStrategy` in Java, as follows:

```
// Java
import org.apache.camel.processor.aggregate.AggregationStrategy;
import org.apache.camel.Exchange;

public class HighestBidAggregationStrategy implements AggregationStrategy
{
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
    {
        float oldBid = oldExchange.getOut().getHeader("Bid", Float.class);
        float newBid = newExchange.getOut().getHeader("Bid", Float.class);
```

```

        return (newBid > oldBid) ? newExchange : oldExchange;
    }
}

```

Where it is assumed that the buyers insert the bid price into a header named, **Bid**. For more details about custom aggregation strategies, see [Section 7.5, “Aggregator”](#).

Parallel processing

By default, the multicast processor invokes each of the recipient endpoints one after another (in the order listed in the `to()` command). In some cases, this might cause unacceptably long latency. To avoid these long latency times, you have the option of enabling parallel processing by adding the `parallelProcessing()` clause. For example, to enable parallel processing in the electronic auction example, define the route as follows:

```

from("cxf:bean:offer")
    .multicast(new HighestBidAggregationStrategy())
    .parallelProcessing()
    .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");

```

Where the multicast processor now invokes the buyer endpoints, using a thread pool that has one thread for each of the endpoints.

If you want to customize the size of the thread pool that invokes the buyer endpoints, you can invoke the `executorService()` method to specify your own custom executor service. For example:

```

from("cxf:bean:offer")
    .multicast(new HighestBidAggregationStrategy())
    .executorService(MyExecutor)
    .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");

```

Where *MyExecutor* is an instance of [java.util.concurrent.ExecutorService](#) type.

When the exchange has an *InOut* pattern, an aggregation strategy is used to aggregate reply messages. The default aggregation strategy takes the latest reply message and discards earlier replies. For example, in the following route, the custom strategy, **MyAggregationStrategy**, is used to aggregate the replies from the endpoints, **direct:a**, **direct:b**, and **direct:c**:

```

from("direct:start")
    .multicast(new MyAggregationStrategy())
    .parallelProcessing()
    .timeout(500)
    .to("direct:a", "direct:b", "direct:c")
    .end()
    .to("mock:result");

```

XML configuration example

The following example shows how to configure a similar route in XML, where the route uses a custom aggregation strategy and a custom thread executor:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

```

```

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd
    ">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
    <from uri="cxf:bean:offer"/>
    <multicast strategyRef="highestBidAggregationStrategy"
    parallelProcessing="true"
    threadPoolRef="myThreadExcutor">
    <to uri="cxf:bean:Buyer1"/>
    <to uri="cxf:bean:Buyer2"/>
    <to uri="cxf:bean:Buyer3"/>
    </multicast>
    </route>
    </camelContext>

    <bean id="highestBidAggregationStrategy"
class="com.acme.example.HighestBidAggregationStrategy"/>
    <bean id="myThreadExcutor" class="com.acme.example.MyThreadExcutor"/>

</beans>

```

Where both the **parallelProcessing** attribute and the **threadPoolRef** attribute are optional. It is only necessary to set them if you want to customize the threading behavior of the multicast processor.

Apply custom processing to the outgoing messages

Before multicast sends a message to one of the recipient endpoints, it creates a message replica, which is a shallow copy of the original message. If you want to perform some custom processing on each message replica before the replica is sent to its endpoint, you can invoke the **onPrepare** DSL command in the **multicast** clause. The **onPrepare** command inserts a custom processor just *after* the message has been shallow-copied and just *before* the message is dispatched to its endpoint. For example, in the following route, the **CustomProc** processor is invoked on the message sent to **direct:a** and the **CustomProc** processor is also invoked on the message sent to **direct:b**.

```

from("direct:start")
    .multicast().onPrepare(new CustomProc())
    .to("direct:a").to("direct:b");

```

A common use case for the **onPrepare** DSL command is to perform a deep copy of some or all elements of a message. For example, the following **CustomProc** processor class performs a deep copy of the message body, where the message body is presumed to be of type, **BodyType**, and the deep copy is performed by the method, **BodyType.deepCopy()**.

```

// Java
import org.apache.camel.*;
...
public class CustomProc implements Processor {

    public void process(Exchange exchange) throws Exception {

```



```

        BodyType body = exchange.getIn().getBody(BodyType.class);

        // Make a _deep_ copy of of the body object
        BodyType clone = BodyType.deepCopy();
        exchange.getIn().setBody(clone);

        // Headers and attachments have already been
        // shallow-copied. If you need deep copies,
        // add some more code here.
    }
}

```



NOTE

Although the **multicast** syntax allows you to invoke the **process** DSL command in the **multicast** clause, this does not make sense semantically and it does *not* have the same effect as **onPrepare** (in fact, in this context, the **process** DSL command has no effect).

Using onPrepare to execute custom logic when preparing messages

The **Multicast** will copy the source **Exchange** and multicast each copy. However the copy is a shallow copy, so in case you have mutable message bodies, then any changes will be visible by the other copied messages. If you want to use a deep clone copy then you need to use a custom **onPrepare** which allows you to do this using the **Processor** interface.

Notice the **onPrepare** can be used for any kind of custom logic which you would like to execute before the **Exchange** is being multicast.



NOTE

Its best practice to design for immutable objects.

For example if you have a mutable message body as this **Animal** class:

```

public class Animal implements Serializable {

    private int id;
    private String name;

    public Animal() {
    }

    public Animal(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public Animal deepClone() {
        Animal clone = new Animal();
        clone.setId(getId());
        clone.setName(getName());
        return clone;
    }
}

```

```

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return id + " " + name;
    }
}

```

Then we can create a deep clone processor which clones the message body:

```

public class AnimalDeepClonePrepare implements Processor {

    public void process(Exchange exchange) throws Exception {
        Animal body = exchange.getIn().getBody(Animal.class);

        // do a deep clone of the body which wont affect when doing
multicasting
        Animal clone = body.deepClone();
        exchange.getIn().setBody(clone);
    }
}

```

Then we can use the `AnimalDeepClonePrepare` class in the [Multicast](#) route using the `onPrepare` option as shown:

```

from("direct:start")
    .multicast().onPrepare(new
AnimalDeepClonePrepare()).to("direct:a").to("direct:b");

```

And the same example in XML DSL

```

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <!-- use on prepare with multicast -->
        <multicast onPrepareRef="animalDeepClonePrepare">
            <to uri="direct:a"/>
            <to uri="direct:b"/>
        </multicast>
    </route>

```

```

<route>
  <from uri="direct:a"/>
  <process ref="processorA"/>
  <to uri="mock:a"/>
</route>
<route>
  <from uri="direct:b"/>
  <process ref="processorB"/>
  <to uri="mock:b"/>
</route>
</camelContext>

<!-- the on prepare Processor which performs the deep cloning -->
<bean id="animalDeepClonePrepare"
class="org.apache.camel.processor.AnimalDeepClonePrepare"/>

<!-- processors used for the last two routes, as part of unit test -->
<bean id="processorA"
class="org.apache.camel.processor.MulticastOnPrepareTest$ProcessorA"/>
<bean id="processorB"
class="org.apache.camel.processor.MulticastOnPrepareTest$ProcessorB"/>

```

Options

The **multicast** DSL command supports the following options:

Name	Default Value	Description
strategyRef		Refers to an AggregationStrategy to be used to assemble the replies from the multicasts, into a single outgoing message from the Multicast . By default Camel will use the last reply as the outgoing message.
parallelProcessing	false	If enables then sending messages to the multicasts occurs concurrently. Note the caller thread will still wait until all messages has been fully processed, before it continues. Its only the sending and processing the replies from the multicasts which happens concurrently.
executorServiceRef		Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatic implied, and you do not have to enable that option as well.

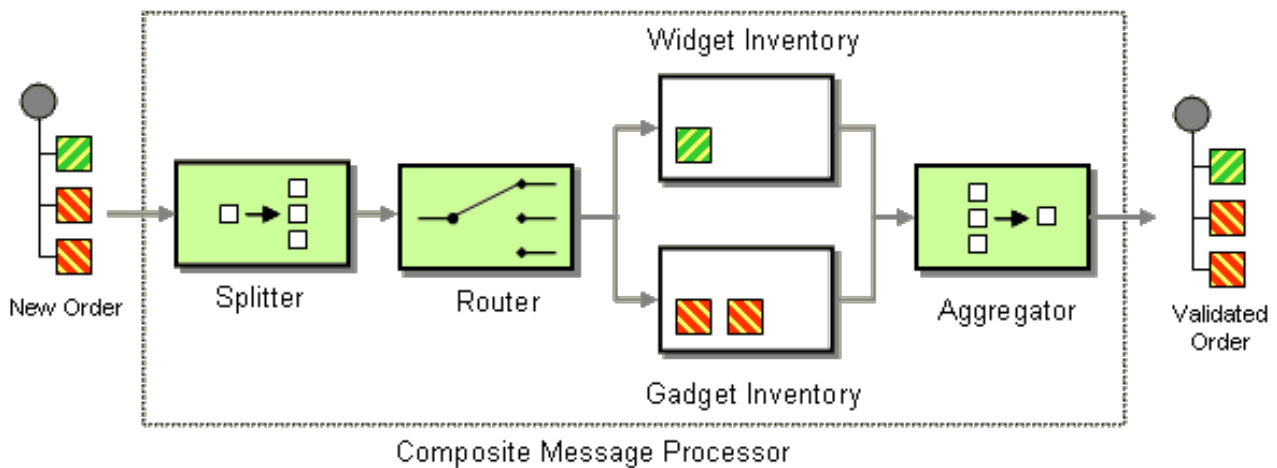
stopOnException	false	Camel 2.2: Whether or not to stop continue processing immediately when an exception occurred. If disable, then Camel will send the message to all multicasts regardless if one of them failed. You can deal with exceptions in the AggregationStrategy class where you have full control how to handle that.
streaming	false	If enabled then Camel will process replies out-of-order, eg in the order they come back. If disabled, Camel will process replies in the same order as multicasted.
timeout		Camel 2.5: Sets a total timeout specified in millis. If the Multicast hasn't been able to send and process all replies within the given timeframe, then the timeout triggers and the Multicast breaks out and continues. Notice if you provide a TimeoutAwareAggregationStrategy then the timeout method is invoked before breaking out.
onPrepareRef		Camel 2.8: Refers to a custom Processor to prepare the copy of the Exchange each multicast will receive. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.
shareUnitOfWork	false	Camel 2.8: Whether the unit of work should be shared. See the same option on Splitter for more details.

7.12. COMPOSED MESSAGE PROCESSOR

Composed Message Processor

The *composed message processor* pattern, as shown in [Figure 7.11](#), “[Composed Message Processor Pattern](#)”, allows you to process a composite message by splitting it up, routing the sub-messages to appropriate destinations, and then re-aggregating the responses back into a single message.

Figure 7.11. Composed Message Processor Pattern



Java DSL example

The following example checks that a multipart order can be filled, where each part of the order requires a check to be made at a different inventory:

```
// split up the order so individual OrderItems can be validated by the
// appropriate bean
from("direct:start")
  .split().body()
  .choice()
    .when().method("orderItemHelper", "isWidget")
      .to("bean:widgetInventory")
    .otherwise()
      .to("bean:gadgetInventory")
  .end()
  .to("seda:aggregate");

// collect and re-assemble the validated OrderItems into an order again
from("seda:aggregate")
  .aggregate(new MyOrderAggregationStrategy())
  .header("orderId")
  .completionTimeout(1000L)
  .to("mock:result");
```

XML DSL example

The preceding route can also be written in XML DSL, as follows:

```
<route>
  <from uri="direct:start"/>
  <split>
    <simple>body</simple>
    <choice>
      <when>
        <method bean="orderItemHelper" method="isWidget"/>
      </when>
      <otherwise>
      </otherwise>
    </choice>
  </split>
  <to uri="bean:widgetInventory"/>
  </to>
  <to uri="bean:gadgetInventory"/>
  </to>
```

```

        </otherwise>
    </choice>
    <to uri="seda:aggregate"/>
</split>
</route>

<route>
    <from uri="seda:aggregate"/>
    <aggregate strategyRef="myOrderAggregatorStrategy"
completionTimeout="1000">
        <correlationExpression>
            <simple>header.orderId</simple>
        </correlationExpression>
        <to uri="mock:result"/>
    </aggregate>
</route>

```

Processing steps

Processing starts by splitting the order, using a [Splitter](#). The [Splitter](#) then sends individual **OrderItems** to a [Content Based Router](#), which routes messages based on the item type. *Widget* items get sent for checking in the **widgetInventory** bean and *gadget* items get sent to the **gadgetInventory** bean. Once these **OrderItems** have been validated by the appropriate bean, they are sent on to the [Aggregator](#) which collects and re-assembles the validated **OrderItems** into an order again.

Each received order has a header containing an *order ID*. We make use of the order ID during the aggregation step: the `.header("orderId")` qualifier on the `aggregate()` DSL command instructs the aggregator to use the header with the key, **orderId**, as the correlation expression.

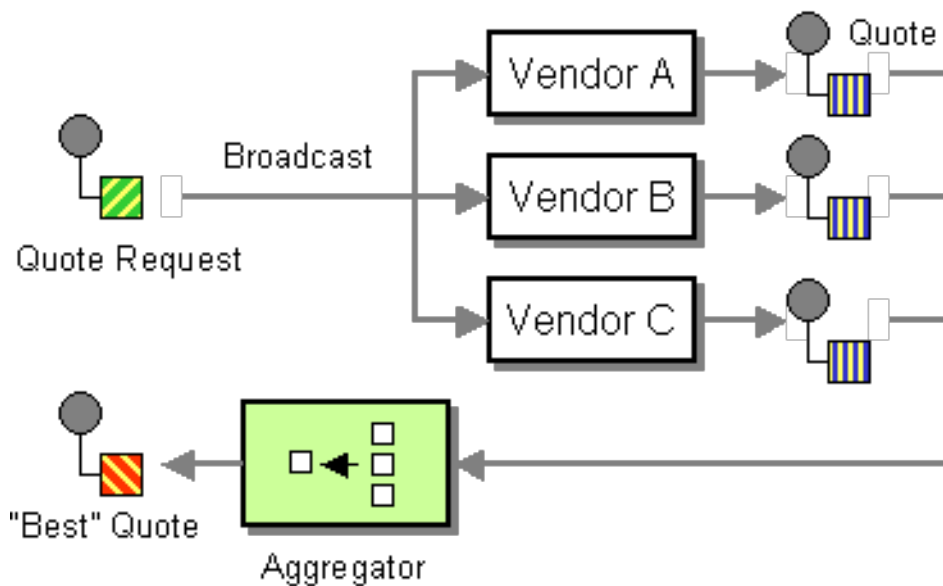
For full details, check the example source here:

7.13. SCATTER-GATHER

Scatter-Gather

The *scatter-gather pattern*, as shown in [Figure 7.12, "Scatter-Gather Pattern"](#), enables you to route messages to a number of dynamically specified recipients and re-aggregate the responses back into a single message.

Figure 7.12. Scatter-Gather Pattern



Dynamic scatter-gather example

The following example outlines an application that gets the best quote for beer from several different vendors. The examples uses a dynamic [Recipient List](#) to request a quote from all vendors and an [Aggregator](#) to pick the best quote out of all the responses. The routes for this application are defined as follows:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <recipientList>
      <header>listOfVendors</header>
    </recipientList>
  </route>
  <route>
    <from uri="seda:quoteAggregator"/>
    <aggregate strategyRef="aggregatorStrategy" completionTimeout="1000">
      <correlationExpression>
        <header>quoteRequestId</header>
      </correlationExpression>
      <to uri="mock:result"/>
    </aggregate>
  </route>
</camelContext>
```

In the first route, the [Recipient List](#) looks at the **listOfVendors** header to obtain the list of recipients. Hence, the client that sends messages to this application needs to add a **listOfVendors** header to the message. [Example 7.1, “Messaging Client Sample”](#) shows some sample code from a messaging client that adds the relevant header data to outgoing messages.

Example 7.1. Messaging Client Sample

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("listOfVendors", "bean:vendor1, bean:vendor2,
bean:vendor3");
```

```
headers.put("quoteRequestId", "quoteRequest-1");
template.sendBodyAndHeaders("direct:start", "<quote_request
item=\"beer\"/>", headers);
```

The message would be distributed to the following endpoints: **bean:vendor1**, **bean:vendor2**, and **bean:vendor3**. These beans are all implemented by the following class:

```
public class MyVendor {
    private int beerPrice;

    @Produce(uri = "seda:quoteAggregator")
    private ProducerTemplate quoteAggregator;

    public MyVendor(int beerPrice) {
        this.beerPrice = beerPrice;
    }

    public void getQuote(@XPath("/quote_request/@item") String item,
Exchange exchange) throws Exception {
        if ("beer".equals(item)) {
            exchange.getIn().setBody(beerPrice);
            quoteAggregator.send(exchange);
        } else {
            throw new Exception("No quote available for " + item);
        }
    }
}
```

The bean instances, **vendor1**, **vendor2**, and **vendor3**, are instantiated using Spring XML syntax, as follows:

```
<bean id="aggregatorStrategy"
class="org.apache.camel.spring.processor.scattergather.LowestQuoteAggregat
ionStrategy"/>

<bean id="vendor1"
class="org.apache.camel.spring.processor.scattergather.MyVendor">
    <constructor-arg>
        <value>1</value>
    </constructor-arg>
</bean>

<bean id="vendor2"
class="org.apache.camel.spring.processor.scattergather.MyVendor">
    <constructor-arg>
        <value>2</value>
    </constructor-arg>
</bean>

<bean id="vendor3"
class="org.apache.camel.spring.processor.scattergather.MyVendor">
    <constructor-arg>
```



```

    <value>3</value>
  </constructor-arg>
</bean>

```

Each bean is initialized with a different price for beer (passed to the constructor argument). When a message is sent to each bean endpoint, it arrives at the `MyVendor.getQuote` method. This method does a simple check to see whether this quote request is for beer and then sets the price of beer on the exchange for retrieval at a later step. The message is forwarded to the next step using [POJO Producing](#) (see the `@Produce` annotation).

At the next step, we want to take the beer quotes from all vendors and find out which one was the best (that is, the lowest). For this, we use an [Aggregator](#) with a custom aggregation strategy. The [Aggregator](#) needs to identify which messages are relevant to the current quote, which is done by correlating messages based on the value of the `quoteRequestId` header (passed to the `correlationExpression`). As shown in [Example 7.1, "Messaging Client Sample"](#), the correlation ID is set to `quoteRequest - 1` (the correlation ID should be unique). To pick the lowest quote out of the set, you can use a custom aggregation strategy like the following:

```

public class LowestQuoteAggregationStrategy implements AggregationStrategy
{
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
    {
        // the first time we only have the new exchange
        if (oldExchange == null) {
            return newExchange;
        }

        if (oldExchange.getIn().getBody(int.class) <
newExchange.getIn().getBody(int.class)) {
            return oldExchange;
        } else {
            return newExchange;
        }
    }
}

```

Static scatter-gather example

You can specify the recipients explicitly in the scatter-gather application by employing a static [Recipient List](#). The following example shows the routes you would use to implement a static scatter-gather scenario:

```

from("direct:start").multicast().to("seda:vendor1", "seda:vendor2",
"seda:vendor3");

from("seda:vendor1").to("bean:vendor1").to("seda:quoteAggregator");
from("seda:vendor2").to("bean:vendor2").to("seda:quoteAggregator");
from("seda:vendor3").to("bean:vendor3").to("seda:quoteAggregator");

from("seda:quoteAggregator")
    .aggregate(header("quoteRequestId"), new
LowestQuoteAggregationStrategy()).to("mock:result")

```

7.14. LOOP

Loop

The *loop* pattern enables you to process a message multiple times. It is used mainly for testing.



DEFAULT MODE

Notice by default the loop uses the same exchange throughout the looping. So the result from the previous iteration is used for the next (eg [Pipes and Filters](#)). From **Camel 2.8** onwards you can enable copy mode instead. See the options table for more details.

Exchange properties

On each loop iteration, two exchange properties are set, which can optionally be read by any processors included in the loop.

Property	Description
CamelLoopSize	Apache Camel 2.0: Total number of loops
CamelLoopIndex	Apache Camel 2.0: Index of the current iteration (0 based)

Java DSL examples

The following examples show how to take a request from the **direct:x** endpoint and then send the message repeatedly to **mock:result**. The number of loop iterations is specified either as an argument to **loop()** or by evaluating an expression at run time, where the expression *must* evaluate to an **int** (or else a **RuntimeException** is thrown).

The following example passes the loop count as a constant:

```
from("direct:a").loop(8).to("mock:result");
```

The following example evaluates a simple expression to determine the loop count:

```
from("direct:b").loop(header("loop")).to("mock:result");
```

The following example evaluates an XPath expression to determine the loop count:

```
from("direct:c").loop().xpath("/hello/@times").to("mock:result");
```

XML configuration example

You can configure the same routes in Spring XML.

The following example passes the loop count as a constant:

```
<route>
```

```

<from uri="direct:a"/>
<loop>
  <constant>8</constant>
  <to uri="mock:result"/>
</loop>
</route>

```

The following example evaluates a simple expression to determine the loop count:

```

<route>
  <from uri="direct:b"/>
  <loop>
    <header>loop</header>
    <to uri="mock:result"/>
  </loop>
</route>

```

Using copy mode

Now suppose we send a message to **direct:start** endpoint containing the letter A. The output of processing this route will be that, each **mock:loop** endpoint will receive AB as message.

```

from("direct:start")
  // instruct loop to use copy mode, which mean it will use a copy of
  // the input exchange
  // for each loop iteration, instead of keep using the same exchange
  // all over
  .loop(3).copy()
    .transform(body().append("B"))
    .to("mock:loop")
  .end()
  .to("mock:result");

```

However if we do *not* enable copy mode then **mock:loop** will receive AB, ABB, AB BB messages.

```

from("direct:start")
  // by default loop will keep using the same exchange so on the 2nd
  // and 3rd iteration its
  // the same exchange that was previous used that are being looped all
  // over
  .loop(3)
    .transform(body().append("B"))
    .to("mock:loop")
  .end()
  .to("mock:result");

```

The equivalent example in XML DSL in copy mode is as follows:

```

<route>
  <from uri="direct:start"/>
  <!-- enable copy mode for loop eip -->
  <loop copy="true">
    <constant>3</constant>
    <transform>

```

```

        <simple>${body}B</simple>
    </transform>
    <to uri="mock:loop"/>
</loop>
    <to uri="mock:result"/>
</route>

```

Options

The **loop** DSL command supports the following options:

Name	Default Value	Description
copy	false	Camel 2.8: Whether or not copy mode is used. If false then the same Exchange is being used throughout the looping. So the result from the previous iteration will be <i>visible</i> for the next iteration. Instead you can enable copy mode, and then each iteration is <i>restarting</i> with a fresh copy of the input Exchange .

7.15. SAMPLING

Sampling Throttler

A sampling throttler allows you to extract a sample of exchanges from the traffic through a route. It is configured with a sampling period during which only a *single* exchange is allowed to pass through. All other exchanges will be stopped.

By default, the sample period is 1 second.

Java DSL example

Use the **sample()** DSL command to invoke the sampler as follows:

```

// Sample with default sampling period (1 second)
from("direct:sample")
    .sample()
    .to("mock:result");

// Sample with explicitly specified sample period
from("direct:sample-configured")
    .sample(1, TimeUnit.SECONDS)
    .to("mock:result");

// Alternative syntax for specifying sampling period
from("direct:sample-configured-via-dsl")
    .sample().samplePeriod(1).timeUnits(TimeUnit.SECONDS)
    .to("mock:result");

```

```

from("direct:sample-messageFrequency")
    .sample(10)
    .to("mock:result");

from("direct:sample-messageFrequency-via-dsl")
    .sample().sampleMessageFrequency(5)
    .to("mock:result");

```

Spring XML example

In Spring XML, use the `sample` element to invoke the sampler, where you have the option of specifying the sampling period using the **samplePeriod** and **units** attributes:

```

<route>
  <from uri="direct:sample"/>
  <sample samplePeriod="1" units="seconds">
    <to uri="mock:result"/>
  </sample>
</route>
<route>
  <from uri="direct:sample-messageFrequency"/>
  <sample messageFrequency="10">
    <to uri="mock:result"/>
  </sample>
</route>
<route>
  <from uri="direct:sample-messageFrequency-via-dsl"/>
  <sample messageFrequency="5">
    <to uri="mock:result"/>
  </sample>
</route>

```

Options

The **sample** DSL command supports the following options:

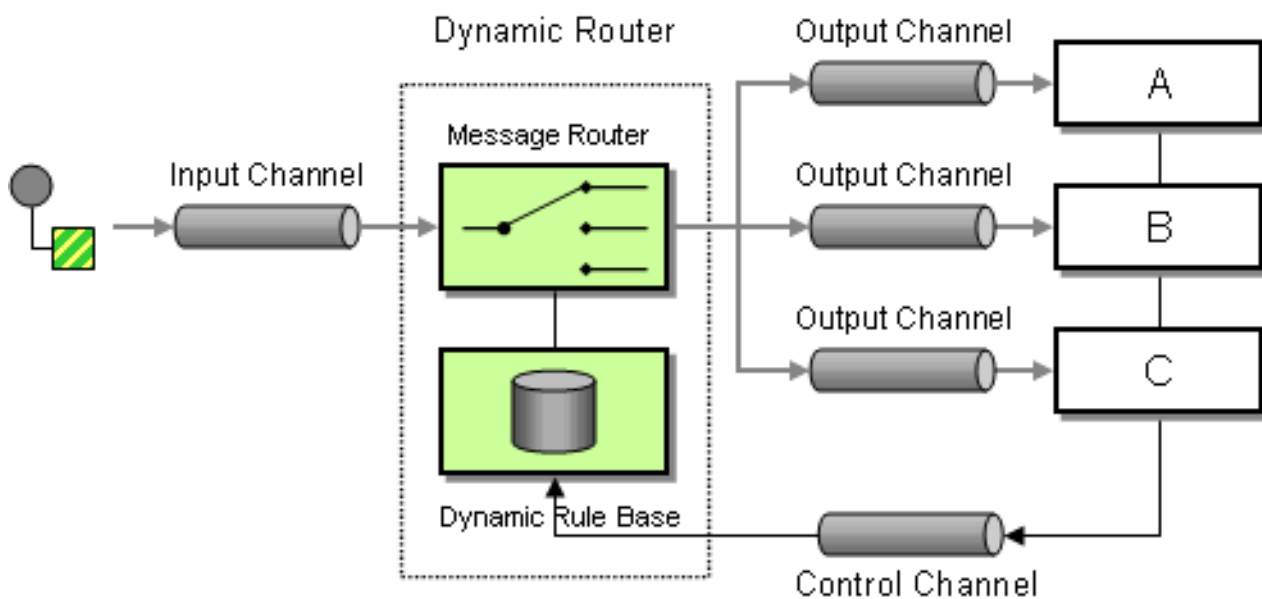
Name	Default Value	Description
messageFrequency		Samples the message every N'th message. You can only use either frequency or period.
samplePeriod	1	Samples the message every N'th period. You can only use either frequency or period.
units	SECOND	Time unit as an enum of java.util.concurrent.TimeUnit from the JDK.

7.16. DYNAMIC ROUTER

Dynamic Router

The [Dynamic Router](#) pattern, as shown in [Figure 7.13, “Dynamic Router Pattern”](#), enables you to route a message consecutively through a series of processing steps, where the sequence of steps is not known at design time. The list of endpoints through which the message should pass is calculated dynamically *at run time*. Each time the message returns from an endpoint, the dynamic router calls back on a bean to discover the next endpoint in the route.

Figure 7.13. Dynamic Router Pattern



In **Camel 2.5** we introduced a **dynamicRouter** in the DSL, which is like a dynamic [Routing Slip](#) that evaluates the slip *on-the-fly*.



BEWARE

You must ensure the expression used for the **dynamicRouter** (such as a bean), returns **null** to indicate the end. Otherwise, the **dynamicRouter** will continue in an endless loop.

Dynamic Router in Camel 2.5 onwards

From Camel 2.5, the [Dynamic Router](#) updates the exchange property, **Exchange.SLIP_ENDPOINT**, with the current endpoint as it advances through the slip. This enables you to find out how far the exchange has progressed through the slip. (It's a slip because the [Dynamic Router](#) implementation is based on [Routing Slip](#)).

Java DSL

In Java DSL you can use the **dynamicRouter** as follows:

■

```
from("direct:start")
    // use a bean as the dynamic router
    .dynamicRouter(bean(DynamicRouterTest.class, "slip"));
```

Which will leverage a [Bean](#) to compute the slip *on-the-fly*, which could be implemented as follows:

```
// Java
/**
 * Use this method to compute dynamic where we should route next.
 *
 * @param body the message body
 * @return endpoints to go, or <tt>null</tt> to indicate the end
 */
public String slip(String body) {
    bodies.add(body);
    invoked++;

    if (invoked == 1) {
        return "mock:a";
    } else if (invoked == 2) {
        return "mock:b,mock:c";
    } else if (invoked == 3) {
        return "direct:foo";
    } else if (invoked == 4) {
        return "mock:result";
    }

    // no more so return null
    return null;
}
```



NOTE

The preceding example is *not* thread safe. You would have to store the state on the **Exchange** to ensure thread safety.

Spring XML

The same example in Spring XML would be:

```
<bean id="mySlip" class="org.apache.camel.processor.DynamicRouterTest"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <dynamicRouter>
            <!-- use a method call on a bean as dynamic router -->
            <method ref="mySlip" method="slip"/>
        </dynamicRouter>
    </route>

    <route>
        <from uri="direct:foo"/>
        <transform><constant>Bye World</constant></transform>
    </route>
</camelContext>
```

```

        <to uri="mock:foo"/>
    </route>

</camelContext>

```

Options

The `dynamicRouter` DSL command supports the following options:

Name	Default Value	Description
<code>uriDelimiter</code>	<code>,</code>	Delimiter used if the Expression returned multiple endpoints.
<code>ignoreInvalidEndpoints</code>	<code>false</code>	If an endpoint uri could not be resolved, should it be ignored. Otherwise Camel will throw an exception stating the endpoint uri is not valid.

@DynamicRouter annotation

You can also use the `@DynamicRouter` annotation. For example:

```

// Java
public class MyDynamicRouter {

    @Consume(uri = "activemq:foo")
    @DynamicRouter
    public String route(@XPath("/customer/id") String customerId,
        @Header("Location") String location, Document body) {
        // query a database to find the best match of the endpoint based
        on the input parameteres
        // return the next endpoint uri, where to go. Return null to
        indicate the end.
    }
}

```

The `route` method is invoked repeatedly as the message progresses through the slip. The idea is to return the endpoint URI of the next destination. Return `null` to indicate the end. You can return multiple endpoints if you like, just as the [Routing Slip](#), where each endpoint is separated by a delimiter.

CHAPTER 8. MESSAGE TRANSFORMATION

Abstract

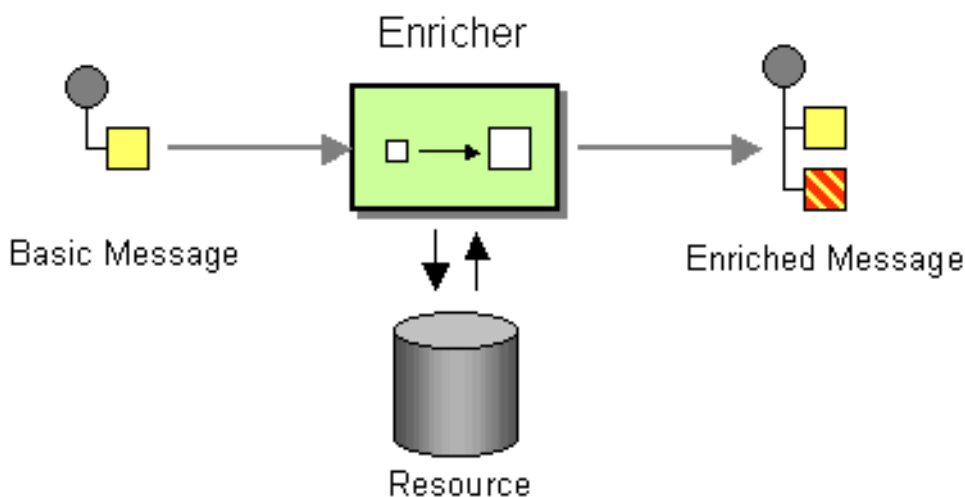
The message transformation patterns describe how to modify the contents of messages for various purposes.

8.1. CONTENT ENRICHER

Overview

The *content enricher* pattern describes a scenario where the message destination requires more data than is present in the original message. In this case, you would use a content enricher to pull in the extra data from an external resource.

Figure 8.1. Content Enricher Pattern



Models of content enrichment

Apache Camel supports two kinds of content enricher, as follows:

- **enrich()**—obtains additional data from the resource by sending a copy of the current exchange to a *producer* endpoint and then using the data from the resulting reply (the exchange created by the enricher is always an *InOut* exchange).
- **pollEnrich()**—obtains the additional data by polling a *consumer* endpoint for data. Effectively, the consumer endpoint from the main route and the consumer endpoint in **pollEnrich()** are coupled, such that exchanges incoming on the main route trigger a poll of the **pollEnrich()** endpoint.

Content enrichment using `enrich()`

```

AggregationStrategy aggregationStrategy = ...

from("direct:start")
  .enrich("direct:resource", aggregationStrategy)
  .to("direct:result");
  
```

```
from("direct:resource")
...
```

The content enricher (**enrich**) retrieves additional data from a *resource endpoint* in order to enrich an incoming message (contained in the *original exchange*). An aggregation strategy combines the original exchange and the *resource exchange*. The first parameter of the **AggregationStrategy.aggregate(Exchange, Exchange)** method corresponds to the original exchange, and the second parameter corresponds to the resource exchange. The results from the resource endpoint are stored in the resource exchange's *Out* message. Here is a sample template for implementing your own aggregation strategy class:

```
public class ExampleAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange original, Exchange resource) {
        Object originalBody = original.getIn().getBody();
        Object resourceResponse = resource.getOut().getBody();
        Object mergeResult = ... // combine original body and resource
        response
        if (original.getPattern().isOutCapable()) {
            original.getOut().setBody(mergeResult);
        } else {
            original.getIn().setBody(mergeResult);
        }
        return original;
    }
}
```

Using this template, the original exchange can have any exchange pattern. The resource exchange created by the enricher is always an *InOut* exchange.

Spring XML enrich example

The preceding example can also be implemented in Spring XML:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <enrich uri="direct:resource" strategyRef="aggregationStrategy"/>
    <to uri="direct:result"/>
  </route>
  <route>
    <from uri="direct:resource"/>
    ...
  </route>
</camelContext>

<bean id="aggregationStrategy" class="..." />
```

Default aggregation strategy

The aggregation strategy is optional. If you do not provide it, Apache Camel will use the body obtained from the resource by default. For example:

```
from("direct:start")
  .enrich("direct:resource")
  .to("direct:result");
```

In the preceding route, the message sent to the **direct:result** endpoint contains the output from the **direct:resource**, because this example does not use any custom aggregation.

In XML DSL, just omit the **strategyRef** attribute, as follows:

```
<route>
  <from uri="direct:start"/>
  <enrich uri="direct:resource"/>
  <to uri="direct:result"/>
</route>
```

Enrich Options

The **enrich** DSL command supports the following options:

Name	Default Value	Description
uri		The endpoint uri for the external service to enrich from. You must use either uri or ref .
ref		Refers to the endpoint for the external service to enrich from. You must use either uri or ref .
strategyRef		Refers to an AggregationStrategy to be used to merge the reply from the external service, into a single outgoing message. By default Camel will use the reply from the external service as outgoing message.

Content enrich using pollEnrich

The **pollEnrich** command treats the resource endpoint as a *consumer*. Instead of sending an exchange to the resource endpoint, it *polls* the endpoint. By default, the poll returns immediately, if there is no exchange available from the resource endpoint. For example, the following route reads a file whose name is extracted from the header of an incoming JMS message:

```
from("activemq:queue:order")
  .pollEnrich("file://order/data/additional?fileName=orderId")
  .to("bean:processOrder");
```

And if you want to wait at most 20 seconds for the file to be ready, you can use a timeout as follows:

```
from("activemq:queue:order")
```

```
.pollEnrich("file://order/data/additional?fileName=orderId", 20000) //
timeout is in milliseconds
.to("bean:processOrder");
```

You can also specify an aggregation strategy for **pollEnrich**, as follows:

```
.pollEnrich("file://order/data/additional?fileName=orderId", 20000,
aggregationStrategy)
```



NOTE

The resource exchange passed to the aggregation strategy's **aggregate()** method might be **null**, if the poll times out before an exchange is received.



DATA FROM CURRENT EXCHANGE NOT USED

pollEnrich does *not* access any data from the current Exchange, so that, when polling, it cannot use any of the existing headers you may have set on the Exchange. For example, you cannot set a filename in the **Exchange.FILE_NAME** header and use **pollEnrich** to consume only that file. For that, you must set the filename in the endpoint URI.

Polling methods used by pollEnrich()

In general, the **pollEnrich()** enricher polls the consumer endpoint using one of the following polling methods:

- **receiveNowait()** (*used by default*)
- **receive()**
- **receive(long timeout)**

The **pollEnrich()** command's timeout argument (specified in milliseconds) determines which method gets called, as follows:

- Timeout is **0** or not specified, **receiveNowait** is called.
- Timeout is negative, **receive** is called.
- Otherwise, **receive(timeout)** is called.

pollEnrich example

In this example we enrich the message by loading the content from the file named `inbox/data.txt`.

```
from("direct:start")
.pollEnrich("file:inbox?fileName=data.txt")
.to("direct:result");
```

And in XML DSL you do:

```
<route>
  <from uri="direct:start"/>
  <pollEnrich uri="file:inbox?fileName=data.txt"/>
  <to uri="direct:result"/>
</route>
```

If there is no file then the message is empty. We can use a timeout to either wait (potential forever) until a file exists, or use a timeout to wait a period. For example to wait up til 5 seconds you can do:

```
<route>
  <from uri="direct:start"/>
  <pollEnrich uri="file:inbox?fileName=data.txt" timeout="5000"/>
  <to uri="direct:result"/>
</route>
```

PollEnrich Options

The `pollEnrich` DSL command supports the following options:

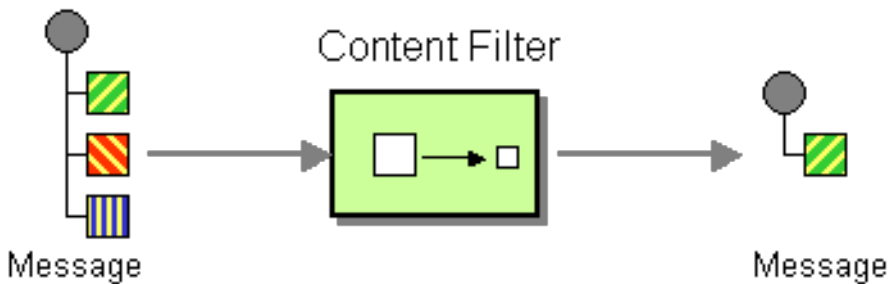
Name	Default Value	Description
<code>uri</code>		The endpoint uri for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .
<code>ref</code>		Refers to the endpoint for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .
<code>strategyRef</code>		Refers to an AggregationStrategy to be used to merge the reply from the external service, into a single outgoing message. By default Camel will use the reply from the external service as outgoing message.
<code>timeout</code>	<code>0</code>	Timeout in millis to use when polling from the external service. See below for important details about the timeout.

8.2. CONTENT FILTER

Overview

The *content filter* pattern describes a scenario where you need to filter out extraneous content from a message before delivering it to its intended recipient. For example, you might employ a content filter to strip out confidential information from a message.

Figure 8.2. Content Filter Pattern



A common way to filter messages is to use an expression in the DSL, written in one of the supported scripting languages (for example, XSLT, XQuery or JoSQL).

Implementing a content filter

A content filter is essentially an application of a message processing technique for a particular purpose. To implement a content filter, you can employ any of the following message processing techniques:

- *Message translator*—see [message translators](#).
- *Processors*—see [chapter "Implementing a Processor" in "Programming EIP Components"](#).
- [Bean integration](#).

XML configuration example

The following example shows how to configure the same route in XML:

```

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="xslt:classpath:com/acme/content_filter.xsl"/>
    <to uri="activemq:Another.Queue"/>
  </route>
</camelContext>
  
```

Using an XPath filter

You can also use XPath to filter out part of the message you are interested in:

```

<route>
  <from uri="activemq:Input"/>
  <setBody><xpath resultType="org.w3c.dom.Document">//foo:bar</xpath>
</setBody>
  <to uri="activemq:Output"/>
</route>
  
```

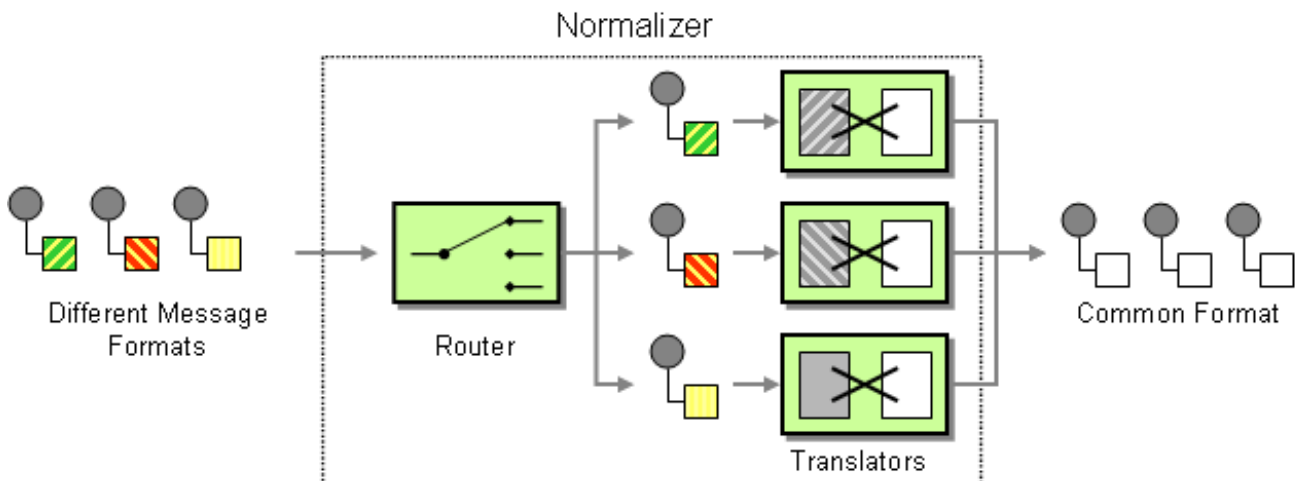
8.3. NORMALIZER

Overview

The *normalizer* pattern is used to process messages that are semantically equivalent, but arrive in different formats. The normalizer transforms the incoming messages into a common format.

In Apache Camel, you can implement the normalizer pattern by combining a [content-based router](#), which detects the incoming message's format, with a collection of different [message translators](#), which transform the different incoming formats into a common format.

Figure 8.3. Normalizer Pattern



Java DSL example

This example shows a Message Normalizer that converts two types of XML messages into a common format. Messages in this common format are then filtered.

Using the Fluent Builders

```
// we need to normalize two types of incoming messages
from("direct:start")
    .choice()
        .when().xpath("/employee").to("bean:normalizer?
method=employeeToPerson")
        .when().xpath("/customer").to("bean:normalizer?
method=customerToPerson")
    .end()
    .to("mock:result");
```

In this case we're using a Java bean as the normalizer. The class looks like this

```
// Java
public class MyNormalizer {
    public void employeeToPerson(Exchange exchange,
    @XPath("/employee/name/text()") String name) {
        exchange.getOut().setBody(createPerson(name));
    }

    public void customerToPerson(Exchange exchange,
    @XPath("/customer/@name") String name) {
        exchange.getOut().setBody(createPerson(name));
    }
}
```

```

    private String createPerson(String name) {
        return "<person name=\"" + name + "\"/>";
    }
}

```

XML configuration example

The same example in the XML DSL

```

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <xpath>/employee</xpath>
        <to uri="bean:normalizer?method=employeeToPerson"/>
      </when>
      <when>
        <xpath>/customer</xpath>
        <to uri="bean:normalizer?method=customerToPerson"/>
      </when>
    </choice>
    <to uri="mock:result"/>
  </route>
</camelContext>

<bean id="normalizer" class="org.apache.camel.processor.MyNormalizer"/>

```

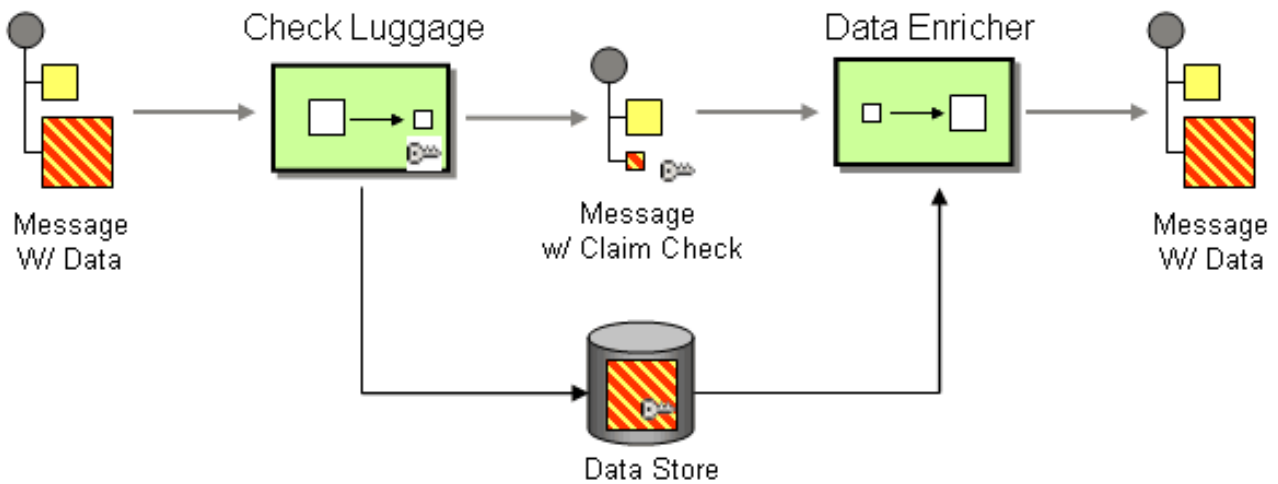
8.4. CLAIM CHECK

Claim Check

The *claim check* pattern, shown in [Figure 8.4, “Claim Check Pattern”](#), allows you to replace message content with a claim check (a unique key), which can be used to retrieve the message content at a later time. The message content is stored temporarily in a persistent store like a database or file system. This pattern is very useful when message content is very large (thus it would be expensive to send around) and not all components require all information.

It can also be useful in situations where you cannot trust the information with an outside party; in this case, you can use the Claim Check to hide the sensitive portions of data.

Figure 8.4. Claim Check Pattern



Java DSL example

The following example shows how to replace a message body with a claim check and restore the body at a later step.

```
from("direct:start").to("bean:checkLuggage", "mock:testCheckpoint",
    "bean:dataEnricher", "mock:result");
```

The next step in the pipeline is the `mock:testCheckpoint` endpoint, which checks that the message body has been removed, the claim check added, and so on.

XML DSL example

The preceding example can also be written in XML, as follows:

```
<route>
  <from uri="direct:start"/>
  <pipeline>
    <to uri="bean:checkLuggage"/>
    <to uri="mock:testCheckpoint"/>
    <to uri="bean:dataEnricher"/>
    <to uri="mock:result"/>
  </pipeline>
</route>
```

checkLuggage bean

The message is first sent to the `checkLuggage` bean which is implemented as follows:

```
public static final class CheckLuggageBean {
    public void checkLuggage(Exchange exchange, @Body String body,
        @XPath("/order/@custId") String custId) {
        // store the message body into the data store, using the custId as
        the claim check
        datastore.put(custId, body);
        // add the claim check as a header
        exchange.getIn().setHeader("claimCheck", custId);
    }
}
```

```

        // remove the body from the message
        exchange.getIn().setBody(null);
    }
}

```

This bean stores the message body into the data store, using the **custId** as the claim check. In this example, we are using a **HashMap** to store the message body; in a real application you would use a database or the file system. The claim check is added as a message header for later use and, finally, we remove the body from the message and pass it down the pipeline.

testCheckpoint endpoint

The example route is just a [Pipeline](#). In a real application, you would substitute some other steps for the **mock:testCheckpoint** endpoint.

dataEnricher bean

To add the message body back into the message, we use the **dataEnricher** bean, which is implemented as follows:

```

public static final class DataEnricherBean {
    public void addDataBackIn(Exchange exchange, @Header("claimCheck")
String claimCheck) {
        // query the data store using the claim check as the key and add
the data
        // back into the message body
        exchange.getIn().setBody(dataStore.get(claimCheck));
        // remove the message data from the data store
        dataStore.remove(claimCheck);
        // remove the claim check header
        exchange.getIn().removeHeader("claimCheck");
    }
}

```

This bean queries the data store, using the claim check as the key, and then adds the recovered data back into the message body. The bean then deletes the message data from the data store and removes the **claimCheck** header from the message.

8.5. SORT

Sort

The *sort* pattern is used to sort the contents of a message body, assuming that the message body contains a list of items that can be sorted.

By default, the contents of the message are sorted using a default comparator that handles numeric values or strings. You can provide your own comparator and you can specify an expression that returns the list to be sorted (the expression must be convertible to **java.util.List**).

Java DSL example

The following example generates the list of items to sort by tokenizing on the line break character:

```
from("file://inbox").sort(body().tokenize("\n")).to("bean:MyServiceBean.processLine");
```

You can pass in your own comparator as the second argument to `sort()`:

```
from("file://inbox").sort(body().tokenize("\n"), new MyReverseComparator()).to("bean:MyServiceBean.processLine");
```

XML configuration example

You can configure the same routes in Spring XML.

The following example generates the list of items to sort by tokenizing on the line break character:

```
<route>
  <from uri="file://inbox"/>
  <sort>
    <simple>body</simple>
  </sort>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>
```

And to use a custom comparator, you can reference it as a Spring bean:

```
<route>
  <from uri="file://inbox"/>
  <sort comparatorRef="myReverseComparator">
    <simple>body</simple>
  </sort>
  <beanRef ref="MyServiceBean" method="processLine"/>
</route>

<bean id="myReverseComparator" class="com.mycompany.MyReverseComparator"/>
```

Besides `<simple>`, you can supply an expression using any language you like, so long as it returns a list.

Options

The `sort` DSL command supports the following options:

Name	Default Value	Description
<code>comparatorRef</code>		Refers to a custom <code>java.util.Comparator</code> to use for sorting the message body. Camel will by default use a comparator which does a A..Z sorting.

8.6. VALIDATE

Overview

The `validate` pattern provides a convenient syntax to check whether the content of a message is valid. The `validate` DSL command takes a predicate expression as its sole argument: if the predicate evaluates to **true**, the route continues processing normally; if the predicate evaluates to **false**, a **`PredicateValidationException`** is thrown.

Java DSL example

The following route validates the body of the current message using a regular expression:

```
from("jms:queue:incoming")
  .validate(body(String.class).regex("^\\w{10}\\.\\.\\.\\d{2}\\.\\.\\.\\w{24}$"))
  .to("bean:MyServiceBean.processLine");
```

You can also validate a message header—for example:

```
from("jms:queue:incoming")
  .validate(header("bar").isGreaterThan(100))
  .to("bean:MyServiceBean.processLine");
```

And you can use `validate` with the [simple](#) expression language:

```
from("jms:queue:incoming")
  .validate(simple("${in.header.bar} == 100"))
  .to("bean:MyServiceBean.processLine");
```

XML DSL example

To use `validate` in the XML DSL, the recommended approach is to use the [simple](#) expression language:

```
<route>
  <from uri="jms:queue:incoming"/>
  <validate>
    <simple>${body} regex ^\\w{10}\\.\\.\\.\\d{2}\\.\\.\\.\\w{24}$</simple>
  </validate>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>

<bean id="myServiceBean" class="com.mycompany.MyServiceBean"/>
```

You can also validate a message header—for example:

```
<route>
  <from uri="jms:queue:incoming"/>
  <validate>
    <simple>${in.header.bar} == 100</simple>
  </validate>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>

<bean id="myServiceBean" class="com.mycompany.MyServiceBean"/>
```

CHAPTER 9. MESSAGING ENDPOINTS

Abstract

The messaging endpoint patterns describe various features and qualities of service that can be configured on an endpoint.

9.1. MESSAGING MAPPER

Overview

The *messaging mapper* pattern describes how to map domain objects to and from a canonical message format, where the message format is chosen to be as platform neutral as possible. The chosen message format should be suitable for transmission through a [message bus](#), where the message bus is the backbone for integrating a variety of different systems, some of which might not be object-oriented.

Many different approaches are possible, but not all of them fulfill the requirements of a messaging mapper. For example, an obvious way to transmit an object is to use *object serialization*, which enables you to write an object to a data stream using an unambiguous encoding (supported natively in Java). However, this is *not* a suitable approach to use for the messaging mapper pattern, however, because the serialization format is understood only by Java applications. Java object serialization creates an impedance mismatch between the original application and the other applications in the messaging system.

The requirements for a messaging mapper can be summarized as follows:

- The canonical message format used to transmit domain objects should be suitable for consumption by non-object oriented applications.
- The mapper code should be implemented separately from both the domain object code and the messaging infrastructure. Apache Camel helps fulfill this requirement by providing hooks that can be used to insert mapper code into a route.
- The mapper might need to find an effective way of dealing with certain object-oriented concepts such as inheritance, object references, and object trees. The complexity of these issues varies from application to application, but the aim of the mapper implementation must always be to create messages that can be processed effectively by non-object-oriented applications.

Finding objects to map

You can use one of the following mechanisms to find the objects to map:

- *Find a registered bean.* — For singleton objects and small numbers of objects, you could use the **CamelContext** registry to store references to beans. For example, if a bean instance is instantiated using Spring XML, it is automatically entered into the registry, where the bean is identified by the value of its **id** attribute.
- *Select objects using the JoSQL language.* — If all of the objects you want to access are already instantiated at runtime, you could use the JoSQL language to locate a specific object (or objects). For example, if you have a class, **org.apache.camel.builder.sql.Person**, with a **name** bean property and the incoming message has a **UserName** header, you could select the object whose **name** property equals the value of the **UserName** header using the following code:

```
import static org.apache.camel.builder.sql.SqlBuilder.sql;
import org.apache.camel.Expression;
...
Expression expression = sql("SELECT * FROM
org.apache.camel.builder.sql.Person where name = :UserName");
Object value = expression.evaluate(exchange);
```

Where the syntax, `:HeaderName`, is used to substitute the value of a header in a JoSQL expression.

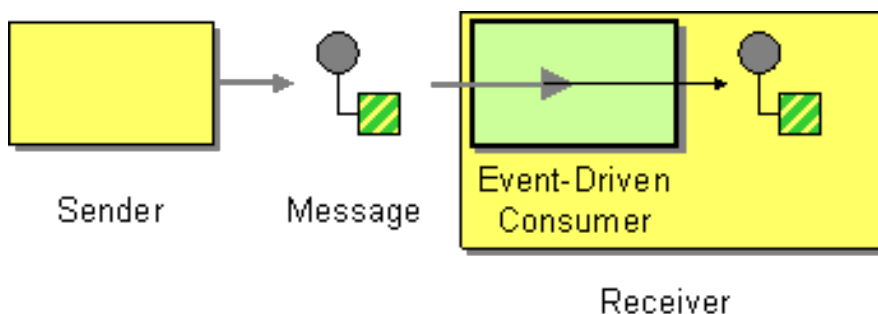
- *Dynamic* — For a more scalable solution, it might be necessary to read object data from a database. In some cases, the existing object-oriented application might already provide a finder object that can load objects from the database. In other cases, you might have to write some custom code to extract objects from a database, and in these cases the JDBC component and the SQL component might be useful.

9.2. EVENT DRIVEN CONSUMER

Overview

The *event-driven consumer* pattern, shown in [Figure 9.1, “Event Driven Consumer Pattern”](#), is a pattern for implementing the consumer endpoint in a Apache Camel component, and is only relevant to programmers who need to develop a custom component in Apache Camel. Existing components already have a consumer implementation pattern hard-wired into them.

Figure 9.1. Event Driven Consumer Pattern



Consumers that conform to this pattern provide an event method that is automatically called by the messaging channel or transport layer whenever an incoming message is received. One of the characteristics of the event-driven consumer pattern is that the consumer endpoint itself does not provide any threads to process the incoming messages. Instead, the underlying transport or messaging channel implicitly provides a processor thread when it invokes the exposed event method (which blocks for the duration of the message processing).

For more details about this implementation pattern, see [section “Consumer Patterns and Threading”](#) in [“Programming EIP Components”](#) and [chapter “Consumer Interface”](#) in [“Programming EIP Components”](#).

9.3. POLLING CONSUMER

Overview

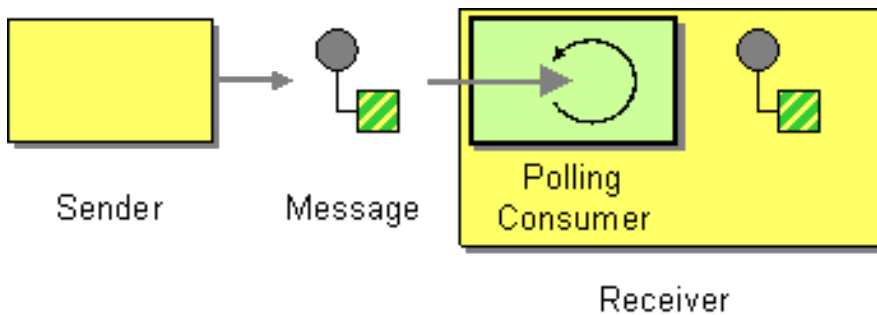
The *polling consumer* pattern, shown in [Figure 9.2, “Polling Consumer Pattern”](#), is a pattern for implementing the consumer endpoint in a Apache Camel component, so it is only relevant to programmers who need to develop a custom component in Apache Camel. Existing components already

have a consumer implementation pattern hard-wired into them.

Consumers that conform to this pattern expose polling methods, `receive()`, `receive(long timeout)`, and `receiveNoWait()` that return a new exchange object, if one is available from the monitored resource. A polling consumer implementation must provide its own thread pool to perform the polling.

For more details about this implementation pattern, see [section "Consumer Patterns and Threading" in "Programming EIP Components"](#), [chapter "Consumer Interface" in "Programming EIP Components"](#), and [section "Using the Consumer Template" in "Programming EIP Components"](#).

Figure 9.2. Polling Consumer Pattern



Scheduled poll consumer

Many of the Apache Camel consumer endpoints employ a scheduled poll pattern to receive messages at the start of a route. That is, the endpoint appears to implement an event-driven consumer interface, but internally a scheduled poll is used to monitor a resource that provides the incoming messages for the endpoint.

See [section "Implementing the Consumer Interface" in "Programming EIP Components"](#) for details of how to implement this pattern.

Quartz component

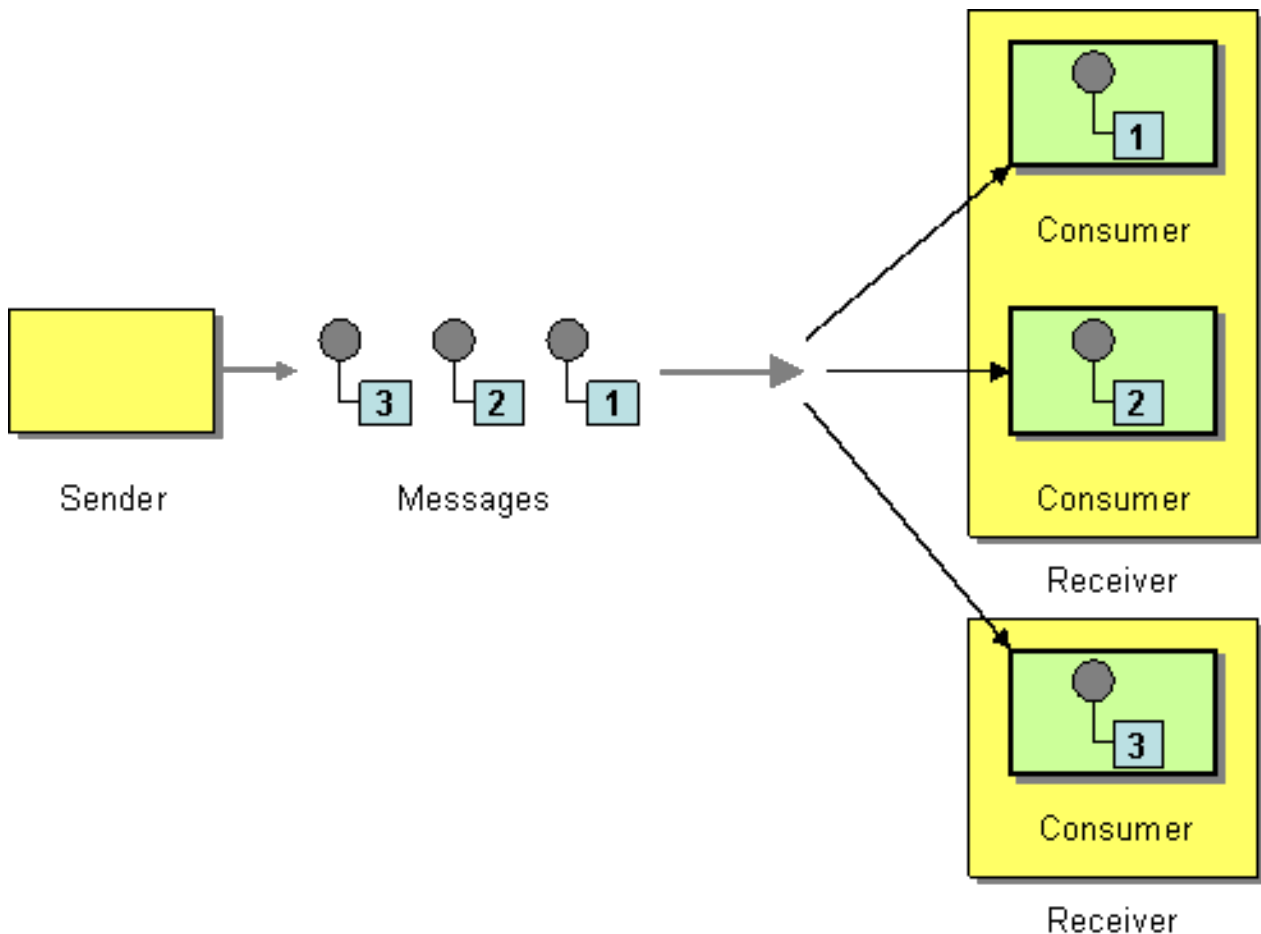
You can use the quartz component to provide scheduled delivery of messages using the *Quartz* enterprise scheduler. See [chapter "Quartz" in "EIP Component Reference"](#) and [Quartz Component](#) for details.

9.4. COMPETING CONSUMERS

Overview

The *competing consumers* pattern, shown in [Figure 9.3, "Competing Consumers Pattern"](#), enables multiple consumers to pull messages from the same queue, with the guarantee that *each message is consumed once only*. This pattern can be used to replace serial message processing with concurrent message processing (bringing a corresponding reduction in response latency).

Figure 9.3. Competing Consumers Pattern



The following components demonstrate the competing consumers pattern:

- [the section called “JMS based competing consumers”](#)
- [the section called “SEDA based competing consumers”](#)

JMS based competing consumers

A regular JMS queue implicitly guarantees that each message can only be consumed at once. Hence, a JMS queue automatically supports the competing consumers pattern. For example, you could define three competing consumers that pull messages from the JMS queue, **HighVolumeQ**, as follows:

```
from("jms:HighVolumeQ").to("cxf:bean:replica01");
from("jms:HighVolumeQ").to("cxf:bean:replica02");
from("jms:HighVolumeQ").to("cxf:bean:replica03");
```

Where the CXF (Web services) endpoints, **replica01**, **replica02**, and **replica03**, process messages from the **HighVolumeQ** queue in parallel.

Alternatively, you can set the JMS query option, **concurrentConsumers**, to create a thread pool of competing consumers. For example, the following route creates a pool of three competing threads that pick messages from the specified queue:

```
from("jms:HighVolumeQ?concurrentConsumers=3").to("cxf:bean:replica01");
```

And the **concurrentConsumers** option can also be specified in XML DSL, as follows:


```
<route>
  <from uri="jms:HighVolumeQ?concurrentConsumers=3"/>
  <to uri="cxf:bean:replica01"/>
</route>
```



NOTE

JMS topics *cannot* support the competing consumers pattern. By definition, a JMS topic is intended to send multiple copies of the same message to different consumers. Therefore, it is not compatible with the competing consumers pattern.

SEDA based competing consumers

The purpose of the SEDA component is to simplify concurrent processing by breaking the computation into stages. A SEDA endpoint essentially encapsulates an in-memory blocking queue (implemented by `java.util.concurrent.BlockingQueue`). Therefore, you can use a SEDA endpoint to break a route into stages, where each stage might use multiple threads. For example, you can define a SEDA route consisting of two stages, as follows:

```
// Stage 1: Read messages from file system.
from("file://var/messages").to("seda:fanout");

// Stage 2: Perform concurrent processing (3 threads).
from("seda:fanout").to("cxf:bean:replica01");
from("seda:fanout").to("cxf:bean:replica02");
from("seda:fanout").to("cxf:bean:replica03");
```

Where the first stage contains a single thread that consumes message from a file endpoint, `file://var/messages`, and routes them to a SEDA endpoint, `seda:fanout`. The second stage contains three threads: a thread that routes exchanges to `cxf:bean:replica01`, a thread that routes exchanges to `cxf:bean:replica02`, and a thread that routes exchanges to `cxf:bean:replica03`. These three threads compete to take exchange instances from the SEDA endpoint, which is implemented using a blocking queue. Because the blocking queue uses locking to prevent more than one thread from accessing the queue at a time, you are guaranteed that each exchange instance can only be consumed once.

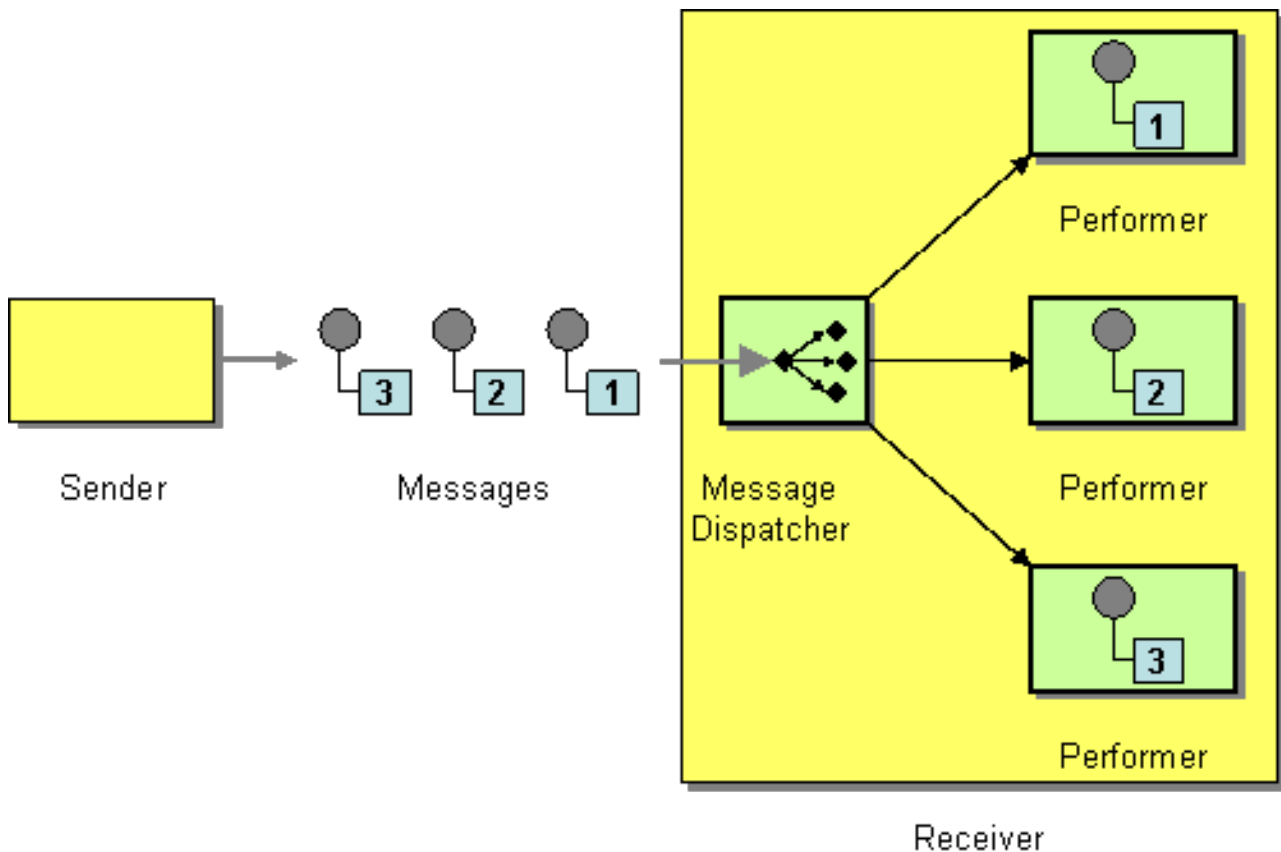
For a discussion of the differences between a SEDA endpoint and a thread pool created by `thread()`, see [chapter "SEDA" in "EIP Component Reference"](#).

9.5. MESSAGE DISPATCHER

Overview

The *message dispatcher* pattern, shown in [Figure 9.4, "Message Dispatcher Pattern"](#), is used to consume messages from a channel and then distribute them locally to *performers*, which are responsible for processing the messages. In a Apache Camel application, performers are usually represented by in-process endpoints, which are used to transfer messages to another section of the route.

Figure 9.4. Message Dispatcher Pattern



You can implement the message dispatcher pattern in Apache Camel using one of the following approaches:

- [the section called “JMS selectors”](#)
- [the section called “JMS selectors in ActiveMQ”](#)
- [the section called “Content-based router”](#)

JMS selectors

If your application consumes messages from a JMS queue, you can implement the message dispatcher pattern using *JMS selectors*. A JMS selector is a predicate expression involving JMS headers and JMS properties. If the selector evaluates to **true**, the JMS message is allowed to reach the consumer, and if the selector evaluates to **false**, the JMS message is blocked. In many respects, a JMS selector is like a [filter processor](#), but it has the additional advantage that the filtering is implemented inside the JMS provider. This means that a JMS selector can block messages before they are transmitted to the Apache Camel application. This provides a significant efficiency advantage.

In Apache Camel, you can define a JMS selector on a consumer endpoint by setting the **selector** query option on a JMS endpoint URI. For example:

```
from("jms:dispatcher?selector=CountryCode='US'").to("cxf:bean:replica01");
from("jms:dispatcher?selector=CountryCode='IE'").to("cxf:bean:replica02");
from("jms:dispatcher?selector=CountryCode='DE'").to("cxf:bean:replica03");
```

Where the predicates that appear in a selector string are based on a subset of the SQL92 conditional expression syntax (for full details, see the [JMS specification](#)). The identifiers appearing in a selector string can refer either to JMS headers or to JMS properties. For example, in the preceding routes, the

sender sets a JMS property called **CountryCode**.

If you want to add a JMS property to a message from within your Apache Camel application, you can do so by setting a message header (either on *In* message or on *Out* messages). When reading or writing to JMS endpoints, Apache Camel maps JMS headers and JMS properties to, and from, its native message headers.

Technically, the selector strings must be URL encoded according to the **application/x-www-form-urlencoded** MIME format (see the [HTML specification](#)). In practice, the **&**(ampersand) character might cause difficulties because it is used to delimit each query option in the URI. For more complex selector strings that might need to embed the **&** character, you can encode the strings using the `java.net.URLEncoder` utility class. For example:

```
from("jms:dispatcher?selector=" +
    java.net.URLEncoder.encode("CountryCode='US'", "UTF-8")).
    to("cxf:bean:replica01");
```

Where the UTF-8 encoding must be used.

JMS selectors in ActiveMQ

You can also define JMS selectors on ActiveMQ endpoints. For example:

```
from("activemq:dispatcher?
    selector=CountryCode='US'").to("cxf:bean:replica01");
from("activemq:dispatcher?
    selector=CountryCode='IE'").to("cxf:bean:replica02");
from("activemq:dispatcher?
    selector=CountryCode='DE'").to("cxf:bean:replica03");
```

For more details, see [ActiveMQ: JMS Selectors](#) and [ActiveMQ Message Properties](#).

Content-based router

The essential difference between the content-based router pattern and the message dispatcher pattern is that a content-based router dispatches messages to physically separate destinations (remote endpoints), and a message dispatcher dispatches messages locally, within the same process space. In Apache Camel, the distinction between these two patterns is determined by the target endpoint. The same router logic is used to implement both a content-based router and a message dispatcher. When the target endpoint is remote, the route defines a content-based router. When the target endpoint is in-process, the route defines a message dispatcher.

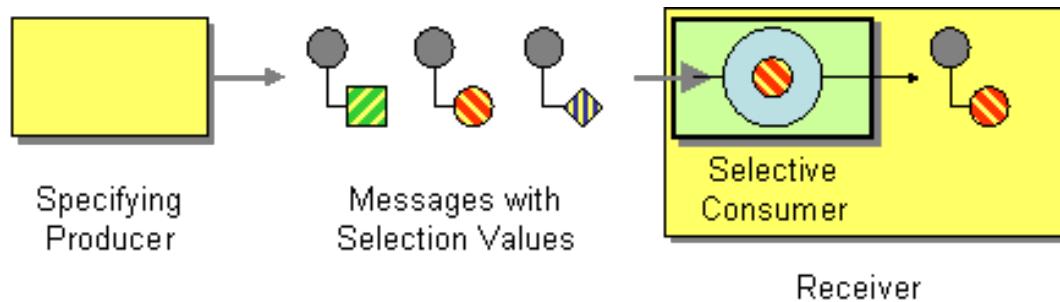
For details and examples of how to use the content-based router pattern see [Section 7.1, “Content-Based Router”](#).

9.6. SELECTIVE CONSUMER

Overview

The *selective consumer* pattern, shown in [Figure 9.5, “Selective Consumer Pattern”](#), describes a consumer that applies a filter to incoming messages, so that only messages meeting specific selection criteria are processed.

Figure 9.5. Selective Consumer Pattern



You can implement the selective consumer pattern in Apache Camel using one of the following approaches:

- [the section called “JMS selector”](#)
- [the section called “JMS selector in ActiveMQ”](#)
- [the section called “Message filter”](#)

JMS selector

A JMS selector is a predicate expression involving JMS headers and JMS properties. If the selector evaluates to **true**, the JMS message is allowed to reach the consumer, and if the selector evaluates to **false**, the JMS message is blocked. For example, to consume messages from the queue, **selective**, and select only those messages whose country code property is equal to **US**, you can use the following Java DSL route:

```
from("jms:selective?selector=" +
    java.net.URLEncoder.encode("CountryCode='US'", "UTF-8")).
    to("cxf:bean:replica01");
```

Where the selector string, **CountryCode='US'**, must be URL encoded (using UTF-8 characters) to avoid trouble with parsing the query options. This example presumes that the JMS property, **CountryCode**, is set by the sender. For more details about JMS selectors, see [the section called “JMS selectors”](#).



NOTE

If a selector is applied to a JMS queue, messages that are not selected remain on the queue and are potentially available to other consumers attached to the same queue.

JMS selector in ActiveMQ

You can also define JMS selectors on ActiveMQ endpoints. For example:

```
from("acivemq:selective?selector=" +
    java.net.URLEncoder.encode("CountryCode='US'", "UTF-8")).
    to("cxf:bean:replica01");
```

For more details, see [ActiveMQ: JMS Selectors](#) and [ActiveMQ Message Properties](#).

Message filter

If it is not possible to set a selector on the consumer endpoint, you can insert a filter processor into your route instead. For example, you can define a selective consumer that processes only messages with a US country code using Java DSL, as follows:

```
from("seda:a").filter(header("CountryCode").isEqualTo("US")).process(myProcessor);
```

The same route can be defined using XML configuration, as follows:

```
<camelContext id="buildCustomProcessorWithFilter"
xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$CountryCode = 'US'</xpath>
      <process ref="#myProcessor"/>
    </filter>
  </route>
</camelContext>
```

For more information about the Apache Camel filter processor, see [Message Filter](#).



WARNING

Be careful about using a message filter to select messages from a JMS *queue*. When using a filter processor, blocked messages are simply discarded. Hence, if the messages are consumed from a queue (which allows each message to be consumed only once—see [Section 9.4, “Competing Consumers”](#)), then blocked messages are not processed at all. This might not be the behavior you want.

9.7. DURABLE SUBSCRIBER

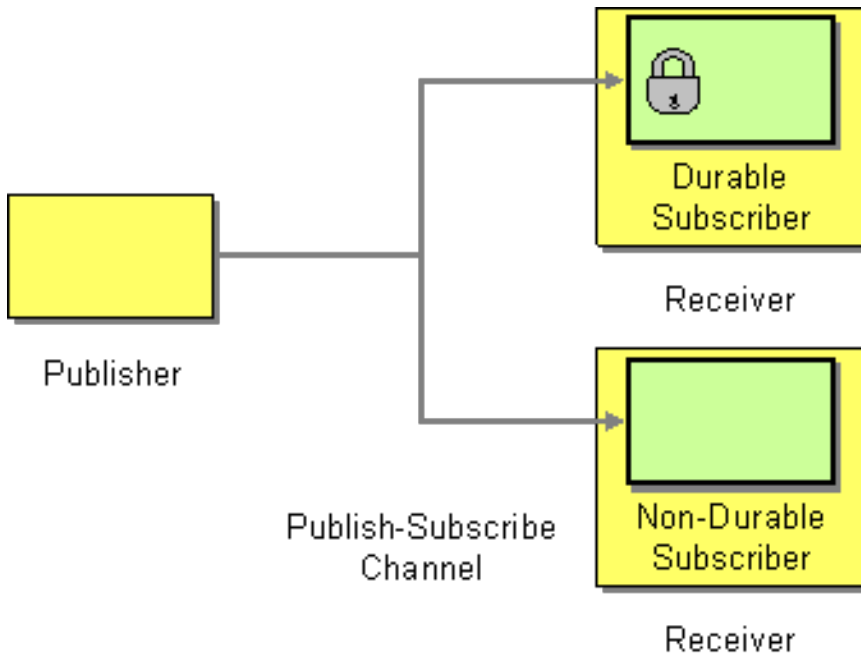
Overview

A *durable subscriber*, as shown in [Figure 9.6, “Durable Subscriber Pattern”](#), is a consumer that wants to receive all of the messages sent over a particular [publish-subscribe](#) channel, including messages sent while the consumer is disconnected from the messaging system. This requires the messaging system to store messages for later replay to the disconnected consumer. There also has to be a mechanism for a consumer to indicate that it wants to establish a durable subscription. Generally, a publish-subscribe channel (or topic) can have both durable and non-durable subscribers, which behave as follows:

- **non-durable subscriber**—Can have two states: *connected* and *disconnected*. While a non-durable subscriber is connected to a topic, it receives all of the topic's messages in real time. However, a non-durable subscriber never receives messages sent to the topic while the subscriber is disconnected.
- **durable subscriber**—Can have two states: *connected* and *inactive*. The inactive state means that the durable subscriber is disconnected from the topic, but wants to receive the messages that arrive in the interim. When the durable subscriber reconnects to the topic, it receives a

replay of all the messages sent while it was inactive.

Figure 9.6. Durable Subscriber Pattern



JMS durable subscriber

The JMS component implements the durable subscriber pattern. In order to set up a durable subscription on a JMS endpoint, you must specify a *client ID*, which identifies this particular connection, and a *durable subscription name*, which identifies the durable subscriber. For example, the following route sets up a durable subscription to the JMS topic, **news**, with a client ID of **conn01** and a durable subscription name of **John.Doe**:

```
from("jms:topic:news?clientId=conn01&durableSubscriptionName=John.Doe").
    to("cxf:bean:newsprocessor");
```

You can also set up a durable subscription using the ActiveMQ endpoint:

```
from("activemq:topic:news?
clientId=conn01&durableSubscriptionName=John.Doe").
    to("cxf:bean:newsprocessor");
```

If you want to process the incoming messages concurrently, you can use a SEDA endpoint to fan out the route into multiple, parallel segments, as follows:

```
from("jms:topic:news?clientId=conn01&durableSubscriptionName=John.Doe").
    to("seda:fanout");

from("seda:fanout").to("cxf:bean:newsproc01");
from("seda:fanout").to("cxf:bean:newsproc02");
from("seda:fanout").to("cxf:bean:newsproc03");
```

Where each message is processed only once, because the SEDA component supports the [competing consumers](#) pattern.

Alternative example

Another alternative is to combine the [Message Dispatcher](#) or [Content-Based Router](#) with [File component](#) or [JPA component](#) components for durable subscribers then something like [SEDA component](#) for non-durable.

Here is a simple example of creating durable subscribers to a [chapter "JMS" in "EIP Component Reference"](#) topic

Using the [Fluent Builders](#)

```
from("direct:start").to("activemq:topic:foo");

from("activemq:topic:foo?
clientId=1&durableSubscriptionName=bar1").to("mock:result1");

from("activemq:topic:foo?
clientId=2&durableSubscriptionName=bar2").to("mock:result2");
```

Using the [Spring XML Extensions](#)

```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:topic:foo"/>
</route>

<route>
  <from uri="activemq:topic:foo?
clientId=1&durableSubscriptionName=bar1"/>
  <to uri="mock:result1"/>
</route>

<route>
  <from uri="activemq:topic:foo?
clientId=2&durableSubscriptionName=bar2"/>
  <to uri="mock:result2"/>
</route>
```

Here is another example of [JMS](#) durable subscribers, but this time using [virtual topics](#) (recommended by AMQ over durable subscriptions)

Using the [Fluent Builders](#)

```
from("direct:start").to("activemq:topic:VirtualTopic.foo");

from("activemq:queue:Consumer.1.VirtualTopic.foo").to("mock:result1");

from("activemq:queue:Consumer.2.VirtualTopic.foo").to("mock:result2");
```

Using the [Spring XML Extensions](#)

```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:topic:VirtualTopic.foo"/>
</route>

<route>
```

```

        <from uri="activemq:queue:Consumer.1.VirtualTopic.foo"/>
        <to uri="mock:result1"/>
    </route>

    <route>
        <from uri="activemq:queue:Consumer.2.VirtualTopic.foo"/>
        <to uri="mock:result2"/>
    </route>

```

9.8. IDEMPOTENT CONSUMER

Overview

The *idempotent consumer* pattern is used to filter out duplicate messages. For example, consider a scenario where the connection between a messaging system and a consumer endpoint is abruptly lost due to some fault in the system. If the messaging system was in the middle of transmitting a message, it might be unclear whether or not the consumer received the last message. To improve delivery reliability, the messaging system might decide to redeliver such messages as soon as the connection is re-established. Unfortunately, this entails the risk that the consumer might receive duplicate messages and, in some cases, the effect of duplicating a message may have undesirable consequences (such as debiting a sum of money twice from your account). In this scenario, an idempotent consumer could be used to weed out undesired duplicates from the message stream.

Camel provides the following Idempotent Consumer implementations:

- **MemoryIdempotentRepository**
- [File](#)
- [HazelcastIdempotentRepository](#)
- [JdbcMessageIdRepository](#)
- [JpaMessageIdRepository](#)

Idempotent consumer with in-memory cache

In Apache Camel, the idempotent consumer pattern is implemented by the `idempotentConsumer()` processor, which takes two arguments:

- **messageIdExpression** — An expression that returns a message ID string for the current message.
- **messageIdRepository** — A reference to a message ID repository, which stores the IDs of all the messages received.

As each message comes in, the idempotent consumer processor looks up the current message ID in the repository to see if this message has been seen before. If yes, the message is discarded; if no, the message is allowed to pass and its ID is added to the repository.

The code shown in [Example 9.1, “Filtering Duplicate Messages with an In-memory Cache”](#) uses the `TransactionID` header to filter out duplicates.

Example 9.1. Filtering Duplicate Messages with an In-memory Cache


```

import static
org.apache.camel.processor.idempotent.MemoryMessageIdRepository.memoryMe
ssageIdRepository;
...
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a")
            .idempotentConsumer(
                header("TransactionID"),
                memoryMessageIdRepository(200)
            ).to("seda:b");
    }
};

```

Where the call to **memoryMessageIdRepository(200)** creates an in-memory cache that can hold up to 200 message IDs.

You can also define an idempotent consumer using XML configuration. For example, you can define the preceding route in XML, as follows:

```

<camelContext id="buildIdempotentConsumer"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <idempotentConsumer messageIdRepositoryRef="MsgIDRepos">
            <simple>header.TransactionID</simple>
            <to uri="seda:b"/>
        </idempotentConsumer>
    </route>
</camelContext>

<bean id="MsgIDRepos"
class="org.apache.camel.processor.idempotent.MemoryMessageIdRepository">
    <!-- Specify the in-memory cache size. -->
    <constructor-arg type="int" value="200"/>
</bean>

```

Idempotent consumer with JPA repository

The in-memory cache suffers from the disadvantages of easily running out of memory and not working in a clustered environment. To overcome these disadvantages, you can use a Java Persistent API (JPA) based repository instead. The JPA message ID repository uses an object-oriented database to store the message IDs. For example, you can define a route that uses a JPA repository for the idempotent consumer, as follows:

```

import org.springframework.orm.jpa.JpaTemplate;

import org.apache.camel.spring.SpringRouteBuilder;
import static
org.apache.camel.processor.idempotent.jpa.JpaMessageIdRepository.jpaMessag
eIdRepository;
...
RouteBuilder builder = new SpringRouteBuilder() {

```

```

    public void configure() {
        from("seda:a").idempotentConsumer(
            header("TransactionID"),
            jpaMessageIdRepository(bean(JpaTemplate.class),
"myProcessorName")
            ).to("seda:b");
    }
};

```

The JPA message ID repository is initialized with two arguments:

- **JpaTemplate** instance—Provides the handle for the JPA database.
- processor name—Identifies the current idempotent consumer processor.

The **SpringRouteBuilder.bean()** method is a shortcut that references a bean defined in the Spring XML file. The **JpaTemplate** bean provides a handle to the underlying JPA database. See the JPA documentation for details of how to configure this bean.

For more details about setting up a JPA repository, see [JPA Component](#) documentation, the [Spring JPA](#) documentation, and the sample code in the [Camel JPA unit test](#).

Spring XML example

The following example uses the **myMessageId** header to filter out duplicates:

```

<!-- repository for the idempotent consumer -->
<bean id="myRepo"
class="org.apache.camel.processor.idempotent.MemoryIdempotentRepository"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <idempotentConsumer messageIdRepositoryRef="myRepo">
            <!-- use the messageId header as key for identifying duplicate
messages -->
            <header>messageId</header>
            <!-- if not a duplicate send it to this mock endpoint -->
            <to uri="mock:result"/>
        </idempotentConsumer>
    </route>
</camelContext>

```

Idempotent consumer with JDBC repository

A JDBC repository is also supported for storing message IDs in the idempotent consumer pattern. The implementation of the JDBC repository is provided by the SQL component, so if you are using the Maven build system, add a dependency on the **camel-sql** artifact.

You can use the **SingleConnectionDataSource** JDBC wrapper class from the Spring persistence API in order to instantiate the connection to a SQL database. For example, to instantiate a JDBC connection to a [HyperSQL](#) database instance, you could define the following JDBC data source:

```

<bean id="dataSource"

```

```

class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:mem:camel_jdbc"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>

```



NOTE

The preceding JDBC data source uses the HyperSQL **mem** protocol, which creates a memory-only database instance. This is a toy implementation of the HyperSQL database which is *not* actually persistent.

Using the preceding data source, you can define an idempotent consumer pattern that uses the JDBC message ID repository, as follows:

```

<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>

```

How to handle duplicate messages in the route

Available as of Camel 2.8

You can now set the **skipDuplicate** option to **false** which instructs the idempotent consumer to route duplicate messages as well. However the duplicate message has been marked as duplicate by having a property on the [Exchange](#) set to true. We can leverage this fact by using a [Content-Based Router](#) or [Message Filter](#) to detect this and handle duplicate messages.

For example in the following example we use the [Message Filter](#) to send the message to a duplicate endpoint, and then stop continue routing that message.

```

from("direct:start")
  // instruct idempotent consumer to not skip duplicates as we will

```

```

filter then our self

.idempotentConsumer(header("messageId")).messageIdRepository(repo).skipDuplicate(
false)
    .filter(property(Exchange.DUPLICATE_MESSAGE).isEqualTo(true))
    // filter out duplicate messages by sending them to someplace
else and then stop
    .to("mock:duplicate")
    .stop()
.end()
// and here we process only new messages (no duplicates)
.to("mock:result");

```

The sample example in XML DSL would be:

```

<!-- idempotent repository, just use a memory based for testing -->
<bean id="myRepo"
class="org.apache.camel.processor.idempotent.MemoryIdempotentRepository"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <!-- we do not want to skip any duplicate messages -->
        <idempotentConsumer messageIdRepositoryRef="myRepo"
skipDuplicate="false">
            <!-- use the messageId header as key for identifying
duplicate messages -->
            <header>messageId</header>
            <!-- we will to handle duplicate messages using a filter -->
            <filter>
                <!-- the filter will only react on duplicate messages,
if this property is set on the Exchange -->
                <property>CamelDuplicateMessage</property>
                <!-- and send the message to this mock, due its part of
an unit test -->
                <!-- but you can of course do anything as its part of the
route -->
                <to uri="mock:duplicate"/>
                <!-- and then stop -->
                <stop/>
            </filter>
            <!-- here we route only new messages -->
            <to uri="mock:result"/>
        </idempotentConsumer>
    </route>
</camelContext>

```

How to handle duplicate message in a clustered environment with a data grid

If you have running Camel in a clustered environment, a in memory idempotent repository doesn't work (see above). You can setup either a central database or use the idempotent consumer implementation based on the [Hazelcast](#) data grid. Hazelcast finds the nodes over multicast (which is default - configure Hazelcast for tcp-ip) and creates automatically a map based repository:

```
HazelcastIdempotentRepository idempotentRepo = new
```

```
HazelcastIdempotentRepository("myrepo");

from("direct:in").idempotentConsumer(header("messageId"),
idempotentRepo).to("mock:out");
```

You have to define how long the repository should hold each message id (default is to delete it never). To avoid that you run out of memory you should create an eviction strategy based on the [Hazelcast configuration](#). For additional information see [camel-hazelcast](#).

See this [little tutorial](#), how setup such an idempotent repository on two cluster nodes using Apache Karaf.

Options

The Idempotent Consumer has the following options:

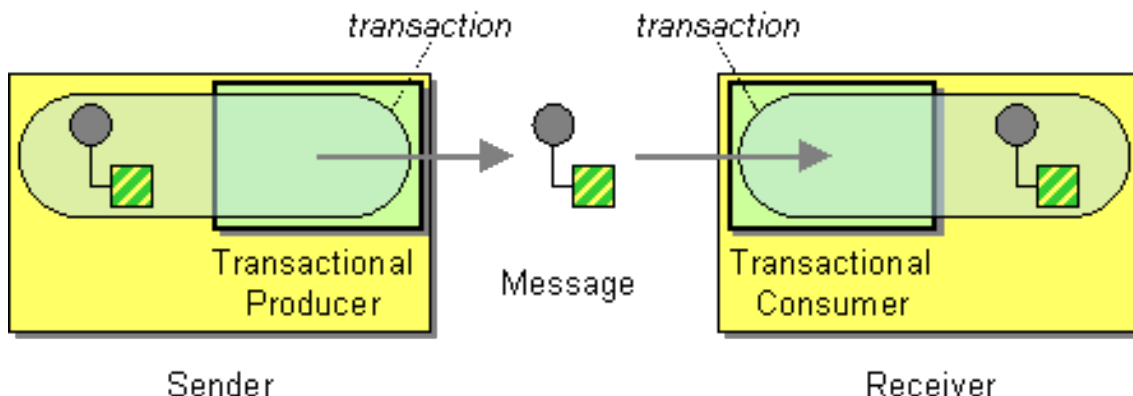
Option	Default	Description
eager	true	Camel 2.0: Eager controls whether Camel adds the message to the repository before or after the exchange has been processed. If enabled before then Camel will be able to detect duplicate messages even when messages are currently in progress. By disabling Camel will only detect duplicates when a message has successfully been processed.
messageIdRepositoryRef	null	A reference to a IdempotentRepository to lookup in the registry. This option is mandatory when using XML DSL.
skipDuplicate	true	Camel 2.8: Sets whether to skip duplicate messages. If set to false then the message will be continued. However the Exchange has been marked as a duplicate by having the Exchange.DUPLICATE_MESSAGE exchange property set to a Boolean.TRUE value.

9.9. TRANSACTIONAL CLIENT

Overview

The *transactional client* pattern, shown in [Figure 9.7, “Transactional Client Pattern”](#), refers to messaging endpoints that can participate in a transaction. Apache Camel supports transactions using [Spring transaction management](#).

Figure 9.7. Transactional Client Pattern



Transaction oriented endpoints

Not all Apache Camel endpoints support transactions. Those that do are called *transaction oriented endpoints* (or TOEs). For example, both the JMS component and the ActiveMQ component support transactions.

To enable transactions on a component, you must perform the appropriate initialization before adding the component to the `CamelContext`. This entails writing code to initialize your transactional components explicitly.

References

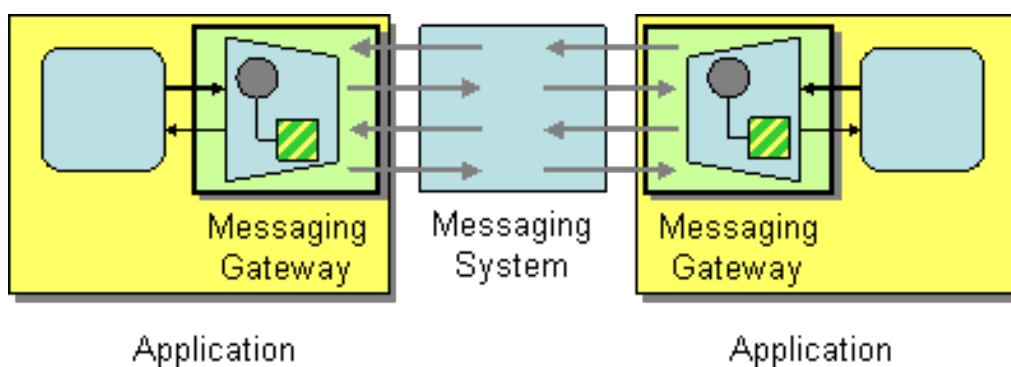
The details of configuring transactions in Apache Camel are beyond the scope of this guide. For full details of how to use transactions, see the Apache Camel *Transaction Guide*.

9.10. MESSAGING GATEWAY

Overview

The *messaging gateway* pattern, shown in [Figure 9.8, “Messaging Gateway Pattern”](#), describes an approach to integrating with a messaging system, where the messaging system's API remains hidden from the programmer at the application level. One of the more common example is when you want to translate synchronous method calls into request/reply message exchanges, without the programmer being aware of this.

Figure 9.8. Messaging Gateway Pattern



The following Apache Camel components provide this kind of integration with the messaging system:

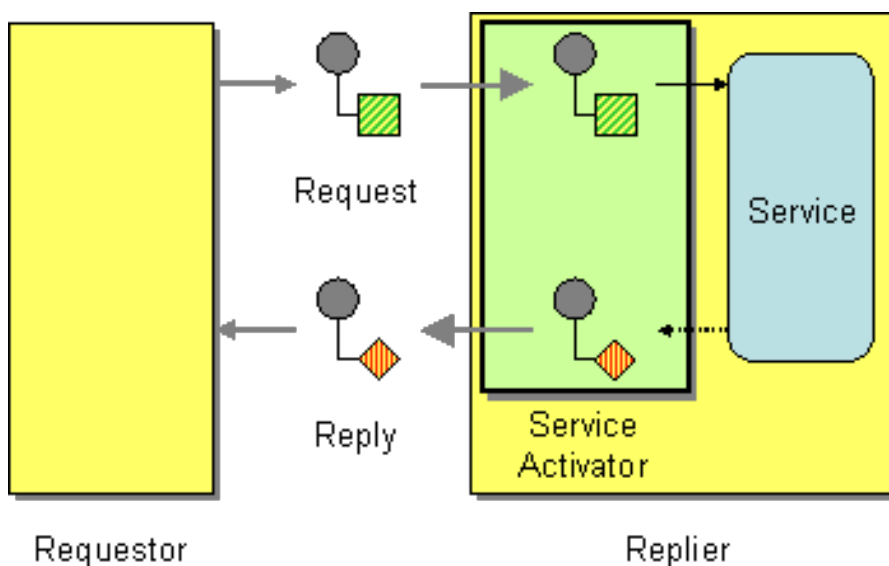
- [chapter "CXF" in "EIP Component Reference"](#)
- [chapter "Bean" in "EIP Component Reference"](#)

9.11. SERVICE ACTIVATOR

Overview

The *service activator* pattern, shown in [Figure 9.9, "Service Activator Pattern"](#), describes the scenario where a service's operations are invoked in response to an incoming request message. The service activator identifies which operation to call and extracts the data to use as the operation's parameters. Finally, the service activator invokes an operation using the data extracted from the message. The operation invocation can be either oneway (request only) or two-way (request/reply).

Figure 9.9. Service Activator Pattern



In many respects, a service activator resembles a conventional remote procedure call (RPC), where operation invocations are encoded as messages. The main difference is that a service activator needs to be more flexible. An RPC framework standardizes the request and reply message encodings (for example, Web service operations are encoded as SOAP messages), whereas a service activator typically needs to improvise the mapping between the messaging system and the service's operations.

Bean integration

The main mechanism that Apache Camel provides to support the service activator pattern is *bean integration*. [Bean integration](#) provides a general framework for mapping incoming messages to method invocations on Java objects. For example, the Java fluent DSL provides the processors `bean()` and `beanRef()` that you can insert into a route to invoke methods on a registered Java bean. The detailed mapping of message data to Java method parameters is determined by the *bean binding*, which can be implemented by adding annotations to the bean class.

For example, consider the following route which calls the Java method, `BankBean.getUserAccBalance()`, to service requests incoming on a JMS/ActiveMQ queue:

```
from("activemq:BalanceQueries")
    .setProperty("userid",
xpath("/Account/BalanceQuery/UserID").stringResult())
    .beanRef("bankBean", "getUserAccBalance")
    .to("velocity:file:src/scripts/acc_balance.vm")
    .to("activemq:BalanceResults");
```

The messages pulled from the ActiveMQ endpoint, `activemq:BalanceQueries`, have a simple XML format that provides the user ID of a bank account. For example:

```
<?xml version='1.0' encoding='UTF-8'?>
<Account>
  <BalanceQuery>
    <UserID>James.Strachan</UserID>
  </BalanceQuery>
</Account>
```

The first processor in the route, `setProperty()`, extracts the user ID from the *In* message and stores it in the `userid` exchange property. This is preferable to storing it in a header, because the *In* headers are not available after invoking the bean.

The service activation step is performed by the `beanRef()` processor, which binds the incoming message to the `getUserAccBalance()` method on the Java object identified by the `bankBean` bean ID. The following code shows a sample implementation of the `BankBean` class:

```
package tutorial;

import org.apache.camel.language.XPath;

public class BankBean {
    public int getUserAccBalance(@XPath("/Account/BalanceQuery/UserID")
String user) {
        if (user.equals("James.Strachan")) {
            return 1200;
        }
        else {
            return 0;
        }
    }
}
```

Where the binding of message data to method parameter is enabled by the `@XPath` annotation, which injects the content of the `UserID` XML element into the `user` method parameter. On completion of the call, the return value is inserted into the body of the *Out* message which is then copied into the *In*

message for the next step in the route. In order for the bean to be accessible to the `beanRef()` processor, you must instantiate an instance in Spring XML. For example, you can add the following lines to the `META-INF/spring/camel-context.xml` configuration file to instantiate the bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <bean id="bankBean" class="tutorial.BankBean"/>
</beans>
```

Where the bean ID, `bankBean`, identifies this bean instance in the registry.

The output of the bean invocation is injected into a Velocity template, to produce a properly formatted result message. The Velocity endpoint, `velocity:file:src/scripts/acc_balance.vm`, specifies the location of a velocity script with the following contents:

```
<?xml version='1.0' encoding='UTF-8'?>
<Account>
  <BalanceResult>
    <UserID>${exchange.getProperty("userid")}</UserID>
    <Balance>${body}</Balance>
  </BalanceResult>
</Account>
```

The exchange instance is available as the Velocity variable, `exchange`, which enables you to retrieve the `userid` exchange property, using `${exchange.getProperty("userid")}`. The body of the current `In` message, `${body}`, contains the result of the `getUserAccBalance()` method invocation.

CHAPTER 10. SYSTEM MANAGEMENT

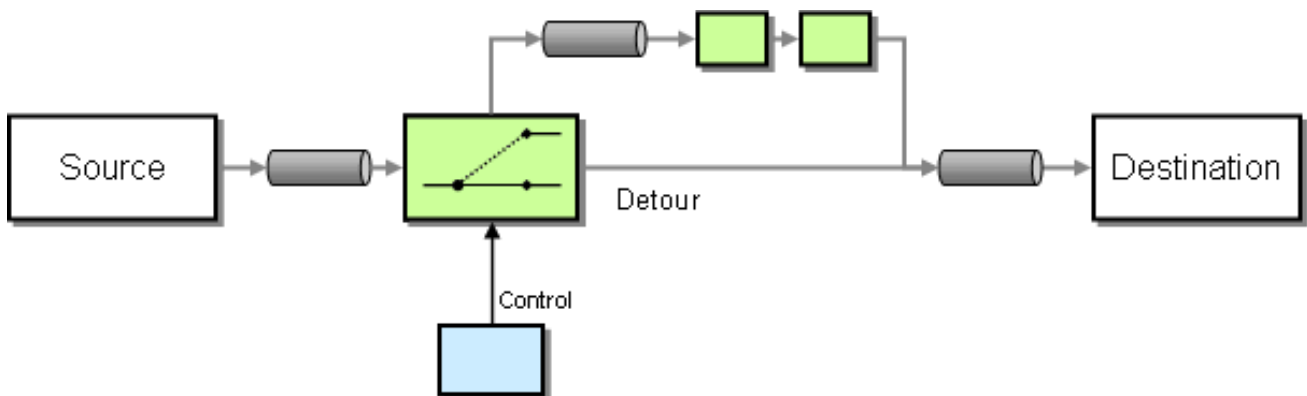
Abstract

The system management patterns describe how to monitor, test, and administer a messaging system.

10.1. DETOUR

Detour

The [Detour](#) from the [Introducing Enterprise Integration Patterns](#) allows you to send messages through additional steps if a control condition is met. It can be useful for turning on extra validation, testing, debugging code when needed.



Example

In this example we essentially have a route like `from("direct:start").to("mock:result")` with a conditional detour to the `mock:detour` endpoint in the middle of the route..

```
from("direct:start").choice()
    .when().method("controlBean", "isDetour").to("mock:detour").end()
    .to("mock:result");
```

Using the [Spring XML Extensions](#)

```
<route>
  <from uri="direct:start"/>
  <choice>
    <when>
      <method bean="controlBean" method="isDetour"/>
    <to uri="mock:detour"/>
    </when>
  </choice>
  <to uri="mock:result"/>
</split>
</route>
```

whether the detour is turned on or off is decided by the **ControlBean**. So, when the detour is on the message is routed to **mock:detour** and then **mock:result**. When the detour is off, the message is routed to **mock:result**.

For full details, check the example source here:

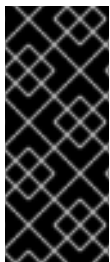
[camel-core/src/test/java/org/apache/camel/processor/DetourTest.java](#)

10.2. LOGEIP

Overview

Apache Camel provides several ways to perform logging in a route:

- Using the **log** DSL command.
- Using the **Log** component, which can log the message content.
- Using the Tracer, which traces message flow.
- Using a **Processor** or a **Bean** endpoint to perform logging in Java.



DIFFERENCE BETWEEN THE LOG DSL COMMAND AND THE LOG COMPONENT

The **log** DSL is much lighter and meant for logging human logs such as **Starting to do . . .** It can only log a message based on the **Simple** language. In contrast, the **Log** component is a fully featured logging component. The **Log** component is capable of logging the message itself and you have many URI options to control the logging.

Java DSL example

Since **Apache Camel 2.2**, you can use the **log** DSL command to construct a log message at run time using the Simple expression language. For example, you can create a log message within a route, as follows:

```
from("direct:start").log("Processing ${id}").to("bean:foo");
```

This route constructs a **String** format message at run time. The log message will be logged at **INFO** level, using the route ID as the log name. By default, routes are named consecutively, **route-1**, **route-2** and so on. But you can use the DSL command, **routeId("myCoolRoute")**, to specify a custom route ID.

The log DSL also provides variants that enable you to set the logging level and the log name explicitly. For example, to set the logging level explicitly to **LogLevel.DEBUG**, you can invoke the log DSL as follows:

has overloaded methods to set the logging level and/or name as well.

```
from("direct:start").log(LogLevel.DEBUG, "Processing  
${id}").to("bean:foo");
```

To set the log name to **fileRoute**, you can invoke the log DSL as follows:

```
from("file://target/files").log(LoggingLevel.DEBUG, "fileRoute",
"Processing file ${file:name}").to("bean:foo");
```

XML DSL example

In XML DSL, the log DSL is represented by the **log** element and the log message is specified by setting the **message** attribute to a Simple expression, as follows:

```
<route id="foo">
  <from uri="direct:foo"/>
  <log message="Got ${body}"/>
  <to uri="mock:foo"/>
</route>
```

The **log** element supports the **message**, **loggingLevel** and **logName** attributes. For example:

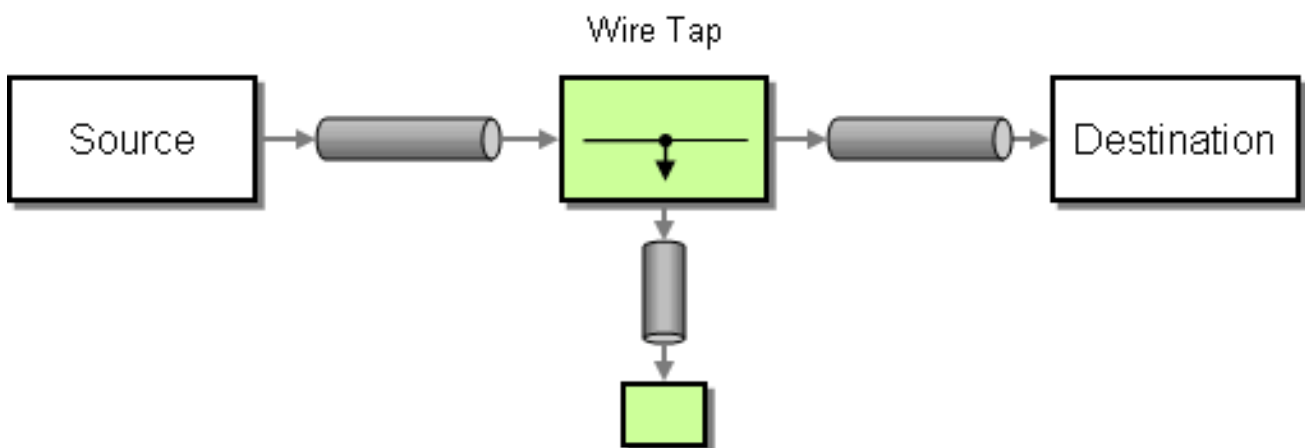
```
<route id="baz">
  <from uri="direct:baz"/>
  <log message="Me Got ${body}" loggingLevel="FATAL" logName="cool"/>
  <to uri="mock:baz"/>
</route>
```

10.3. WIRE TAP

Wire Tap

The *wire tap* pattern, as shown in [Figure 10.1, “Wire Tap Pattern”](#), enables you to route a copy of the message to a separate tap location, while the original message is forwarded to the ultimate destination.

Figure 10.1. Wire Tap Pattern



STREAMS

If you [Wire Tap](#) a stream message body, you should consider enabling [Stream Caching](#) to ensure the message body can be re-read. See more details at [Stream Caching](#)

WireTap node

Apache Camel 2.0 introduces the **wireTap** node for doing wire taps. The **wireTap** node copies the original exchange to a tapped exchange, whose exchange pattern is set to *InOnly*, because the tapped exchange should be propagated in a *oneway* style. The tapped exchange is processed in a separate thread, so that it can run concurrently with the main route.

The **wireTap** supports two different approaches to tapping an exchange:

- Tap a copy of the original exchange.
- Tap a new exchange instance, enabling you to customize the tapped exchange.

Tap a copy of the original exchange

Using the Java DSL:

```
from("direct:start")
    .to("log:foo")
    .wireTap("direct:tap")
    .to("mock:result");
```

Using Spring XML extensions:

```
<route>
  <from uri="direct:start"/>
  <to uri="log:foo"/>
  <wireTap uri="direct:tap"/>
  <to uri="mock:result"/>
</route>
```

Tap and modify a copy of the original exchange

Using the Java DSL, Apache Camel supports using either a processor or an expression to modify a copy of the original exchange. Using a processor gives you full power over how the exchange is populated, because you can set properties, headers and so on. The expression approach can only be used to modify the *In* message body.

For example, to modify a copy of the original exchange using the *processor* approach:

```
from("direct:start")
    .wireTap("direct:foo", new Processor() {
        public void process(Exchange exchange) throws Exception {
            exchange.getIn().setHeader("foo", "bar");
        }
    }).to("mock:result");

from("direct:foo").to("mock:foo");
```

And to modify a copy of the original exchange using the *expression* approach:

```
from("direct:start")
    .wireTap("direct:foo", constant("Bye World"))
    .to("mock:result");
```

```
from("direct:foo").to("mock:foo");
```

Using the Spring XML extensions, you can modify a copy of the original exchange using the *processor* approach, where the **processorRef** attribute references a spring bean with the **myProcessor** ID:

```
<route>
  <from uri="direct:start2"/>
  <wireTap uri="direct:foo" processorRef="myProcessor"/>
  <to uri="mock:result"/>
</route>
```

And to modify a copy of the original exchange using the *expression* approach:

```
<route>
  <from uri="direct:start"/>
  <wireTap uri="direct:foo">
    <body><constant>Bye World</constant></body>
  </wireTap>
  <to uri="mock:result"/>
</route>
```

Tap a new exchange instance

You can define a wiretap with a new exchange instance by setting the copy flag to **false** (the default is **true**). In this case, an initially empty exchange is created for the wiretap.

For example, to create a new exchange instance using the *processor* approach:

```
from("direct:start")
  .wireTap("direct:foo", false, new Processor() {
    public void process(Exchange exchange) throws Exception {
      exchange.getIn().setBody("Bye World");
      exchange.getIn().setHeader("foo", "bar");
    }
  }).to("mock:result");

from("direct:foo").to("mock:foo");
```

Where the second **wireTap** argument sets the copy flag to **false**, indicating that the original exchange is *not* copied and an empty exchange is created instead.

To create a new exchange instance using the *expression* approach:

```
from("direct:start")
  .wireTap("direct:foo", false, constant("Bye World"))
  .to("mock:result");

from("direct:foo").to("mock:foo");
```

Using the Spring XML extensions, you can indicate that a new exchange is to be created by setting the **wireTap** element's **copy** attribute to **false**.

To create a new exchange instance using the *processor* approach, where the **processorRef** attribute references a spring bean with the **myProcessor** ID, as follows:

```
<route>
  <from uri="direct:start2"/>
  <wireTap uri="direct:foo" processorRef="myProcessor" copy="false"/>
  <to uri="mock:result"/>
</route>
```

And to create a new exchange instance using the *expression* approach:

```
<route>
  <from uri="direct:start"/>
  <wireTap uri="direct:foo" copy="false">
    <body><constant>Bye World</constant></body>
  </wireTap>
  <to uri="mock:result"/>
</route>
```

Sending a new **Exchange** and set headers in DSL

Available as of Camel 2.8

If you send a new messages using the **Wire Tap** then you could only set the message body using an **Expression** from the DSL. If you also need to set new headers you would have to use a **Processor** for that. So in Camel 2.8 onwards we have improved this situation so you can now set headers as well in the DSL.

The following example sends a new message which has

- "Bye World" as message body
- a header with key "id" with the value 123
- a header with key "date" which has current date as value

Java DSL

```
from("direct:start")
  // tap a new message and send it to direct:tap
  // the new message should be Bye World with 2 headers
  .wireTap("direct:tap")
    // create the new tap message body and headers
    .newExchangeBody(constant("Bye World"))
    .newExchangeHeader("id", constant(123))
    .newExchangeHeader("date", simple("${date:now:yyyyMMdd}"))
  .end()
  // here we continue routing the original messages
  .to("mock:result");

// this is the tapped route
from("direct:tap")
  .to("mock:tap");
```

XML DSL

The XML DSL is slightly different than Java DSL as how you configure the message body and headers. In XML you use `<body>` and `<setHeader>` as shown:

```
<route>
  <from uri="direct:start"/>
  <!-- tap a new message and send it to direct:tap -->
  <!-- the new message should be Bye World with 2 headers -->
  <wireTap uri="direct:tap">
    <!-- create the new tap message body and headers -->
    <body><constant>Bye World</constant></body>
    <setHeader headerName="id"><constant>123</constant></setHeader>
    <setHeader headerName="date"><simple>${date:now:yyyyMMdd}
  </simple></setHeader>
  </wireTap>
  <!-- here we continue routing the original message -->
  <to uri="mock:result"/>
</route>
```

Using onPrepare to execute custom logic when preparing messages

Available as of Camel 2.8

For details, see [Multicast](#).

Options

The `wireTap` DSL command supports the following options:

Name	Default Value	Description
<code>uri</code>		The endpoint uri where to send the wire tapped message. You should use either uri or ref .
<code>ref</code>		Refers to the endpoint where to send the wire tapped message. You should use either uri or ref .
<code>executorServiceRef</code>		Refers to a custom Thread Pool to be used when processing the wire tapped messages. If not set then Camel uses a default thread pool.
<code>processorRef</code>		Refers to a custom Processor to be used for creating a new message (eg the send a new message mode). See below.

copy	true	Camel 2.3: Should a copy of the Exchange to used when wire tapping the message.
onPrepareRef		Camel 2.8: Refers to a custom Processor to prepare the copy of the Exchange to be wire tapped. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.

APPENDIX A. MIGRATING FROM SERVICEMIX EIP

Abstract

If you are currently an Apache ServiceMix 3.x user, you might already have implemented some Enterprise Integration Patterns using the *ServiceMix EIP module*. It is recommended that you migrate these legacy patterns to Apache Camel, which has more extensive support for Enterprise Integration Patterns. After migrating, you can deploy your patterns into a Red Hat JBoss Fuse container.

A.1. MIGRATING ENDPOINTS

Overview

A typical ServiceMix EIP route exposes a service that consumes exchanges from the NMR. The route also defines one or more target destinations, to which exchanges are sent. In the Apache Camel environment, the exposed ServiceMix service maps to a *consumer endpoint* and the ServiceMix target destinations map to *producer endpoints*. The Apache Camel consumer endpoints and producer endpoints are both defined using *endpoint URIs*.

When migrating endpoints from ServiceMix EIP to Apache Camel, you must express the ServiceMix services/endpoints as Apache Camel endpoint URIs. You can adopt one of the following approaches:

- Connect to an existing ServiceMix service/endpoint through the ServiceMix Camel module (which integrates Apache Camel with the NMR).
- If the existing ServiceMix service/endpoint represents a ServiceMix binding component, you can replace the ServiceMix binding component with an equivalent Apache Camel component (thus bypassing the NMR).

The ServiceMix Camel module

The integration between Apache Camel and ServiceMix is provided by the `servicemix-camel` module. This module is provided with ServiceMix, but actually implements a plug-in for the Apache Camel product: the *JBI component* (see [chapter "JBI" in "EIP Component Reference"](#) and [JBI Component](#)).

To access the JBI component from Apache Camel, make sure that the `servicemix-camel` JAR file is included on your Classpath or, if you are using Maven, include a dependency on the `servicemix-camel` artifact in your project POM. You can then access the JBI component by defining Apache Camel endpoint URIs with the `jbi` : component prefix.

Translating ServiceMix URIs into Apache Camel endpoint URIs

ServiceMix defines a flexible format for defining URIs, which is described in detail in [ServiceMix URIs](#). To translate a ServiceMix URI into a Apache Camel endpoint URI, perform the following steps:

1. If the ServiceMix URI contains a namespace prefix, replace the prefix by its corresponding namespace.

For example, after modifying the ServiceMix URI, `service:test:messageFilter`, where `test` corresponds to the namespace, `http://progress.com/demos/test`, you get `service:http://progress.com/demos/test:messageFilter`.

2. Modify the separator character, depending on what kind of namespace appears in the URI:

- If the namespace starts with **http://**, use the **/** character as the separator between namespace, service name, and endpoint name (if present).

For example, the URI,

service:http://progress.com/demos/test:messageFilter, would be modified to **service:http://progress.com/demos/test/messageFilter**.

- If the namespace starts with **urn:**, use the **:** character as the separator between namespace, service name, and endpoint name (if present).

For example, **service:urn:progress.com:demos:test:messageFilter**.

3. Create a JBI endpoint URI by adding the **jbi:** prefix.

For example, **jbi:service:http://progress.com/demos/test/messageFilter**.

Example of mapping ServiceMix URIs

For example, consider the following configuration of the [static recipient list](#) pattern in ServiceMix EIP. The **eip:exchange-target** elements define some targets using the ServiceMix URI format.

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
      xmlns:eip="http://servicemix.apache.org/eip/1.0"
      xmlns:test="http://progress.com/demos/test" >
  ...
  <eip:static-recipient-list service="test:recipients"
endpoint="endpoint">
    <eip:recipients>
      <eip:exchange-target uri="service:test:messageFilter" />
      <eip:exchange-target uri="service:test:trace4" />
    </eip:recipients>
  </eip:static-recipient-list>
  ...
</beans>
```

When the preceding ServiceMix configuration is mapped to an equivalent Apache Camel configuration, you get the following route:

```
<route>
  <from
uri="jbi:endpoint:http://progress.com/demos/test/recipients/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/messageFilter"/>
  <to uri="jbi:service:http://progress.com/demos/test/trace4"/>
</route>
```

Replacing ServiceMix bindings with Apache Camel components

Instead of using the Apache Camel JBI component to route all your messages through the ServiceMix NMR, you can use one of the many supported Apache Camel components to connect directly to a consumer or a producer endpoint. In particular, when sending messages to an external endpoint, it is more efficient to send the messages directly through a Apache Camel component than sending them through the NMR and a ServiceMix binding.

For details of all the Apache Camel components that are available, see ["EIP Component Reference"](#) and [Apache Camel Components](#).

A.2. COMMON ELEMENTS

Overview

When configuring ServiceMix EIP patterns in a ServiceMix configuration file, there are some common elements that occur in many of the pattern schemas. This section provides a brief overview of these common elements and explains how they can be mapped to equivalent constructs in Apache Camel.

Exchange target

All of the patterns supported by ServiceMix EIP use the `eip:exchange-target` element to specify JBI target endpoints. [Table A.1, "Mapping the Exchange Target Element"](#) shows examples of how to map sample `eip:exchange-target` elements to Apache Camel endpoint URIs, where it is assumed that the `test` prefix maps to the `http://progress.com/demos/test` namespace.

Table A.1. Mapping the Exchange Target Element

ServiceMix EIP Target	Apache Camel Endpoint URI
<code><eip:exchange-target interface="HelloWorld" /></code>	<code>jbi:interface:HelloWorld</code>
<code><eip:exchange-target service="test:HelloWorldService" /></code>	<code>jbi:service:http://progress.com/demos/test/HelloWorldService</code>
<code><eip:exchange-target service="test:HelloWorldService" endpoint="secure" /></code>	<code>jbi:service:http://progress.com/demos/test/HelloWorldService/secure</code>
<code><eip:exchange-target uri="service:test:HelloWorldService" /></code>	<code>jbi:service:http://progress.com/demos/test/HelloWorldService</code>

Predicates

The ServiceMix EIP component allows you to define predicate expressions in the XPath language. For example, XPath predicates can appear in `eip:xpath-predicate` elements or in `eip:xpath-splitter` elements, where the XPath predicate is specified using an `xpath` attribute.

ServiceMix XPath predicates can easily be migrated to equivalent constructs in Apache Camel: that is, either the `xpath` element (in XML configuration) or the `xpath()` command (in Java DSL). For example, the message filter pattern in Apache Camel can incorporate an XPath predicate as follows:

```
<route>
  <from
uri="jbi:endpoint:http://progress.com/demos/test/messageFilter/endpoint">
  <filter>
    <xpath>count(/test:world) = 1</xpath>
  <to uri="jbi:service:http://progress.com/demos/test/trace3"/>
```

```
</filter>
</route>
```

Where the **xpath** element specifies that only messages containing the **test:world** element will pass through the filter.



NOTE

Apache Camel also supports a wide range of other scripting languages including XQuery, PHP, Python, and Ruby, which can be used to define predicates. For details of all the supported predicate languages, see [Expression and Predicate Languages](#).

Namespace contexts

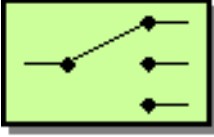
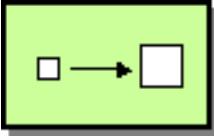


When using XPath predicates in the ServiceMix EIP configuration, it is necessary to define a namespace context using the **eip:namespace-context** element. The namespace is then referenced using a **namespaceContext** attribute.

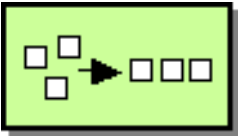
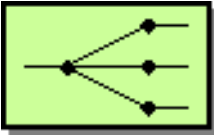
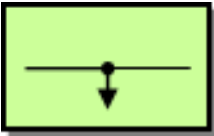
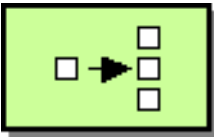
When ServiceMix EIP configuration is migrated to Apache Camel, there is no need to define namespace contexts, because Apache Camel allows you to define XPath predicates without referencing a namespace context. You can simply drop the **eip:namespace-context** elements when you migrate to Apache Camel.

A.3. SERVICEMIX EIP PATTERNS

The patterns supported by ServiceMix EIP are shown in [Table A.2, “ServiceMix EIP Patterns”](#).

Table A.2. ServiceMix EIP Patterns

	Content-Based Router	How we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems.
	Content Enricher	How we communicate with another system if the message originator does not have all the required data items available.
	Message Filter	How a component avoids receiving uninteresting messages.
	Pipeline	How we perform complex processing on a message while maintaining independence and flexibility.

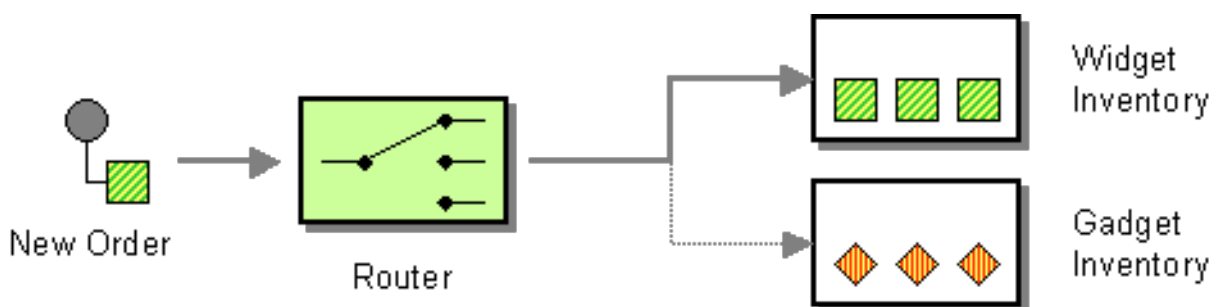
	Resequencer	How we get a stream of related but out-of-sequence messages back into the correct order.
	Static Recipient List	How we route a message to a list of specified recipients.
	Static Routing Slip	How we route a message consecutively through a series of processing steps.
	Wire Tap	How you inspect messages that travel on a point-to-point channel.
	XPath Splitter	How we process a message if it contains multiple elements, each of which may have to be processed in a different way.

A.4. CONTENT-BASED ROUTER

Overview

A *content-based router* enables you to route messages to the appropriate destination, where the routing decision is based on the message contents. This pattern maps to the corresponding [content-based router](#) pattern in Apache Camel.

Figure A.1. Content-based Router Pattern



Example ServiceMix EIP route

Example A.1, “[ServiceMix EIP Content-based Route](#)” shows how to define a content-based router using the ServiceMix EIP component. If a `test:echo` element is present in the message body, the message is routed to the `http://test/pipeLine/endpoint` endpoint. Otherwise, the message is routed to the `test:recipients` endpoint.

Example A.1. ServiceMix EIP Content-based Route

```

<eip:content-based-router service="test:router" endpoint="endpoint">
  <eip:rules>
    <eip:routing-rule>
      <eip:predicate>
        <eip:xpath-predicate xpath="count(/test:echo) = 1"
namespaceContext="#nsContext" />
      </eip:predicate>
      <eip:target>
        <eip:exchange-target uri="endpoint:test:pipeline:endpoint" />
      </eip:target>
    </eip:routing-rule>
    <eip:routing-rule>
      <!-- There is no predicate, so this is the default destination --
>
      <eip:target>
        <eip:exchange-target service="test:recipients" />
      </eip:target>
    </eip:routing-rule>
  </eip:rules>
</eip:content-based-router>

```

Equivalent Apache Camel XML route

[Example A.2, “Apache Camel Content-based Router Using XML Configuration”](#) shows how to define an equivalent route using Apache Camel XML configuration.

Example A.2. Apache Camel Content-based Router Using XML Configuration

```

<route>
  <from
uri="jbi:endpoint:http://progress.com/demos/test/router/endpoint"/>
  <choice>
    <when>
      <xpath>count(/test:echo) = 1</xpath>
      <to
uri="jbi:endpoint:http://progress.com/demos/test/pipeline/endpoint"/>
    </when>
    <otherwise>
      <!-- This is the default destination -->
      <to uri="jbi:service:http://progress.com/demos/test/recipients"/>
    </otherwise>
  </choice>
</route>

```

Equivalent Apache Camel Java DSL route

[Example A.3, “Apache Camel Content-based Router Using Java DSL”](#) shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.3. Apache Camel Content-based Router Using Java DSL

```

from("jbi:endpoint:http://progress.com/demos/test/router/endpoint").
    choice().when(xpath("count(/test:echo) =
1")).to("jbi:endpoint:http://progress.com/demos/test/pipeline/endpoint")
.
otherwise().to("jbi:service:http://progress.com/demos/test/recipients");

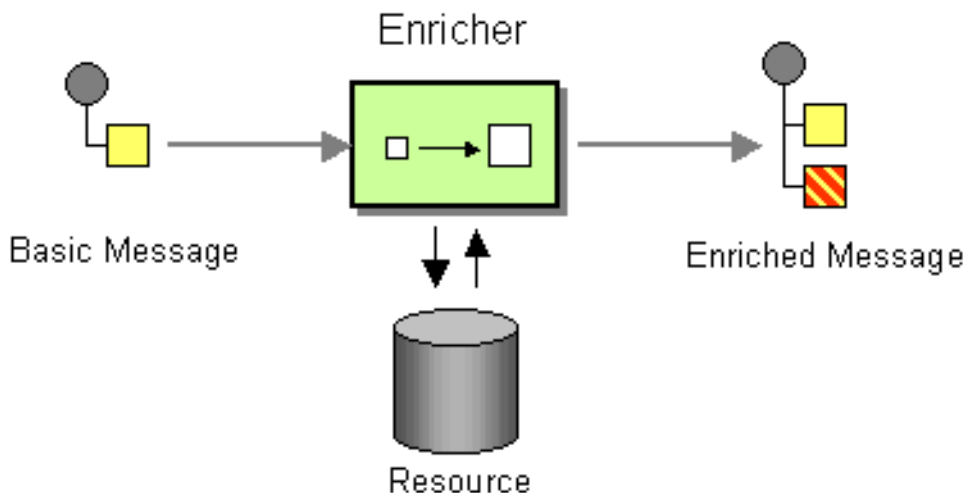
```

A.5. CONTENT ENRICHER

Overview

A *content enricher*, shown in [Figure A.2, “Content Enricher Pattern”](#), is a pattern for augmenting a message with missing information. The ServiceMix EIP content enricher is roughly equivalent to a pipeline that adds missing data as the message passes through an enricher target. Consequently, when migrating to Apache Camel, you can re-implement the ServiceMix content enricher as a Apache Camel pipeline.

Figure A.2. Content Enricher Pattern



Example ServiceMix EIP route

[Example A.4, “ServiceMix EIP Content Enricher”](#) shows how to define a content enricher using the ServiceMix EIP component. Incoming messages pass through the enricher target, **test:additionalInformationExtractor**, which adds missing data to the message. The message is then sent on to its ultimate destination, **test:myTarget**.

Example A.4. ServiceMix EIP Content Enricher

```

<eip:content-enricher service="test:contentEnricher"
endpoint="endpoint">
  <eip:enricherTarget>
    <eip:exchange-target service="test:additionalInformationExtractor"
/>
  </eip:enricherTarget>
</eip:target>

```



```

    <eip:exchange-target service="test:myTarget" />
  </eip:target>
</eip:content-enricher>

```

Equivalent Apache Camel XML route

[Example A.5, “Apache Camel Content Enricher using XML Configuration”](#) shows how to define an equivalent route using Apache Camel XML configuration.

Example A.5. Apache Camel Content Enricher using XML Configuration

```

<route>
  <from
uri="jbi:endpoint:http://progress.com/demos/test/contentEnricher/endpoin
t"/>
  <to
uri="jbi:service:http://progress.com/demos/test/additionalInformationExt
racter"/>
  <to uri="jbi:service:http://progress.com/demos/test/myTarget"/>
</route>

```

Equivalent Apache Camel Java DSL route

[Example A.6, “Apache Camel Content Enricher using Java DSL”](#) shows how to define an equivalent route using the Apache Camel Java DSL:

Example A.6. Apache Camel Content Enricher using Java DSL

```

from("jbi:endpoint:http://progress.com/demos/test/contentEnricher/endpoi
nt").
to("jbi:service:http://progress.com/demos/test/additionalInformationExtr
acter").
  to("jbi:service:http://progress.com/demos/test/myTarget");

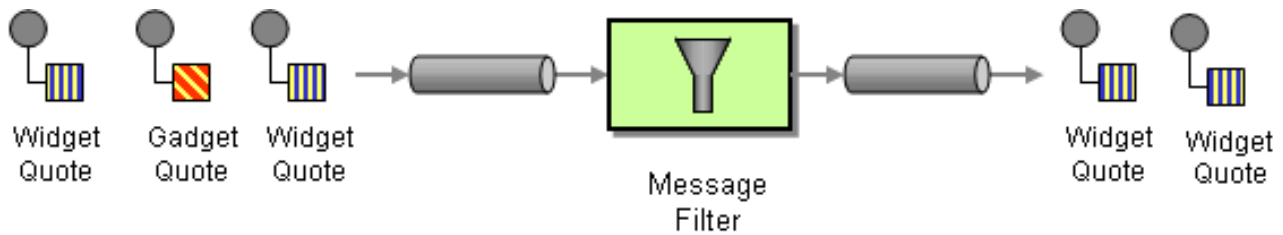
```

A.6. MESSAGE FILTER

Overview

A *message filter*, shown in [Figure A.3, “Message Filter Pattern”](#), is a processor that eliminates undesired messages based on specific criteria. Filtering is controlled by specifying a predicate in the filter: when the predicate is **true**, the incoming message is allowed to pass; otherwise, it is blocked. This pattern maps to the corresponding [message filter](#) pattern in Apache Camel.

Figure A.3. Message Filter Pattern



Example ServiceMix EIP route

Example A.7, “ServiceMix EIP Message Filter” shows how to define a message filter using the ServiceMix EIP component. Incoming messages are passed through a filter mechanism that blocks messages that lack a `test:world` element.

Example A.7. ServiceMix EIP Message Filter

```
<eip:message-filter service="test:messageFilter" endpoint="endpoint">
  <eip:target>
    <eip:exchange-target service="test:trace3" />
  </eip:target>
  <eip:filter>
    <eip:xpath-predicate xpath="count(/test:world) = 1"
namespaceContext="#nsContext"/>
  </eip:filter>
</eip:message-filter>
```

Equivalent Apache Camel XML route

Example A.8, “Apache Camel Message Filter Using XML” shows how to define an equivalent route using Apache Camel XML configuration.

Example A.8. Apache Camel Message Filter Using XML

```
<route>
  <from
uri="jbi:endpoint:http://progress.com/demos/test/messageFilter/endpoint"
  >
    <filter>
      <xpath>count(/test:world) = 1</xpath>
      <to uri="jbi:service:http://progress.com/demos/test/trace3"/>
    </filter>
  </route>
```

Equivalent Apache Camel Java DSL route

Example A.9, “Apache Camel Message Filter Using Java DSL” shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.9. Apache Camel Message Filter Using Java DSL

```

from("jbi:endpoint:http://progress.com/demos/test/messageFilter/endpoint
").
  filter(xpath("count(/test:world) = 1")).
  to("jbi:service:http://progress.com/demos/test/trace3");

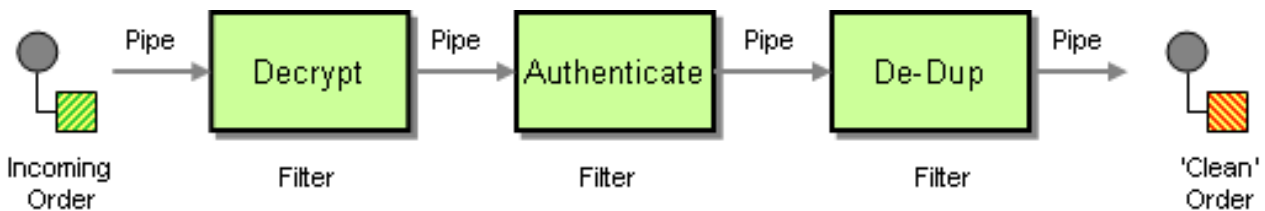
```

A.7. PIPELINE

Overview

The ServiceMix EIP *pipeline* pattern, shown in [Figure A.4, “Pipes and Filters Pattern”](#), is used to pass messages through a single transformer endpoint, where the transformer's input is taken from the source endpoint and the transformer's output is routed to the target endpoint. This pattern is thus a special case of the more general Apache Camel [pipes and filters](#) pattern, which enables you to pass an *In* message through *multiple* transformer endpoints.

Figure A.4. Pipes and Filters Pattern



Example ServiceMix EIP route

[Example A.10, “ServiceMix EIP Pipeline”](#) shows how to define a pipeline using the ServiceMix EIP component. Incoming messages are passed into the transformer endpoint, `test:decrypt`, and the output from the transformer endpoint is then passed into the target endpoint, `test:plaintextOrder`.

Example A.10. ServiceMix EIP Pipeline

```

<eip:pipeline service="test:pipeline" endpoint="endpoint">
  <eip:transformer>
    <eip:exchange-target service="test:decrypt" />
  </eip:transformer>
  <eip:target>
    <eip:exchange-target service="test:plaintextOrder" />
  </eip:target>
</eip:pipeline>

```

Equivalent Apache Camel XML route

[Example A.11, “Apache Camel Pipeline Using XML”](#) shows how to define an equivalent route using Apache Camel XML configuration.

Example A.11. Apache Camel Pipeline Using XML

```

<route>
  <from

```

```
uri="jbi:endpoint:http://progress.com/demos/test/pipeline/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/decrypt"/>
  <to uri="jbi:service:http://progress.com/demos/test/plaintextOrder"/>
</route>
```

Equivalent Apache Camel Java DSL route

Example A.12, “Apache Camel Pipeline Using Java DSL” shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.12. Apache Camel Pipeline Using Java DSL

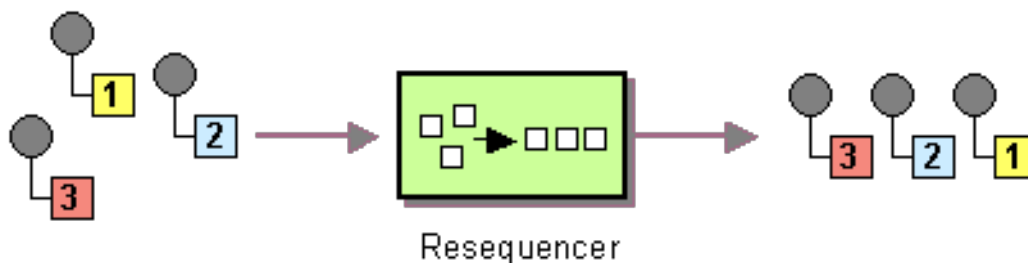
```
from("jbi:endpoint:http://progress.com/demos/test/pipeline/endpoint").
  pipeline("jbi:service:http://progress.com/demos/test/decrypt",
  "jbi:service:http://progress.com/demos/test/plaintextOrder");
```

A.8. RESEQUENCER

Overview

The *resequencer* pattern, shown in Figure A.5, “Resequencer Pattern”, enables you to resequence messages according to the sequence number stored in an NMR property. The ServiceMix EIP resequencer pattern maps to the Apache Camel *resequencer* configured with the *stream resequencing* algorithm.

Figure A.5. Resequencer Pattern



Sequence number property

The sequence of messages emitted from the resequencer is determined by the value of the sequence number property: messages with a low sequence number are emitted first and messages with a higher number are emitted later. By default, the sequence number is read from the **org.apache.servicemix.eip.sequence.number** property in ServiceMix, but you can customize the name of this property using the **eip:default-comparator** element in ServiceMix.

The equivalent concept in Apache Camel is a *sequencing expression*, which can be any message-dependent expression. When migrating from ServiceMix EIP, you normally define an expression that extracts the sequence number from a header (a Apache Camel header is equivalent to an NMR message property). For example, to extract a sequence number from a **seqnum** header, you can use the simple expression, **header.seqnum**.

Example ServiceMix EIP route

[Example A.13, “ServiceMix EIP Resequencer”](#) shows how to define a resequencer using the ServiceMix EIP component.

Example A.13. ServiceMix EIP Resequencer

```
<eip:resequencer
  service="sample:Resequencer"
  endpoint="ResequencerEndpoint"
  comparator="#comparator"
  capacity="100"
  timeout="2000">
  <eip:target>
    <eip:exchange-target service="sample:SampleTarget" />
  </eip:target>
</eip:resequencer>

<!-- Configure default comparator with custom sequence number property -
->
<eip:default-comparator xml:id="comparator"
  sequenceNumberKey="seqnum"/>
```

Equivalent Apache Camel XML route

[Example A.14, “Apache Camel Resequencer Using XML”](#) shows how to define an equivalent route using Apache Camel XML configuration.

Example A.14. Apache Camel Resequencer Using XML

```
<route>
  <from uri="jbi:endpoint:sample:Resequencer:ResequencerEndpoint"/>
  <resequencer>
    <simple>header.seqnum</simple>
    <to uri="jbi:service:sample:SampleTarget" />
    <stream-config capacity="100" timeout="2000"/>
  </resequencer>
</route>
```

Equivalent Apache Camel Java DSL route

[Example A.15, “Apache Camel Resequencer Using Java DSL”](#) shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.15. Apache Camel Resequencer Using Java DSL

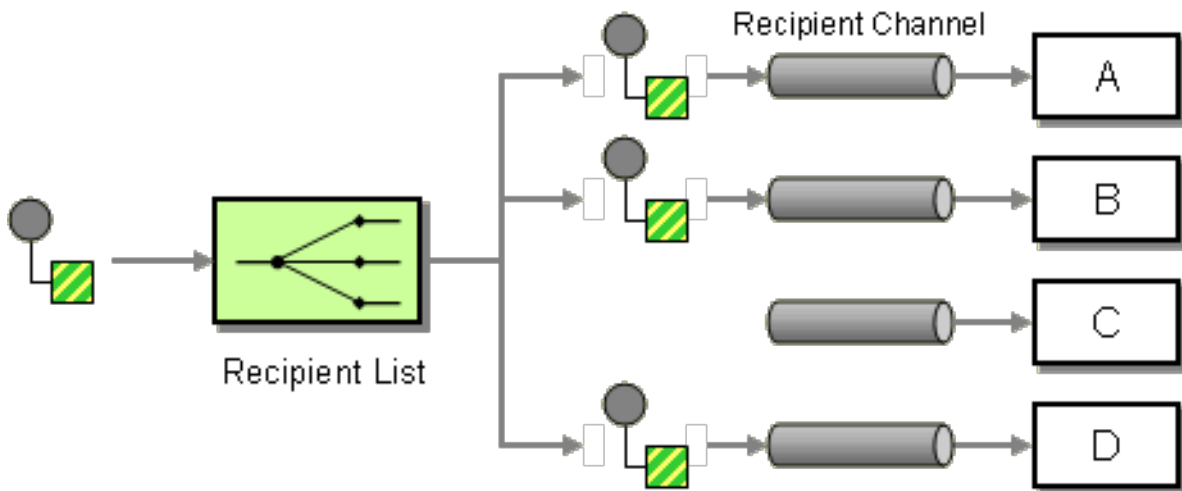
```
from("jbi:endpoint:sample:Resequencer:ResequencerEndpoint").
  resequencer(header("seqnum")).
  stream(new StreamResequencerConfig(100, 2000L)).
  to("jbi:service:sample:SampleTarget");
```

A.9. STATIC RECIPIENT LIST

Overview

A *recipient list*, shown in [Figure A.6, “Static Recipient List Pattern”](#), is a type of router that sends each incoming message to multiple different destinations. The ServiceMix EIP recipient list is restricted to processing *InOnly* and *RobustInOnly* exchange patterns. Moreover, the list of recipients must be static. This pattern maps to the [recipient list](#) with fixed destination pattern in Apache Camel.

Figure A.6. Static Recipient List Pattern



Example ServiceMix EIP route

[Example A.16, “ServiceMix EIP Static Recipient List”](#) shows how to define a static recipient list using the ServiceMix EIP component. Incoming messages are copied to the `test:messageFilter` endpoint and to the `test:trace4` endpoint.

Example A.16. ServiceMix EIP Static Recipient List

```
<eip:static-recipient-list service="test:recipients"
                        endpoint="endpoint">
  <eip:recipients>
    <eip:exchange-target service="test:messageFilter" />
    <eip:exchange-target service="test:trace4" />
  </eip:recipients>
</eip:static-recipient-list>
```

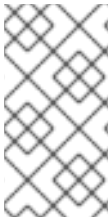
Equivalent Apache Camel XML route

[Example A.17, “Apache Camel Static Recipient List Using XML”](#) shows how to define an equivalent route using Apache Camel XML configuration.

Example A.17. Apache Camel Static Recipient List Using XML

```
<route>
  <from
uri="jbi:endpoint:http://progress.com/demos/test/recipients/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/messageFilter"/>
```

```
<to uri="jbi:service:http://progress.com/demos/test/trace4"/>
</route>
```



NOTE

The preceding route configuration appears to have the same syntax as a Apache Camel pipeline pattern. The key difference is that the preceding route is intended for processing *InOnly* message exchanges, which are processed in a different way. See [Section 4.4, “Pipes and Filters”](#) for more details.

Equivalent Apache Camel Java DSL route

[Example A.18, “Apache Camel Static Recipient List Using Java DSL”](#) shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.18. Apache Camel Static Recipient List Using Java DSL

```
from("jbi:endpoint:http://progress.com/demos/test/recipients/endpoint").
    to("jbi:service:http://progress.com/demos/test/messageFilter",
        "jbi:service:http://progress.com/demos/test/trace4");
```

A.10. STATIC ROUTING SLIP

Overview

The *static routing slip* pattern in the ServiceMix EIP component is used to route an *InOut* message exchange through a series of endpoints. Semantically, it is equivalent to the [pipeline](#) pattern in Apache Camel.

Example ServiceMix EIP route

[Example A.19, “ServiceMix EIP Static Routing Slip”](#) shows how to define a static routing slip using the ServiceMix EIP component. Incoming messages pass through each of the endpoints, **test:procA**, **test:procB**, and **test:procC**, where the output of each endpoint is connected to the input of the next endpoint in the chain. The final endpoint, **test:procC**, sends its output (*Out* message) back to the caller.

Example A.19. ServiceMix EIP Static Routing Slip

```
<eip:static-routing-slip service="test:routingSlip"
    endpoint="endpoint">
    <eip:targets>
        <eip:exchange-target service="test:procA" />
        <eip:exchange-target service="test:procB" />
        <eip:exchange-target service="test:procC" />
    </eip:targets>
</eip:static-routing-slip>
```

Equivalent Apache Camel XML route

Example A.20, “[Apache Camel Static Routing Slip Using XML](#)” shows how to define an equivalent route using Apache Camel XML configuration.

Example A.20. Apache Camel Static Routing Slip Using XML

```
<route>
  <from
    uri="jbi:endpoint:http://progress.com/demos/test/routingSlip/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/procA"/>
  <to uri="jbi:service:http://progress.com/demos/test/procB"/>
  <to uri="jbi:service:http://progress.com/demos/test/procC"/>
</route>
```

Equivalent Apache Camel Java DSL route

Example A.21, “[Apache Camel Static Routing Slip Using Java DSL](#)” shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.21. Apache Camel Static Routing Slip Using Java DSL

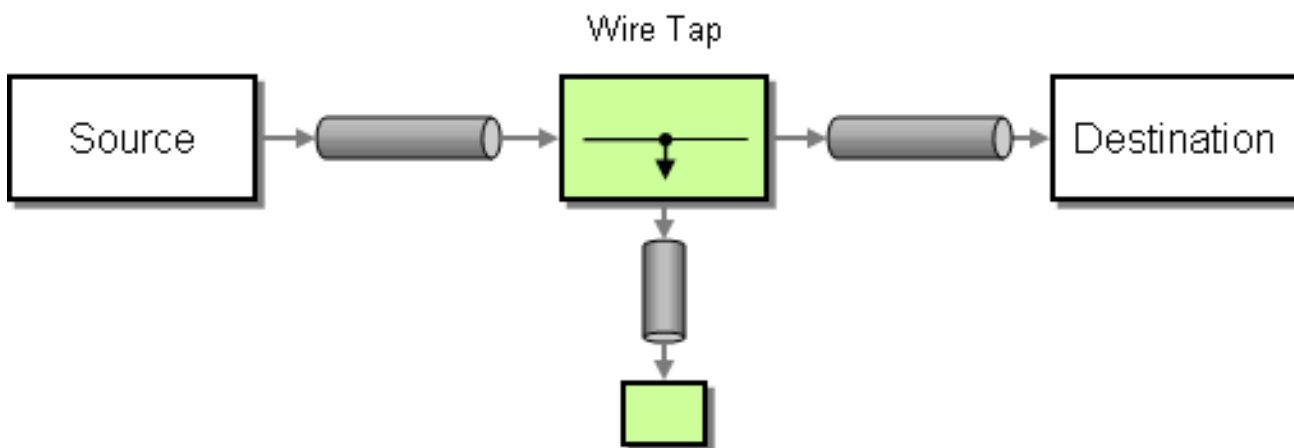
```
from("jbi:endpoint:http://progress.com/demos/test/routingSlip/endpoint")
    .
    pipeline("jbi:service:http://progress.com/demos/test/procA",
            "jbi:service:http://progress.com/demos/test/procB",
            "jbi:service:http://progress.com/demos/test/procC");
```

A.11. WIRE TAP

Overview

The *wire tap* pattern, shown in [Figure A.7, “Wire Tap Pattern”](#), allows you to route messages to a separate tap location before it is forwarded to the ultimate destination. The ServiceMix EIP wire tap pattern maps to the [wire tap](#) pattern in Apache Camel.

Figure A.7. Wire Tap Pattern



Example ServiceMix EIP route

[Example A.22, “ServiceMix EIP Wire Tap”](#) shows how to define a wire tap using the ServiceMix EIP component. The *In* message from the source endpoint is copied to the *In*-listener endpoint, before being forwarded on to the target endpoint. If you want to monitor any returned *Out* messages or *Fault* messages from the target endpoint, you also must define an *Out* listener (using the `eip:outListener` element) and a *Fault* listener (using the `eip:faultListener` element).

Example A.22. ServiceMix EIP Wire Tap

```
<eip:wire-tap service="test:wireTap" endpoint="endpoint">
  <eip:target>
    <eip:exchange-target service="test:target" />
  </eip:target>
  <eip:inListener>
    <eip:exchange-target service="test:trace1" />
  </eip:inListener>
</eip:wire-tap>
```

Equivalent Apache Camel XML route

[Example A.23, “Apache Camel Wire Tap Using XML”](#) shows how to define an equivalent route using Apache Camel XML configuration.

Example A.23. Apache Camel Wire Tap Using XML

```
<route>
  <from
    uri="jbi:endpoint:http://progress.com/demos/test/wireTap/endpoint"/>
    <to uri="jbi:service:http://progress.com/demos/test/trace1"/>
    <to uri="jbi:service:http://progress.com/demos/test/target"/>
  </route>
```

Equivalent Apache Camel Java DSL route

[Example A.24, “Apache Camel Wire Tap Using Java DSL”](#) shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.24. Apache Camel Wire Tap Using Java DSL

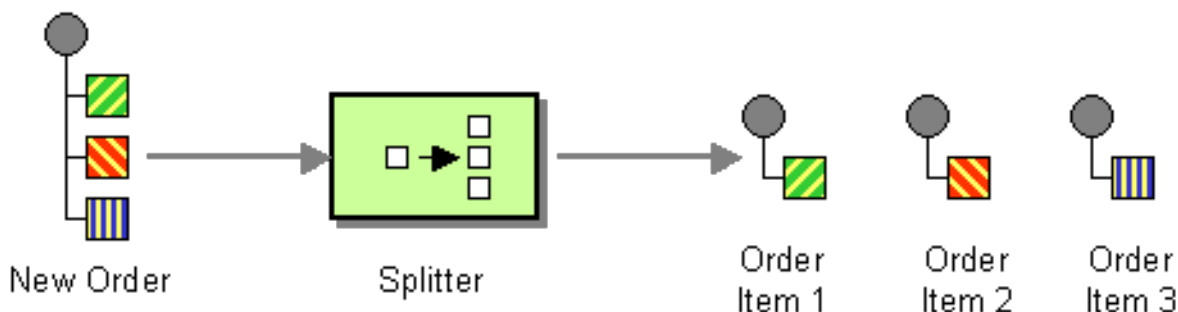
```
from("jbi:endpoint:http://progress.com/demos/test/wireTap/endpoint")
  .to("jbi:service:http://progress.com/demos/test/trace1",
      "jbi:service:http://progress.com/demos/test/target");
```

A.12. XPATH SPLITTER

Overview

A *splitter*, shown in [Figure A.8, “XPath Splitter Pattern”](#), is a type of router that splits an incoming message into a series of outgoing messages, where each of the messages contains a piece of the original message. The ServiceMix EIP XPath splitter pattern is restricted to using the *InOnly* and *RobustInOnly* exchange patterns. The expression that defines how to split up the original message is defined in the XPath language. The XPath splitter pattern maps to the [splitter](#) pattern in Apache Camel.

Figure A.8. XPath Splitter Pattern



Forwarding NMR attachments and properties

The `eip:xpath-splitter` element supports a `forwardAttachments` attribute and a `forwardProperties` attribute, either of which can be set to `true`, if you want the splitter to copy the incoming message's attachments or properties to the outgoing messages. The corresponding splitter pattern in Apache Camel does not support any such attributes. By default, the incoming message's headers are copied to each of the outgoing messages by the Apache Camel splitter.

Example ServiceMix EIP route

[Example A.25, “ServiceMix EIP XPath Splitter”](#) shows how to define a splitter using the ServiceMix EIP component. The specified XPath expression, `/*/ *`, causes an incoming message to split at every occurrence of a nested XML element (for example, the `/foo/bar` and `/foo/car` elements are split into distinct messages).

Example A.25. ServiceMix EIP XPath Splitter

```
<eip:xpath-splitter service="test:xpathSplitter"
  endpoint="endpoint"
  xpath="/*/ *"
  namespaceContext="#nsContext">
  <eip:target>
    <eip:exchange-target uri="service:http://test/router" />
  </eip:target>
</eip:xpath-splitter>
```

Equivalent Apache Camel XML route

[Example A.26, “Apache Camel XPath Splitter Using XML”](#) shows how to define an equivalent route using Apache Camel XML configuration.

Example A.26. Apache Camel XPath Splitter Using XML

```
<route>
  <from
```

```
uri="jbi:endpoint:http://progress.com/demos/test/xpathSplitter/endpoint"
/>
  <splitter>
    <xpath>/**/*</xpath>
    <to uri="jbi:service:http://test/router"/>
  </splitter>
</route>
```

Equivalent Apache Camel Java DSL route

[Example A.27, “Apache Camel XPath Splitter Using Java DSL”](#) shows how to define an equivalent route using the Apache Camel Java DSL.

Example A.27. Apache Camel XPath Splitter Using Java DSL

```
from("jbi:endpoint:http://progress.com/demos/test/xpathSplitter/endpoint")
    .splitter(xpath("/**/*")).to("jbi:service:http://test/router");
```

INDEX

P

performer, [Overview](#)

W

wire tap pattern, [System Management](#)