Red Hat JBoss Fuse 6.3

Apache Camel Development Guide

Develop applications with Apache Camel

Red Hat JBoss Fuse 6.3 Apache Camel Development Guide

Develop applications with Apache Camel

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

Legal Notice

Abstract

This guide describes how to develop JBoss Fuse applications with Apache Camel. It covers the basic building blocks, enterprise integration patterns, basic syntax for routing expression and predicate languages, creating web services with the Apache CXF component, using the Apache Camel API, and how to create a Camel component that wraps any Java API.

Table of Contents

PART I. IMPLEMENTING ENTERPRISE INTEGRATION PATTERNS

Abstract

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

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.
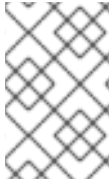
> **NOTE**
>
> When you use the *contextScan* with Spring or Blueprint to filter **RouteBuilder** classes, by default Apache Camel will look for singleton beans. However, you can turn on the old behavior to include prototype scoped with the new option **includeNonSingletons**.

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
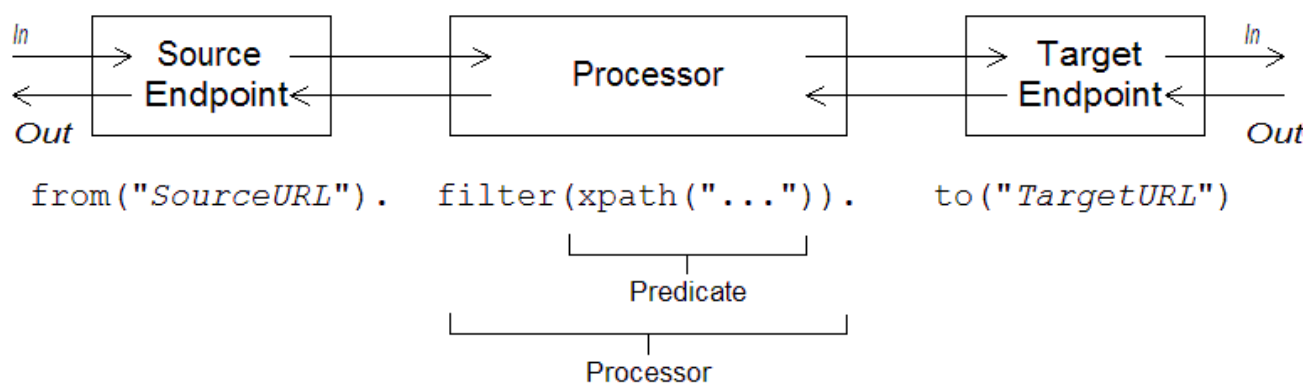
A local rule always starts with a **from("*EndpointURL*")** method, which specifies the source of messages (*consumer endpoint*) for the routing rule. You can then add an arbitrarily long chain of processors to the rule (for example, **filter()**). You typically finish off the rule with a **to("*EndpointURL*")** 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, **cxf:MyServiceName**, or an Apache ActiveMQ endpoint, **activemq:queue:MyQName**. For a complete list of component types, see "Apache Camel 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 8.5, "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

Working with Long Endpoint URIs

Sometimes endpoint URIs can become quite long due to all the accompanying configuration information supplied. In JBoss Fuse 6.2 onwards, there are two approaches you can take to make your working with lengthy URIs more manageable.

Configure Endpoints Separately

> You can configure the endpoint separately, and from the routes refer to the endpoints using their shorthand IDs.

> ```
> <camelContext ...>
>
>   <endpoint id="foo" uri="ftp://foo@myserver">
>     <property name="password" value="secret"/>
>     <property name="recursive" value="true"/>
>     <property name="ftpClient.dataTimeout" value="30000"/>
>     <property name="ftpClient.serverLanguageCode" value="fr"/>
>   </endpoint>
> ```

```
  <route>
    <from uri="ref:foo"/>
    ...
  </route>
</camelContext>
```

You can also configure some options in the URI and then use the **property** attribute to specify additional options (or to override options from the URI).

```
<endpoint id="foo" uri="ftp://foo@myserver?recursive=true">
  <property name="password" value="secret"/>
  <property name="ftpClient.dataTimeout" value="30000"/>
  <property name="ftpClient.serverLanguageCode" value="fr"/>
</endpoint>
```

Split Endpoint Configuration Across New Lines

You can split URI attributes using new lines.

```
<route>
  <from uri="ftp://foo@myserver?password=secret&amp;
        recursive=true&amp;ftpClient.dataTimeout=30000&amp;
        ftpClientConfig.serverLanguageCode=fr"/>
  <to uri="bean:doSomething"/>
</route>
```
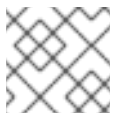
> **NOTE**
>
> You can specify one or more options on each line, each separated by **&**.

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");
```

Specifying raw values in URI options

By default, the option values that you specify in a URI are automatically URI-encoded. In some cases this is undesirable beahavior. For example, when setting a password option, it is preferable to transmit the raw character string *without* URI encoding.

It is possible to switch off URI encoding by specifying an option value with the syntax, **RAW(*RawValue*)**. For example,

```
from("SourceURI")
  .to("ftp:joe@myftpserver.com?password=RAW(se+re?t&23)&binary=true")
```

In this example, the password value is transmitted as the literal value, **se+re?t&23**.

Case-insensitive enum options

Some endpoint URI options get mapped to Java **enum** constants. For example, the **level** option of the Log component, which can take the **enum** values, **INFO**, **WARN**, **ERROR**, and so on. This type conversion is case-insensitive, so any of the following alternatives could be used to set the logging level of a Log producer endpoint:

```
<to uri="log:foo?level=info"/>
<to uri="log:foo?level=INfo"/>
<to uri="log:foo?level=InFo"/>
```

Specifying URI Resources

From Camel 2.17, the resource based components such as XSLT, Velocity can load the resource file from the Registry by using *ref:* as prefix.

For example, *ifmyvelocityscriptbean* and *mysimplescriptbean* are the IDs of two beans in the registry, you can use the contents of these beans as follows:

```
Velocity endpoint:
------------------
from("velocity:ref:myvelocityscriptbean").<rest_of_route>.

Language endpoint (for invoking a scripting language):
-----------------------------------------------------
from("direct:start")
  .to("language:simple:ref:mysimplescriptbean")
 Where Camel implicitly converts the bean to a String.
```

Apache Camel components

Each URI *scheme* maps to an *Apache Camel component*, where an 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>2.17.0.redhat-630xxx</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.

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
aggregate()	**aggregate**	Aggregator EIP: Creates an aggregator, which combines multiple incoming exchanges into a single exchange.

Java DSL	XML DSL	Description
aop()	**aop**	Use Aspect Oriented Programming (AOP) to do work before and after a specified sub-route. See ???.
bean(), **beanRef()**	**bean**	Process the current exchange by invoking a method on a Java object (or bean). See Section 2.4, "Bean Integration".
choice()	**choice**	Content Based Router EIP: Selects a particular sub-route based on the exchange content, using **when** and **otherwise** clauses.
convertBodyTo()	**convertBodyTo**	Converts the *In* message body to the specified type.
delay()	**delay**	Delayer EIP: Delays the propagation of the exchange to the latter part of the route.
doTry()	**doTry**	Creates a try/catch block for handling exceptions, using **doCatch**, **doFinally**, and **end** clauses.
end()	*N/A*	Ends the current command block.
enrich(),**enrichRef()**	**enrich**	Content Enricher EIP: Combines the current exchange with data requested from a specified *producer* endpoint URI.
filter()	**filter**	Message Filter EIP: Uses a predicate expression to filter incoming exchanges.
idempotentConsumer()	**idempotentConsumer**	Idempotent Consumer EIP: Implements a strategy to suppress duplicate messages.
inheritErrorHandler()	**@inheritErrorHandler**	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).

Java DSL	XML DSL	Description
inOnly()	**inOnly**	Either sets the current exchange's MEP to *InOnly* (if no arguments) or sends the exchange as an *InOnly* to the specified endpoint(s).
inOut()	**inOut**	Either sets the current exchange's MEP to *InOut* (if no arguments) or sends the exchange as an *InOut* to the specified endpoint(s).
loadBalance()	**loadBalance**	Load Balancer EIP: Implements load balancing over a collection of endpoints.
log()	**log**	Logs a message to the console.
loop()	**loop**	Loop EIP: Repeatedly resends each exchange to the latter part of the route.
markRollbackOnly()	**@markRollbackOnly**	*(Transactions)* 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 "Transaction Guide".
markRollbackOnlyLast()	**@markRollbackOnlyLast**	*(Transactions)* 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 "Transaction Guide".
marshal()	**marshal**	Transforms into a low-level or binary format using the specified data format, in preparation for sending over a particular transport protocol.
multicast()	**multicast**	Multicast EIP: Multicasts the current exchange to multiple destinations, where each destination gets its own copy of the exchange.

Java DSL	XML DSL	Description
onCompletion()	**onCompletion**	Defines a sub-route (terminated by **end()** in the Java DSL) that gets executed after the main route has completed. See also Section 2.11, "OnCompletion".
onException()	**onException**	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).
pipeline()	**pipeline**	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".
policy()	**policy**	Apply a policy to the current route (currently only used for transactional policies—see "Transaction Guide").
pollEnrich(),**pollEnrichRef()**	**pollEnrich**	Content Enricher EIP: Combines the current exchange with data polled from a specified *consumer* endpoint URI.
process(),**processRef**	**process**	Execute a custom processor on the current exchange. See the section called "Custom processor" and Part IV, "Programming EIP Components".
recipientList()	**recipientList**	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).
removeHeader()	**removeHeader**	Removes the specified header from the exchange's *In* message.
removeHeaders()	**removeHeaders**	Removes the headers matching the specified pattern from the exchange's *In* 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.

Java DSL	XML DSL	Description
removeProperty()	**removeProperty**	Removes the specified exchange property from the exchange.
removeProperties()	**removeProperties**	Removes the properties matching the specified pattern from the exchange. Takes a comma separated list of 1 or more strings as arguments. The first string is the pattern (see **removeHeaders()** above). Subsequent strings specify exceptions – these properties remain.
resequence()	**resequence**	Resequencer EIP: Re-orders incoming exchanges on the basis of a specified comparotor operation. Supports a *batch* mode and a *stream* mode.
rollback()	**rollback**	*(Transactions)* Marks the current transaction for rollback only (also raising an exception, by default). See "Transaction Guide".
routingSlip()	**routingSlip**	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.
sample()	**sample**	Creates a sampling throttler, allowing you to extract a sample of exchanges from the traffic on a route.
setBody()	**setBody**	Sets the message body of the exchange's *In* message.
setExchangePattern()	**setExchangePattern**	Sets the current exchange's MEP to the specified value. See the section called "Message exchange patterns".
setHeader()	**setHeader**	Sets the specified header in the exchange's *In* message.
setOutHeader()	**setOutHeader**	Sets the specified header in the exchange's *Out* message.
setProperty()	**setProperty()**	Sets the specified exchange property.

Java DSL	XML DSL	Description
sort()	**sort**	Sorts the contents of the *In* message body (where a custom comparator can optionally be specified).
split()	**split**	Splitter EIP: Splits the current exchange into a sequence of exchanges, where each split exchange contains a fragment of the original message body.
stop()	**stop**	Stops routing the current exchange and marks it as completed.
threads()	**threads**	Creates a thread pool for concurrent processing of the latter part of the route.
throttle()	**throttle**	Throttler EIP: Limit the flow rate to the specified level (exchanges per second).
throwException()	**throwException**	Throw the specified Java exception.
to()	**to**	Send the exchange to one or more endpoints. See Section 2.1, "Pipeline Processing".
toF()	*N/A*	Send the exchange to an endpoint, using string formatting. That is, the endpoint URI string can embed substitutions in the style of the C **printf()** function.
transacted()	**transacted**	Create a Spring transaction scope that encloses the latter part of the route. See "Transaction Guide".
transform()	**transform**	Message Translator EIP: Copy the *In* message headers to the *Out* message headers and set the *Out* message body to the specified value.
unmarshal()	**unmarshal**	Transforms the *In* message body from a low-level or binary format to a high-level format, using the specified data format.

Java DSL	XML DSL	Description
validate()	**validate**	Takes a predicate expression to test whether the current message is valid. If the predicate returns **false**, throws a **PredicateValidationExceptio n** exception.
wireTap()	**wireTap**	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(bodyAs(String.class).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
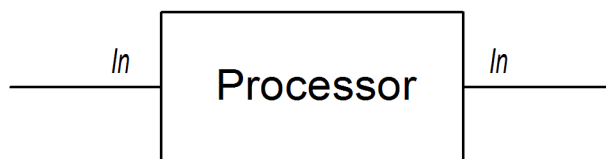
The following route shows a **setHeader()** command that modifies the current *In* message by adding (or modifying) the **BillingSystem** heading:

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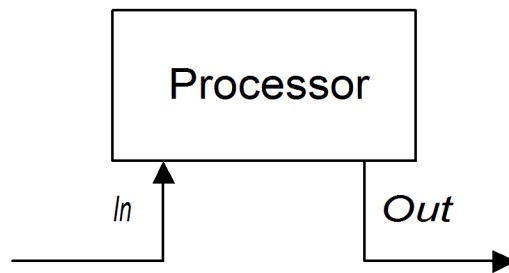
The following route shows a **transform()** command that creates an *Out* message with a message body containing the string, **DummyBody**:

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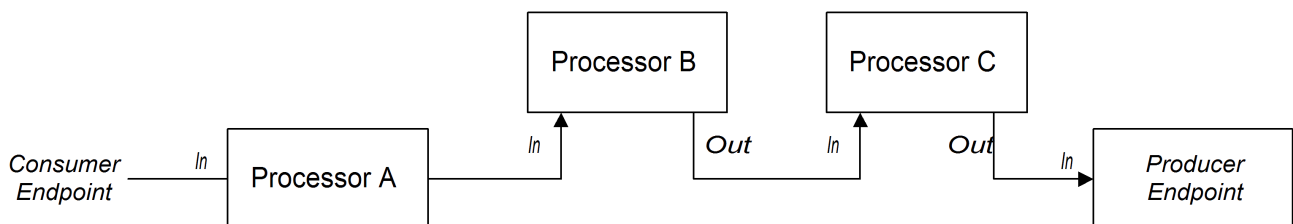
Apache Camel employs the pipeline pattern by default, so you do not need to use any special syntax to create a pipeline in your routes. For example, the following route pulls messages from a **userdataQueue** queue, pipes the message through a Velocity template (to produce a customer address in text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

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

Where the Velocity endpoint, **velocity:file:AdressTemplate.vm**, specifies the location of a Velocity template file, **file:AdressTemplate.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 *Velocity* in the *Apache Camel Component Reference Guide*.

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
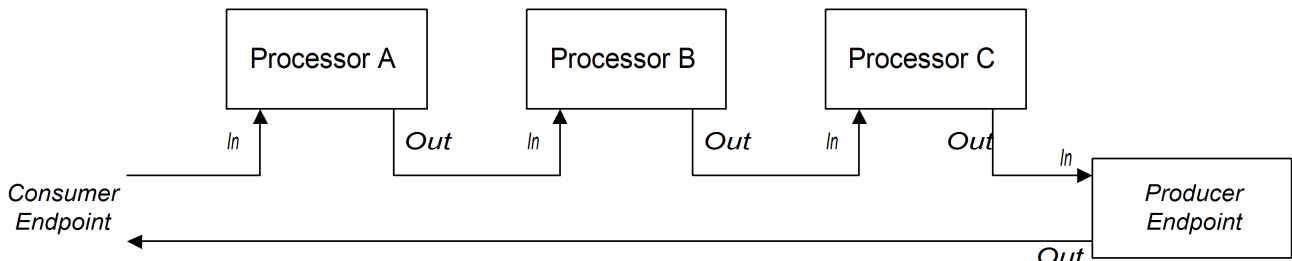
Note that in order to support the *InOut* exchange pattern, it is *essential* that the last node in the route (whether it is a producer endpoint or some other kind of processor) creates an *Out* message. Otherwise, any client that connects to the consumer endpoint would hang and wait indefinitely for 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 9.1, "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 6.3, "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
// 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:

```xml
<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**, **IOException**, 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.

> **NOTE**
>
> The **throwException** EIP enables you to create a new exception instance from a simple language expression. You can make it dynamic, based on the available information from the current exchange. for example,
>
> ```
> <throwException exceptionType="java.lang.IllegalArgumentException"
> message="${body}"/>
> ```

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 6.3, "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 by 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>
```
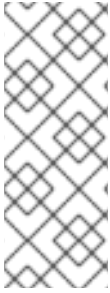
NOTE

By default, Camel makes a copy of the original message at the start of the route, which ensures that the original message is available when you call **useOriginalMessage()**. But if the **setAllowUseOriginalMessage()** option is set to **false** on the Camel context, the original message will *not* be accessible and you cannot call **useOriginalMessage()** (for example, you might want to choose this behaviour to optimize performance when processing large messages).

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 6.3, "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" redeliveryPolicyRef="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.
// Noitce: 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.

> **NOTE**
>
> Using a custom processor, the Camel Exception Clause and Error Handler get invoked, soon after it throws an exception using the new ***onExceptionOccurred*** option.

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:

```xml
<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:

```xml
<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

Java DSL Builder	XML DSL Type Attribute	Description
defaultErrorHandler()	**DefaultErrorHandler**	Propagates exceptions back to the caller and supports the redelivery policy, but it does not support a dead letter queue.
deadLetterChannel()	**DeadLetterChannel**	Supports the same features as the default error handler and, in addition, supports a dead letter queue.
loggingErrorChannel()	**LoggingErrorChannel**	Logs the exception text whenever an exception occurs.
noErrorHandler()	**NoErrorHandler**	Dummy handler implementation that can be used to disable the error handler.
	TransactionErrorHandler	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.
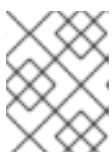
  > **NOTE**
  >
  > This algorithm is different from the exception matching algorithm used by the **onException** clause—see Section 2.3.1, "onException Clause" for details.

- *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:

```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>
```

Nested Conditions in doTry

There are various options available to add Camel exception handling to a JavaDSL route. **dotry()** creates a try or catch block for handling exceptions and is useful for route specific error handling.

If you want to catch the exception inside of **ChoiceDefinition**, you can use the following **doTry** blocks:

```java
from("direct:wayne-get-token").setExchangePattern(ExchangePattern.InOut)
        .doTry()
            .to("https4://wayne-token-service")
            .choice()
                .when().simple("${header.CamelHttpResponseCode} == '200'")
                    .convertBodyTo(String.class)
.setHeader("wayne-token").groovy("body.replaceAll('\"',')")
                    .log(">> Wayne Token : ${header.wayne-token}")
                .endChoice()

doCatch(java.lang.Class (java.lang.Exception>)
            .log(">> Exception")
        .endDoTry();

from("direct:wayne-get-token").setExchangePattern(ExchangePattern.InOut)
        .doTry()
            .to("https4://wayne-token-service")
```

```
            .doCatch(Exception.class)
              .log(">> Exception")
            .endDoTry();
```

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* in the *Apache Camel Component Reference Guide* .

How to propagate stack trace information

It is possible to configure a CXF endpoint so that, when a Java exception is thrown on the server side, the stack trace for the exception is marshalled into a fault message and returned to the client. To enable this feaure, set the **dataFormat** to **PAYLOAD** and set the **faultStackTraceEnabled** property to **true** in the **cxfEndpoint** element, as follows:

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable sending the stack trace back to client; the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" />
    <entry key="dataFormat" value="PAYLOAD" />
  </cxf:properties>
</cxf:cxfEndpoint>
```

For security reasons, the stack trace does not include the causing exception (that is, the part of a stack trace that follows **Caused by**). If you want to include the causing exception in the stack trace, set the **exceptionMessageCauseEnabled** property to **true** in the **cxfEndpoint** element, as follows:

```
<cxf:cxfEndpoint id="router" address="http://localhost:9002/TestMessage"
    wsdlURL="ship.wsdl"
    endpointName="s:TestSoapEndpoint"
    serviceName="s:TestService"
    xmlns:s="http://test">
  <cxf:properties>
    <!-- enable to show the cause exception message and the default value is false -->
  <entry key="exceptionMessageCauseEnabled" value="true" />
    <!-- enable to send the stack trace back to client,  the default value is false-->
    <entry key="faultStackTraceEnabled" value="true" />
    <entry key="dataFormat" value="PAYLOAD" />
  </cxf:properties>
</cxf:cxfEndpoint>
```

> **WARNING**
>
> You should only enable the **exceptionMessageCauseEnabled** flag for testing and diagnostic purposes. It is normal practice for servers to conceal the original cause of an exception to make it harder for hostile users to probe the server.

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**

Registry Implementation	Camel Component with Registry Plug-In
Guice bean registry	**camel-guice**
Blueprint bean registry	**camel-blueprint**
OSGi service registry	deployed in *OSGi container*
JNDI registry	

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 **MyBeanBrocessor** 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 30, *The Simple Language*). 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 Spring bean from 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.

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

For a slight efficiency gain, you can set the **cache** option to **true**, which avoids looking up the registry every time a bean is used. For example, to enable caching, you can set the **cache** attribute on the **bean** element as follows:

```
<bean ref="myBeanId" method="processBody" cache="true"/>
```

Accessing a Spring bean from Java

When you create an object instance using the Spring **bean** element, you can reference it from Java 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");
```

Alternatively, you can reference the Spring bean by injection, using the **@BeanInject** annotation as follows:

```
// Java
import org.apache.camel.@BeanInject;
...
public class MyRouteBuilder extends RouteBuilder {

  @BeanInject("myBeanId")
  com.acme.MyBeanProcessor bean;

  public void configure() throws Exception {
    ..
  }
}
```

If you omit the bean ID from the **@BeanInject** annotation, Camel looks up the registry by type, but this only works if there is just a single bean of the given type. For example, to look up and inject the bean of **com.acme.MyBeanProcessor** type:

```
@BeanInject
com.acme.MyBeanProcessor bean;
```

Bean shutdown order in Spring XML

For the beans used by a Camel context, the correct shutdown order is usually:

1. Shut down the **camelContext** instance, followed by;

2. Shut down the used beans.

If this shutdown order is reversed, then it could happen that the Camel context tries to access a bean that is already destroyed (either leading directly to an error; or the Camel context tries to create the missing bean while it is being destroyed, which also causes an error). The default shutdown order in Spring XML depends on the order in which the beans and the **camelContext** appear in the Spring XML file. In order to avoid random errors due to incorrect shutdown order, therefore, the **camelContext** is configured to shut down *before* any of the other beans in the Spring XML file. This is the default behaviour since Apache Camel 2.13.0.

If you need to change this behaviour (so that the Camel context is *not* forced to shut down before the other beans), you can set the **shutdownEager** attribute on the **camelContext** element to **false**. In this case, you could potentially exercise more fine-grained control over shutdown order using the Spring **depends-on** attribute.

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?
@Attachments	Binds to a list of attachments.	

Annotation	Meaning	Parameter?
@Body	Binds to an inbound message body.	
@Header	Binds to an inbound message header.	String name of the header.
@Headers	Binds to a **java.util.Map** of the inbound message headers.	
@OutHeaders	Binds to a **java.util.Map** of the outbound message headers.	
@Property	Binds to a named exchange property.	String name of the property.
@Properties	Binds to a **java.util.Map** 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
// 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
@Bean	Injects a Bean expression.
@Constant	Injects a Constant expression
@EL	Injects an EL expression.
@Groovy	Injects a Groovy expression.
@Header	Injects a Header expression.
@JavaScript	Injects a JavaScript expression.
@OGNL	Injects an OGNL expression.
@PHP	Injects a PHP expression.
@Python	Injects a Python expression.
@Ruby	Injects a Ruby expression.
@Simple	Injects a Simple expression.
@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
// 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
// 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 **MyIdGenerator** 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.
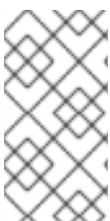
> **NOTE**
>
> Some of the language annotations are available in the core component (**@Bean**, **@Constant**, **@Simple**, and **@XPath**). For non-core components, however, you will have to make sure that you load the relevant component. For example, to use the OGNL script, you must load the **camel-ognl** 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. CREATING EXCHANGE INSTANCES

Overview

When processing messages with Java code (for example, in a bean class or in a processor class), it is often necessary to create a fresh exchange instance. If you need to create an **Exchange** object, the easiest approach is to invoke the methods of the **ExchangeBuilder** class, as described here.

ExchangeBuilder class

The fully qualified name of the **ExchangeBuilder** class is as follows:

```
org.apache.camel.builder.ExchangeBuilder
```

The **ExchangeBuilder** exposes the static method, **anExchange**, which you can use to start building an exchange object.

Example

For example, the following code creates a new exchange object containing the message body string, **Hello World!**, and with headers containing username and password credentials:

```
// Java
import org.apache.camel.Exchange;
import org.apache.camel.builder.ExchangeBuilder;
...
Exchange exch = ExchangeBuilder.anExchange(camelCtx)
            .withBody("Hello World!")
            .withHeader("username", "jdoe")
            .withHeader("password", "pass")
            .build();
```

ExchangeBuilder methods

The **ExchangeBuilder** class supports the following methods:

ExchangeBuilder anExchange(CamelContext context)

  (static method) Initiate building an exchange object.

Exchange build()

  Build the exchange.

ExchangeBuilder withBody(Object body)

  Set the message body on the exchange (that is, sets the exchange's *In* message body).

ExchangeBuilder withHeader(String key, Object value)

  Set a header on the exchange (that is, sets a header on the exchange's *In* message).

ExchangeBuilder withPattern(ExchangePattern pattern)

Sets the exchange pattern on the exchange.

ExchangeBuilder withProperty(String key, Object value)

Sets a property on the exchange.

2.6. TRANSFORMING MESSAGE CONTENT

Abstract

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.

2.6.1. Simple Message Transformations

Overview

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.

API for simple transformations

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 convertBodyTo(Class type)	Converts the IN message body to the specified type.
Type removeFaultHeader(String name)	Adds a processor which removes the header on the FAULT message.
Type removeHeader(String name)	Adds a processor which removes the header on the IN message.
Type removeProperty(String name)	Adds a processor which removes the exchange property.
ExpressionClause<ProcessorDefinition<Type>> setBody()	Adds a processor which sets the body on the IN message.
Type setFaultBody(Expression expression)	Adds a processor which sets the body on the FAULT message.
Type setFaultHeader(String name, Expression expression)	Adds a processor which sets the header on the FAULT message.
ExpressionClause<ProcessorDefinition<Type>> setHeader(String name)	Adds a processor which sets the header on the IN message.
Type setHeader(String name, Expression expression)	Adds a processor which sets the header on the IN message.
ExpressionClause<ProcessorDefinition<Type>> setOutHeader(String name)	Adds a processor which sets the header on the OUT message.
Type setOutHeader(String name, Expression expression)	Adds a processor which sets the header on the OUT message.
ExpressionClause<ProcessorDefinition<Type>> setProperty(String name)	Adds a processor which sets the exchange property.
Type setProperty(String name, Expression expression)	Adds a processor which sets the exchange property.
ExpressionClause<ProcessorDefinition<Type>> transform()	Adds a processor which sets the body on the OUT message.
Type transform(Expression expression)	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
static <E extends Exchange> ValueBuilder<E> body()	Returns a predicate and value builder for the inbound body on an exchange.
static <E extends Exchange,T> ValueBuilder<E> bodyAs(Class<T> type)	Returns a predicate and value builder for the inbound message body as a specific type.
static <E extends Exchange> ValueBuilder<E> constant(Object value)	Returns a constant expression.
static <E extends Exchange> ValueBuilder<E> faultBody()	Returns a predicate and value builder for the fault body on an exchange.
static <E extends Exchange,T> ValueBuilder<E> faultBodyAs(Class<T> type)	Returns a predicate and value builder for the fault message body as a specific type.
static <E extends Exchange> ValueBuilder<E> header(String name)	Returns a predicate and value builder for headers on an exchange.
static <E extends Exchange> ValueBuilder<E> outBody()	Returns a predicate and value builder for the outbound body on an exchange.
static <E extends Exchange> ValueBuilder<E> outBodyAs(Class<T> type)	Returns a predicate and value builder for the outbound message body as a specific type.
static ValueBuilder property(String name)	Returns a predicate and value builder for properties on an exchange.
static ValueBuilder regexReplaceAll(Expression content, String regex, Expression replacement)	Returns an expression that replaces all occurrences of the regular expression with the given replacement.
static ValueBuilder regexReplaceAll(Expression content, String regex, String replacement)	Returns an expression that replaces all occurrences of the regular expression with the given replacement.
static ValueBuilder sendTo(String uri)	Returns an expression processing the exchange to the given endpoint uri.
static <E extends Exchange> ValueBuilder<E> systemProperty(String name)	Returns an expression for the given system property.

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

Method	Description
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 occurrencies of the regular expression with the given replacement.
ValueBuilder<E> regexReplaceAll(String regex, String replacement)	Replaces all occurrencies 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.
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.

2.6.2. Marshalling and Unmarshalling

Java DSL commands

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.

Data formats

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

- Java serialization

- JAXB

- XMLBeans

- XStream

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("GeneratedPackageName");
```

```
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://www.xml.com/pub/a/2004/08/18/xstream.html). 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.

```
from("SourceURL").unmarshal().xstream()
.<FurtherProcessing>.to("TargetURL");
```
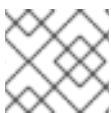
> **NOTE**
>
> The XStream data format is currently *not* supported in Spring XML.

2.6.3. Endpoint Bindings

What is a binding?

In Apache Camel, a binding is a way of wrapping an endpoint in a contract—for example, by applying a Data Format, a Content Enricher or a validation step. A condition or transformation is applied to the messages coming in, and a complementary condition or transformation is applied to the messages going out.

DataFormatBinding

The **DataFormatBinding** class is useful for the specific case where you want to define a binding that marshals and unmarshals a particular data format (see Section 2.6.2, "Marshalling and Unmarshalling"). In this case, all that you need to do to create a binding is to create a **DataFormatBinding** instance, passing a reference to the relevant data format in the constructor.

For example, the XML DSL snippet in Example 2.2, "JAXB Binding" shows a binding (with ID, **jaxb**) that is capable of marshalling and unmarshalling the JAXB data format when it is associated with an Apache Camel endpoint:

Example 2.2. JAXB Binding

```
<beans ... >
  ...
  <bean id="jaxb" class="org.apache.camel.processor.binding.DataFormatBinding">
    <constructor-arg ref="jaxbformat"/>
  </bean>

  <bean id="jaxbformat" class="org.apache.camel.model.dataformat.JaxbDataFormat">
    <property name="prettyPrint" value="true"/>
    <property name="contextPath" value="org.apache.camel.example"/>
  </bean>

</beans>
```

Associating a binding with an endpoint

The following alternatives are available for associating a binding with an endpoint:

- the section called "Binding URI"

- the section called "BindingComponent"

Binding URI

To associate a binding with an endpoint, you can prefix the endpoint URI with **binding:*NameOfBinding***, where ***NameOfBinding*** is the bean ID of the binding (for example, the ID of a binding bean created in Spring XML).

For example, the following example shows how to associate ActiveMQ endpoints with the JAXB binding defined in Example 2.2, "JAXB Binding".

```
<beans ...>
```

```
...
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="binding:jaxb:activemq:orderQueue"/>
    <to uri="binding:jaxb:activemq:otherQueue"/>
  </route>
</camelContext>
...
</beans>
```

BindingComponent

Instead of using a prefix to associate a binding with an endpoint, you can make the association implicit, so that the binding does not need to appear in the URI. For existing endpoints that do not have an implicit binding, the easiest way to achieve this is to wrap the endpoint using the **BindingComponent** class.

For example, to associate the **jaxb** binding with **activemq** endpoints, you could define a new **BindingComponent** instance as follows:

```
<beans ... >
  ...
  <bean id="jaxbmq" class="org.apache.camel.component.binding.BindingComponent">
    <constructor-arg ref="jaxb"/>
    <constructor-arg value="activemq:foo."/>
  </bean>

  <bean id="jaxb" class="org.apache.camel.processor.binding.DataFormatBinding">
    <constructor-arg ref="jaxbformat"/>
  </bean>

  <bean id="jaxbformat" class="org.apache.camel.model.dataformat.JaxbDataFormat">
    <property name="prettyPrint" value="true"/>
    <property name="contextPath" value="org.apache.camel.example"/>
  </bean>

</beans>
```

Where the (optional) second constructor argument to **jaxbmq** defines a URI prefix. You can now use the **jaxbmq** ID as the scheme for an endpoint URI. For example, you can define the following route using this binding component:

```
<beans ...>
  ...
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="jaxbmq:firstQueue"/>
      <to uri="jaxbmq:otherQueue"/>
    </route>
  </camelContext>
  ...
</beans>
```

The preceding route is equivalent to the following route, which uses the binding URI approach:

```
<beans ...>
    ...
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="binding:jaxb:activemq:foo.firstQueue"/>
            <to uri="binding:jaxb:activemq:foo.otherQueue"/>
        </route>
    </camelContext>
    ...
</beans>
```

> **NOTE**
>
> For developers that implement a custom Apache Camel component, it is possible to achieve this by implementing an endpoint class that inherits from the **org.apache.camel.spi.HasBinding** interface.

BindingComponent constructors

The **BindingComponent** class supports the following constructors:

public BindingComponent()

No arguments form. Use property injection to configure the binding component instance.

public BindingComponent(Binding binding)

Associate this binding component with the specified **Binding** object, **binding**.

public BindingComponent(Binding binding, String uriPrefix)

Associate this binding component with the specified **Binding** object, **binding**, and URI prefix, **uriPrefix**. This is the most commonly used constructor.

public BindingComponent(Binding binding, String uriPrefix, String uriPostfix)

This constructor supports the additional URI post-fix, **uriPostfix**, argument, which is automatically appended to any URIs defined using this binding component.

Implementing a custom binding

In addition to the **DataFormatBinding**, which is used for marshalling and unmarshalling data formats, you can implement your own custom bindings. Define a custom binding as follows:

1. Implement an **org.apache.camel.Processor** class to perform a transformation on messages incoming to a consumer endpoint (appearing in a **from** element).

2. Implement a complementary **org.apache.camel.Processor** class to perform the reverse transformation on messages outgoing from a producer endpoint (appearing in a **to** element).

3. Implement the **org.apache.camel.spi.Binding** interface, which acts as a factory for the processor instances.

Binding interface

Example 2.3, "The org.apache.camel.spi.Binding Interface" shows the definition of the
org.apache.camel.spi.Binding interface, which you must implement to define a custom binding.

Example 2.3. The org.apache.camel.spi.Binding Interface

```java
// Java
package org.apache.camel.spi;

import org.apache.camel.Processor;

/**
 * Represents a <a href="http://camel.apache.org/binding.html">Binding</a> or contract
 * which can be applied to an Endpoint; such as ensuring that a particular
 * <a href="http://camel.apache.org/data-format.html">Data Format</a> is used on messages in
and out of an endpoint.
 */
public interface Binding {

    /**
     * Returns a new {@link Processor} which is used by a producer on an endpoint to implement
     * the producer side binding before the message is sent to the underlying endpoint.
     */
    Processor createProduceProcessor();

    /**
     * Returns a new {@link Processor} which is used by a consumer on an endpoint to process the
     * message with the binding before its passed to the endpoint consumer producer.
     */
    Processor createConsumeProcessor();
}
```

When to use bindings

Bindings are useful when you need to apply the same kind of transformation to many different kinds of
endpoint.

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
```

> **NOTE**
>
> Property Placeholders support an encoding option that enables you to read the
> **.properties** file, using a specific character set such as UTF-8. However, by default, it
> implements the ISO-8859-1 character set.

Apache Camel using the **PropertyPlaceholders** support the following:

- Specify the default value together with the key to lookup.

- No need to define the **PropertiesComponent**, if all the placeholder keys consist of default
  values, which are to be used.

- Use third-party functions to lookup the property values. It enables you to implement your own
  logic.

> **NOTE**
>
> Provide three out of the box functions to lookup values from OS environmental
> variable, JVM system properties, or the service name idiom.

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 2.4, "Sample Property File".

Example 2.4. 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:

```
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 **propertyPlacholder** 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).

Additionally, if you want the properties component to ignore any missing locations that are specified using Java system properties or O/S environment variables, you can set the **ignoreMissingLocation** option to **true**.

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 2.4, "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 placholder syntax also uses double braces, **{{*Key*}}**. For example, you could define a **jmxAgent** element using property placeholders, as follows:

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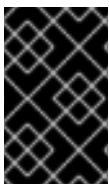
> **IMPORTANT**
>
> The **prop** prefix must be explicitly assigned to the **http://camel.apache.org/schema/placeholder** namespace in your Spring file, as shown in the **beans** element of the preceding example.

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}?");
```

You can specify a default value for the property, using the syntax, **${properties:*Key*:*DefaultVal*}**. For example:

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

It is also possible to override the location of the property file using the syntax, **${properties-location:*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-location:com/mycompany/bar.properties:bar.quote}.");
```

Using Property Placeholders in the XML DSL

In older releases, the **xs:string** type attributes were used to support placeholders in the XML DSL. For example, the timeout attribute would be a **xs:int** type. Therefore, you cannot set a string value as the placeholder key.

From Apache Camel 2.7 onwards, this is now possible by using a special placeholder namespace. The following example illustrates the prop prefix for the namespace. It enables you to use the prop prefix in the attributes in the XML DSLs.

> **NOTE**
>
> In the Multicast, set the option **stopOnException** as the value of the placeholder with the key **stop**. Also, in the properties file, define the value as
>
> ```
> stop=true
> ```

```xml
<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"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
    ">

    <!-- Notice in the declaration above, we have defined the prop prefix as the Camel placeholder
namespace -->

    <bean id="damn" class="java.lang.IllegalArgumentException">
        <constructor-arg index="0" value="Damn"/>
    </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"/>
            <!-- use prop namespace, to define a property placeholder, which maps to
                option stopOnException={{stop}} -->
            <multicast prop:stopOnException="stop">
                <to uri="mock:a"/>
                <throwException ref="damn"/>
                <to uri="mock:b"/>
            </multicast>
        </route>

    </camelContext>

</beans>
```

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
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <!-- OSGI blueprint property placeholder -->
    <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint">
        <!-- list some properties 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
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <!-- OSGI blueprint property placeholder -->
    <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint">
        <!-- list some properties 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: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/context
  http://www.springframework.org/schema/context/spring-context.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>
```

> **NOTE**
>
> Alternatively, you can set the **location** attribute of the
> **BridgePropertyPlaceholderConfigurer** to point at a Spring properties file. The Spring
> properties file syntax is fully supported.

2.8. THREADING MODEL

Java thread pool API

The Apache Camel threading model is based on the powerful Java concurrency API, Package
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 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	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
multicast	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
recipientList	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef
split	parallelProcessing() executorService() executorServiceRef()	@parallelProcessing @executorServiceRef

Processor	Java DSL	XML DSL
threads	executorService() executorServiceRef() poolSize() maxPoolSize() keepAliveTime() timeUnit() maxQueueSize() rejectedPolicy()	@executorServiceRef @poolSize @maxPoolSize @keepAliveTime @timeUnit @maxQueueSize @rejectedPolicy
wireTap	wireTap(String uri, ExecutorService executorService) wireTap(String uri, String executorServiceRef)	@executorServiceRef

threads DSL options

The **threads** processor is a general-purpose DSL command, which you can use to introduce a thread pool into a route. It supports the following options to customize the thread pool:

poolSize()

Minimum number of threads in the pool (and initial pool size).

maxPoolSize()

Maximum number of threads in the pool.

keepAliveTime()

If any threads are idle for longer than this period of time (specified in seconds), they are terminated.

timeUnit()

Time unit for keep alive, specified using the **java.util.concurrent.TimeUnit** type.

maxQueueSize()

Maximum number of pending tasks that this thread pool can store in its incoming task queue.

rejectedPolicy()

Specifies what course of action to take, if the incoming task queue is full. See Table 2.10, "Thread Pool Builder Options"

> **NOTE**
>
> The preceding thread pool options are *not* compatible with the **executorServiceRef** option (for example, you cannot use these options to override the settings in the thread pool referenced by an **executorServiceRef** option). Apache Camel validates the DSL to enforce this.

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**

Thread Option	Default Value
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.

Builder Option	Description
rejectedPolicy()	Specifies what course of action to take, if the incoming task queue is full. You can specify four possible values: **CallerRuns** *(Default value)* 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. **Abort** Aborts the latest incoming task by throwing an exception. **Discard** Quietly discards the latest incoming task. **DiscardOldest** Discards the oldest unhandled task and then attempts to enqueue the latest incoming task in the task queue.
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>
```

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>
```

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 see *File2* in the *Apache Camel Component Reference Guide*. and the *Ftp2* in the *Apache Camel Component Reference Guide* both expose the **scheduledExecutorService** property, which you can use to specify the component's **ExecutorService** object.

Customizing thread names

To make the application logs more readable, it is often a good idea to customize the thread names (which are used to identify threads in the log). To customize thread names, you can configure the *thread name pattern* by calling the **setThreadNamePattern** method on the **ExecutorServiceStrategy** class or the **ExecutorServiceManager** class. Alternatively, an easier way to set the thread name pattern is to set the **threadNamePattern** property on the **CamelContext** object.

The following placeholders can be used in a thread name pattern:

#camelId#

The name of the current **CamelContext**.

#counter#

A unique thread identifier, implemented as an incrementing counter.

#name#

The regular Camel thread name.

#longName#

The long thread name—which can include endpoint parameters and so on.

The following is a typical example of a thread name pattern:

```
Camel (#camelId#) thread #counter# - #name#
```

The following example shows how to set the **threadNamePattern** attribute on a Camel context using XML DSL:

```
<camelContext xmlns="http://camel.apache.org/schema/spring"
        threadNamePattern="Riding the thread #counter#" >
  <route>
    <from uri="seda:start"/>
    <to uri="log:result"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

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 2.5. 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 2.6. 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 finshed processing. The treatment of running tasks is configurable, however.

3. 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 2.5, "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 repititions. 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 2.7, "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 2.7. 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 2.8, "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 2.8. XML DSL Example of Simple Scheduled Route

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

<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
// 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.

Logging Inflight Exchanges on Timeout

If a graceful shutdown fails to shutdown cleanly within the given timeout period, then Apache Camel performs more aggressive shut down. It forces routes, threadpools etc to shutdown.

After the timeout, Apache Camel logs information about the current inflight exchanges. It logs the origin of the exchange and current route of exchange.

For example, the log below shows that there is one inflight exchange, that origins from route1 and is currently on the same route1 at the delay1 node.

During graceful shutdown, If you enable the DEBUG logging level on **org.apache.camel.impl.DefaultShutdownStrategy**, then it logs the same inflight exchange information.

```
2015-01-12 13:23:23,656 [- ShutdownTask] INFO DefaultShutdownStrategy - There are 1 inflight
exchanges:
InflightExchange: [exchangeId=ID-davsclaus-air-62213-1421065401253-0-3, fromRouteId=route1,
routeId=route1, nodeId=delay1, elapsed=2007, duration=2017]
```

If you do not want to see these logs, you can turn this off by setting the option **logInflightExchangesOnTimeout** to false.

```
context.getShutdownStrategegy().setLogInflightExchangesOnTimeout(false);
```

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**	*None*	Specifies the date and time when the route is started for the first time.
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**	*None*	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.

Parameter	Type	Default	Description
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
routeSuspendDate	**java.util.Date**	*None*	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**	*None*	Specifies the date and time when the route is resumed for the first time.

Parameter	Type	Default	Description
routeResumeRepeat Count	**int**	0	When set to a non-zero value, specifies how many times the route should be resumed.
routeResumeRepeatI nterval	**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 2.9, "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 2.9. 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 2.10, "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 2.10. XML DSL Example of a Cron Scheduled Route**
>
> ```xml
> <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
routeStartString	**String**	*None*	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
routeStopTime	**String**	*None*	Specifies a cron expression that triggers one or more route stop events.

Parameter	Type	Default	Description
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
routeSuspendTime	**String**	*None*	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
routeResumeTime	**String**	*None*	Specifies a cron expression that triggers one or more route resume events.

2.10.4. Route Policy Factory

Using Route Policy Factory

Available as of Camel 2.14

If you want to use a route policy for every route, you can use a **org.apache.camel.spi.RoutePolicyFactory** as a factory for creating a **RoutePolicy** instance for each route. This can be used when you want to use the same kind of route policy for every route. Then you need to only configure the factory once, and every route created will have the policy assigned.

There is API on CamelContext to add a factory, as shown below:

```
context.addRoutePolicyFactory(new MyRoutePolicyFactory());
```

From XML DSL you only define a **<bean>** with the factory

```
<bean id="myRoutePolicyFactory" class="com.foo.MyRoutePolicyFactory"/>
```

The factory contains the createRoutePolicy method for creating route policies.

```
/**
 * Creates a new {@link org.apache.camel.spi.RoutePolicy} which will be assigned to the given route.
 *
 * @param camelContext the camel context
 * @param routeId      the route id
 * @param route        the route definition
 * @return the created {@link org.apache.camel.spi.RoutePolicy}, or <tt>null</tt> to not use a policy
 for this route
 */
RoutePolicy createRoutePolicy(CamelContext camelContext, String routeId, RouteDefinition route);
```

Note you can have as many route policy factories as you want. Just call the **addRoutePolicyFactory** again, or declare the other factories as **<bean>** in XML.

2.11. ONCOMPLETION

Overview

The *OnCompletion* DSL name is used to define an action that is to take place when a **Unit of Work** is completed. A **Unit of Work** is a Camel concept that encompasses an entire exchange. See Section 43.1, "Exchanges". The **onCompletion** command has the following features:

- The scope of the **OnCompletion** command can be global or per route. A route scope overrides global scope.

- **OnCompletion** can be configured to be triggered on success for failure.

- The **onWhen** predicate can be used to only trigger the **onCompletion** in certain situations.

- You can define whether or not to use a thread pool, though the default is no thread pool.

Route Only Scope for onCompletion

When an **onCompletion** DSL is specified on an exchange, Camel spins off a new thread. This allows the original thread to continue without interference from the **onCompletion** task. A route will only support one **onCompletion**. In the following example, the **onCompletion** is triggered whether the exchange completes with success or failure. This is the default action.

```
from("direct:start")
    .onCompletion()
        // this route is only invoked when the original route is complete as a kind
        // of completion callback
        .to("log:sync")
        .to("mock:sync")
    // must use end to denote the end of the onCompletion route
```

```
    .end()
    // here the original route contiues
    .process(new MyProcessor())
    .to("mock:result");
```

For XML the format is as follows:

```
<route>
    <from uri="direct:start"/>
    <!-- this onCompletion block will only be executed when the exchange is done being routed -->
    <!-- this callback is always triggered even if the exchange failed -->
    <onCompletion>
        <!-- so this is a kinda like an after completion callback -->
        <to uri="log:sync"/>
        <to uri="mock:sync"/>
    </onCompletion>
    <process ref="myProcessor"/>
    <to uri="mock:result"/>
</route>
```

To trigger the **onCompletion** on failure, the **onFailureOnly** parameter can be used. Similarly, to trigger the **onCompletion** on success, use the **onCompleteOnly** parameter.

```
from("direct:start")
    // here we qualify onCompletion to only invoke when the exchange failed (exception or FAULT body)
    .onCompletion().onFailureOnly()
        .to("log:sync")
        .to("mock:sync")
    // must use end to denote the end of the onCompletion route
    .end()
    // here the original route continues
    .process(new MyProcessor())
    .to("mock:result");
```

For XML, **onFailureOnly** and **onCompleteOnly** are expressed as booleans on the **onCompletion** tag:

```
<route>
    <from uri="direct:start"/>
    <!-- this onCompletion block will only be executed when the exchange is done being routed -->
    <!-- this callback is only triggered when the exchange failed, as we have onFailure=true -->
    <onCompletion onFailureOnly="true">
        <to uri="log:sync"/>
        <to uri="mock:sync"/>
    </onCompletion>
    <process ref="myProcessor"/>
    <to uri="mock:result"/>
</route>
```

Global Scope for onCompletion

To define **onCompletion** for more than just one route:

```
// define a global on completion that is invoked when the exchange is complete
```

```
onCompletion().to("log:global").to("mock:sync");

from("direct:start")
    .process(new MyProcessor())
    .to("mock:result");
```

Using onWhen

To trigger the **onCompletion** under certain circumstances, use the **onWhen** predicate. The following example will trigger the **onCompletion** when the body of the message contains the word **Hello**:

```
/from("direct:start")
    .onCompletion().onWhen(body().contains("Hello"))
        // this route is only invoked when the original route is complete as a kind
        // of completion callback. And also only if the onWhen predicate is true
        .to("log:sync")
        .to("mock:sync")
    // must use end to denote the end of the onCompletion route
    .end()
    // here the original route contiues
    .to("log:original")
    .to("mock:result");
```

Using onCompletion with or without a thread pool

As of Camel 2.14, **onCompletion** will not use a thread pool by default. To force the use of a thread pool, either set an **executorService** or set **parallelProcessing** to true. For example, in Java DSL, use the following format:

```
onCompletion().parallelProcessing()
    .to("mock:before")
    .delay(1000)
    .setBody(simple("OnComplete:${body}"));
```

For XML the format is:

```
<onCompletion parallelProcessing="true">
  <to uri="before"/>
  <delay><constant>1000</constant></delay>
  <setBody><simple>OnComplete:${body}<simple></setBody>
</onCompletion>
```

Use the **executorServiceRef** option to refer to a specific thread pool:

```
<onCompletion executorServiceRef="myThreadPool"
  <to uri="before"/>
  <delay><constant>1000</constant></delay>
  <setBody><simple>OnComplete:${body}</simple></setBody>
</onCompletion>>
```

Run onCompletion before Consumer Sends Response

onCompletion can be run in two modes:

- AfterConsumer – The default mode which runs after the consumer is finished

- BeforeConsumer – Runs before the consumer writes a response back to the callee. This allows **onCompletion** to modify the Exchange, such as adding special headers, or to log the Exchange as a response logger.

For example, to add a **created by** header to the response, use **modeBeforeConsumer()** as shown below:

```
.onCompletion().modeBeforeConsumer()
    .setHeader("createdBy", constant("Someone"))
.end()
```

For XML, set the mode attribute to **BeforeConsumer**:

```
<onCompletion mode="BeforeConsumer">
  <setHeader headerName="createdBy">
   <constant>Someone</constant>
  </setHeader>
</onCompletion>
```

2.12. METRICS

Overview

Available as of Camel 2.14

While Camel provides a lot of existing metrics integration with Codahale metrics has been added for Camel routes. This allows end users to seamless feed Camel routing information together with existing data they are gathering using Codahale metrics.

To use the Codahale metrics you will need to:

1. Add camel-metrics component

2. Enable route metrics in XML or Java code

Note that performance metrics are only usable if you have a way of displaying them; any kind of monitoring tooling which can integrate with JMX can be used, as the metrics are available over JMX. In addition, the actual data is 100% Codehale JSON.

Metrics Route Policy

Obtaining Codahale metrics for a single route can be accomplished by defining a **MetricsRoutePolicy** on a per route basis.

From Java create an instance of **MetricsRoutePolicy** to be assigned as the route's policy. This is shown below:

```
from("file:src/data?noop=true").routePolicy(new MetricsRoutePolicy()).to("jms:incomingOrders");
```

From XML DSL you define a **<bean>** which is specified as the route's policy; for example:

```
<bean id="policy" class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicy"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route routePolicyRef="policy">
    <from uri="file:src/data?noop=true"/>
[...]
```

Metrics Route Policy Factory

This factory allows one to add a **RoutePolicy** for each route which exposes route utilization statistics using Codahale metrics. This factory can be used in Java and XML as the examples below demonstrate.

From Java you just add the factory to the **CamelContext** as shown below:

```
context.addRoutePolicyFactory(new MetricsRoutePolicyFactory());
```

And from XML DSL you define a **<bean>** as follows:

```
<!-- use camel-metrics route policy to gather metrics for all routes -->
<bean id="metricsRoutePolicyFactory"
class="org.apache.camel.component.metrics.routepolicy.MetricsRoutePolicyFactory"/>
```

From Java code you can get hold of the **com.codahale.metrics.MetricRegistry** from the **org.apache.camel.component.metrics.routepolicy.MetricsRegistryService** as shown below:

```
MetricRegistryService registryService = context.hasService(MetricsRegistryService.class);
if (registryService != null) {
  MetricsRegistry registry = registryService.getMetricsRegistry();
  ...
}
```

Options

The **MetricsRoutePolicyFactory** and **MetricsRoutePolicy** supports the following options:

Name	Default	Description
durationUnit	**TimeUnit.MILLISECONDS**	The unit to use for duration in the metrics reporter or when dumping the statistics as json.
jmxDomain	**org.apache.camel.metrics**	The JXM domain name.
metricsRegistry		Allow to use a shared **com.codahale.metrics.Metric Registry**. If none is provided then Camel will create a shared instance used by the this CamelContext.

prettyPrint	**false**	Whether to use pretty print when outputting statistics in json format.
rateUnit	**TimeUnit.SECONDS**	The unit to use for rate in the metrics reporter or when dumping the statistics as json.
useJmx	**false**	Whether to report fine grained statistics to JMX by using the **com.codahale.metrics.JmxReporter**. Notice that if JMX is enabled on CamelContext then a **MetricsRegistryService** mbean is enlisted under the services type in the JMX tree. That mbean has a single operation to output the statistics using json. Setting useJmx to true is only needed if you want fine grained mbeans per statistics type.

2.13. 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 deployed in an OSGi bundle is equal to the *OSGi symbolic name* of the bundle. For example, if the OSGi symbolic name is **MyCamelBundle**, the JMX name would be **MyCamelBundle**. In cases where there is more than one **CamelContext** in the bundle, the JMX name is disambiguated by adding a counter value as a suffix. For example, if there are multiple Camel contexts in the **MyCamelBundle** bundle, the corresponding JMX MBeans are named as follows:

```
MyCamelBundle-1
MyCamelBundle-2
MyCamelBundle-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 *(OSGi only)*.
#symbolicName#	The OSGi symbolic name *(OSGi only)*.
#version#	The OSGi bundle version *(OSGi only)*.

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.

2.14. PERFORMANCE AND OPTIMIZATION

Avoid unnecessary message copying

You can avoid making an unnecessary copy of the original message, by setting the **allowUseOriginalMessage** option to **false** on the **CamelContext** object. For example, in Blueprint XML you can set this option as follows:

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

You can safely set **allowUseOriginalMessage** to **false**, if the following conditions are satisfied:

- You do not set **useOriginalMessage=true** on any of the error handlers or on the **onException** element.

- You do not use the **getOriginalMessage** method anywhere in your Java application code.

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
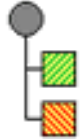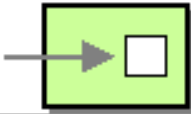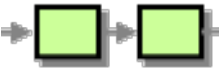
Enterprise Integration Patterns book

Apache Camel supports most of the patterns from the book, Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf.
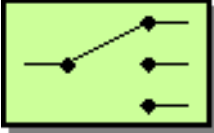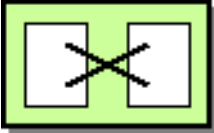
Messaging systems

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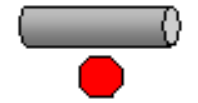
Icon	Name	Use Case
	Message Router	How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of defined conditions?
	Message Translator	How do systems using different data formats communicate with each other using messaging?

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.
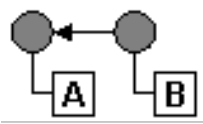
Table 3.2. Messaging Channels

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.
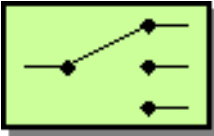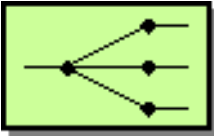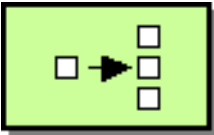
Table 3.3. Message Construction
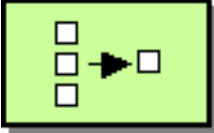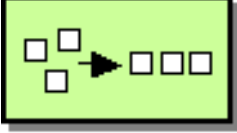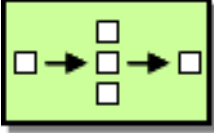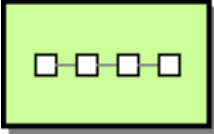
Icon	Name	Use Case
	Correlation Identifier	How does a requestor identify the request that generated the received reply?
	Return Address	How does a replier know where to send the reply?

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?
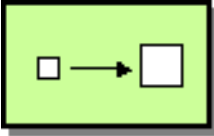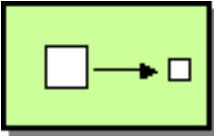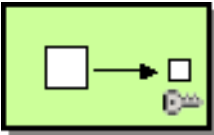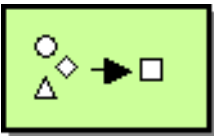
Icon	Name	Use Case
	Aggregator	How do we combine the results of individual, but related messages so that they can be processed as a whole?
	Resequencer	How can we get a stream of related, but out-of-sequence, messages back into the correct order?
	Composed Message Processor	How can you maintain the overall 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
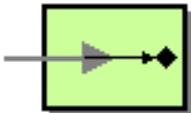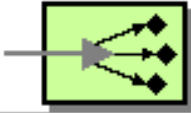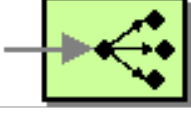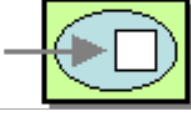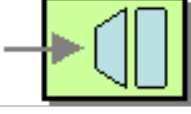
Icon	Name	Use Case
	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?
	Content Filter	How do you simplify dealing with a large message, when you are interested in only a few data items?
	Claim Check	How can we reduce the data volume of messages 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?
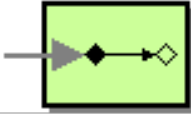
Messaging endpoints

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
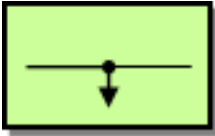
Icon	Name	Use Case
	Messaging Mapper	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	Event Driven Consumer	How can an application automatically consume messages as they become available?
	Polling Consumer	How can an application 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. DEFINING REST SERVICES

Abstract

Apache Camel supports multiple approaches to defining REST services. In particular, Apache Camel provides the REST DSL (Domain Specific Language), which is a simple but powerful fluent API that can be layered over any REST component and provides integration with Swagger.

4.1. OVERVIEW OF REST IN CAMEL

Overview

Apache Camel provides many different approaches and components for defining REST services in your Camel applications. This section provides a quick overview of these different approaches and components, so that you can decide which implementation and API best suits your requirements.

What is REST?

Representational State Transfer (REST) is an architecture for distributed applications that centers around the transmission of data over HTTP, using only the four basic HTTP verbs: **GET**, **POST**, **PUT**, and **DELETE**.

In contrast to a protocol such as SOAP, which treats HTTP as a mere transport protocol for SOAP messages, the REST architecture exploits HTTP directly. The key insight is that the HTTP protocol *itself*, augmented by a few simple conventions, is eminently suitable to serve as the framework for distributed applications.

A sample REST invocation

Because the REST architecture is built around the standard HTTP verbs, in many cases you can use a regular browser as a REST client. For example, to invoke a simple *Hello World* REST service running on the host and port, **localhost:9091**, you could navigate to a URL like the following in your browser:

```
http://localhost:9091/say/hello/Garp
```

The *Hello World* REST service might then return a response string, such as:

```
Hello Garp
```

Which gets displayed in your browser window. The ease with which you can invoke REST services, using nothing more than a standard browser (or the **curl** command-line utility), is one of the many reasons why the REST protocol has rapidly gained popularity.

REST wrapper layers

The following REST wrapper layers offer a simplified syntax for defining REST services and can be layered on top of different REST implementations:

REST DSL

> The REST DSL (in **camel-core**) is a facade or wrapper layer that provides a simplified builder API for defining REST services. The REST DSL does *not* itself provide a REST implementation: it must be

combined with an underlying REST implementation. For example, the following Java code shows how to define a simple *Hello World* service using the REST DSL:

```
rest("/say")
    .get("/hello/{name}").route().transform().simple("Hello ${header.name}");
```

For more details, see Section 4.2, "Defining Services with REST DSL" .

Rest component

The Rest component (in **camel-core**) is a wrapper layer that enables you to define REST services using a URI syntax. Like the REST DSL, the Rest component does *not* itself provide a REST implementation: it must be combined with an underlying REST implementation. For example, the following Java code shows how to define a simple *Hello World* service using the Rest component:

```
from("rest:get:say:/hello/{name}").transform().simple("Hello ${header.name}");
```

REST implementations

Apache Camel provides several different REST implementations, through the following components:

Spark–Rest component

The Spark-Rest component (in **camel-spark-rest**) is a REST implementation that enables you to define REST services using a URI syntax. The Spark framework itself is a Java API, which is loosely based on the Sinatra framework (a Python API). For example, the following Java code shows how to define a simple *Hello World* service using the Spark–Rest component:

```
from("spark-rest:get:/say/hello/:name").transform().simple("Hello ${header.name}");
```

Notice that, in contrast to the Rest component, the syntax for a variable in the URI is **:name** instead of **{name}**.
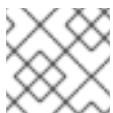
> **NOTE**
>
> The Spark–Rest component requires Java 8.

Restlet component

The Restlet component (in **camel-restlet**) is a REST implementation that can, in principle, be layered above different transport protocols (although this component is only tested against the HTTP protocol). This component also provides an integration with the Restlet Framework, which is a commercial framework for developing REST services in Java. For example, the following Java code shows how to define a simple *Hello World* service using the Restlet component:

```
from("restlet:http://0.0.0.0:9091/say/hello/{name}?restletMethod=get")
    .transform().simple("Hello ${header.name}");
```

For more details, see *Restlet* in the *Apache Camel Component Reference Guide* .

Servlet component

The Servlet component (in **camel-servlet**) is a component that binds a Java servlet to a Camel route. In other words, the Servlet component enables you to package and deploy a Camel route as if

it was a standard Java servlet. The Servlet component is therefore particularly useful, *if you need to deploy a Camel route inside a servlet container* (for example, into an Apache Tomcat HTTP server or into a JBoss Enterprise Application Platform container).

The Servlet component on its own, however, does not provide any convenient REST API for defining REST services. The easiest way to use the Servlet component, therefore, is to combine it with the REST DSL, so that you can define REST services with a user-friendly API.

For more details, see *Servlet* in the *Apache Camel Component Reference Guide*.

JAX-RS REST implementation

JAX-RS (Java API for RESTful Web Services) is a framework for binding REST requests to Java objects, where the Java classes must be decorated with JAX-RS annotations in order to define the binding. The JAX-RS framework is relatively mature and provides a sophisticated framework for developing REST services, but it is also somewhat complex to program.

The JAX-RS integration with Apache Camel is implemented by the CXFRS component, which is layered over Apache CXF. In outline, JAX-RS binds a REST request to a Java class using the following annotations (where this is only an incomplete sample of the many available annotations):

@**Path**

Annotation that can map a context path to a Java class or map a sub-path to a particular Java method.

@**GET,** @**POST,** @**PUT,** @**DELETE**

Annotations that map a HTTP method to a Java method.

@**PathParam**

Annotation that either maps a URI parameter to a Java method argument, or injects a URI parameter into a field.

@**QueryParam**

Annotation that either maps a query parameter to a Java method argument, or injects a query parameter into a field.

The body of a REST request or REST response is normally expected to be in JAXB (XML) data format. But Apache CXF also supports conversion of JSON format to JAXB format, so that JSON messages can also be parsed.

For more details, see *CXFRS* in the *Apache Camel Component Reference Guide* and part "Developing RESTful Web Services" in "Apache CXF Development Guide".

> **NOTE**
>
> The CXFRS component is *not* integrated with the REST DSL.

4.2. DEFINING SERVICES WITH REST DSL

REST DSL is a facade

The REST DSL is effectively a *facade* that provides a simplified syntax for defining REST services in a Java DSL or an XML DSL (Domain Specific Language). REST DSL does not actually provide the REST implementation, it is just a wrapper around an *existing* REST implementation (of which there are several in Apache Camel).

Advantages of the REST DSL

The REST DSL wrapper layer offers the following advantages:

- A modern easy-to-use syntax for defining REST services.

- Compatible with multiple different Apache Camel components.

- Swagger integration (through the **camel-swagger** component).

Components that integrate with REST DSL

Because the REST DSL is not an actual REST implementation, one of the first things you need to do is to choose a Camel component to provide the underlying implementation. The following Camel components are currently integrated with the REST DSL:

- Servlet component (**camel-servlet**).

- Spark-Rest component (**camel-spark-rest**).

- Netty HTTP component (**camel-netty-http**).

- Netty4 HTTP component (**camel-netty4-http**).

- Jetty component (**camel-jetty**).

- Restlet component (**camel-restlet**).

> **NOTE**
>
> The Rest component (part of **camel-core**) is not a REST implementation. Like the REST DSL, the Rest component is a facade, providing a simplified syntax to define REST services using a URI syntax. The Rest component also requires an underlying REST implementation.

Configuring REST DSL to use a REST implementation

To specify the REST implementation, you use either the **restConfiguration()** builder (in Java DSL) or the **restConfiguration** element (in XML DSL). For example, to configure REST DSL to use the Spark-Rest component, you would use a builder expression like the following in the Java DSL:

```
restConfiguration().component("spark-rest").port(9091);
```

And you would use an element like the following (as a child of **camelContext**) in the XML DSL:

```
<restConfiguration component="spark-rest" port="9091"/>
```

Syntax

The Java DSL syntax for defining a REST service is as follows:

```
rest("BasePath").Option()+.
    .Verb("Path").Option()+.[to() | route().CamelRoute.endRest()]
    .Verb("Path").Option()+.[to() | route().CamelRoute.endRest()]
    ...
    .Verb("Path").Option()+.[to() | route().CamelRoute];
```

Where **CamelRoute** is an optional embedded Camel route (defined using the standard Java DSL syntax for routes).

The REST service definition starts with the **rest()** keyword, followed by one or more verb clauses that handle specific URL path segments. The HTTP verb can be one of **get()**, **head()**, **put()**, **post()**, **delete()**, or **verb()**. Each verb clause can use either of the following syntaxes:

- Verb clause ending in **to()** keyword. For example:

  ```
  get("...").Option()+.to("...")
  ```

- Verb clause ending in **route()** keyword (for embedding a Camel route). For example:

  ```
  get("...").Option()+.route("...").CamelRoute.endRest()
  ```

REST DSL with Java

In Java, to define a service with the REST DSL, put the REST definition into the body of a **RouteBuilder.configure()** method, just like you do for regular Apache Camel routes. For example, to define a simple *Hello World* service using the REST DSL with the Spark-Rest component, define the following Java code:

```
restConfiguration().component("spark-rest").port(9091);

rest("/say")
    .get("/hello").to("direct:hello")
    .get("/bye").to("direct:bye");

from("direct:hello")
    .transform().constant("Hello World");
from("direct:bye")
    .transform().constant("Bye World");
```

The preceding example features three different kinds of builder:

restConfiguration()

Configures the REST DSL to use a specific REST implementation (Spark-Rest).

rest()

Defines a service using the REST DSL. Each of the verb clauses are terminated by a **to()** keyword, which forwards the incoming message to a **direct** endpoint (the **direct** component splices routes together within the same application).

from()

Defines a regular Camel route.

REST DSL with XML

In XML, to define a service with the XML DSL, define a **rest** element as a child of the **camelContext** element. For example, to define a simple *Hello World* service using the REST DSL with the Spark-Rest component, define the following XML code (in Blueprint):

```xml
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <restConfiguration component="spark-rest" port="9091"/>

  <rest path="/say">
    <get uri="/hello">
      <to uri="direct:hello"/>
    </get>
    <get uri="/bye">
      <to uri="direct:bye"/>
    </get>
  </rest>

  <route>
    <from uri="direct:hello"/>
    <transform>
      <constant>Hello World</constant>
    </transform>
  </route>
  <route>
    <from uri="direct:bye"/>
    <transform>
      <constant>Bye World</constant>
    </transform>
  </route>
</camelContext>
```

Specifying a base path

The **rest()** keyword (Java DSL) or the **path** attribute of the **rest** element (XML DSL) allows you to define a base path, which is then prefixed to the paths in all of the verb clauses. For example, given the following snippet of Java DSL:

```java
rest("/say")
    .get("/hello").to("direct:hello")
    .get("/bye").to("direct:bye");
```

Or given the following snippet of XML DSL:

```xml
<rest path="/say">
  <get uri="/hello">
    <to uri="direct:hello"/>
  </get>
  <get uri="/bye" consumes="application/json">
```

```
      <to uri="direct:bye"/>
    </get>
</rest>
```

The REST DSL builder gives you the following URL mappings:

```
/say/hello
/say/bye
```

The base path is optional. If you prefer, you could (less elegantly) specify the full path in each of the verb clauses:

```
rest()
    .get("/say/hello").to("direct:hello")
    .get("/say/bye").to("direct:bye");
```

Using Dynamic To

The REST DSL supports the **toD** dynamic to parameter. Use this parameter to specify URIs.

For example, in JMS a dynamic endpoint URI could be defined in the following way:

```
public void configure() throws Exception {
    rest("/say")
        .get("/hello/{language}").toD("jms:queue:hello-${header.language}");
}
```

In XML DSL, the same details would look like this:

```
<rest uri="/say">
  <get uri="/hello//{language}">
    <toD uri="jms:queue:hello-${header.language}"/>
  </get>
<rest>
```

For more information about the **toD** dynamic to parameter, see the section called "Dynamic To".

URI templates

In a verb argument, you can specify a URI template, which enables you to capture specific path segments in named properties (which are then mapped to Camel message headers). For example, if you would like to personalize the *Hello World* application so that it greets the caller by name, you could define a REST service like the following:

```
rest("/say")
    .get("/hello/{name}").to("direct:hello")
    .get("/bye/{name}").to("direct:bye");

from("direct:hello")
    .transform().simple("Hello ${header.name}");
from("direct:bye")
    .transform().simple("Bye ${header.name}");
```

The URI template captures the text of the **{name}** path segment and copies this captured text into the **name** message header. If you invoke the service by sending a GET HTTP Request with the URL ending in **/say/hello/Joe**, the HTTP Response is **Hello Joe**.

Embedded route syntax

Instead of terminating a verb clause with the **to()** keyword (Java DSL) or the **to** element (XML DSL), you have the option of embedding an Apache Camel route directly into the REST DSL, using the **route()** keyword (Java DSL) or the **route** element (XML DSL). The **route()** keyword enables you to embed a route into a verb clause, with the following syntax:

RESTVerbClause.route("...").*CamelRoute*.endRest()

Where the **endRest()** keyword (Java DSL only) is a necessary punctuation mark that enables you to separate the verb clauses (when there is more than one verb clause in the **rest()** builder).

For example, you could refactor the *Hello World* example to use embedded Camel routes, as follows in Java DSL:

```
rest("/say")
    .get("/hello").route().transform().constant("Hello World").endRest()
    .get("/bye").route().transform().constant("Bye World");
```

And as follows in XML DSL:

```xml
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  ...
  <rest path="/say">
   <get uri="/hello">
     <route>
       <transform>
         <constant>Hello World</constant>
       </transform>
     </route>
   </get>
   <get uri="/bye">
     <route>
       <transform>
         <constant>Bye World</constant>
       </transform>
     </route>
   </get>
  </rest>
</camelContext>
```

NOTE

If you define any exception clauses (using **onException()**) or interceptors (using **intercept()**) in the current **CamelContext**, these exception clauses and interceptors are also active in the embedded routes.

Specifying the content type of requests and responses

You can filter the *content type* of HTTP requests and responses using the **consumes()** and **produces()** options in Java, or the **consumes** and **produces** attributes in XML. For example, some common content types (officially known as *Internet media types*) are the following:

- **text/plain**

- **text/html**

- **text/xml**

- **application/json**

- **application/xml**

The content type is specified as an option on a verb clause in the REST DSL. For example, to restrict a verb clause to accept only **text/plain** HTTP requests, and to send only **text/html** HTTP responses, you would use Java code like the following:

```
rest("/email")
    .post("/to/{recipient}").consumes("text/plain").produces("text/html").to("direct:foo");
```

And in XML, you can set the **consumes** and **produces** attributes, as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  ...
  <rest path="/email">
    <post uri="/to/{recipient}" consumes="text/plain" produces="text/html">
      <to "direct:foo"/>
    </get>
  </rest>
</camelContext>
```

You can also specify the argument to **consumes()** or **produces()** as a comma-separated list. For example, **consumes("text/plain, application/json")**.

Additional HTTP methods

Some HTTP server implementations support additional HTTP methods, which are not provided by the standard set of verbs in the REST DSL, **get()**, **head()**, **put()**, **post()**, **delete()**. To access additional HTTP methods, you can use the generic keyword, **verb()**, in Java DSL and the generic element, **verb**, in XML DSL.

For example, to implement the TRACE HTTP method in Java:

```
rest("/say")
    .verb("TRACE", "/hello").route().transform();
```

Where **transform()** copies the body of the *IN* message to the body of the *OUT* message, thus echoing the HTTP request.

To implement the TRACE HTTP method in XML:

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  ...
  <rest path="/say">
```

```xml
    <verb uri="/hello" method="TRACE">
     <route>
      <transform/>
     </route>
    </get>
</camelContext>
```

Defining custom HTTP error messages

If your REST service needs to send an error message as its response, you can define a custom HTTP error message as follows:

1. Specify the HTTP error code by setting the **Exchange.HTTP_RESPONSE_CODE** header key to the error code value (for example, **400**, **404**, and so on). This setting indicates to the REST DSL that you want to send an error message reply, instead of a regular response.

2. Populate the message body with your custom error message.

3. Set the **Content-Type** header, if required.

4. If your REST service is configured to marshal to and from Java objects (that is, **bindingMode** is enabled), you should ensure that the **skipBindingOnErrorCode** option is enabled (which it is, by default). This is to ensure that the REST DSL does not attempt to unmarshal the message body when sending the response.

   For more details about object binding, see Section 4.3, "Marshalling to and from Java Objects" .

The following Java example shows how to define a custom error message:

```java
// Java
// Configure the REST DSL, with JSON binding mode
restConfiguration().component("restlet").host("localhost").port(portNum).bindingMode(RestBindingMode.json);

// Define the service with REST DSL
rest("/users/")
    .post("lives").type(UserPojo.class).outType(CountryPojo.class)
        .route()
          .choice()
            .when().simple("${body.id} < 100")
                .bean(new UserErrorService(), "idTooLowError")
            .otherwise()
                .bean(new UserService(), "livesWhere");
```

In this example, if the input ID is a number less than 100, we return a custom error message, using the **UserErrorService** bean, which is implemented as follows:

```java
// Java
public class UserErrorService {
    public void idTooLowError(Exchange exchange) {
        exchange.getIn().setBody("id value is too low");
        exchange.getIn().setHeader(Exchange.CONTENT_TYPE, "text/plain");
        exchange.getIn().setHeader(Exchange.HTTP_RESPONSE_CODE, 400);
    }
}
```

In the **UserErrorService** bean we define the custom error message and set the HTTP error code to **400**.

Parameter Default Values

Default values can be specified for the headers of an incoming Camel message.

You can specify a default value by using a key word such as **verbose** on the query parameter. For example, in the code below, the default value is **false**. This means that if no other value is provided for a header with the **verbose** key, **false** will be inserted as a default.

```
rest("/customers/")
    .get("/{id}").to("direct:customerDetail")
    .get("/{id}/orders")
      .param()
  .name("verbose")
  .type(RestParamType.query)
  .defaultValue("false")
  .description("Verbose order details")
      .endParam()
        .to("direct:customerOrders")
    .post("/neworder").to("direct:customerNewOrder");
```

Wrapping a JsonParserException in a custom HTTP error message

A common case where you might want to return a custom error message is in order to wrap a **JsonParserException** exception. For example, you can conveniently exploit the Camel exception handling mechanism to create a custom HTTP error message, with HTTP error code 400, as follows:

```
// Java
onException(JsonParseException.class)
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(400))
    .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))
    .setBody().constant("Invalid json data");
```

REST DSL options

In general, REST DSL options can be applied either directly to the base part of the service definition (that is, immediately following **rest()**), as follows:

```
rest("/email").consumes("text/plain").produces("text/html")
    .post("/to/{recipient}").to("direct:foo")
    .get("/for/{username}").to("direct:bar");
```

In which case the specified options apply to all of the subordinate verb clauses. Or the options can be applied to each individual verb clause, as follows:

```
rest("/email")
    .post("/to/{recipient}").consumes("text/plain").produces("text/html").to("direct:foo")
    .get("/for/{username}").consumes("text/plain").produces("text/html").to("direct:bar");
```

In which case the specified options apply only to the relevant verb clause, overriding any settings from the base part.

Table 4.1, "REST DSL Options" summarizes the options supported by the REST DSL.

Table 4.1. REST DSL Options

Java DSL	XML DSL	Description
bindingMode()	**@bindingMode**	Specifies the binding mode, which can be used to marshal incoming messages to Java objects (and, optionally, unmarshal Java objects to outgoing messages). Can have the following values: **off** (default), **auto**, **json**, **xml**, **json_xml**.
consumes()	**@consumes**	Restricts the verb clause to accept only the specified Internet media type (MIME type) in a HTTP Request. Typical values are: **text/plain**, **text/http**, **text/xml**, **application/json**, **application/xml**.
customId()	**@customId**	Defines a custom ID for JMX management.
description()	**description**	Document the REST service or verb clause. Useful for JMX management and tooling.
enableCORS()	**@enableCORS**	If **true**, enables CORS (cross-origin resource sharing) headers in the HTTP response. Default is **false**.
id()	**@id**	Defines a unique ID for the REST service, which is useful to define for JMX management and other tooling.
method()	**@method**	Specifies the HTTP method processed by this verb clause. Usually used in conjunction with the generic **verb()** keyword.
outType()	**@outType**	When object binding is enabled (that is, when **bindingMode** option is enabled), this option specifies the Java type that represents a **HTTP Response** message.

Java DSL	XML DSL	Description
produces()	**produces**	Restricts the verb clause to produce only the specified Internet media type (MIME type) in a HTTP Response. Typical values are: **text/plain**, **text/http**, **text/xml**, **application/json**, **application/xml**.
type()	**@type**	When object binding is enabled (that is, when **bindingMode** option is enabled), this option specifies the Java type that represents a **HTTP Request** message.
VerbURIArgument	**@uri**	Specifies a path segment or URI template as an argument to a verb. For example, **get(*VerbURIArgument*)**.
BasePathArgument	**@path**	Specifies the base path in the **rest()** keyword (Java DSL) or in the **rest** element (XML DSL).

4.3. MARSHALLING TO AND FROM JAVA OBJECTS

Marshalling Java objects for transmission over HTTP

One of the most common ways to use the REST protocol is to transmit the contents of a Java bean in the message body. In order for this to work, you need to have a mechanism for marshalling the Java object to and from a suitable data format. The following data formats, which are suitable for encoding Java objects, are supported by the REST DSL:

JSON

JSON (JavaScript object notation) is a lightweight data format that can easily be mapped to and from Java objects. The JSON syntax is compact, lightly typed, and easy for humans to read and write. For all of these reasons, JSON has become popular as a message format for REST services.

For example, the following JSON code could represent a **User** bean with two property fields, **id** and **name**:

```
{
    "id" : 1234,
    "name" : "Jane Doe"
}
```

JAXB

JAXB (Java Architecture for XML Binding) is an XML-based data format that can easily be mapped to and from Java objects. In order to marshal the XML to a Java object, you must also annotate the Java class that you want to use.

For example, the following JAXB code could represent a **User** bean with two property fields, **id** and **name**:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<User>
  <Id>1234</Id>
  <Name>Jane Doe</Name>
</User>
```
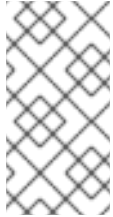
> **NOTE**
>
> From Camel 2.17.0, JAXB data format and type converter supports the conversion from XML to POJO for classes, that use **ObjectFactory** instead of **XmlRootElement**. Also, the camel context should include the CamelJaxbObjectFactory property with value true. However, due to optimization the default value is false.

Integration of JSON and JAXB with the REST DSL

You could, of course, write the required code to convert the message body to and from a Java object yourself. But the REST DSL offers the convenience of performing this conversion automatically. In particular, the integration of JSON and JAXB with the REST DSL offers the following advantages:

- Marshalling to and from Java objects is performed automatically (given the appropriate configuration).

- The REST DSL can automatically detect the data format (either JSON or JAXB) and perform the appropriate conversion.

- The REST DSL provides an *abstraction layer*, so that the code you write is not specific to a particular JSON or JAXB implementation. So you can switch the implementation later on, with minimum impact to your application code.

Supported data format components

Apache Camel provides a number of different implementations of the JSON and JAXB data formats. The following data formats are currently supported by the REST DSL:

- JSON

    - Jackson data format (**camel-jackson**) *(default)*

    - GSon data format (**camel-gson**)

    - XStream data format (**camel-xstream**)

- JAXB

    - JAXB data format (**camel-jaxb**)

How to enable object marshalling

To enable object marshalling in the REST DSL, observe the following points:

1. Enable binding mode, by setting the **bindingMode** option (there are several levels at which you can set the binding mode—for details, see the section called "Configuring the binding mode").

2. Specify the Java type to convert to (or from), on the incoming message with the **type** option (required), and on the outgoing message with the **outType** option (optional).

3. If you want to convert your Java object to and from the JAXB data format, you must remember to annotate the Java class with the appropriate JAXB annotations.

4. Specify the underlying data format implementation (or implementations), using the **jsonDataFormat** option and/or the **xmlDataFormat** option (which can be specified on the **restConfiguration** builder).

5. If your route provides a return value in JAXB format, you are normally expected to set the *Out* message of the exchange body to be an instance of a class with JAXB annotations (a JAXB element). If you prefer to provide the JAXB return value directly in XML format, however, set the **dataFormatProperty** with the key, **xml.out.mustBeJAXBElement**, to **false** (which can be specified on the **restConfiguration** builder). For example, in the XML DSL syntax:

   ```
   <restConfiguration ...>
     <dataFormatProperty key="xml.out.mustBeJAXBElement"
                 value="false"/>
     ...
   </restConfiguration>
   ```

6. Add the required dependencies to your project build file. For example, if you are using the Maven build system and you are using the Jackson data format, you would add the following dependency to your Maven POM file:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <project ...>
     ...
     <dependencies>
       ...
       <!-- use for json binding -->
       <dependency>
         <groupId>org.apache.camel</groupId>
         <artifactId>camel-jackson</artifactId>
       </dependency>
       ...
     </dependencies>
   </project>
   ```

7. When deploying your application to the OSGi container, remember to install the requisite feature for your chosen data format. For example, if you are using the Jackson data format (the default), you would install the **camel-jackson** feature, by entering the following Karaf console command:

   ```
   JBossFuse:karaf@root> features:install camel-jackson
   ```

Alternatively, if you are deploying into a Fabric environment, you would add the feature to a Fabric profile. For example, if you are using the profile, **MyRestProfile**, you could add the feature by entering the following console command:

```
JBossFuse:karaf@root> fabric:profile-edit --features camel-jackson MyRestProfile
```

Configuring the binding mode

The **bindingMode** option is **off** by default, so you must configure it explicitly, in order to enable marshalling of Java objects. TABLE shows the list of supported binding modes.

NOTE

From Camel 2.16.3 onwards the binding from POJO to JSon/JAXB will only happen if the content-type header includes *json* or *xml*. This allows you to specify a custom content-type if the message body should not attempt to be marshalled using the binding. This is useful if, for example, the message body is a custom binary payload.

Table 4.2. REST DSL BInding Modes

Binding Mode	Description
off	Binding is turned off *(default)*.
auto	Binding is enabled for JSON and/or XML. In this mode, Camel auto-selects either JSON or XML (JAXB), based on the format of the incoming message. You are *not* required to enable both kinds of data format, however: either a JSON implementation, an XML implementation, or both can be provided on the classpath.
json	Binding is enabled for JSON only. A JSON implementation *must* be provided on the classpath (by default, Camel tries to enable the **camel-jackson** implementation).
xml	Binding is enabled for XML only. An XML implementation *must* be provided on the classpath (by default, Camel tries to enable the **camel-jaxb** implementation).
json_xml	Binding is enabled for both JSON and XML. In this mode, Camel auto-selects either JSON or XML (JAXB), based on the format of the incoming message. You are required to provide *both* kinds of data format on the classpath.

In Java, these binding mode values are represented as instances of the following **enum** type:

```
org.apache.camel.model.rest.RestBindingMode
```

There are several different levels at which you can set the **bindingMode**, as follows:

REST DSL configuration

You can set the **bindingMode** option from the **restConfiguration** builder, as follows:

```
restConfiguration().component("servlet").port(8181).bindingMode(RestBindingMode.json);
```

Service definition base part

You can set the **bindingMode** option immediately following the **rest()** keyword (before the verb clauses), as follows:

```
rest("/user").bindingMode(RestBindingMode.json).get("/{id}").VerbClause
```

Verb clause

You can set the **bindingMode** option in a verb clause, as follows:

```
rest("/user")
    .get("/{id}").bindingMode(RestBindingMode.json).to("...");
```

Example

For a complete code example, showing how to use the REST DSL, using the Servlet component as the REST implementation, take a look at the Apache Camel **camel-example-servlet-rest-blueprint** example. You can find this example by installing the standalone Apache Camel distribution, **apache-camel-2.17.0.redhat-630xxx.zip**, which is provided in the **extras/** subdirectory of your JBoss Fuse installation.

After installing the standalone Apache Camel distribution, you can find the example code under the following directory:

```
ApacheCamelInstallDir/examples/camel-example-servlet-rest-blueprint
```

Configure the Servlet component as the REST implementation

In the **camel-example-servlet-rest-blueprint** example, the underlying implementation of the REST DSL is provided by the Servlet component. The Servlet component is configured in the Blueprint XML file, as shown in Example 4.1, "Configure Servlet Component for REST DSL" .

Example 4.1. Configure Servlet Component for REST DSL

```xml
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ...>

    <!-- to setup camel servlet with OSGi HttpService -->
    <reference id="httpService" interface="org.osgi.service.http.HttpService"/>

    <bean class="org.apache.camel.component.servlet.osgi.OsgiServletRegisterer"
        init-method="register"
        destroy-method="unregister">
      <property name="alias" value="/camel-example-servlet-rest-blueprint/rest"/>
      <property name="httpService" ref="httpService"/>
      <property name="servlet" ref="camelServlet"/>
```

```
    </bean>

    <bean id="camelServlet"
  class="org.apache.camel.component.servlet.CamelHttpTransportServlet"/>
    ...
    <camelContext xmlns="http://camel.apache.org/schema/blueprint">

      <restConfiguration component="servlet"
                  bindingMode="json"
                  contextPath="/camel-example-servlet-rest-blueprint/rest"
                  port="8181">
        <dataFormatProperty key="prettyPrint" value="true"/>
      </restConfiguration>
      ...
    </camelContext>

  </blueprint>
```

To configure the Servlet component with REST DSL, you need to configure a stack consisting of the following three layers:

REST DSL layer

The REST DSL layer is configured by the **restConfiguration** element, which integrates with the Servlet component by setting the **component** attribute to the value, **servlet**.

Servlet component layer

The Servlet component layer is implemented as an instance of the class, **CamelHttpTransportServlet**, where the example instance has the bean ID, **camelServlet**.

HTTP container layer

The Servlet component must be deployed into a HTTP container. The Karaf container is normally configured with a default HTTP container (a Jetty HTTP container), which listens for HTTP requests on the port, 8181. To deploy the Servlet component to the default Jetty container, you need to do the following:

1. Get an OSGi reference to the **org.osgi.service.http.HttpService** OSGi service, where this service is a standardised OSGi interface that provides access to the default HTTP server in OSGi.

2. Create an instance of the utility class, **OsgiServletRegisterer**, to register the Servlet component in the HTTP container. The **OsgiServletRegisterer** class is a utility that simplifies managing the lifecycle of the Servlet component. When an instance of this class is created, it automatically calls the **registerServlet** method on the **HttpService** OSGi service; and when the instance is destroyed, it automatically calls the **unregister** method.

Required dependencies

This example has two dependencies which are of key importance to the REST DSL, as follows:

Servlet component

Provides the underlying implementation of the REST DSL. This is specified in the Maven POM file, as follows:

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

And before you deploy the application bundle to the OSGi container, you must install the Servlet component feature, as follows:

```
JBossFuse:karaf@root> features:install camel-servlet
```

Jackson data format

Provides the JSON data format implementation. This is specified in the Maven POM file, as follows:

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

And before you deploy the application bundle to the OSGi container, you must install the Jackson data format feature, as follows:

```
JBossFuse:karaf@root> features:install camel-jackson
```

Java type for responses

The example application passes **User** type objects back and forth in HTTP Request and Response messages. The **User** Java class is defined as shown in Example 4.2, "User Class for JSON Response".

Example 4.2. User Class for JSON Response

```java
// Java
package org.apache.camel.example.rest;

public class User {

    private int id;
    private String name;

    public User() {
    }

    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }

    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;
        }
    }
```

The **User** class has a relatively simple representation in the JSON data format. For example, a typical instance of this class expressed in JSON format is:

```
{
    "id" : 1234,
    "name" : "Jane Doe"
}
```

Sample REST DSL route with JSON binding

The REST DSL configuration and the REST service definition for this example are shown in Example 4.3, "REST DSL Route with JSON Binding".

Example 4.3. REST DSL Route with JSON Binding

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        ...>
  ...
  <!-- a bean for user services -->
  <bean id="userService" class="org.apache.camel.example.rest.UserService"/>

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

    <restConfiguration component="servlet"
                bindingMode="json"
                contextPath="/camel-example-servlet-rest-blueprint/rest"
                port="8181">
      <dataFormatProperty key="prettyPrint" value="true"/>
    </restConfiguration>

    <!-- defines the REST services using the  base path, /user -->
    <rest path="/user" consumes="application/json" produces="application/json">
      <description>User rest service</description>

      <!-- this is a rest GET to view a user with the given id -->
      <get uri="/{id}" outType="org.apache.camel.example.rest.User">
        <description>Find user by id</description>
        <to uri="bean:userService?method=getUser(${header.id})"/>
```

```
      </get>

      <!-- this is a rest PUT to create/update a user -->
      <put type="org.apache.camel.example.rest.User">
        <description>Updates or create a user</description>
        <to uri="bean:userService?method=updateUser"/>
      </put>

      <!-- this is a rest GET to find all users -->
      <get uri="/findAll" outType="org.apache.camel.example.rest.User[]">
        <description>Find all users</description>
        <to uri="bean:userService?method=listUsers"/>
      </get>

    </rest>

  </camelContext>

</blueprint>
```

REST operations

The REST service from Example 4.3, "REST DSL Route with JSON Binding" defines the following REST operations:

GET /camel-example-servlet-rest-blueprint/rest/user/{id}

Get the details for the user identified by **{id}**, where the HTTP response is returned in JSON format.

PUT /camel-example-servlet-rest-blueprint/rest/user

Create a new user, where the user details are contained in the body of the PUT message, encoded in JSON format (to match the **User** object type).

GET /camel-example-servlet-rest-blueprint/rest/user/findAll

Get the details for *all* users, where the HTTP response is returned as an array of users, in JSON format.

URLs to invoke the REST service

By inspecting the REST DSL definitions from Example 4.3, "REST DSL Route with JSON Binding" , you can piece together the URLs required to invoke each of the REST operations. For example, to invoke the first REST operation, which returns details of a user with a given ID, the URL is built up as follows:

http://localhost:8181

In **restConfiguration**, the protocol defaults to **http** and the port is set explicitly to **8181**.

/camel-example-servlet-rest-blueprint/rest

Specified by the **contextPath** attribute of the **restConfiguration** element.

/user

Specified by the **path** attribute of the **rest** element.

/{id}

Specified by the **uri** attribute of the **get** verb element.

Hence, it is possible to invoke this REST operation with the **curl** utility, by entering the following command at the command line:

```
curl -X GET -H "Accept: application/json" http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/123
```

Similarly, the remaining REST operations could be invoked with **curl**, by entering the following sample commands:

```
curl -X GET -H "Accept: application/json" http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user/findAll

curl -X PUT -d "{ \"id\": 666, \"name\": \"The devil\"}" -H "Accept: application/json" http://localhost:8181/camel-example-servlet-rest-blueprint/rest/user
```

4.4. CONFIGURING THE REST DSL

Configuring with Java

In Java, you can configure the REST DSL using the **restConfiguration()** builder API. For example, to configure the REST DSL to use the Servlet component as the underlying implementation:

```
restConfiguration().component("servlet").bindingMode("json").port("8181")
   .contextPath("/camel-example-servlet-rest-blueprint/rest");
```

Configuring with XML

In XML, you can configure the REST DSL using the **restConfiguration** element. For example, to configure the REST DSL to use the Servlet component as the underlying implementation:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<blueprint ...>
  ...
  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
   ...
   <restConfiguration component="servlet"
              bindingMode="json"
              contextPath="/camel-example-servlet-rest-blueprint/rest"
              port="8181">
     <dataFormatProperty key="prettyPrint" value="true"/>
   </restConfiguration>
   ...
  </camelContext>

</blueprint>
```

Configuration options

Table 4.3, "Options for Configuring REST DSL" shows options for configuring the REST DSL using the **restConfiguration()** builder (Java DSL) or the **restConfiguration** element (XML DSL).

Table 4.3. Options for Configuring REST DSL

Java DSL	XML DSL	Description
component()	**@component**	Specifies the Camel component to use as the REST transport (for example, **servlet**, **restlet**, **spark-rest**, and so on). The value can either be the standard component name or the bean ID of a custom instance. If this option is not specified, Camel looks for an instance of **RestConsumerFactory** on the classpath or in the bean registry.
scheme()	**@scheme**	The protocol to use for exposing the REST service. Depends on the underlying REST implementation, but **http** and **https** are usually supported. Default is **http**.
host()	**@host**	The hostname to use for exposing the REST service.
port()	**@port**	The port number to use for exposing the REST service. Note: This setting is *ignored* by the Servlet component, which uses the container's standard HTTP port instead. In the case of the Apache Karaf OSGi container, the standard HTTP port is normally 8181. It is good practice to set the port value nonetheless, for the sake of JMX and tooling.
contextPath()	**@contextPath**	Sets a leading context path for the REST services. This can be used with components such as Servlet, where the deployed Web application is deployed using a **context-path** setting.

Java DSL	XML DSL	Description
hostNameResolver()	**@hostNameResolver**	If a hostname is not set explicitly, this resolver determines the host for the REST service. Possible values are **RestHostNameResolver.localHostName** (Java DSL) or **localHostName** (XML DSL), which resolves to the host name format; and **RestHostNameResolver.localIp** (Java DSL) or **localIp** (XML DSL), which resolves to the dotted decimal IP address format. From Camel 2.17 **RestHostNameResolver.allLocalIp** can be used to resolve to all local IP addresses. The default is **localHostName** up to Camel 2.16. From Camel 2.17 the default is **allLocalIp**.
bindingMode()	**@bindingMode**	Enables binding mode for JSON or XML format messages. Possible values are: **off**, **auto**, **json**, **xml**, or **json_xml**. Default is **off**.
skipBindingOnErrorCode()	**@skipBindingOnErrorCode**	Specifies whether to skip binding on output, if there is a custom HTTP error code header. This allows you to build custom error messages that do not bind to JSON or XML, as successful messages would otherwise do. Default is **true**.
enableCORS()	**@enableCORS**	If **true**, enables CORS (cross-origin resource sharing) headers in the HTTP response. Default is **false**.
jsonDataFormat()	**@jsonDataFormat**	Specifies the component that Camel uses to implement the JSON data format. Possible values are: **json-jackson**, **json-gson**, **json-xstream**. Default is **json-jackson**.
xmlDataFormat()	**@xmlDataFormat**	Specifies the component that Camel uses to implement the XML data format. Possible value is: **jaxb**. Default is **jaxb**.

Java DSL	XML DSL	Description
componentProperty()	**componentProperty**	Enables you to set arbitrary *component level* properties on the underlying REST implementation.
endpointProperty()	**endpointProperty**	Enables you to set arbitrary *endpoint level* properties on the underlying REST implementation.
consumerProperty()	**consumerProperty**	Enables you to set arbitrary *consumer endpoint* properties on the underlying REST implementation.
dataFormatProperty()	**dataFormatProperty**	Enables you to set arbitrary properties on the underlying data format component (for example, Jackson or JAXB). From Camel 2.14.1 onwards, you can attach the following prefixes to the property keys: • **json.in** • **json.out** • **xml.in** • **xml.out** To restrict the property setting to a specific format type (JSON or XML) and a particular message direction (*IN* or *OUT*).
corsHeaderProperty()	**corsHeaders**	Enables you to specify custom CORS headers, as key/value pairs.

Default CORS headers

If CORS (cross-origin resource sharing) is enabled, the following headers are set by default. You can optionally override the default settings, by invoking the **corsHeaderProperty** DSL command.

Table 4.4. Default CORS Headers

Header Key	Header Value
Access-Control-Allow-Origin	*
Access-Control-Allow-Methods	**GET**, **HEAD**, **POST**, **PUT**, **DELETE**, **TRACE**, **OPTIONS**, **CONNECT**, **PATCH**

Header Key	Header Value
Access-Control-Allow-Headers	**Origin**, **Accept**, **X-Requested-With**, **Content-Type**, **Access-Control-Request-Method**, **Access-Control-Request-Headers**
Access-Control-Max-Age	**3600**

Enabling or disabling Jackson JSON features

You can enable or disable specific Jackson JSON features by configuring the following keys in the **dataFormatProperty** option:

- **json.in.disableFeatures**

- **json.in.enableFeatures**

For example, to disable Jackson's **FAIL_ON_UNKNOWN_PROPERTIES** feature (which causes Jackson to fail if a JSON input has a property that cannot be mapped to a Java object):

```
restConfiguration().component("jetty")
    .host("localhost").port(getPort())
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("json.in.disableFeatures", "FAIL_ON_UNKNOWN_PROPERTIES");
```

You can disable *multiple* features by specifying a comma-separated list. For example:

```
.dataFormatProperty("json.in.disableFeatures",
"FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE");
```

Here is an example that shows how to disable and enable Jackson JSON features in the Java DSL:

```
restConfiguration().component("jetty")
    .host("localhost").port(getPort())
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("json.in.disableFeatures",
"FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE")
    .dataFormatProperty("json.in.enableFeatures",
"FAIL_ON_NUMBERS_FOR_ENUMS,USE_BIG_DECIMAL_FOR_FLOATS");
```

Here is an example that shows how to disable and enable Jackson JSON features in the XML DSL:

```
<restConfiguration component="jetty" host="localhost" port="9090" bindingMode="json">
  <dataFormatProperty key="json.in.disableFeatures"
value="FAIL_ON_UNKNOWN_PROPERTIES,ADJUST_DATES_TO_CONTEXT_TIME_ZONE"/>
  <dataFormatProperty key="json.in.enableFeatures"
value="FAIL_ON_NUMBERS_FOR_ENUMS,USE_BIG_DECIMAL_FOR_FLOATS"/>
</restConfiguration>
```

The Jackson features that can be disabled or enabled correspond to the **enum** IDs from the following Jackson classes

- com.fasterxml.jackson.databind.SerializationFeature

- com.fasterxml.jackson.databind.DeserializationFeature

- com.fasterxml.jackson.databind.MapperFeature

4.5. SWAGGER INTEGRATION

Overview

You can use a Swagger service to create API documentation for any REST-defined routes and endpoints in a CamelContext file. To do this, use the Camel REST DSL with the **camel-swagger-java** module, which is purely Java-based. The **camel-swagger-java** module creates a servlet that is integrated with the CamelContext and that pulls the information from each REST endpoint to generate the API documentation in JSON or YAML format.

If you use Maven then edit your **pom.xml** file to add a dependency on the **camel-swagger-java** component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-swagger-java</artifactId>
  <version>x.x.x</version>
  <!-- Specify the version of your camel-core module. -->
</dependency>
```

Configuring a CamelContext to enable Swagger

To enable the use of the Swagger API in the Camel REST DSL, invoke **apiContextPath()** to set the context path for the Swagger-generated API documentation. For example:

```
public class UserRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        // Configure the Camel REST DSL to use the netty4-http component:
        restConfiguration().component("netty4-http").bindingMode(RestBindingMode.json)
            // Generate pretty print output:
            .dataFormatProperty("prettyPrint", "true")
            // Set the context path and port number that netty will use:
            .contextPath("/").port(8080)
            // Add the context path for the Swagger-generated API documentation:
            .apiContextPath("/api-doc")
                .apiProperty("api.title", "User API").apiProperty("api.version", "1.2.3")
                // Enable CORS:
                .apiProperty("cors", "true");

        // This user REST service handles only JSON files:
        rest("/user").description("User rest service")
            .consumes("application/json").produces("application/json")
            .get("/{id}").description("Find user by id").outType(User.class)
                .param().name("id").type(path).description("The id of the user to
get").dataType("int").endParam()
                .to("bean:userService?method=getUser(${header.id})")
            .put().description("Updates or create a user").type(User.class)
```

```
                    .param().name("body").type(body).description("The user to update or create").endParam()
                    .to("bean:userService?method=updateUser")
                .get("/findAll").description("Find all users").outTypeList(User.class)
                    .to("bean:userService?method=listUsers");
        }
    }
```

Swagger module configuration options

The options described in the table below let you configure the Swagger module. Set an option as follows:

- If you are using the **camel-swagger-java** module as a servlet, set an option by updating the **web.xml** file and specifying an **init-param** element for each configuration option you want to set.

- If you are using the **camel-swagger-java** module from Camel REST components, set an option by invoking the appropriate **RestConfigurationDefinition** method, such as **enableCORS()**, **host()**, or **contextPath()**. Set the **api.xxx** options with the **RestConfigurationDefinition.apiProperty()** method.

Option	Type	Description
api.contact.email	String	Email address to be used for API-related correspondence.
api.contact.name	String	Name of person or organization to contact.
api.contact.url	String	URL to a website for more contact information.

Option	Type	Description
apiContextIdListing	Boolean	If your application uses more than one **CamelContext** object, the default behavior is to list the REST endpoints in only the current **CamelContext**. If you want a list of the REST endpoints in each **CamelContext** that is running in the JVM that is running the REST service then set this option to true. When **apiContextIdListing** is true then Swagger outputs the **CamelContext** IDs in the root path, for example, /**api-docs**, as a list of names in JSON format. To access the Swagger-generated documentation, append the REST context path to the **CamelContext** ID, for example, **api-docs/myCamel**. You can use the **apiContextIdPattern** option to filter the names in this output list.
apiContextIdPattern	String	Pattern that filters which CamelContext IDs appear in the context listing. You can specify regular expressions and use * as a wildcard. This is the same pattern matching facility as used by the Camel Intercept feature.
api.license.name	String	License name used for the API.
api.license.url	String	URL to the license used for the API.
api.path	String	Sets the path where the REST API to generate documentation for is available, for example, /**api-docs**. Specify a relative path. Do not specify, for example, **http** or **https**. The **camel-swagger-java** module calculates the absolute path at runtime in this format: **protocol://host:port/context-path/api-path**.
api.termsOfService	String	URL to the terms of service of the API.

Option	Type	Description
api.title	String	Title of the application.
api.version	String	Version of the API. The default is 0.0.0.
base.path	String	Required. Sets the path where the REST services are available. Specify a relative path. That is, do not specify, for example, **http** or **https**. The **camel-swagger-java** modul calculates the absolute path at runtime in this format: **protocol://host:port/context-path/base.path**.
cors	Boolean	Whether to enable HTTP Access Control (CORS). This enable CORS only for viewing the REST API documentation, and not for access to the REST service. The default is false. The recommendation is to use the **CorsFilter** option instead, as described after this table.
host	String	Set the name of the host that the Swagger service is running on. The default is to calculate the host name based on **localhost**.
schemes	String	Protocol schemes to use. Separate multiple values with a comma, for example, **"http,https".** The default is **http**.
swagger.version	String	Swagger specification version. The default is 2.0.

Using the CORS filter to enable CORS support

If you use the Swagger user interface to view your REST API documentation then you probably need to enable support for HTTP Access Control (CORS). This support is required when the Swagger user interface is hosted and running on a hostname/port that is different from the hostname/port on which your REST APIs are running.

To enable support for CORS, add the **RestSwaggerCorsFilter** to your **web.xml** file. The CORS filter adds the HTTP headers that enable CORS. For example:

```
<!-- Enable CORS filter to allow use of Swagger UI for browsing and testing APIs. -->
```

```
<filter>
    <filter-name>RestSwaggerCorsFilter</filter-name>
    <filter-class>org.apache.camel.swagger.rest.RestSwaggerCorsFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>RestSwaggerCorsFilter</filter-name>
    <url-pattern>/api-docs/*</url-pattern>
    <url-pattern>/rest/*</url-pattern>
</filter-mapping>
```

The **RestSwaggerCorsFilter** sets the following headers for all requests:

- Access-Control-Allow-Origin= *

- Access-Control-Allow-Methods = GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH

- Access-Control-Max-Age = 3600'

- Access-Control-Allow-Headers = Origin, Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers

RestSwaggerCorsFilter is a simple filter. You might need a more sophisticated filter if you need to block certain clients or set the header values differently for a given client.

Obtaining JSON or YAML output

Starting with Camel 2.17, the **camel-swagger-java** module supports both JSON and YAML formatted output. To specify the output you want, add /**swagger.json** or /**swagger.yaml** to the request URL. If a request URL does not specify a format then the **camel-swagger-java** module inspects the HTTP Accept header to detect whether JSON or YAML can be accepted. If both are accepted or if none was set as accepted then JSON is the default return format.

Examples

In the Apache Camel distribution, **camel-example-swagger-cdi** and **camel-example-swagger-java** demonstrate the use of the **camel-swagger-java** module.

Enhancing documentation generated by Swagger

Starting with Camel 2.16, you can enhance the documentation generated by Swagger by defining parameter details such as name, description, data type, parameter type and so on. If you are using XML, specify the **param** element to add this information. The following example shows how to provide information about the ID path parameter:

```
<!-- This is a REST GET request to view information for the user with the given ID: -->
<get uri="/{id}" outType="org.apache.camel.example.rest.User">
    <description>Find user by ID.</description>
    <param name="id" type="path" description="The ID of the user to get information about."
dataType="int"/>
    <to uri="bean:userService?method=getUser(${header.id})"/>
</get>
```

Following is the same example in Java DSL:

```
.get("/{id}").description("Find user by ID.").outType(User.class)
  .param().name("id").type(path).description("The ID of the user to get information
about.").dataType("int").endParam()
  .to("bean:userService?method=getUser(${header.id})")
```

If you define a parameter whose name is **body** then you must also specify **body** as the type of that parameter. For example:

```
<!-- This is a REST PUT request to create/update information about a user. -->
<put type="org.apache.camel.example.rest.User">
    <description>Updates or creates a user.</description>
    <param name="body" type="body" description="The user to update or create."/>
    <to uri="bean:userService?method=updateUser"/>
</put>
```

Following is the same example in Java DSL:

```
.put().description("Updates or create a user").type(User.class)
    .param().name("body").type(body).description("The user to update or create.").endParam()
    .to("bean:userService?method=updateUser")
```

See also: **examples/camel-example-servlet-rest-tomcat** in the Apache Camel distribution.

CHAPTER 5. MESSAGING SYSTEMS

Abstract

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

5.1. 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 5.1, "Message Pattern".

Figure 5.1. Message Pattern
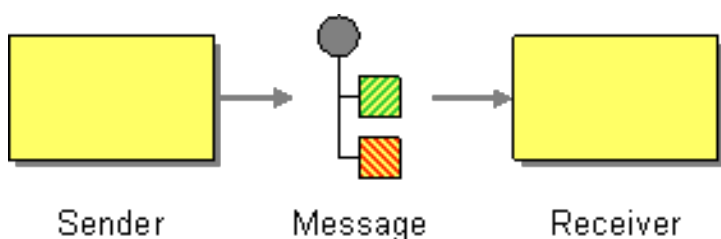
Types of message

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);
```

5.2. 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 5.2, "Message Channel Pattern" .

Figure 5.2. Message Channel Pattern
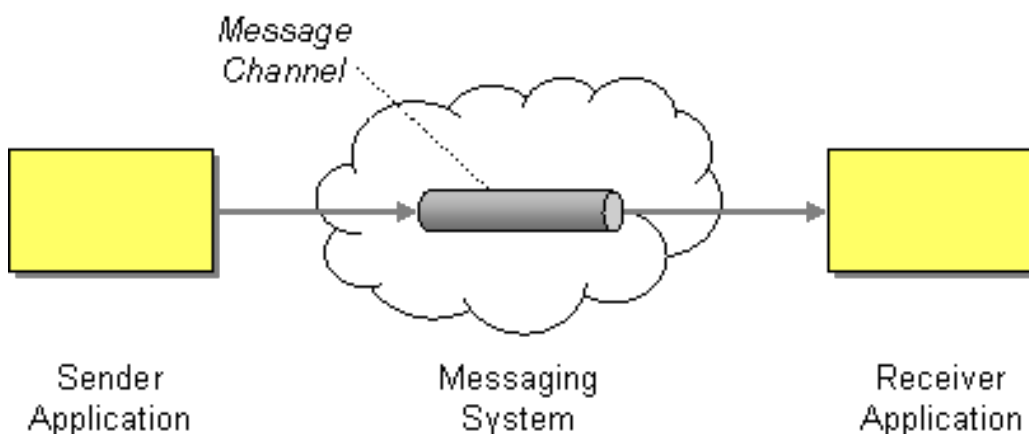
Message-oriented components

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 see *ActiveMQ* in the *Apache Camel Component Reference Guide* 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 *Jms* in the *Apache Camel Component Reference Guide* 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 see *Amqp* in the *Apache Camel Component Reference Guide*. for more details and instructions on setting up the AMQP component.

5.3. MESSAGE ENDPOINT

Overview

A *message endpoint* is the interface between an application and a messaging system. As shown in Figure 5.3, "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 5.3. Message Endpoint Pattern
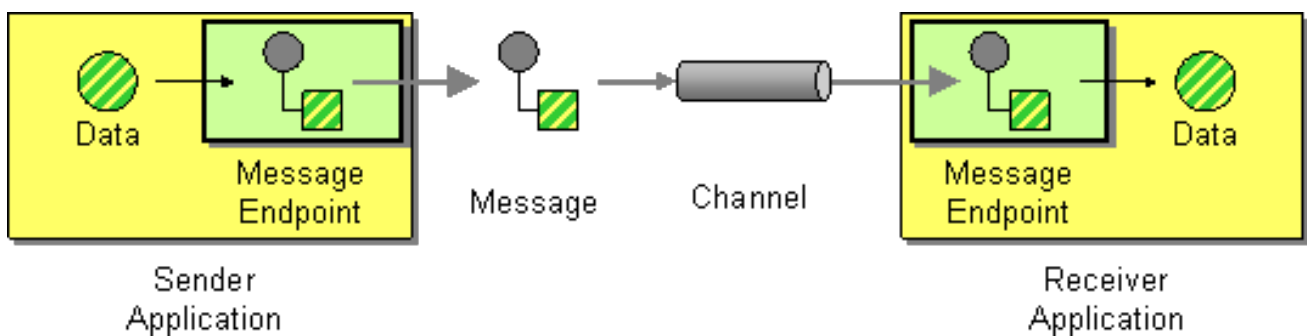
Types of endpoint

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 "Apache Camel 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:

```xml
<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>
```

Dynamic To

The **<toD>** parameter allows you to send a message to a dynamically computed endpoint using one or more expressions that are concatenated together.

By default, the Simple language is used to compute the endpoint. The following example sends a message to an endpoint defined by a header:

```xml
<route>
  <from uri="direct:start"/>
  <toD uri="${header.foo}"/>
</route>
```

In Java DSL the format for the same command is:

```
from("direct:start")
  .toD("${header.foo}");
```

The URI can also be prefixed with a literal, as shown in the following example:

```
<route>
  <from uri="direct:start"/>
  <toD uri="mock:${header.foo}"/>
</route>
```

In Java DSL the format for the same command is:

```
from("direct:start")
  .toD("mock:${header.foo}");
```

In the example above, if the value of header.foo is *orange*, the URI will resolve as **mock:orange**.

To use a language other than Simple, you need to define the *language:* parameter. See Part II, "Routing Expression and Predicate Languages".

The format for using a different language is to use **language:*languagename*:** in the URI. For example, to use Xpath use the following format:

```
<route>
  <from uri="direct:start"/>
  <toD uri="language:xpath:/order/@uri/">
</route>
```

Here is the same example in Java DSL:

```
from("direct:start")
  .toD("language:xpath:/order/@uri");
```

If you do not specify **language:** then the endpoint is a component name. In some cases a component and a language have the same name, such as xquery.

You can concatenate multiple languages using a **+** sign. In the example below, the URI is a combination of Simple and Xpath languages. Simple is the default so the language does not have to be defined. After the **+** sign is the Xpath instruction, indicated by **language:xpath**.

```
<route>
  <from uri="direct:start"/>
  <toD uri="jms:${header.base}+language:xpath:/order/@id"/>
</route>
```

In Java DSL the format is as follows:

```
from("direct:start")
  .toD("jms:${header.base}+language:xpath:/order/@id");
```

Many languages can be concatenated at one time, just separate each with a **+** and specify each language with **language:*languagename***.

The following options are available with **toD**:

Name	Default Value	Description
uri		Mandatory: The URI to use.
pattern		Set a specific Exchange Pattern to use when sending to the endpoint. The original MEP is restored afterwards.
cacheSize		Configure the cache size of the **ProducerCache**, which caches producers for reuse. The default cache size is 1000, which will be used if no other value is specified. Setting the value to -1 turns off the cache completely.
ignoreInvalidEndpoint	**false**	Specifies whether to ignore an endpoint URI that could not be resolved. If disabled, Camel will throw an exception identifying the invalid endpoint URI.

5.4. PIPES AND FILTERS

Overview

The *pipes and filters* pattern, shown in Figure 5.4, "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 5.4. Pipes and Filters Pattern
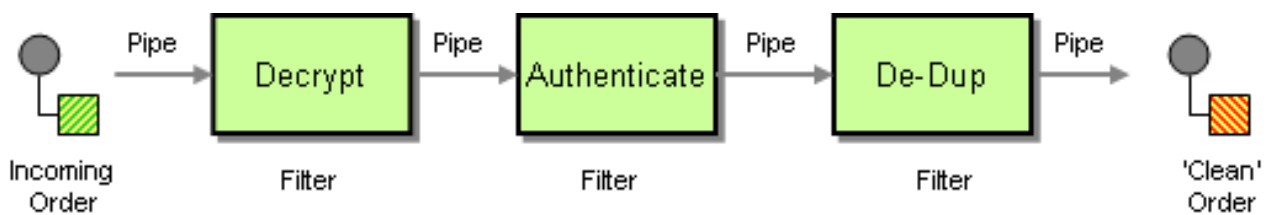
Pipeline for the InOut exchange pattern

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 5.5, "Pipeline for InOut Exchanges".

Figure 5.5. Pipeline for InOut Exchanges
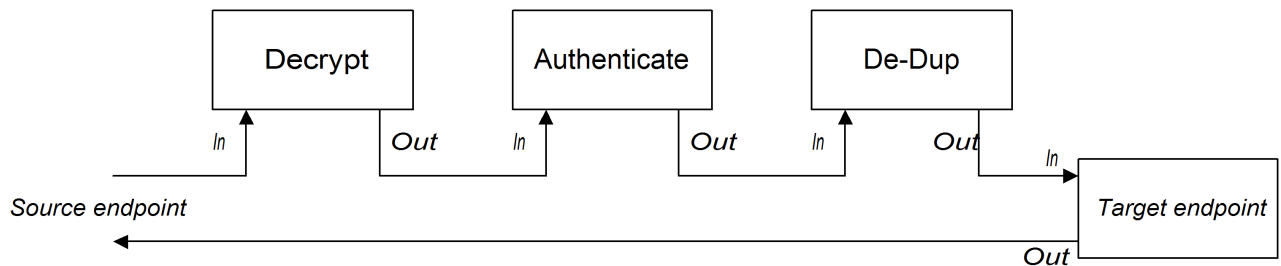
The pipeline connects the output of each endpoint to the input of the next endpoint. The *Out* message from the final endpoint is sent back to the original caller. You can define a route for this pipeline, as follows:

```
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 5.6, "Pipeline for InOnly Exchanges". This type of pipeline is equivalent to a recipient list with fixed destinations(see Section 8.3, "Recipient List").

Figure 5.6. Pipeline for InOnly Exchanges
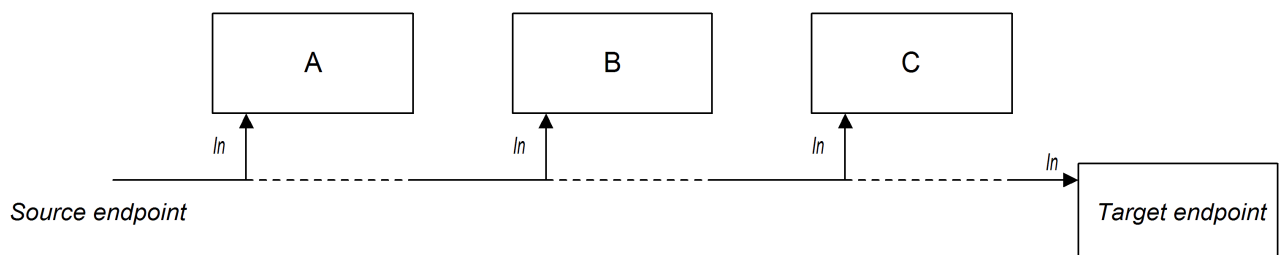
The route for this pipeline is defined using the same syntax as an *InOut* pipeline (either in Java DSL or in XML).

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 8.11, "Multicast").

5.5. MESSAGE ROUTER

Overview

A *message router*, shown in Figure 5.7, "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 5.7. Message Router Pattern
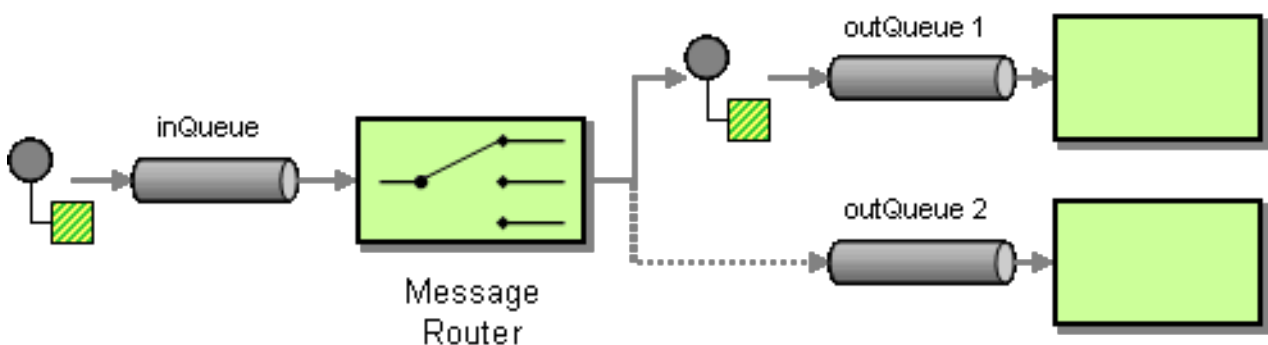
A message router can easily be implemented in Apache Camel using the **choice()** processor, where each of the alternative target endpoints can be selected using a **when()** subclause (for details of the choice processor, see Section 1.5, "Processors").

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:

```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.

5.6. MESSAGE TRANSLATOR

Overview

The *message translator* pattern, shown in Figure 5.8, "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 5.8. Message Translator Pattern
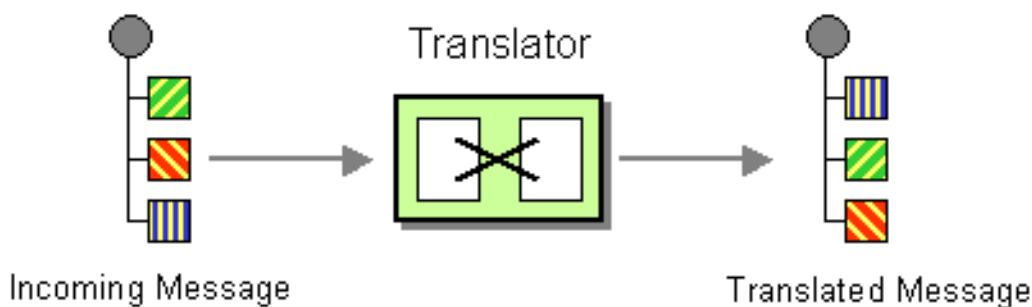
Bean integration

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");
```

5.7. MESSAGE HISTORY

Overview

The *Message History* from EIP pattern enables you to analyze and debug the flow of messages in a loosely coupled system. If you attach a message history to the message, it displays a list of all applications that the message passed through since its origination.

In Apache Camel, using the getTracedRouteNodes method, you can trace a message flow using the Tracer or access information using the Java API from UnitOfWork.

Limiting Character Length in Logs

When you run Apache Camel with logging mechanism, it enables you to log the messages and its content from time to time.

Some messages may contain very big payloads. By default, Apache Camel will clip the log message and show only the first 1000 characters. For example, it displays the following log as:

> [DEBUG ProducerCache - >>>> Endpoint[direct:start] Exchange[Message: 01234567890123456789... [Body clipped after 20 characters, total length is 1000]

You can customize the limit when Apache Camel clips the body in the log. You can also set zero or a negative value, such as -1, means the message body is not logged.

Customizing the Limit using Java DSL

You can set the limit in Camel properties using Java DSL. For example,

```
context.getProperties().put(Exchange.LOG_DEBUG_BODY_MAX_CHARS, "500");
```

Customizing the Limit using Spring DSL

You can set the limit in Camel properties using Spring DSL. For example,

```
<camelContext>
  <properties>
      <property key="CamelLogDebugBodyMaxChars" value="500"/>
  </properties>
</camelContext>
```

CHAPTER 6. 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.

6.1. POINT-TO-POINT CHANNEL

Overview

A *point-to-point channel*, shown in Figure 6.1, "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 6.1. Point to Point Channel Pattern
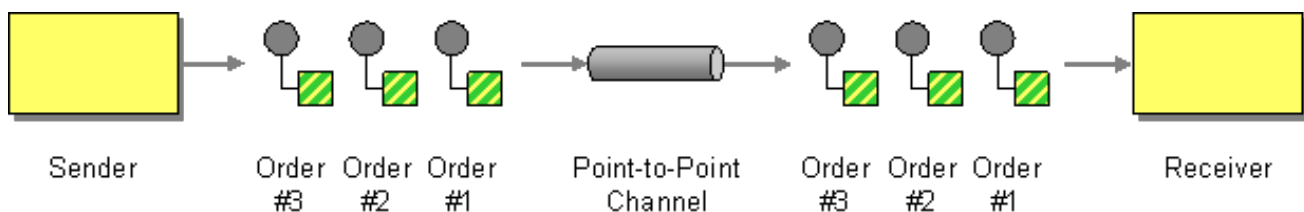
Components that support point-to-point channel

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 *Jms* in the *Apache Camel Component Reference Guide* 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 *ActiveMQ* in the *Apache Camel Component Reference Guide* 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 *Jpa* in the *Apache Camel Component Reference Guide* 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 *Xmpp* in the *Apache Camel Component Reference Guide* for more details.

6.2. PUBLISH-SUBSCRIBE CHANNEL

Overview

A *publish-subscribe channel*, shown in Figure 6.2, "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 6.2. Publish Subscribe Channel Pattern
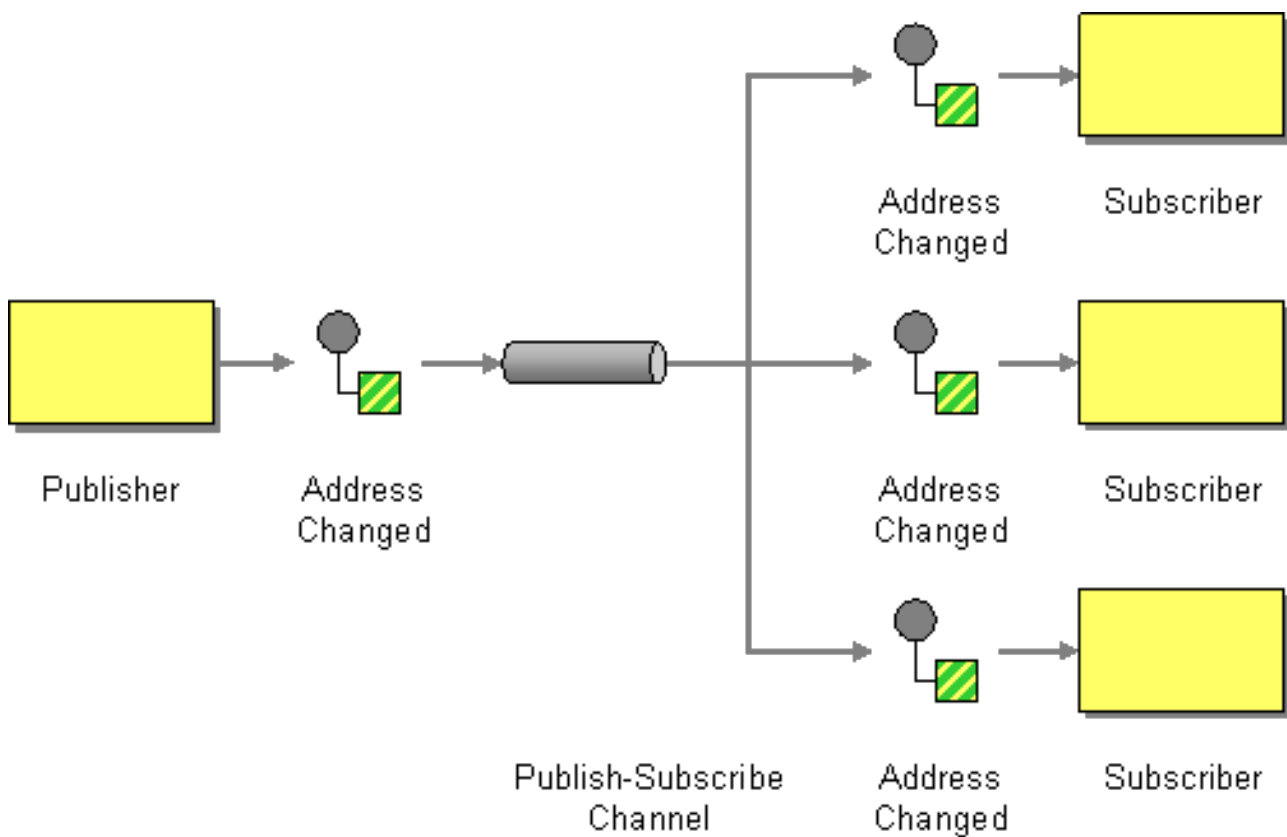
Components that support publish-subscribe channel

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.

- see *VM* in the *Apache Camel Component Reference Guide* 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 *Jms* in the *Apache Camel Component Reference Guide* 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 *ActiveMQ* in the *Apache Camel Component Reference Guide* for more details.

XMPP

The XMPP (Jabber) component supports the publish-subscribe channel pattern when it is used in the group communication mode. See *Xmpp* in the *Apache Camel Component Reference Guide* 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>
```

6.3. DEAD LETTER CHANNEL

Overview

The *dead letter channel* pattern, shown in Figure 6.3, "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 6.3. Dead Letter Channel Pattern
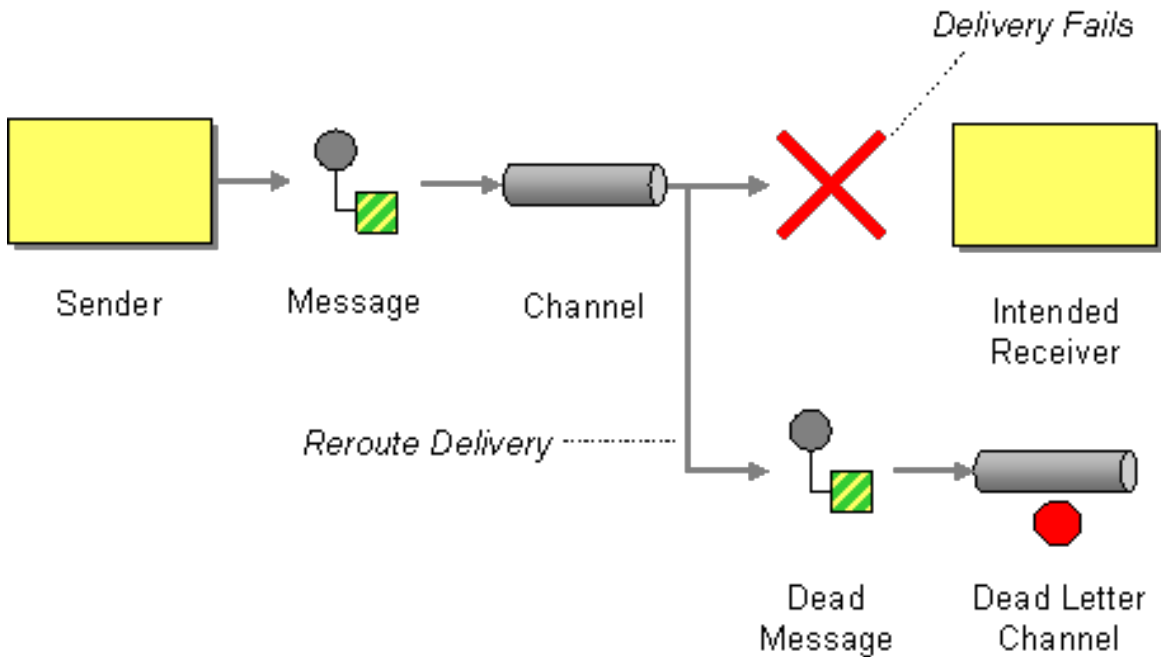
Creating a dead letter channel in Java DSL

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 6.1, "Redelivery Policy Settings" summarizes the methods that you can use to set redelivery policies.

Table 6.1. Redelivery Policy Settings

Method Signature	Default	Description
allowRedeliveryWhileStopping()	**true**	Controls whether redelivery is attempted during graceful shutdown or while a route is stopping. A delivery that is already in progress when stopping is initiated will *not* be interrupted.
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: `d, m*d, m*m*d, m*m*m*d, ...`
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.

Method Signature	Default	Description
deadLetterHandleNewException	**true**	Camel 2.15: Specifies whether or not to handle an exception that occurs while processing a message in the dead letter channel. If **true**, the exception is handled and a logged at the WARN level (so that the dead letter channel is guaranteed to complete). If **false**, the exception is not handled, so the dead letter channel fails, and propagates the new exception.
delayPattern(String delayPattern)	*None*	Apache Camel 2.0: See Redeliver delay pattern later in this section.
disableRedelivery()	**true**	Apache Camel 2.0: Disables the redelivery feature. To enable redelivery, set **maximumRedeliveries()** to a positive integer value.
handled(boolean handled)	**true**	Apache Camel 2.0: If **true**, the current exception is cleared when the message is moved to the dead letter channel; if **false**, the exception is propagated back to the client.
initialRedeliveryDelay(long initialRedeliveryDelay)	**1000**	Specifies the delay (in milliseconds) before attempting the first redelivery.
logNewException	**true**	Specifies whether to log at WARN level, when an exception is raised in the dead letter channel.
logStackTrace(boolean logStackTrace)	**false**	Apache Camel 2.0: If **true**, the JVM stack trace is included in the error logs.
maximumRedeliveries(int maximumRedeliveries)	**0**	Apache Camel 2.0: Maximum number of delivery attempts.

Method Signature	Default	Description
maximumRedeliveryDelay(long maxDelay)	**60000**	Apache Camel 2.0: When using an exponential backoff strategy (see **useExponentialBackOff()**), it is theoretically possible for the redelivery delay to increase without limit. This property imposes an upper limit on the redelivery delay (in milliseconds)
onRedelivery(Processor processor)	*None*	Apache Camel 2.0: Configures a processor that gets called before every redelivery attempt.
redeliveryDelay(long int)	**0**	Apache Camel 2.0: Specifies the delay (in milliseconds) between redelivery attempts. Apache Camel 2.16.0 : The default redelivery delay is one second.
retriesExhaustedLogLevel(LoggingLevel logLevel)	**LoggingLevel.ERROR**	Apache Camel 2.0: Specifies the logging level at which to log delivery failure (specified as an **org.apache.camel.LoggingLevel** constant).
retryAttemptedLogLevel(LoggingLevel logLevel)	**LoggingLevel.DEBUG**	Apache Camel 2.0: Specifies the logging level at which to redelivery attempts (specified as an **org.apache.camel.LoggingLevel** constant).
useCollisionAvoidance()	**false**	Enables collision avoidance, which adds some randomization to the backoff timings to reduce contention probability.
useOriginalMessage()	**false**	Apache Camel 2.0: If this feature is enabled, the message sent to the dead letter channel is a copy of the *original* message exchange, as it existed at the beginning of the route (in the **from()** node).
useExponentialBackOff()	**false**	Enables exponential backoff.

Redelivery headers

If Apache Camel attempts to redeliver a message, it automatically sets the headers described in Table 6.2, "Dead Letter Redelivery Headers" on the *In* message.

Table 6.2. Dead Letter Redelivery Headers

Header Name	Type	Description
CamelRedeliveryCounter	**Integer**	Apache Camel 2.0: Counts the number of unsuccessful delivery attempts. This value is also set in **Exchange.REDELIVERY_CO UNTER**.
CamelRedelivered	**Boolean**	Apache Camel 2.0: True, if one or more redelivery attempts have been made. This value is also set in **Exchange.REDELIVERED**.
CamelRedeliveryMaxCounter	**Integer**	Apache Camel 2.6: Holds the maximum redelivery setting (also set in the **Exchange.REDELIVERY_MA X_COUNTER** exchange property). This header is absent if you use **retryWhile** or have unlimited maximum redelivery configured.

Redelivery exchange properties

If Apache Camel attempts to redeliver a message, it automatically sets the exchange properties described in Table 6.3, "Redelivery Exchange Properties" .

Table 6.3. Redelivery Exchange Properties

Exchange Property Name	Type	Description
Exchange.FAILURE_ROUTE _ID	**String**	Provides the route ID of the route that failed. The literal name of this property is **CamelFailureRouteId**.

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 2.0The **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 occcured. 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.

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 teting faster
    .redeliveryDelay(0L));
```

Where the **MyRedeliveryProcessor** 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);
  }
}
```

Control redelivery during shutdown or stopping

If you stop a route or initiate graceful shutdown, the default behavior of the error handler is to continue attempting redelivery. Because this is typically not the desired behavior, you have the option of disabling redelivery during shutdown or stopping, by setting the **allowRedeliveryWhileStopping** option to **false**, as shown in the following example:

```
errorHandler(deadLetterChannel("jms:queue:dead")
    .allowRedeliveryWhileStopping(false)
    .maximumRedeliveries(20)
    .redeliveryDelay(1000)
    .retryAttemptedLogLevel(LoggingLevel.INFO));
```

> **NOTE**
>
> The **allowRedeliveryWhileStopping** option is **true** by default, for backwards compatibility reasons. During aggressive shutdown, however, redelivery is always suppressed, irrespective of this option setting (for example, after graceful shutdown has timed out).

Using onExceptionOccurred Processor

Dead Letter channel supports the **onExceptionOccurred** processor to allow the custom processing of a message, after an exception occurs. You can use it for custom logging too. Any new exceptions thrown from the **onExceptionOccurred** processor is logged as WARN and ignored, not to override the existing exception.

The difference between the onRedelivery processor and onExceptionOccurred processor is you can process the former exactly before the redelivery attempt. However, it does not happen immediately after an exception occurs. For example, If you configure the error handler to do five seconds delay between the redelivery attempts, then the redelivery processor is invoked five seconds later, after an exception occurs.

The following example explains how to do the custom logging when an exception occurs. You need to configure the **onExceptionOccurred** to use the custom processor.

```
errorHandler(defaultErrorHandler().maximumRedeliveries(3).redeliveryDelay(5000).onExceptionOccur
red(myProcessor));
```

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 6.1, "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.

OnPrepareFailure

Before you pass the exchange to the dead letter queue, you can use the **onPrepare** option to allow a custom processor to prepare the exchange. It enables you to add information about the exchange, such as the cause of exchange failure. For example, the following processor adds a header with the exception message.

```java
public class MyPrepareProcessor implements Processor {
    @Override
    public void process(Exchange exchange) throws Exception {
        Exception cause = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);
        exchange.getIn().setHeader("FailedBecause", cause.getMessage());
    }
}
```

You can configue the error handler to use the processor as follows.

```java
errorHandler(deadLetterChannel("jms:dead").onPrepareFailure(new MyPrepareProcessor()));
```

However, the **onPrepare** option is also available using the default error handler.

```xml
<bean id="myPrepare"
class="org.apache.camel.processor.DeadLetterChannelOnPrepareTest.MyPrepareProcessor"/>
```

```
<errorHandler id="dlc" type="DeadLetterChannel" deadLetterUri="jms:dead"
onPrepareFailureRef="myPrepare"/>
```

6.4. 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 6.4, "Guaranteed Delivery Pattern", by writing messages to persistent storage before attempting to deliver them to their destination.

Figure 6.4. Guaranteed Delivery Pattern
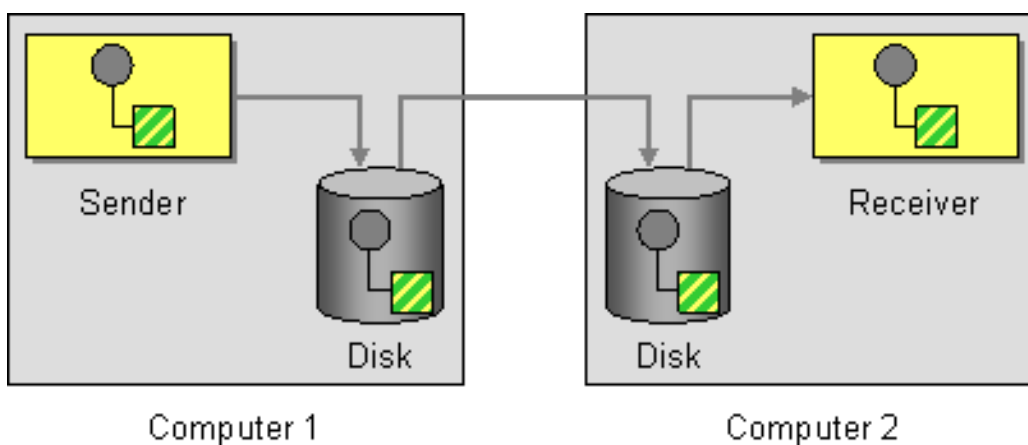
Components that support guaranteed delivery

The following Apache Camel components support the guaranteed delivery pattern:

- JMS

- ActiveMQ

- ActiveMQ Journal

- *File Component* in the *Apache Camel Component Reference Guide*

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 *Jms* in the *Apache Camel Component Reference Guide* > 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 enabe message persistence in ActiveMQ.

The simplest option (different from Figure 6.4, "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 been 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 6.4, "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>
```

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 amqPersistenceAdapter 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 *ActiveMQ* in the *Apache Camel Component Reference Guide* 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.

6.5. MESSAGE BUS

Overview

Message bus refers to a messaging architecture, shown in Figure 6.5, "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 6.5. Message Bus Pattern
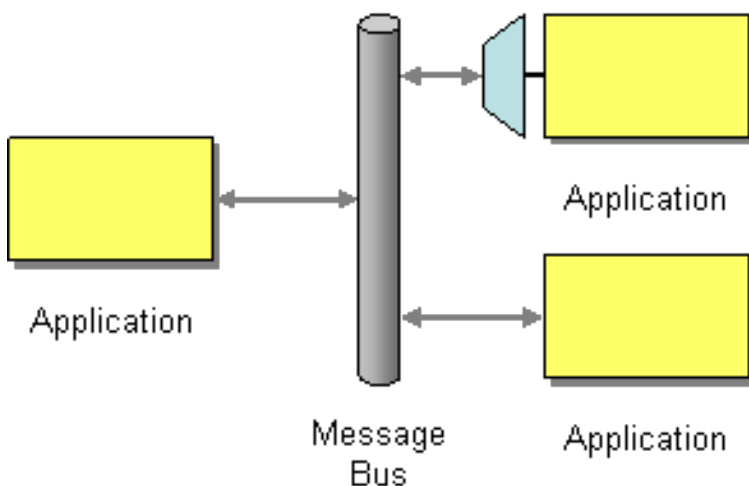
The following features of the message bus pattern are reflected in Apache Camel:

- *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 7. MESSAGE CONSTRUCTION

Abstract

The message construction patterns describe the various forms and functions of the messages that pass through the system.

7.1. CORRELATION IDENTIFIER

Overview

The *correlation identifier* pattern, shown in Figure 7.1, "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 Exchanges as a property with they key, **Exchange.CORRELATION_ID**, which links back to the source Exchanges. For example, the Splitter, Multicast, Recipient List, and Wire Tap EIPs do this.

Figure 7.1. Correlation Identifier Pattern
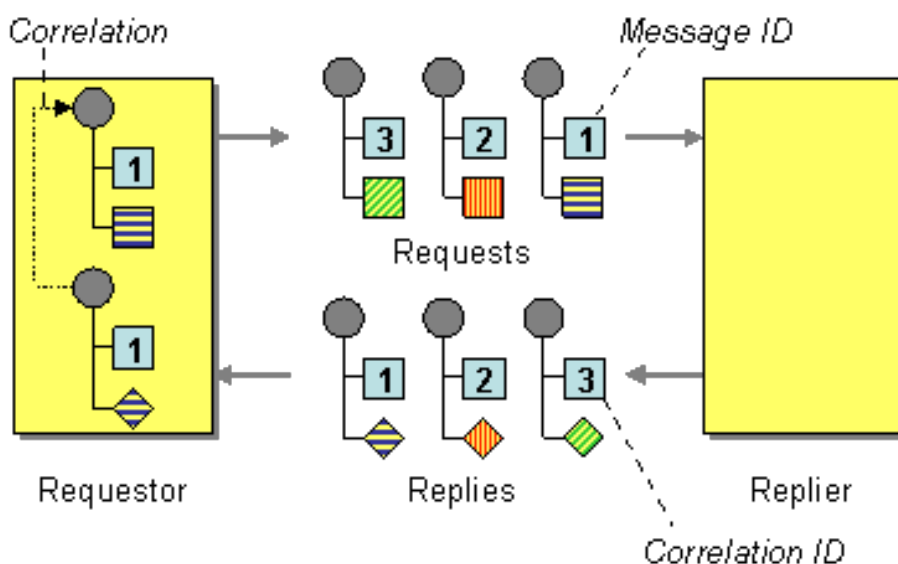
7.2. EVENT MESSAGE

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.

E = aPriceChangedEvent
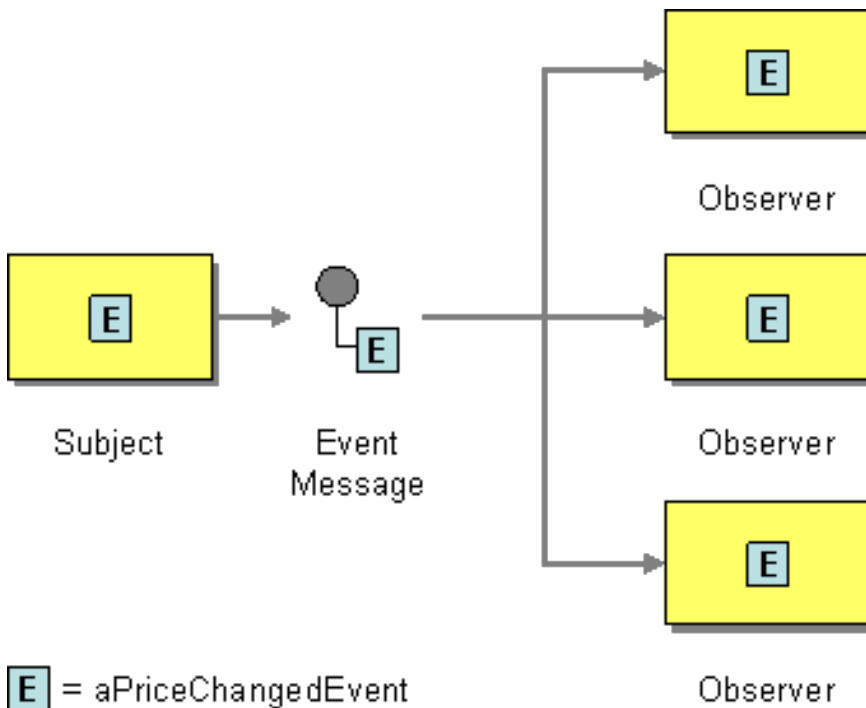
The default behaviour of many Components is InOnly such as for JMS, File or SEDA

Explicitly specifying InOnly

If you are using a component which defaults to InOut you can override the Message Exchange Pattern for an endpoint using the pattern property.

```
foo:bar?exchangePattern=InOnly
```

From 2.0 onwards on Camel you can specify the Message 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>
```

7.3. RETURN ADDRESS

Return Address

Apache Camel supports the Return Address from the Introducing Enterprise Integration Patterns using the **JMSReplyTo** header.
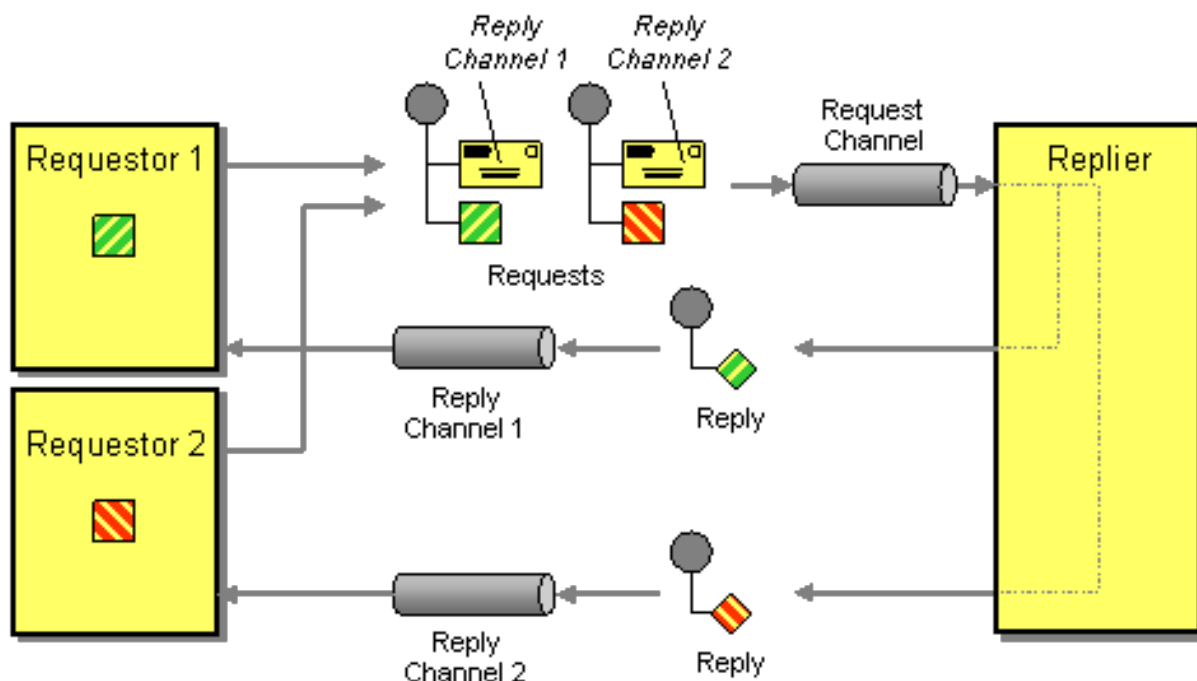
For example when using JMS with *InOut*, the component will by default be returned to the address given in **JMSReplyTo**.

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 8. 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).

8.1. CONTENT-BASED ROUTER

Overview

A *content-based router*, shown in Figure 8.1, "Content-Based Router Pattern" , enables you to route messages to the appropriate destination based on the message contents.

Figure 8.1. Content-Based Router Pattern
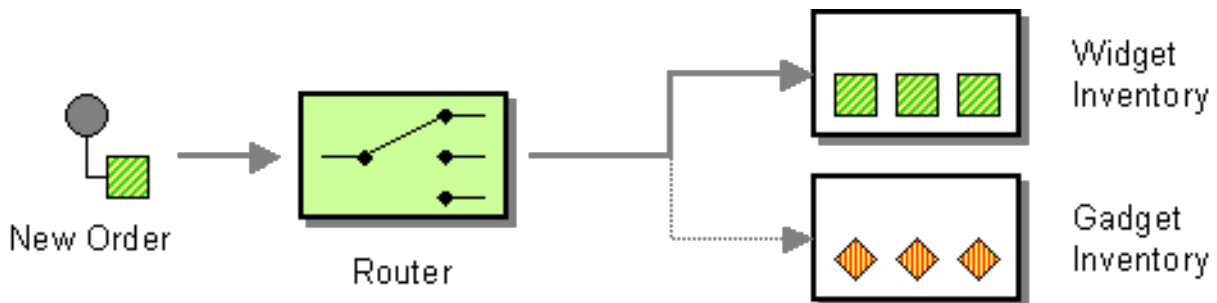
Java DSL example

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>
```

```
      <when>
       <xpath>$foo = 'cheese'</xpath>
       <to uri="seda:c"/>
      </when>
      <otherwise>
       <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

8.2. 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 8.2, "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 8.2. Message Filter Pattern

Widget Quote Gadget Quote Widget Quote

Message Filter

Widget Quote Widget Quote
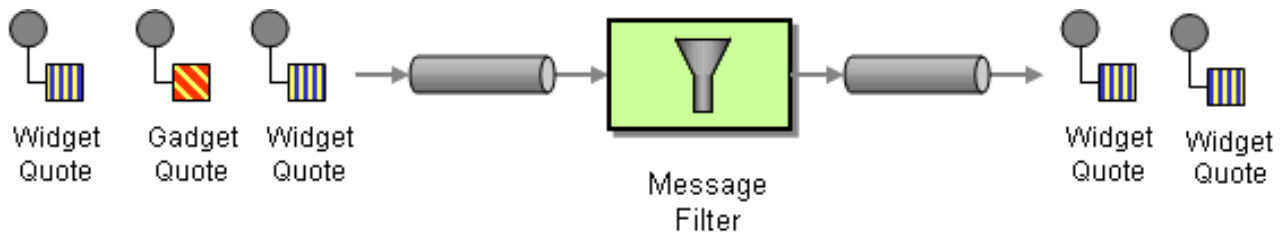
Java DSL example

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>
```
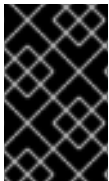
FILTERED ENDPOINT REQUIRED INSIDE </FILTER> TAG

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(bodyAs(String.class).contains("Hello")).to("mock:hello")
        .when(bodyAs(String.class).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 **Exchannge.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.

8.3. RECIPIENT LIST

Overview

A *recipient list*, shown in Figure 8.3, "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 8.3. Recipient List Pattern
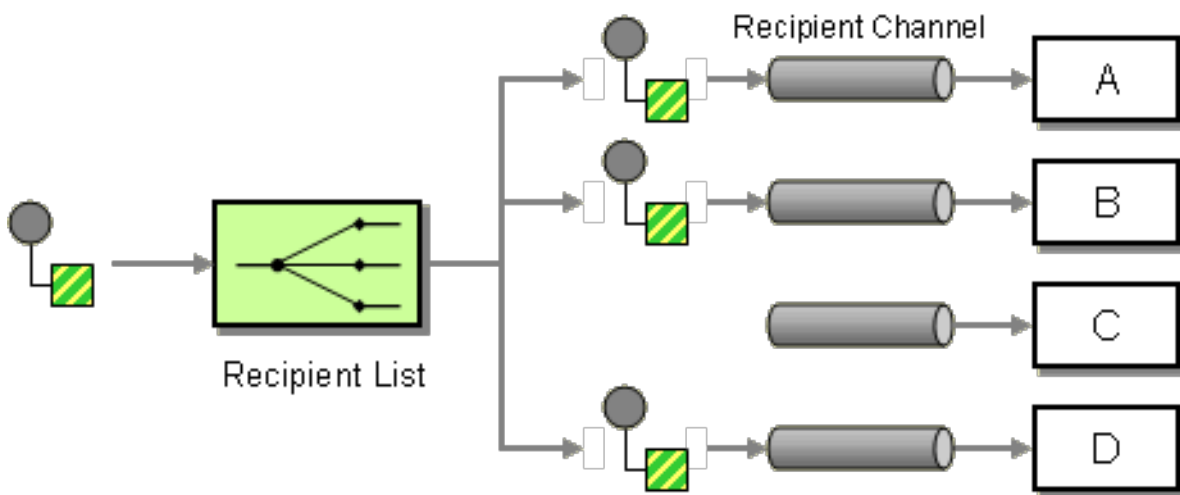
Recipient list with fixed destinations

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 43.3, "Built-In Type Converters" .

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>
```
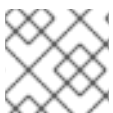
> **NOTE**
>
> You can combine **parallelProcessing** and **stopOnException** in the same route.

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 &lt;tt>null&lt;/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.
strategyMethodName		This option can be used to explicitly specify the method name to use, when using POJOs as the **AggregationStrategy**.

strategyMethodAllowNull	false	This option can be used, when using POJOs as the **AggregationStrategy**. If **false**, the **aggregate** method is not used, when there is no data to enrich. If **true**, **null** values are used for the **oldExchange**, when there is no data to enrich.
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.
parallelAggregate	false	If enabled, the **aggregate** method on **AggregationStrategy** can be called concurrently. Note that this requires the implementation of **AggregationStrategy** to be thread-safe. By default, this option is **false**, which means that Camel automatically synchronizes calls to the **aggregate** method. In some use-cases, however, you can improve performance by implementing **AggregationStrategy** as thread-safe and setting this option to **true**.
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.
timeout		**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 **timeout** method is invoked before breaking out.
onPrepareRef		**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.
shareUnitOfWork	**false**	**Camel 2.8:** Whether the unit of work should be shared. See the same option on Splitter for more details.

cacheSize	0	Camel 2.13.1/2.12.4: Allows to configure the cache size for the **ProducerCache** which caches producers for reuse in the routing slip. Will by default use the default cache size which is 0. Setting the value to -1 allows to turn off the cache all together.

Using Exchange Pattern in Recipient List

By default, the Recipient List uses the current exchange pattern. However, there may be few cases where you can send a message to a recipient using a different exchange pattern.

For example, you may have a route that initiates as a **InOnly** route. Now, If you want to use **InOut** exchange pattern with a recipient list, you need to configure the exchange pattern directly in the recipient endpoints.

The following example illustrates the route where the new files will start as InOnly and then route to a recipient list. If you want to use InOut with the ActiveMQ (JMS) endpoint, you need to specify this using the exchangePattern equals to InOut option. However, the response form the JMS request or reply will then be continued routed, and thus the response is stored in as a file in the outbox directory.

```
from("file:inbox")
    // the exchange pattern is InOnly initially when using a file route
    .recipientList().constant("activemq:queue:inbox?exchangePattern=InOut")
    .to("file:outbox");
```

NOTE

The **InOut** exchange pattern must get a response during the timeout. However, it fails if the response is not recieved.

8.4. 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 8.4, "Splitter Pattern" , is implemented by the **split()** Java DSL command.

Figure 8.4. Splitter Pattern
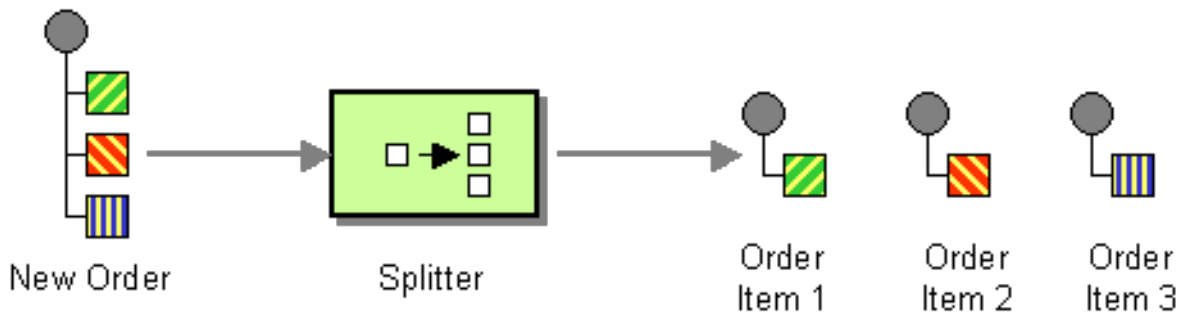
The Apache Camel splitter actually supports two patterns, as follows:

- *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 Part II, "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.

Skip first item

To skip the first item in the message you can use the **skipFirst** option.

In Java DSL, make the third option in the **tokenize** parameter **true**:

```
from("direct:start")
 // split by new line and group by 3, and skip the very first element
    .split().tokenize("\n", 3, true).streaming()
      .to("mock:group");
```

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

```
<route>
  <from uri="file:inbox"/>
    <split streaming="true">
    <tokenize token="\n" group="1000" skipFirst="true" />
    <to uri="activemq:queue:order"/>
  </split>
</route>
```

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("activemq: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 iteratable 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 couse
        // 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 iteratable 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.
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")
```
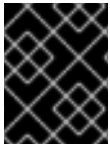
STREAMING AND XPATH

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 specifing 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 *What does the splitter return* below for whats used by default.
strategyMethodName		This option can be used to explicitly specify the method name to use, when using POJOs as the **AggregationStrategy**.
strategyMethodAllowNull	**false**	This option can be used, when using POJOs as the **AggregationStrategy**. If **false**, the **aggregate** method is not used, when there is no data to enrich. If **true**, **null** values are used for the **oldExchange**, when there is no data to enrich.
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.

parallelAggregate	**false**	If enabled, the **aggregate** method on **AggregationStrategy** can be called concurrently. Note that this requires the implementation of **AggregationStrategy** to be thread-safe. By default, this option is **false**, which means that Camel automatically synchronizes calls to the **aggregate** method. In some use-cases, however, you can improve performance by implementing **AggregationStrategy** as thread-safe and setting this option to **true**.
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.

timeout		**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 **timeout** method is invoked before breaking out.
onPrepareRef		**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.
shareUnitOfWork	**false**	**Camel 2.8:** Whether the unit of work should be shared. See further below for more details.

8.5. AGGREGATOR

Overview

The *aggregator* pattern, shown in Figure 8.5, "Aggregator Pattern", enables you to combine a batch of related messages into a single message.

Figure 8.5. Aggregator Pattern
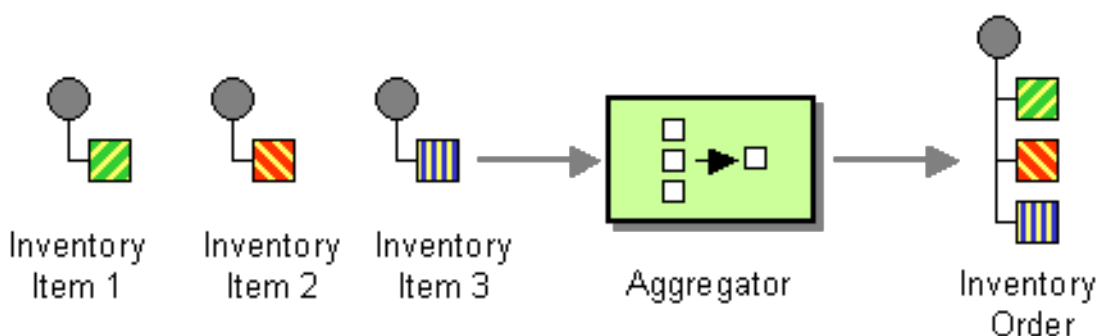
To control the aggregator's behavior, Apache Camel allows you to specify the properties described in *Enterprise Integration Patterns*, as follows:

- *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.)

> **NOTE**
>
> The Aggregator now enlists in JMX using a ManagedAggregateProcessorMBean that includes more information. It enables you to use the aggregate controller to control it.

How the aggregator works

Figure 8.6, "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 8.6. Aggregator Implementation
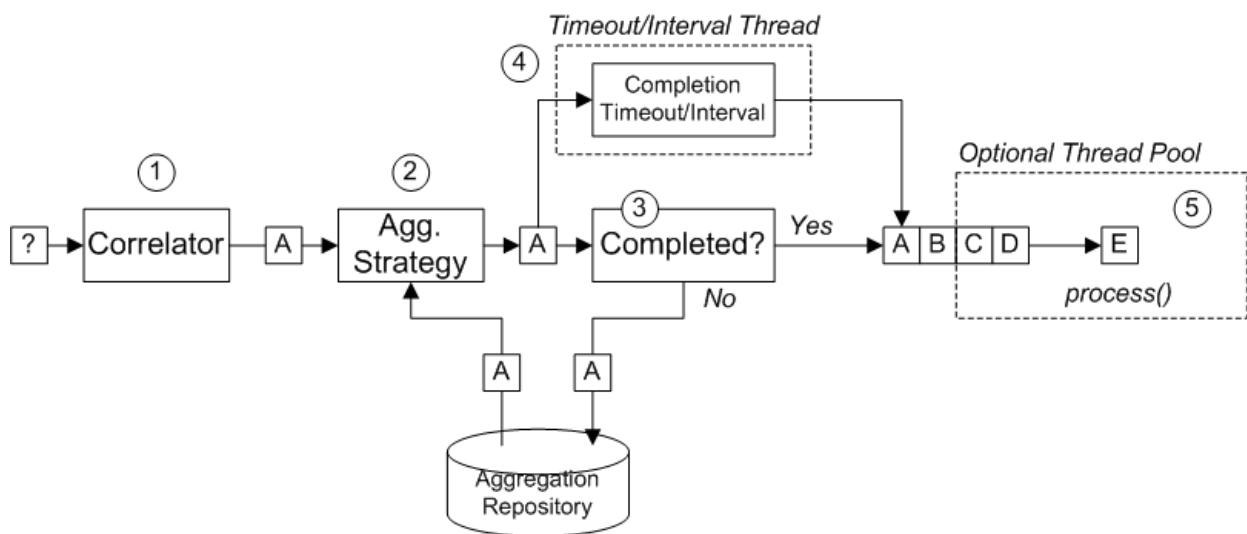
The incoming stream of exchanges shown in Figure 8.6, "Aggregator Implementation" is processed as follows:

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 8.6, "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.

> **NOTE**
>
> From Camel 2.16, the new XSLT Aggregation Strategy allows you to merge two messages with an XSLT file. You can access the **AggregationStrategies.xslt()** file from the toolbox.

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 8.6, "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 exampe, 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;
          }
        }
    }
```
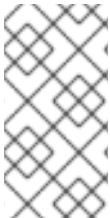
NOTE

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"/>
```

Controlling the lifecycle of a custom aggregation strategy

You can implement a custom aggregation strategy so that its lifecycle is aligned with the lifecycle of the enterprise integration pattern that is controlling it. This can be useful for ensuring that the aggregation strategy can shut down gracefully.

To implement an aggregation strategy with lifecycle support, you must implement the **org.apache.camel.Service** interface (in addition to the **AggregationStrategy** interface) and provide implementations of the **start()** and **stop()** lifecycle methods. For example, the following code example shows an outline of an aggregation strategy with lifecycle support:

```
// Java
import org.apache.camel.processor.aggregate.AggregationStrategy;
import org.apache.camel.Service;
import java.lang.Exception;
...
class MyAggStrategyWithLifecycleControl
    implements AggregationStrategy, Service {

  public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
     // Implementation not shown...
      ...
  }

  public void start() throws Exception {
     // Actions to perform when the enclosing EIP starts up
      ...
  }

  public void stop() throws Exception {
     // Actions to perform when the enclosing EIP is stopping
      ...
  }
}
```

Exchange properties

The following properties are set on each aggregated exchange:

Table 8.1. Aggregated Exchange Properties

Header	Type	Description
Exchange.AGGREGATED_SIZE	**int**	The total number of exchanges aggregated into this exchange.
Exchange.AGGREGATED_COMPLETED_BY	**String**	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 8.2. Redelivered Exchange Properties

Header	Type	Description
Exchange.REDELIVERY_CO UNTER	**int**	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. Alternatively, instead of setting this option, you can define a custom **AggregationStrategy** that implements the **Predicate** interface, in which case the **AggregationStrategy** will be used as the completion predicate.

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:

```xml
<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.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:

```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 of a single group from within an AggregationStrategy

If you implement a custom **AggregationStrategy** class, there is a mechanism available to force the completion of the current message group, by setting the **Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP** exchange property to **true** on the exchange returned from the **AggregationStrategy.aggregate()** method. This mechanism *only* affects the current group: other message groups (with different correlation IDs) are *not* forced to complete. This mechanism overrides any other completion mechanisms, such as predicate, size, timeout, and so on.

For example, the following sample **AggregationStrategy** class completes the current group, if the message body size is larger than 5:

```java
// Java
public final class MyCompletionStrategy implements AggregationStrategy {
    @Override
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        if (oldExchange == null) {
            return newExchange;
        }
        String body = oldExchange.getIn().getBody(String.class) + "+"
            + newExchange.getIn().getBody(String.class);
        oldExchange.getIn().setBody(body);
        if (body.length() >= 5) {
            oldExchange.setProperty(Exchange.AGGREGATION_COMPLETE_CURRENT_GROUP,
true);
        }
        return oldExchange;
    }
}
```

Forcing completion of all groups with a special message

It is possible to force completion of all outstanding aggregate messages, by sending a message with a special header to the route. There are two alternative header settings you can use to force completion:

Exchange.AGGREGATION_COMPLETE_ALL_GROUPS

Set to **true**, to force completion of the current aggregation cycle. This message acts purely as a signal and is *not* included in any aggregation cycle. After processing this signal message, the content of the message is discarded.

Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE

Set to **true**, to force completion of the current aggregation cycle. This message is *included* in the current aggregation cycle.

Using AggregateController

The org.apache.camel.processor.aggregate.AggregateController enables you to control the aggregate at runtime using Java or JMX API. This can be used to force completing groups of exchanges, or query the current runtime statistics.

If no custom have been configured, the aggregator provides a default implementation which you can access using the ***getAggregateController()*** method. However, it is easy to configure a controller in the route using aggregateController.

```
private AggregateController controller = new DefaultAggregateController();

from("direct:start")
  .aggregate(header("id"), new MyAggregationStrategy()).completionSize(10).id("myAggregator")
    .aggregateController(controller)
    .to("mock:aggregated");
```

Also, you can use the API on ***AggregateController*** to force completion. For example, to complete a group with key foo

```
int groups = controller.forceCompletionOfGroup("foo");
```

The number return would be the number of groups completed. Following is an API to complete all groups:

```
int groups = controller.forceCompletionOfAllGroups();
```

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 sent to **mock:result** contains the list of aggregated exchanges in the message body. 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.getIn().getBody(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).

> **NOTE**
>
> The old approach of accessing the grouped exchanges from a property on the outgoing exchange is now deprecated and will be removed in a future release.

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 8.7, "Recoverable Aggregation Repository" shows an overview of the recovery mechanism.

Figure 8.7. Recoverable Aggregation Repository
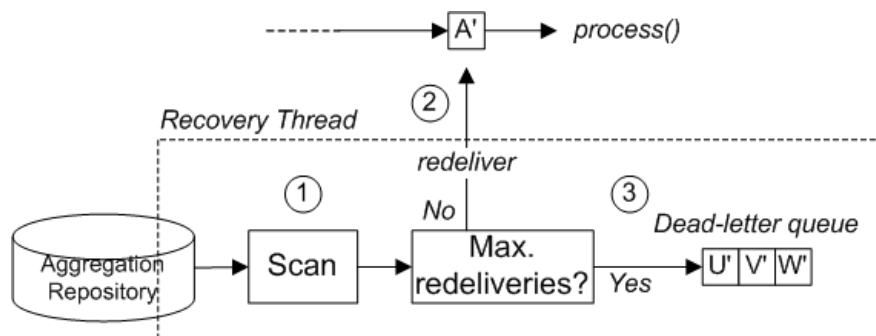
The recovery mechanism works as follows:

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 *HawtDB* in the *Apache Camel Component Reference Guide*.

Threading options

As shown in Figure 8.6, "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).

Aggregating into a List

A common aggregation scenario involves aggregating a series of incoming message bodies into a **List** object. To facilitate this scenario, Apache Camel provides the **AbstractListAggregationStrategy** abstract class, which you can quickly extend to create an aggregation strategy for this case. Incoming message bodies of type, **T**, are aggregated into a completed exchange, with a message body of type **List<T>**.

For example, to aggregate a series of **Integer** message bodies into a **List<Integer>** object, you could use an aggregation strategy defined as follows:

```
import org.apache.camel.processor.aggregate.AbstractListAggregationStrategy;
...
/**
 * Strategy to aggregate integers into a List<Integer>.
 */
public final class MyListOfNumbersStrategy extends AbstractListAggregationStrategy<Integer> {

    @Override
    public Integer getValue(Exchange exchange) {
        // the message body contains a number, so just return that as-is
        return exchange.getIn().getBody(Integer.class);
    }
}
```

Aggregator options

The aggregator supports the following options:

Table 8.3. Aggregator Options

Option	Default	Description
correlationExpression		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 **ignoreBadCorrelationKeys** option.
aggregationStrategy		Mandatory **AggregationStrategy** which is used to *merge* 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. From Camel 2.16 onwards, the strategy can also be a *PreCompletionAwareAggregationStrategy* implementation. It runs the completion check in a pre-completion mode.
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**.

Option	Default	Description
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 completionInterval, only one of the two can be used.
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 completionTimeout, only one of them can be used.
completionPredicate		Specifies a predicate (of **org.apache.camel.Predicate** type), which signals when an aggregated exchange is complete. Alternatively, instead of setting this option, you can define a custom **AggregationStrategy** that implements the **Predicate** interface, in which case the **AggregationStrategy** will be used as the completion predicate.
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 see *File* endpoint in that given poll.

Option	Default	Description
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 *aggregated* 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 *incoming* Exchange, 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.
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.

Option	Default	Description
closeCorrelationKeyOnCompletion		Whether or not *late* 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).
aggregationRepository		Allows you to plug in you 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.

Option	Default	Description
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
timeoutCheckerExecutorSer vice		If using one of the **completionTimeout**, **completionTimeoutExpressi on**, 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.
timeoutCheckerExecutorSer viceRef		Reference to look up a **timeoutCheckerExecutorSer vice** in the registry.
completeAllOnStop		When you stop the Aggregator, this option allows it to complete all pending exchanges from the aggregation repository.
optimisticLocking	**false**	Turns on optimistic locking, which can be used in combination with an aggregation repository.
optimisticLockRetryPolicy		Configures the retry policy for optimistic locking.

8.6. RESEQUENCER

Overview

The *resequencer* pattern, shown in Figure 8.8, "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 8.8. Resequencer Pattern
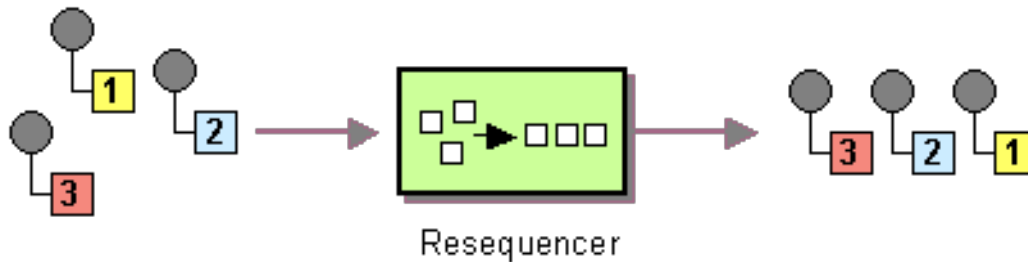
Apache Camel supports two resequencing algorithms:

- *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 8.4, "Batch Resequencer Options" shows the options that are available in batch mode only.

Table 8.4. Batch Resequencer Options

Java DSL	XML DSL	Default	Description
allowDuplicates()	**batch-config/@allowDuplicates**	**false**	If **true**, do not discard duplicate messages from the batch (where *duplicate* means that the message expression evaluates to the same value).
reverse()	**batch-config/@reverse**	**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");
```

Reject old messages

The **rejectOld** option can be used to prevent messages being sent out of order, regardless of the mechanism used to resequence messages. When the **rejectOld** option is enabled, the resequencer rejects an incoming message (by throwing a **MessageRejectedException** exception), if the incoming messages is *older* (as defined by the current comparator) than the last delivered message.

```
from("direct:start")
  .onException(MessageRejectedException.class).handled(true).to("mock:error").end()
  .resequence(header("seqno")).stream().timeout(1000).rejectOld()
  .to("mock:result");
```

8.7. ROUTING SLIP

Overview

The *routing slip* pattern, shown in Figure 8.9, "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 8.9. Routing Slip Pattern
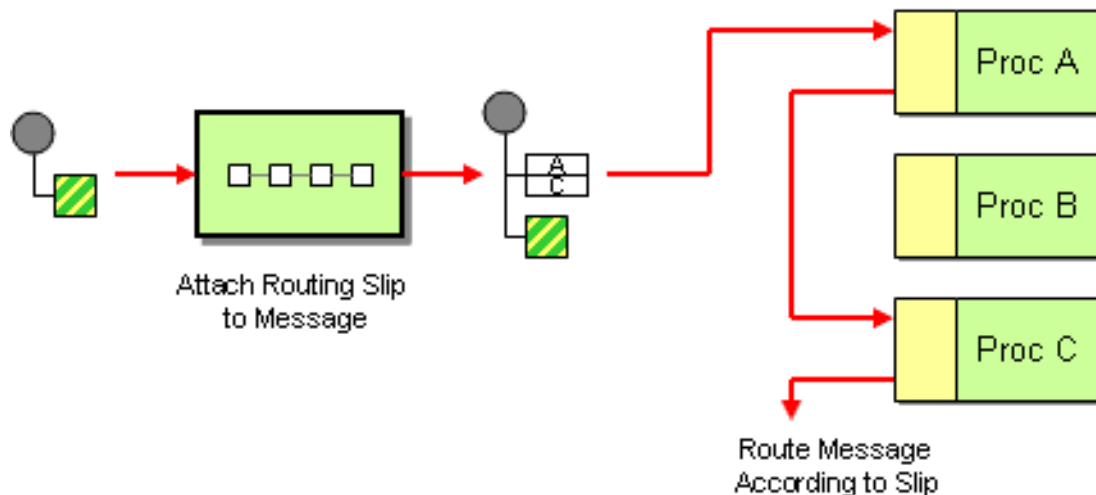
The slip header

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 though 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 thrown an exception stating the endpoint uri is not valid.

cacheSize	0	Camel 2.13.1/2.12.4: Allows to configure the cache size for the **ProducerCache** which caches producers for reuse in the routing slip. Will by default use the default cache size which is 0. Setting the value to -1 allows to turn off the cache all together.

8.8. 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 os 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");
```

> **NOTE**
>
> From Camel 2.17, the Throttler will use the rolling window for time periods that give a better flow of messages. However, It will enhance the performance of a throttler.

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.

8.9. 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&lt;/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.
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.

8.10. 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"](#)

- [Circuit Breaker](#)

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>
        <correlationExpression>
          <simple>header.username</simple>
        </correlationExpression>
      </sticky>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

> **NOTE**
>
> When you add the sticky option to the failover load balancer, the load balancer starts from the last known good endpoint.

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.
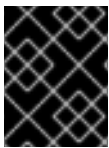
ENABLE STREAM CACHING IF USING STREAMS

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	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**.
			For example, the **errorHandler** might be configured to redeliver messages and use delays between attempts. Apache Camel will initially try to redeliver to the *original* endpoint, and only fail over to the next endpoint when the **errorHandler** is exhausted.
maximumFailoverAttempts	int	-1	Camel 2.3: Specifies the maximum number of attempts to fail over to a new endpoint. The value, **0**, implies that *no* failover attempts are made and the value, **-1**, implies an infinite number of failover attempts.
roundRobin	boolean	false	Camel 2.3: Specifies whether the **failover** load balancer should operate in round robin mode or not. If not, it will *always* 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 *stick* 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:

```xml
<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:

```xml
<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 described in the following table.

Table 8.5. 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;
    }
}
```

Circuit Breaker

The Circuit Breaker load balancer is a stateful pattern that is used to monitor all calls for certain exceptions. Initially, the Circuit Breaker is in closed state and passes all messages. If there are failures and the threshold is reached, it moves to open state and rejects all calls until **halfOpenAfter** timeout is reached. After the timeout, if there is a new call, the Circuit Breaker passes all the messages. If the result is success, the Circuit Breaker moves to a closed state, if not, it moves back to open state.

Java DSL example:

```
from("direct:start").loadBalance()
    .circuitBreaker(2, 1000L, MyCustomException.class)
    .to("mock:result");
```

Spring XML example:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
    <from uri="direct:start"/>
    <loadBalance>
        <circuitBreaker threshold="2" halfOpenAfter="1000">
            <exception>MyCustomException</exception>
        </circuitBreaker>
        <to uri="mock:result"/>
    </loadBalance>
    </route>
</camelContext>
```

8.11. MULTICAST

Overview

The *multicast* pattern, shown in Figure 8.10, "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 8.10. Multicast Pattern
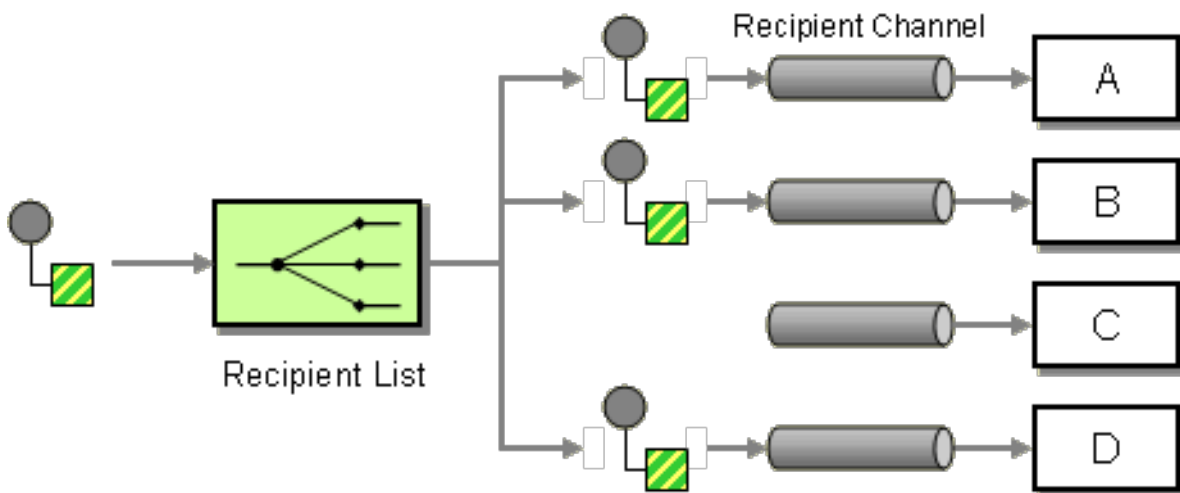
Multicast with a custom aggregation strategy

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 8.5, "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 mutateable 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 multicasted.
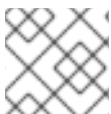
NOTE

It is 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.
strategyMethodName		This option can be used to explicitly specify the method name to use, when using POJOs as the **AggregationStrategy**.
strategyMethodAllowNull	**false**	This option can be used, when using POJOs as the **AggregationStrategy**. If **false**, the **aggregate** method is not used, when there is no data to enrich. If **true**, **null** values are used for the **oldExchange**, when there is no data to enrich.
parallelProcessing	**false**	If enabled, 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.

parallelAggregate	**false**	If enabled, the **aggregate** method on **AggregationStrategy** can be called concurrently. Note that this requires the implementation of **AggregationStrategy** to be thread-safe. By default, this option is **false**, which means that Camel automatically synchronizes calls to the **aggregate** method. In some use-cases, however, you can improve performance by implementing **AggregationStrategy** as thread-safe and setting this option to **true**.
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.

8.12. COMPOSED MESSAGE PROCESSOR

Composed Message Processor

The *composed message processor* pattern, as shown in Figure 8.11, "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 8.11. Composed Message Processor Pattern
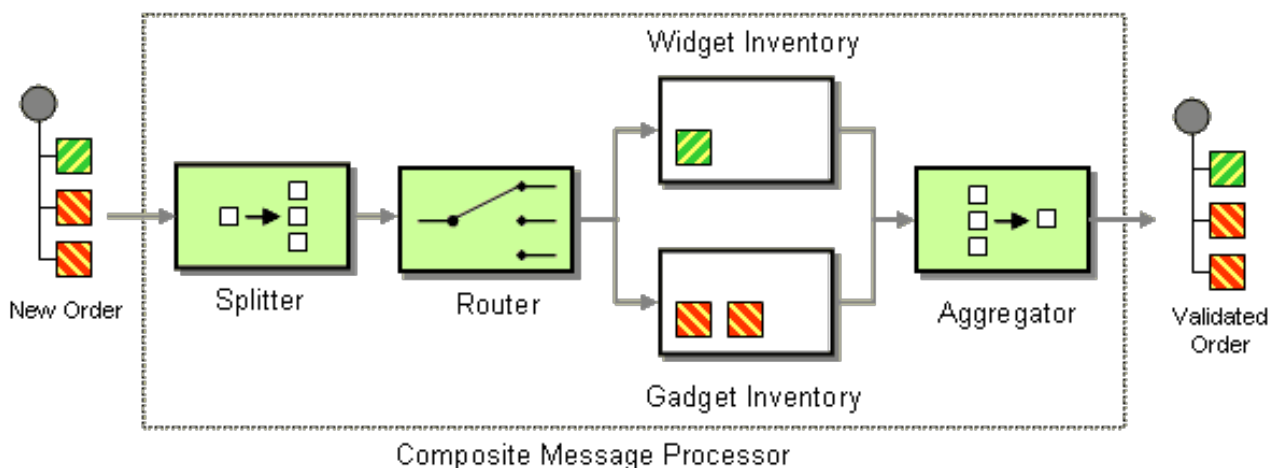
Java DSL example

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:

```xml
<route>
  <from uri="direct:start"/>
  <split>
    <simple>body</simple>
    <choice>
      <when>
        <method bean="orderItemHelper" method="isWidget"/>
  <to uri="bean:widgetInventory"/>
      </when>
      <otherwise>
  <to uri="bean:gadgetInventory"/>
      </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 **ComposedMessageProcessorTest.java** example source at camel-core/src/test/java/org/apache/camel/processor.

8.13. SCATTER-GATHER

Scatter-Gather

The *scatter-gather pattern*, as shown in Figure 8.12, "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 8.12. Scatter-Gather Pattern
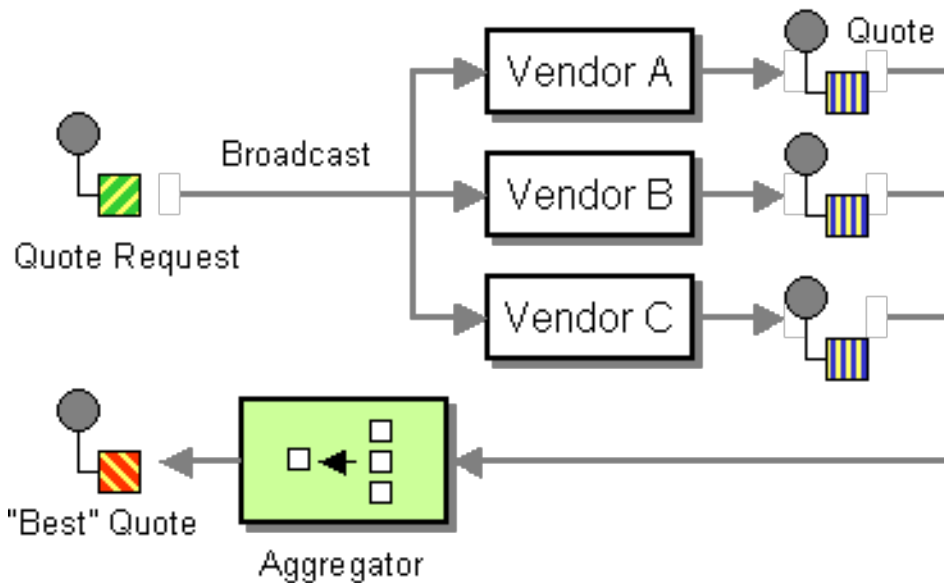
Dynamic scatter-gather example

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:

```xml
<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 8.1, "Messaging Client Sample" shows some sample code from a messaging client that adds the relevant header data to outgoing messages.

Example 8.1. 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.LowestQuoteAggregationStrategy"/>

<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 8.1, "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:

```java
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:

```java
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")
```

8.14. LOOP

Loop

The *loop* pattern enables you to process a message multiple times. It is used mainly for testing.
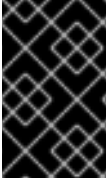
DEFAULT MODE

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 **RuntimeCamelException** 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, ABBB 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 *visible* for the next iteration. Instead you can enable copy mode, and then each iteration is *restarting* with a fresh copy of the input Exchange.

Do While Loop

You can perform the loop until a condition is met using a **do while** loop. The condition will either be true or false.

In DSL, the command is **LoopDoWhile**. The following example will perform the loop until the message body length is 5 characters or less:

```
from("direct:start")
    .loopDoWhile(simple("${body.length} <= 5"))
        .to("mock:loop")
        .transform(body().append("A"))
    .end()
    .to("mock:result");
```

In XML, the command is **loop doWhile**. The following example also performs the loop until the message body length is 5 characters or less:

```
<route>
  <from uri="direct:start"/>
  <loop doWhile="true">
    <simple>${body.length} <= 5</simple>
    <to uri="mock:loop"/>
    <transform>
      <simple>A${body}</simple>
    </transform>
  </loop>
  <to uri="mock:result"/>
</route>
```

8.15. 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.

8.16. DYNAMIC ROUTER

Dynamic Router

The Dynamic Router pattern, as shown in Figure 8.13, "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 8.13. Dynamic Router Pattern
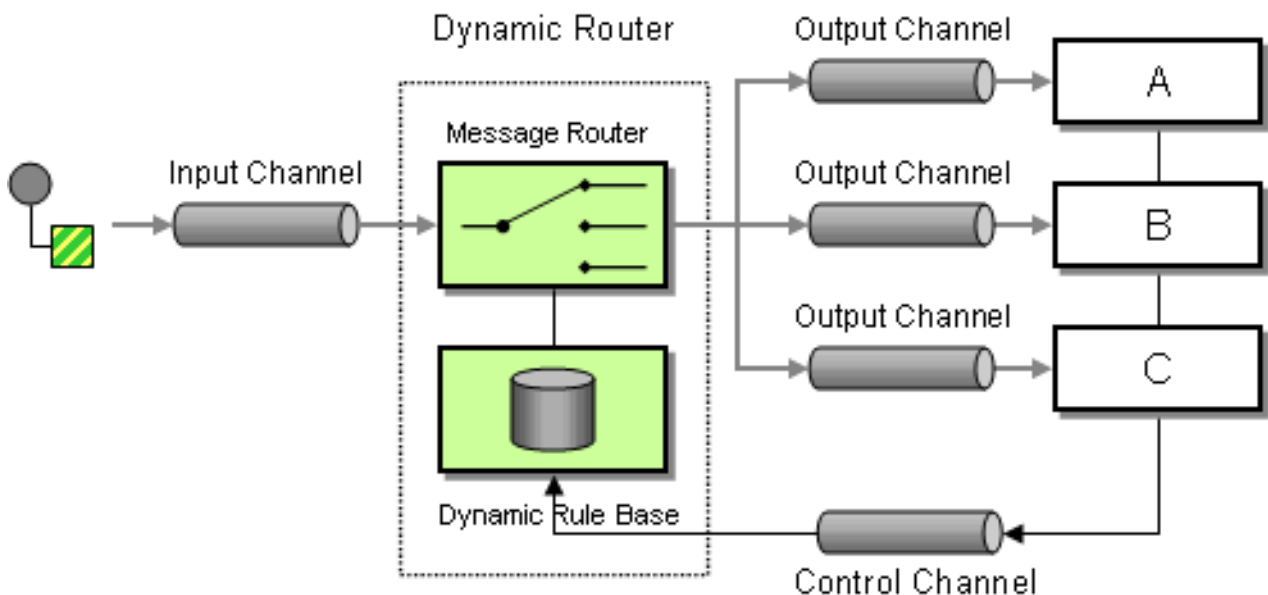
In **Camel 2.5** we introduced a **dynamicRouter** in the DSL, which is like a dynamic Routing Slip that evaluates the slip *on-the-fly*.

> **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
// 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:

```xml
<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>
    <to uri="mock:foo"/>
  </route>

</camelContext>
```

Options

The **dynamicRouter** 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 thrown an exception stating the endpoint uri is not valid.

@DynamicRouter annotation

You can also use the **@DynamicRouter** annotation. For example:

```java
// 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 9. MESSAGE TRANSFORMATION

Abstract

The message transformation patterns describe how to modify the contents of messages for various purposes.

9.1. 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 9.1. Content Enricher Pattern
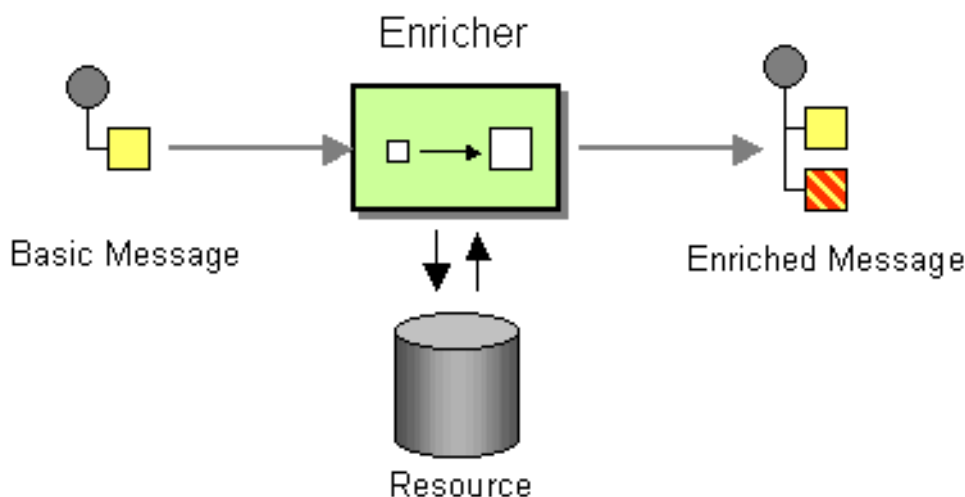
Models of content enrichment

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.

> **NOTE**
>
> Content Enricher with enrich and pollEnrich pattern supports dynamic endpoint **uris**. You can compute **uris** using an expression that enables you to use values from the current exchange. For example, you can poll a file with a name that is computed from the data exchange. This change breaks the XML DSL and enables you to migrate easily. The Java DSL stays backwards compatible.

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 *orginal 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 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 strategyRef="aggregationStrategy">
      <constant>direct:resource</constant>
    <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:

```
&lt;route&gt;
&lt;from uri="direct:start"/&gt;
&lt;enrich uri="direct:resource"/&gt;
&lt;to uri="direct:result"/&gt;
&lt;/route&gt;
```

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.
aggregateOnException		Refers to the Aggregate method. The aggregateOnException enables you to deal with exceptions. For example, you can suppress the exception or set a custom message.

shareUnitOfWork	false	Camel 2.16: Shares the unit of work with the parent and the resource exchange. However, by default, Enrich does not share the unit of work between the parent exchange and the resource exchange. Also, the resource exchange has its own individual unit of work.

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
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.
timeout	0	Timeout in millis to use when polling from the external service. See below for important details about the timeout.

9.2. 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 9.2. Content Filter Pattern
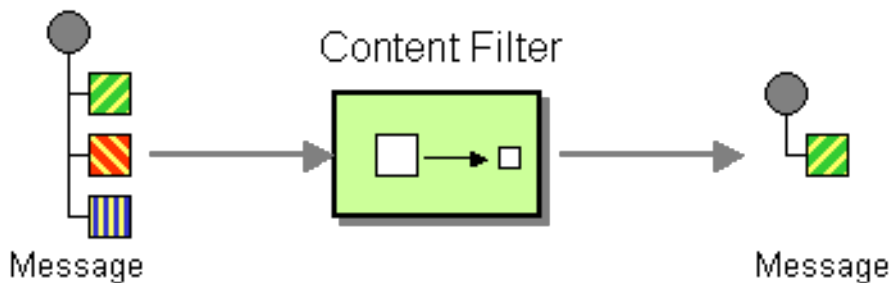
A common way to filter messages is to use an expression in the DSL, written in one of the supported scripting languages (for example, XSLT, XQuery or JoSQL).

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 44, *Implementing a Processor*.

- 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>
```

9.3. 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 9.3. Normalizer Pattern
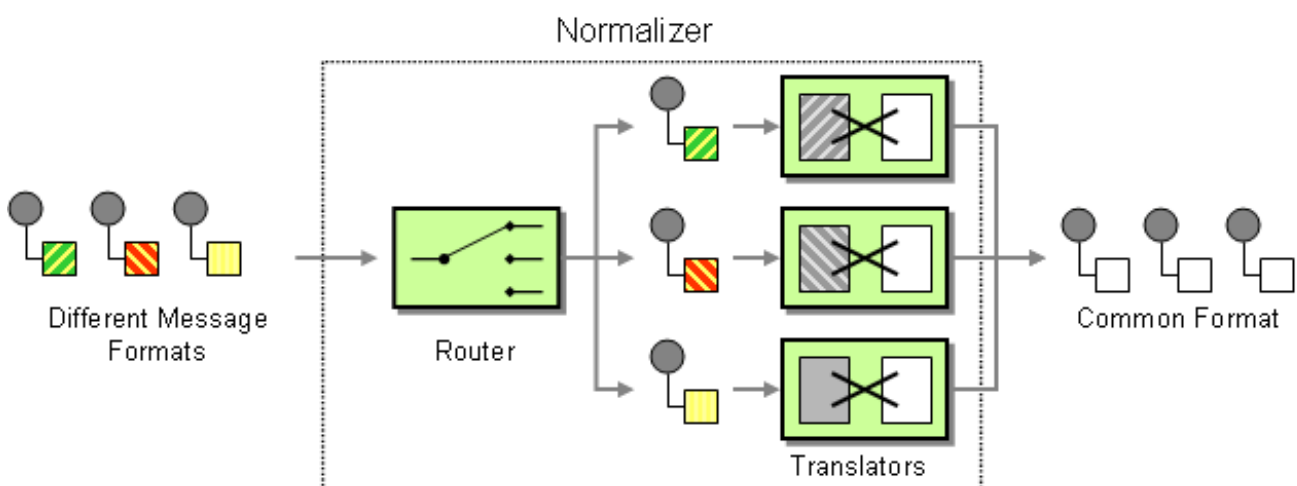
Java DSL example

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"/>
```

9.4. CLAIM CHECK

Claim Check

The *claim check* pattern, shown in Figure 9.4, "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 9.4. Claim Check Pattern
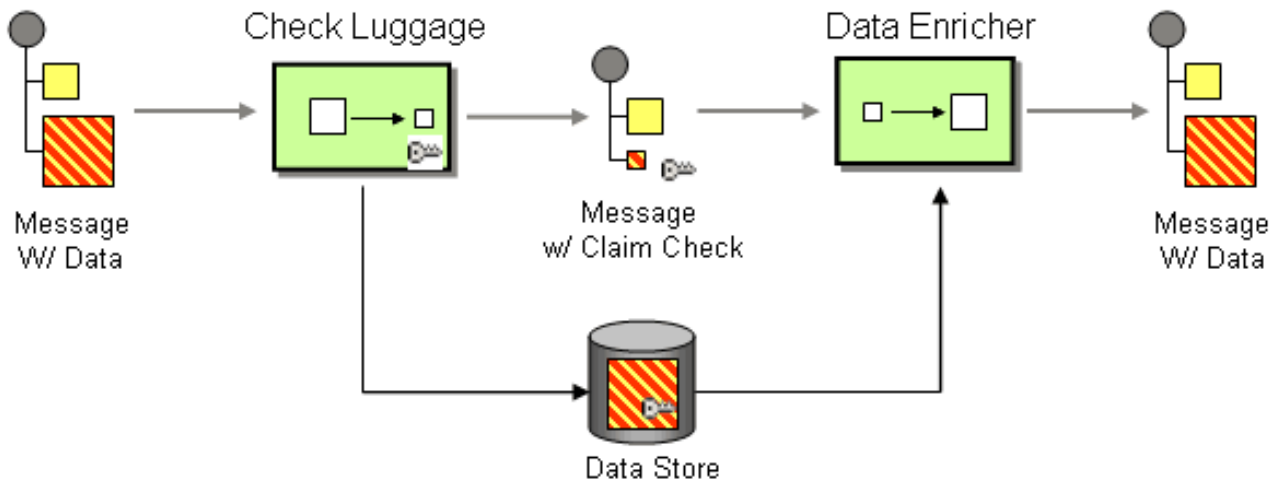
Java DSL example

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.

9.5. 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
comparatorRef		Refers to a custom **java.util.Comparator** to use for sorting the message body. Camel will by default use a comparator which does a A..Z sorting.

9.6. 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 10. MESSAGING ENDPOINTS

Abstract

The messaging endpoint patterns describe various features and qualities of service that can be configured on an endpoint.

10.1. 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.

10.2. EVENT DRIVEN CONSUMER

Overview

The _event-driven consumer_ pattern, shown in Figure 10.1, "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 10.1. Event Driven Consumer Pattern
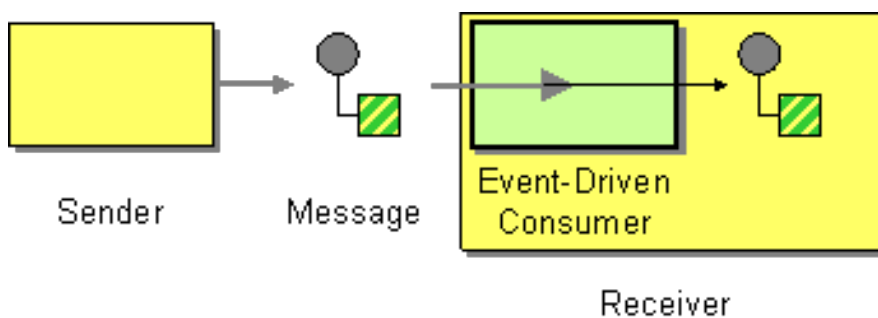
Consumers that conform to this pattern provide an event method that is automatically called by the messaging channel or transport layer whenever an incoming message is received. One of the characteristics of the event-driven consumer pattern is that the consumer endpoint itself does not provide any threads 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 47.1.3, "Consumer Patterns and Threading" and Chapter 50, _Consumer Interface_ .

10.3. POLLING CONSUMER

Overview

The _polling consumer_ pattern, shown in Figure 10.2, "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 47.1.3, "Consumer Patterns and Threading", Chapter 50, *Consumer Interface*, and Section 46.2, "Using the Consumer Template".

Figure 10.2. Polling Consumer Pattern
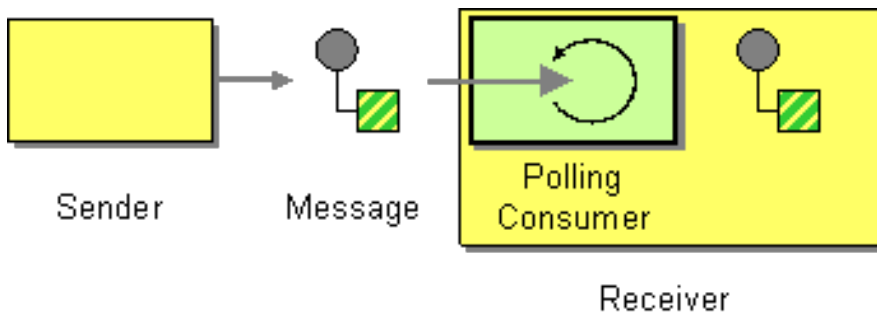
Scheduled poll consumer

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 50.2, "Implementing the Consumer Interface" 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 *Quartz* in the *Apache Camel Component Reference Guide* and Quartz Component for details.

10.4. COMPETING CONSUMERS

Overview

The *competing consumers* pattern, shown in Figure 10.3, "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 10.3. Competing Consumers Pattern
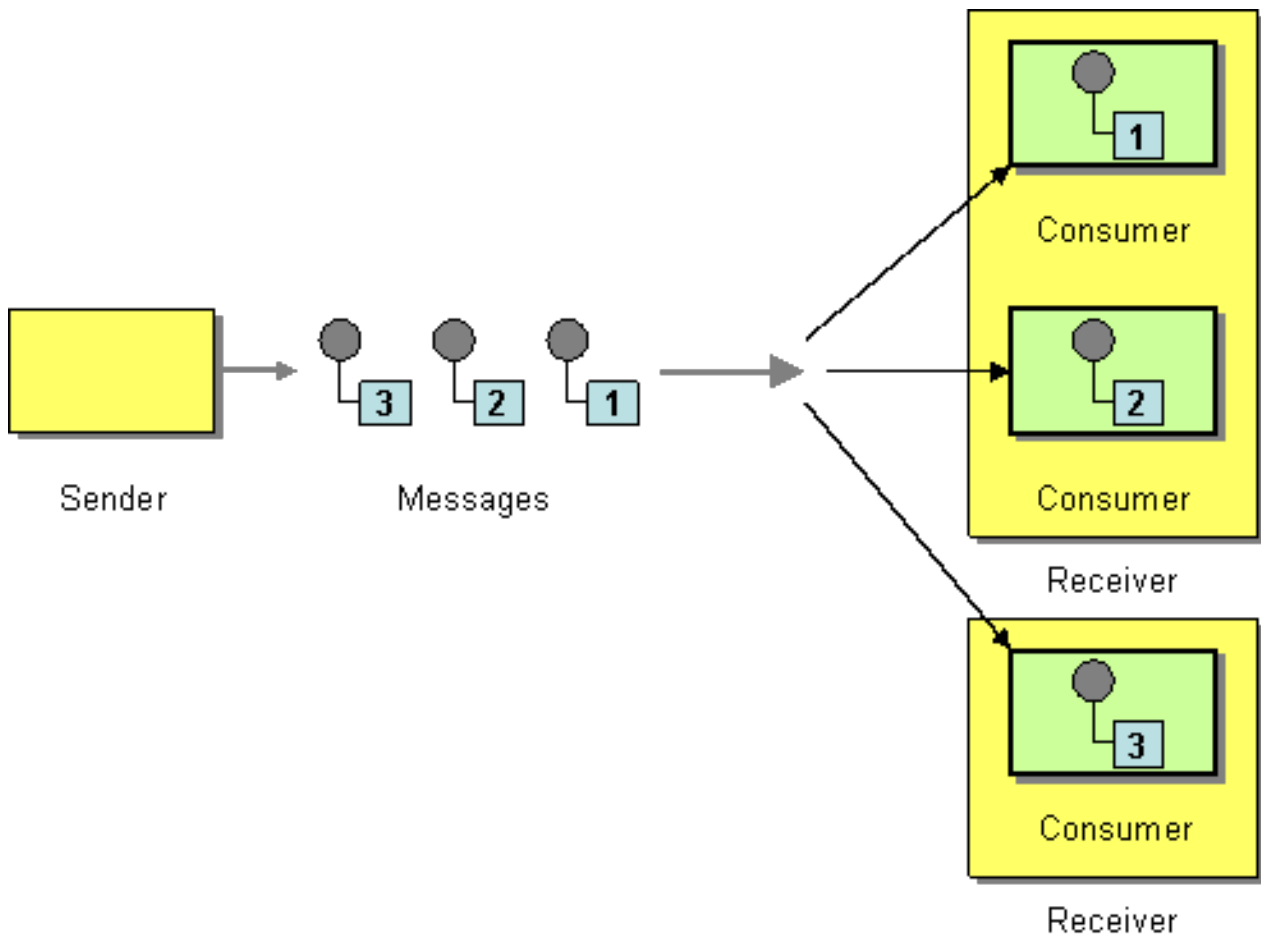
The following components demonstrate the competing consumers pattern:

- 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 "Apache Camel Component Reference" .

10.5. MESSAGE DISPATCHER

Overview

The *message dispatcher* pattern, shown in Figure 10.4, "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 10.4. Message Dispatcher Pattern
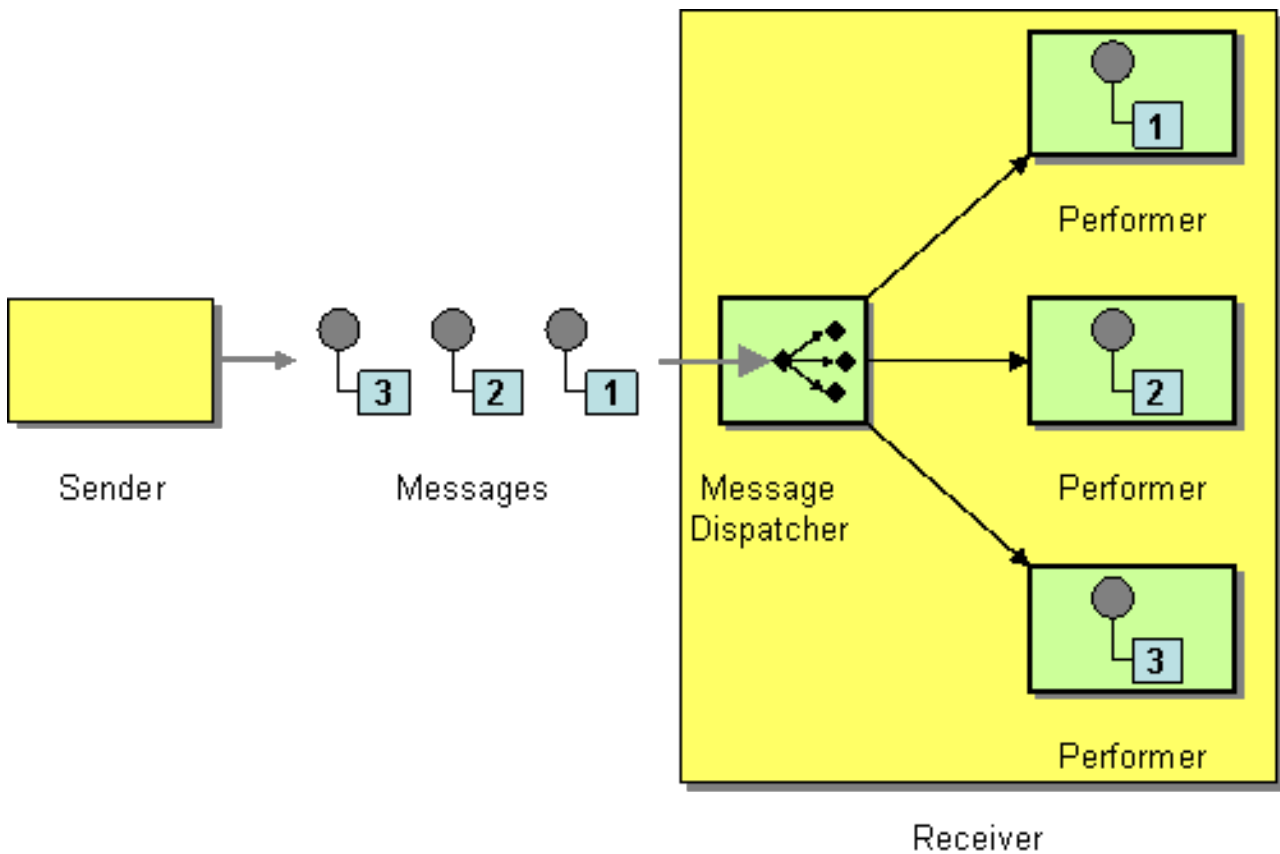
You can implement the message dispatcher pattern in Apache Camel using one of the following approaches:

- 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 8.1, "Content-Based Router".

10.6. SELECTIVE CONSUMER

Overview

The *selective consumer* pattern, shown in Figure 10.5, "Selective Consumer Pattern" , describes a consumer that applies a filter to incoming messages, so that only messages meeting specific selection criteria are processed.

Figure 10.5. Selective Consumer Pattern
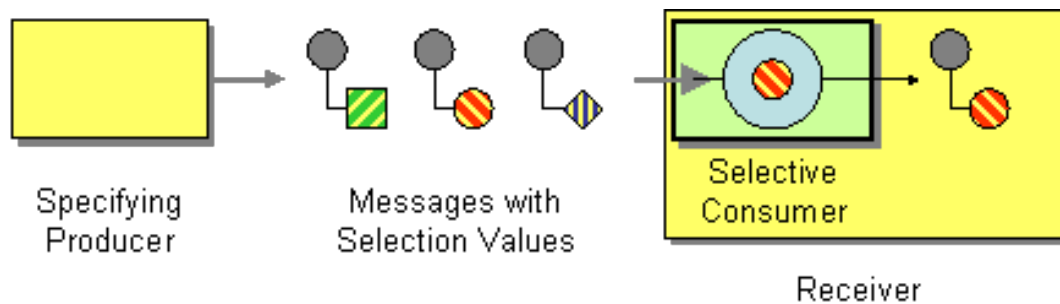
You can implement the selective consumer pattern in Apache Camel using one of the following approaches:

- 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 10.4, "Competing Consumers"), then blocked messages are not processed at all. This might not be the behavior you want.

10.7. DURABLE SUBSCRIBER

Overview

A *durable subscriber*, as shown in Figure 10.6, "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 10.6. Durable Subscriber Pattern
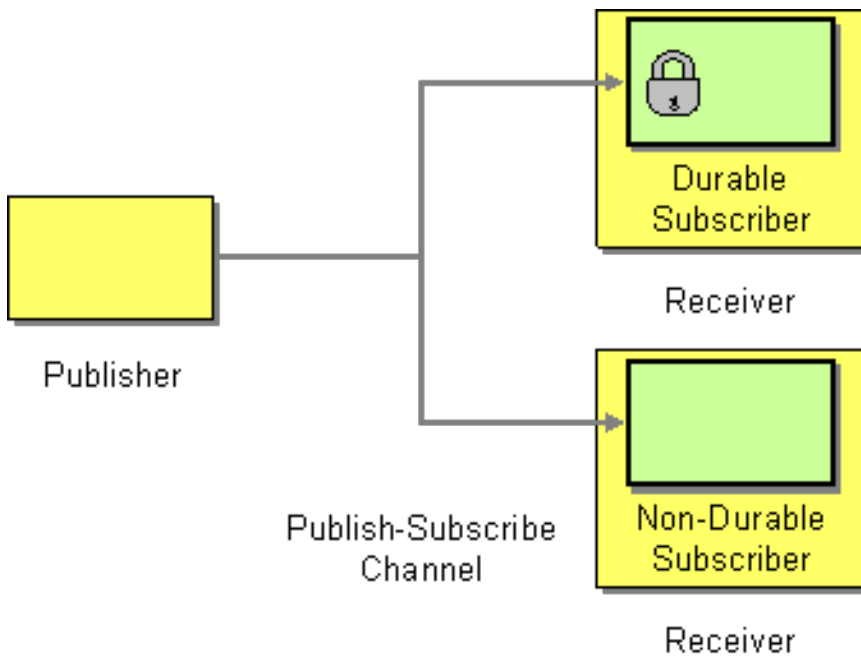
JMS durable subscriber

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 "Apache Camel 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>
```

10.8. 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 10.1, "Filtering Duplicate Messages with an In-memory Cache" uses the **TransactionID** header to filter out duplicates.

Example 10.1. Filtering Duplicate Messages with an In-memory Cache

```
import static
org.apache.camel.processor.idempotent.MemoryMessageIdRepository.memoryMessageIdRepositor
y;
...
```

```
    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>
```

> **NOTE**
>
> From Camel 2.17, Idempotent Repository supports optional serialized headers.

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.jpaMessageIdRepository;
...
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 -->
        <exchangeProperty>CamelDuplicateMessage</exchangeProperty>
        <!-- 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 Idempotent Repository tutorial to learn more about how to 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_MESSAG** exchange property set to a **Boolean.TRUE** value.

completionEager	**false**	**Camel 2.16:** Sets whether to complete the Idempotent consumer eager, when the exchange is done. If you set the **completeEager** option true, then the Idempotent Consumer triggers its completion when the exchange reaches till the end of the idempotent consumer pattern block. However, if the exchange continues to route even after the end block, then it does not affect the state of the idempotent consumer. If you set the **completeEager** option false, then the Idempotent Consumer triggers its completion after the exchange is done and is being routed. However, if the exchange continues to route even after the block ends, then it also affects the state of the idempotent consumer. For example, due to an exception if the exchange fails, then the state of the idempotent consumer will be a rollback.

10.9. TRANSACTIONAL CLIENT

Overview

The *transactional client* pattern, shown in Figure 10.7, "Transactional Client Pattern" , refers to messaging endpoints that can participate in a transaction. Apache Camel supports transactions using Spring transaction management.

Figure 10.7. Transactional Client Pattern
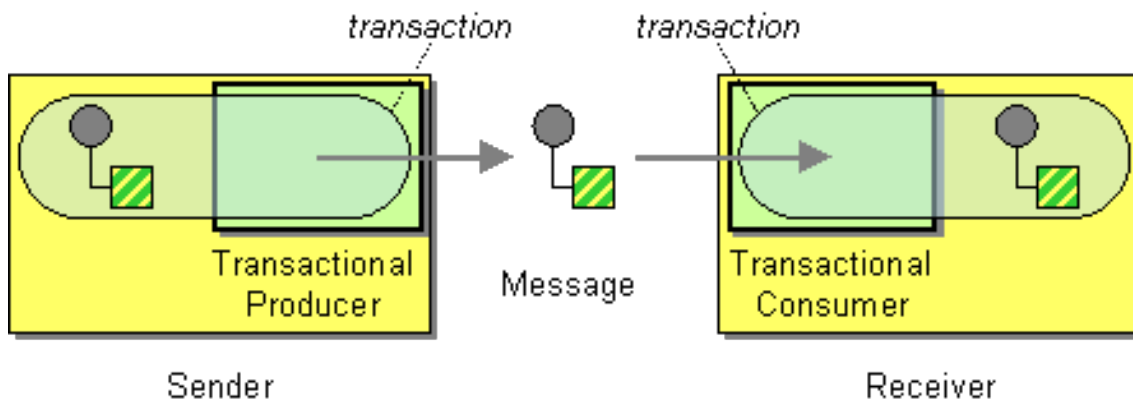
Transaction oriented endpoints

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*.

10.10. MESSAGING GATEWAY

Overview

The *messaging gateway* pattern, shown in Figure 10.8, "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 10.8. Messaging Gateway Pattern
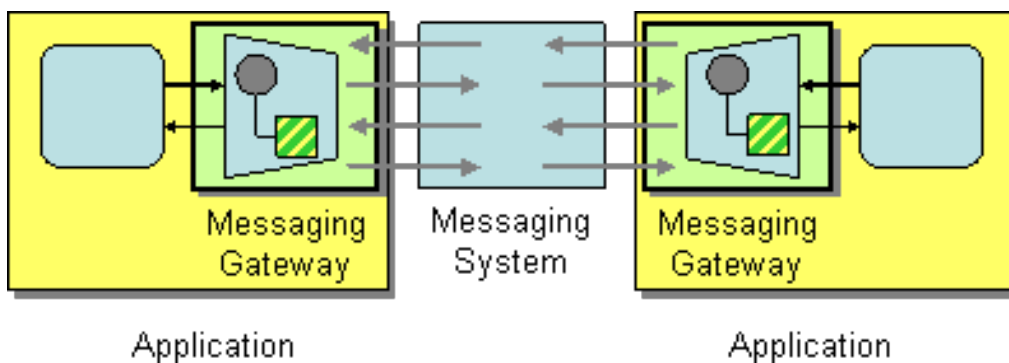
The following Apache Camel components provide this kind of integration with the messaging system:

- chapter "CXF" in "Apache Camel Component Reference"

- chapter "Bean" in "Apache Camel Component Reference"

10.11. SERVICE ACTIVATOR

Overview

The *service activator* pattern, shown in Figure 10.9, "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 10.9. Service Activator Pattern
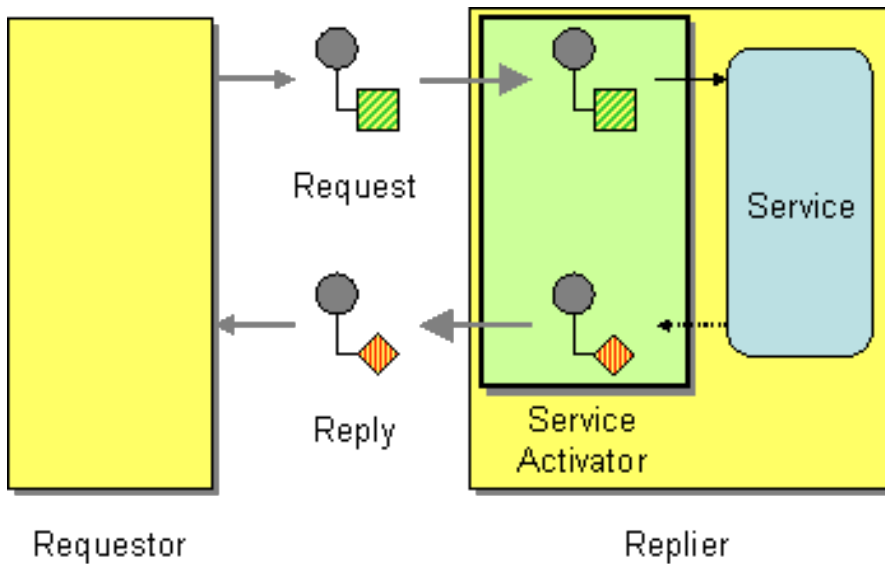
In many respects, a service activator resembles a conventional remote procedure call (RPC), where operation invocations are encoded as messages. The main difference is that a service activator needs to be more flexible. An RPC framework standardizes the request and reply message encodings (for example, 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**, identifes 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 11. SYSTEM MANAGEMENT

Abstract

The system management patterns describe how to monitor, test, and administer a messaging system.

11.1. 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.
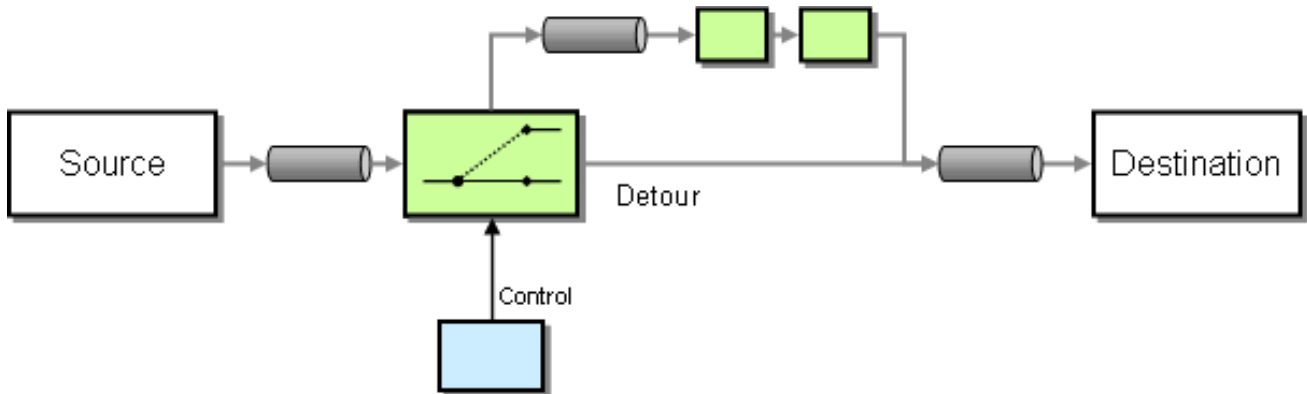
Example

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

11.2. 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 by 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 **LoggingLevel.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(LoggingLevel.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>
```

Global Log Name

The route ID is used as the the default log name. Since **Apache Camel 2.17** the log name can be changed by configuring a logname parameter.

Java DSL, configure the log name based on the following example:

```
CamelContext context = ...
context.getProperties().put(Exchange.LOG_EIP_NAME, "com.foo.myapp");
```

In XML, configure the log name in the following way:

```
<camelContext ...>
  <properties>
    <property key="CamelLogEipName" value="com.foo.myapp"/>
  </properties>
```

If you have more than one log and you want to have the same log name on all of them, you must add the configuration to each log.

11.3. WIRE TAP

Wire Tap

The *wire tap* pattern, as shown in Figure 11.1, "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 11.1. Wire Tap Pattern
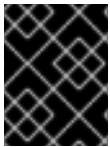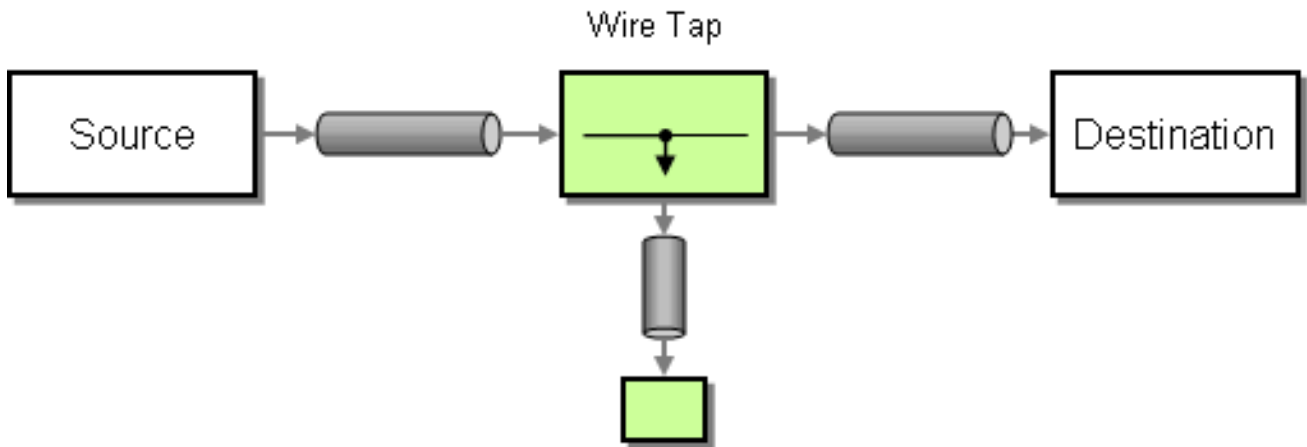
STREAMS

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.

NOTE

From Camel 2.16, the Wire Tap EIP emits event notifications when you send the exchange to the wire tap destination.

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 Processors 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 Dynamic URIs

Wire Tap supports dynamic endpoint URIs. The following example displays how to wire tap to a JMS queue where the header ID is a part of the queue name.

```
from("direct:start")
  .wireTap("jms:queue:backup-${header.id}")
  .to("bean:doSomething");
```

For more information about dynamic endpoint URIs, see the section called "Dynamic To".

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
uri		The endpoint uri where to send the wire tapped message. You should use either **uri** or **ref**.
ref		Refers to the endpoint where to send the wire tapped message. You should use either **uri** or **ref**.
executorServiceRef		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.
processorRef		Refers to a custom Processors 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 Processors 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.

CHAPTER 12. SERVICE COMPONENT RUNTIME

ABSTRACT

Service Component Runtime (SCR) is an implementation of OSGi Declarative Services specification. It enables any plain old Java object to expose and use OSGi services with no boilerplate code.

WORKING WITH CAMEL AND SCR

Apache Camel SCR component provides an integration of Camel with the OSGi Service Component Runtime.

OSGi framework has ability to know the object by looking at SCR descriptor file in its bundle which are generated from Java annotations by a plugin such as **org.apache.felix:maven-scr-plugin**. Using SCR, the bundle remains completely in Java world. There is no need to edit **XML** or properties file. It offers you full control over the project.

CREATING A SERVICE COMPONENT

Following are the steps to create a service component out of your java class.

1. Add the required **org.apache.felix.scr.annotations** at class level. For example,

   ```
   @Component
   @References({
       @Reference(name = "camelComponent",referenceInterface = ComponentResolver.class,
           cardinality = ReferenceCardinality.MANDATORY_MULTIPLE, policy =
   ReferencePolicy.DYNAMIC,
           policyOption = ReferencePolicyOption.GREEDY, bind = "gotCamelComponent", unbind
   = "lostCamelComponent")
   })
   ```

2. Implement the **getRouteBuilders()** method that returns the Camel route you want to run. For example,

   ```
   @Override
   protected List<RoutesBuilder> getRouteBuilders() {
       List<RoutesBuilder> routesBuilders = new ArrayList<>();
       routesBuilders.add(new YourRouteBuilderHere(registry));
       routesBuilders.add(new AnotherRouteBuilderHere(registry));
       return routesBuilders;
   }
   ```

3. Finally, enter the default configuration in annotations.

   ```
   @Properties({
       @Property(name = "camelContextId", value = "my-test"),
       @Property(name = "active", value = "true"),
       @Property(name = "...", value = "..."),
   ```

```
        ...
    })
```

EXAMPLE OF A SERVICE COMPONENT CLASS

The following example illustrates the use of **camel-archetype-scr** to generate a complete service component class:

```java
// This file was generated from org.apache.camel.archetypes/camel-archetype-scr/2.15-SNAPSHOT
package example;

import java.util.ArrayList;
import java.util.List;

import org.apache.camel.scr.AbstractCamelRunner;
import example.internal.CamelScrExampleRoute;
import org.apache.camel.RoutesBuilder;
import org.apache.camel.spi.ComponentResolver;
import org.apache.felix.scr.annotations.*;

@Component(label = CamelScrExample.COMPONENT_LABEL, description =
CamelScrExample.COMPONENT_DESCRIPTION, immediate = true, metatype = true)
@Properties({
    @Property(name = "camelContextId", value = "camel-scr-example"),
    @Property(name = "camelRouteId", value = "foo/timer-log"),
    @Property(name = "active", value = "true"),
    @Property(name = "from", value = "timer:foo?period=5000"),
    @Property(name = "to", value = "log:foo?showHeaders=true"),
    @Property(name = "messageOk", value = "Success: {{from}} -> {{to}}"),
    @Property(name = "messageError", value = "Failure: {{from}} -> {{to}}"),
    @Property(name = "maximumRedeliveries", value = "0"),
    @Property(name = "redeliveryDelay", value = "5000"),
    @Property(name = "backOffMultiplier", value = "2"),
    @Property(name = "maximumRedeliveryDelay", value = "60000")
})
@References({
    @Reference(name = "camelComponent",referenceInterface = ComponentResolver.class,
        cardinality = ReferenceCardinality.MANDATORY_MULTIPLE, policy =
ReferencePolicy.DYNAMIC,
        policyOption = ReferencePolicyOption.GREEDY, bind = "gotCamelComponent", unbind =
"lostCamelComponent")
})
public class CamelScrExample extends AbstractCamelRunner {

    public static final String COMPONENT_LABEL = "example.CamelScrExample";
    public static final String COMPONENT_DESCRIPTION = "This is the description for camel-scr-
example.";

    @Override
    protected List<RoutesBuilder> getRouteBuilders() {
        List<RoutesBuilder> routesBuilders = new ArrayList<>();
        routesBuilders.add(new CamelScrExampleRoute(registry));
        return routesBuilders;
```

```
      }
  }
```

EXAMPLE OF A ROUTEBUILDER CLASS

The following example illustrates the use of **camel-archetype-scr** to generate a **RouteBuilder** class:

```java
// This file was generated from org.apache.camel.archetypes/camel-archetype-scr/2.15-SNAPSHOT
package example.internal;

import org.apache.camel.LoggingLevel;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.SimpleRegistry;
import org.apache.commons.lang.Validate;

public class CamelScrExampleRoute extends RouteBuilder {

    SimpleRegistry registry;

    // Configured fields
    private String camelRouteId;
    private Integer maximumRedeliveries;
    private Long redeliveryDelay;
    private Double backOffMultiplier;
    private Long maximumRedeliveryDelay;

    public CamelScrExampleRoute(final SimpleRegistry registry) {
        this.registry = registry;
    }

    @Override
    public void configure() throws Exception {
        checkProperties();

        // Add a bean to Camel context registry
        registry.put("test", "bean");

        errorHandler(defaultErrorHandler()
            .retryAttemptedLogLevel(LoggingLevel.WARN)
            .maximumRedeliveries(maximumRedeliveries)
            .redeliveryDelay(redeliveryDelay)
            .backOffMultiplier(backOffMultiplier)
            .maximumRedeliveryDelay(maximumRedeliveryDelay));

        from("{{from}}")
            .startupOrder(2)
            .routeId(camelRouteId)
            .onCompletion()
                .to("direct:processCompletion")
            .end()
            .removeHeaders("CamelHttp*")
            .to("{{to}}");
```

```
    from("direct:processCompletion")
        .startupOrder(1)
        .routeId(camelRouteId + ".completion")
        .choice()
            .when(simple("${exception} == null"))
                .log("{{messageOk}}")
            .otherwise()
                .log(LoggingLevel.ERROR, "{{messageError}}")
        .end();
    }
}

public void checkProperties() {
    Validate.notNull(camelRouteId, "camelRouteId property is not set");
    Validate.notNull(maximumRedeliveries, "maximumRedeliveries property is not set");
    Validate.notNull(redeliveryDelay, "redeliveryDelay property is not set");
    Validate.notNull(backOffMultiplier, "backOffMultiplier property is not set");
    Validate.notNull(maximumRedeliveryDelay, "maximumRedeliveryDelay property is not set");
    }
}
```

USING APACHE CAMEL SCR BUNDLE AS A TEMPLATE

If you have a Camel SCR bundle that implements an integration pattern that you use frequently, then, probably there is no need to create a separate bundle for every instance.

The following example illustrates how to use a Camel SCR bundle as a template. It includes the following steps:

1. Create a configuration PID for your service component and add a tail with a dash.

2. Camel SCR will use the configuration to create a new instance of your component.

3. Finally, you can start a new **CamelContext** with your overridden properties.

   ```
   # Create a PID with a tail
   karaf@root> config:edit example.CamelScrExample-anotherone

   # Override some properties
   karaf@root> config:propset camelContextId my-other-context
   karaf@root> config:propset to "file://removeme?fileName=removemetoo.txt"

   # Save the PID
   karaf@root> config:update
   ```

   NOTE

   Make sure that your service component does not start with the default configuration. To prevent this, add **policy = ConfigurationPolicy.REQUIRE** to the class level at component annotation.

USING APACHE CAMEL-ARCHETYPE-SCR

With the help of Apache **camel-archetype-scr** and maven, you can easily create Apache Camel SCR bundle project. It includes the following steps:

1. Run the following command:

```
$ mvn archetype:generate -Dfilter=org.apache.camel.archetypes:camel-archetype-scr

Choose archetype:
local -> org.apache.camel.archetypes:camel-archetype-scr (Creates a new Camel SCR
bundle project for Karaf)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): : 1
Define value for property 'groupId': : example
[INFO] Using property: groupId = example
Define value for property 'artifactId': : camel-scr-example
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': example: :
[INFO] Using property: archetypeArtifactId = camel-archetype-scr
[INFO] Using property: archetypeGroupId = org.apache.camel.archetypes
[INFO] Using property: archetypeVersion = 2.15-SNAPSHOT
Define value for property 'className': : CamelScrExample
Confirm properties configuration:
groupId: example
artifactId: camel-scr-example
version: 1.0-SNAPSHOT
package: example
archetypeArtifactId: camel-archetype-scr
archetypeGroupId: org.apache.camel.archetypes
archetypeVersion: 2.15-SNAPSHOT
className: CamelScrExample
Y: :
```

2. Run Apache Maven.
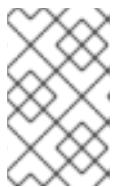
> **NOTE**
>
> For details on setting up Apache Maven to work with Red Hat JBoss Fuse, see Building with Maven in Red Hat JBoss Fuse Deploying into the Container on the Red Hat Customer Portal

3. You can now deploy the bundle. To deploy the bundle on Apache Karaf, perform the following steps on Karaf command line:

```
# Add Camel feature repository
karaf@root> features:chooseurl camel 2.15-SNAPSHOT

# Install camel-scr feature
karaf@root> features:install camel-scr

# Install commons-lang, used to validate parameters
karaf@root> osgi:install mvn:commons-lang/commons-lang/2.6

# Install and start your bundle
karaf@root> osgi:install -s mvn:example/camel-scr-example/1.0-SNAPSHOT
```

```
# View the log.
karaf@root> log:tail -n 10
```

Press ctrl-c to stop the log.

4. By default, the Service Component's configuration PID equals the fully qualified name of its class. You can change the properties of a bundle with Apache Karaf's **config.*** commands:

```
# Override the messageOk property
karaf@root> config:propset -p example.CamelScrExample messageOk "This is better logging".
```

You can also change the configuration by editing the property file in Apache Karaf's **etc** directory.

PART II. ROUTING EXPRESSION AND PREDICATE LANGUAGES

Abstract

This guide describes the basic syntax used by the evaluative languages supported by Apache Camel.

CHAPTER 13. INTRODUCTION

Abstract

This chapter provides an overview of all the expression languages supported by Apache Camel.

13.1. OVERVIEW OF THE LANGUAGES

Table of expression and predicate languages

Table 13.1, "Expression and Predicate Languages" gives an overview of the different syntaxes for invoking expression and predicate languages.

Table 13.1. Expression and Predicate Languages

Language	Static Method	Fluent DSL Method	XML Element	Annotation	Artifact
Section 2.4, "Bean Integration"	**bean()**	**EIP().method()**	**method**	**@Bean**	*Camel core*
Constant	**constant()**	**EIP().constant()**	**constant**	**@Constant**	*Camel core*
EL	**el()**	**EIP().el()**	**el**	**@EL**	**camel-juel**
Groovy	**groovy()**	**EIP().groovy()**	**groovy**	**@Groovy**	**camel-groovy**
Header	**header()**	**EIP().header()**	**header**	**@Header**	*Camel core*
JavaScript	**javaScript()**	**EIP().javaScript()**	**javaScript**	**@JavaScript**	**camel-script**
JoSQL	**sql()**	**EIP().sql()**	**sql**	**@SQL**	**camel-josql**
JSonPath	*None*	**EIP().jsonpath()**	**jsonpath**	**@JSonPath**	**camel-jsonpath**
JXPath	*None*	**EIP().jxpath()**	**jxpath**	**@JXPath**	**camel-jxpath**
MVEL	**mvel()**	**EIP().mvel()**	**mvel**	**@MVEL**	**camel-mvel**
OGNL	**ognl()**	**EIP().ognl()**	**ognl**	**@OGNL**	**camel-ognl**
PHP	**php()**	**EIP().php()**	**php**	**@PHP**	**camel-script**

Language	Static Method	Fluent DSL Method	XML Element	Annotation	Artifact
Property	**property()**	*EIP*().**property()**	**property**	**@Property**	*Camel core*
Python	**python()**	*EIP*().**python()**	**python**	**@Python**	**camel-script**
Ref	**ref()**	*EIP*().**ref()**	**ref**	*N/A*	*Camel core*
Ruby	**ruby()**	*EIP*().**ruby()**	**ruby**	**@Ruby**	**camel-script**
Simple/File	**simple()**	*EIP*().**simple()**	**simple**	**@Simple**	*Camel core*
SpEL	**spel()**	*EIP*().**spel()**	**spel**	**@SpEL**	**camel-spring**
XPath	**xpath()**	*EIP*().**xpath()**	**xpath**	**@XPath**	*Camel core*
XQuery	**xquery()**	*EIP*().**xquery()**	**xquery**	**@XQuery**	**camel-saxon**

13.2. HOW TO INVOKE AN EXPRESSION LANGUAGE

Prerequisites

Before you can use a particular expression language, you must ensure that the required JAR files are available on the classpath. If the language you want to use is not included in the Apache Camel core, you must add the relevant JARs to your classpath.

If you are using the Maven build system, you can modify the build-time classpath simply by adding the relevant dependency to your POM file. For example, if you want to use the Ruby language, add the following dependency to your POM file:

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

If you are going to deploy your application in a Red Hat JBoss Fuse OSGi container, you also need to ensure that the relevant language features are installed (features are named after the corresponding Maven artifact). For example, to use the Groovy language in the OSGi container, you must first install the **camel-groovy** feature by entering the following OSGi console command:

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

> **NOTE**
>
> If you are using an expression or predicate in the routes, refer the value as an external resource by using **resource:classpath:path** or **resource:file:path**. For example, **resource:classpath:com/foo/myscript.groovy**.

Camel on EAP deployment

The **camel-groovy** component is supported by the Camel on EAP (Wildfly Camel) framework, which offers a simplified deployment model on the Red Hat JBoss Enterprise Application Platform (JBoss EAP) container. For details of this model, see chapter "Apache Camel on JBoss EAP" in "Deploying into a Web Server".

Approaches to invoking

As shown in Table 13.1, "Expression and Predicate Languages", there are several different syntaxes for invoking an expression language, depending on the context in which it is used. You can invoke an expression language:

- the section called "As a static method" .

- the section called "As a fluent DSL method" .

- the section called "As an XML element".

- the section called "As an annotation".

As a static method

Most of the languages define a static method that can be used in *any* context where an **org.apache.camel.Expression** type or an **org.apache.camel.Predicate** type is expected. The static method takes a string expression (or predicate) as its argument and returns an **Expression** object (which is usually also a **Predicate** object).

For example, to implement a content-based router that processes messages in XML format, you could route messages based on the value of the **/order/address/countryCode** element, as follows:

```
from("SourceURL")
  .choice
    .when(xpath("/order/address/countryCode = 'us'"))
      .to("file://countries/us/")
    .when(xpath("/order/address/countryCode = 'uk'"))
      .to("file://countries/uk/")
    .otherwise()
      .to("file://countries/other/")
  .to("TargetURL");
```

As a fluent DSL method

The Java fluent DSL supports another style of invoking expression languages. Instead of providing the expression as an argument to an Enterprise Integration Pattern (EIP), you can provide the expression as a sub-clause of the DSL command. For example, instead of invoking an XPath expression as **filter(xpath("*Expression*"))**, you can invoke the expression as, **filter().xpath("*Expression*")**.

For example, the preceding content-based router can be re-implemented in this style of invocation, as follows:

```
from("SourceURL")
  .choice
    .when().xpath("/order/address/countryCode = 'us'")
      .to("file://countries/us/")
    .when().xpath("/order/address/countryCode = 'uk'")
      .to("file://countries/uk/")
    .otherwise()
      .to("file://countries/other/")
  .to("TargetURL");
```

As an XML element

You can also invoke an expression language in XML, by putting the expression string inside the relevant XML element.

For example, the XML element for invoking XPath in XML is **xpath** (which belongs to the standard Apache Camel namespace). You can use XPath expressions in a XML DSL content-based router, as follows:

```
<from uri="file://input/orders"/>
<choice>
  <when>
  <xpath>/order/address/countryCode = 'us'</xpath>
    <to uri="file://countries/us/"/>
  </when>
  <when>
  <xpath>/order/address/countryCode = 'uk'</xpath>
    <to uri="file://countries/uk/"/>
  </when>
  <otherwise>
    <to uri="file://countries/other/"/>
  </otherwise>
</choice>
```

Alternatively, you can specify a language expression using the **language** element, where you specify the name of the language in the **language** attribute. For example, you can define an XPath expression using the **language** element as follows:

```
<language language="xpath">/order/address/countryCode = 'us'</language>
```

As an annotation

Language annotations are used in the context of bean integration (see Section 2.4, "Bean Integration"). The annotations provide a convenient way of extracting information from a message or header and then injecting the extracted data into a bean's method parmeters.

For example, consider the bean, **myBeanProc**, which is invoked as a predicate of the **filter()** EIP. If the bean's **checkCredentials** method returns **true**, the message is allowed to proceed; but if the method returns **false**, the message is blocked by the filter. The filter pattern is implemented as follows:

```
// Java
```

```
MyBeanProcessor myBeanProc = new MyBeanProcessor();

from("SourceURL")
  .filter().method(myBeanProc, "checkCredentials")
  .to("TargetURL");
```

The implementation of the **MyBeanProcessor** class exploits the **@XPath** annotation to extract the **username** and **password** from the underlying XML message, as follows:

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

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

The **@XPath** annotation is placed just before the parameter into which it gets injected. Notice how the XPath expression *explicitly* selects the text node, by appending /**text()** to the path, which ensures that just the content of the element is selected, not the enclosing tags.

As a Camel endpoint URI

Using the Camel Language component, you can invoke a supported language in an endpoint URI. There are two alternative syntaxes.

To invoke a language script stored in a file (or other resource type defined by ***Scheme***), use the following URI syntax:

```
language://LanguageName:resource:Scheme:Location[?Options]
```

Where the scheme can be **file:**, **classpath:**, or **http:**.

For example, the following route executes the **mysimplescript.txt** from the classpath:

```
from("direct:start")
  .to("language:simple:classpath:org/apache/camel/component/language/mysimplescript.txt")
  .to("mock:result");
```

To invoke an embedded language script, use the following URI syntax:

```
language://LanguageName[:Script][?Options]
```

For example, to run the Simple language script stored in the **script** string:

```
String script = URLEncoder.encode("Hello ${body}", "UTF-8");
from("direct:start")
  .to("language:simple:" + script)
  .to("mock:result");
```

For more details about the Language component, see *Language* in the *Apache Camel Component Reference Guide*.

CHAPTER 14. CONSTANT

OVERVIEW

The constant language is a trivial built-in language that is used to specify a plain text string. This makes it possible to provide a plain text string in any context where an expression type is expected.

XML EXAMPLE

In XML, you can set the **username** header to the value, **Jane Doe** as follows:

```xml
<camelContext>
  <route>
    <from uri="SourceURL"/>
    <setHeader headerName="username">
      <constant>Jane Doe</constant>
    </setHeader>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

JAVA EXAMPLE

In Java, you can set the **username** header to the value, **Jane Doe** as follows:

```java
from("SourceURL")
  .setHeader("username", constant("Jane Doe"))
  .to("TargetURL");
```

CHAPTER 15. EL

OVERVIEW

The Unified Expression Language (EL) was originally specified as part of the JSP 2.1 standard (JSR–245), but it is now available as a standalone language. Apache Camel integrates with JUEL (http://juel.sourceforge.net/), which is an open source implementation of the EL language.

ADDING JUEL PACKAGE

To use EL in your routes you need to add a dependency on **camel-juel** to your project as shown in Example 15.1, "Adding the camel–juel dependency".

Example 15.1. Adding the camel–juel dependency

```
<!-- Maven POM File -->
<properties>
  <camel-version>2.17.0.redhat-630xxx</camel-version>
  ...
</properties>

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

STATIC IMPORT

To use the **el()** static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.language.juel.JuelExpression.el;
```

VARIABLES

Table 15.1, "EL variables" lists the variables that are accessible when using EL.

Table 15.1. EL variables

Variable	Type	Value
exchange	**org.apache.camel.Exchange**	The current Exchange
in	**org.apache.camel.Message**	The IN message

Variable	Type	Value
out	**org.apache.camel.Message**	The OUT message

EXAMPLE

Example 15.2, "Routes using EL" shows two routes that use EL.

Example 15.2. Routes using EL

```xml
<camelContext>
  <route>
    <from uri="seda:foo"/>
    <filter>
      <language language="el">${in.headers.foo == 'bar'}</language>
      <to uri="seda:bar"/>
    </filter>
  </route>
  <route>
    <from uri="seda:foo2"/>
    <filter>
      <language language="el">${in.headers['My Header'] == 'bar'}</language>
      <to uri="seda:bar"/>
    </filter>
  </route>
</camelContext>
```

CHAPTER 16. THE FILE LANGUAGE

Abstract

The file language is an extension to the simple language, not an independent language in its own right. But the file language extension can only be used in conjunction with File or FTP endpoints.

16.1. WHEN TO USE THE FILE LANGUAGE

Overview

The file language is an extension to the simple language which is not always available. You can use it under the following circumstances:

- the section called "In a File or FTP consumer endpoint" .

- the section called "On exchanges created by a File or FTP consumer" .

> **NOTE**
>
> The escape character, \, is not available in the file language.

In a File or FTP consumer endpoint

There are several URI options that you can set on a File or FTP consumer endpoint, which take a file language expression as their value. For example, in a File consumer endpoint URI you can set the **fileName**, **move**, **preMove**, **moveFailed**, and **sortBy** options using a file expression.

In a File consumer endpoint, the **fileName** option acts as a filter, determining which file will actually be read from the starting directory. If a plain text string is specified (for example, **fileName=report.txt**), the File consumer reads the same file each time it is updated. You can make this option more dynamic, however, by specifying a simple expression. For example, you could use a counter bean to select a different file each time the File consumer polls the starting directory, as follows:

```
file://target/filelanguage/bean/?fileName=${bean:counter.next}.txt&delete=true
```

Where the **${bean:counter.next}** expression invokes the **next()** method on the bean registered under the ID, **counter**.

The **move** option is used to move files to a backup location after then have been read by a File consumer endpoint. For example, the following endpoint moves files to a backup directory, after they have been processed:

```
file://target/filelanguage/?
move=backup/${date:now:yyyyMMdd}/${file:name.noext}.bak&recursive=false
```

Where the **${file:name.noext}.bak** expression modifies the original file name, replacing the file extension with **.bak**.

You can use the **sortBy** option to specify the order in which file should be processed. For example, to process files according to the alphabetical order of their file name, you could use the following File consumer endpoint:

> file://target/filelanguage/?sortBy=file:name

To process file according to the order in which they were last modified, you could use the following File consumer endpoint:

> file://target/filelanguage/?sortBy=file:modified

You can reverse the order by adding the **reverse:** prefix—for example:

> file://target/filelanguage/?sortBy=reverse:file:modified

On exchanges created by a File or FTP consumer

When an exchange originates from a File or FTP consumer endpoint, it is possible to apply file language expressions to the exchange throughout the route (as long as the original message headers are not erased). For example, you could define a content-based router, which routes messages according to their file extension, as follows:

```xml
<from uri="file://input/orders"/>
<choice>
  <when>
    <simple>${file:ext} == 'txt'</simple>
    <to uri="bean:orderService?method=handleTextFiles"/>
  </when>
  <when>
    <simple>${file:ext} == 'xml'</simple>
    <to uri="bean:orderService?method=handleXmlFiles"/>
  </when>
  <otherwise>
    <to uri="bean:orderService?method=handleOtherFiles"/>
  </otherwise>
</choice>
```

16.2. FILE VARIABLES

Overview

File variables can be used whenever a route starts with a File or FTP consumer endpoint, which implies that the underlying message body is of **java.io.File** type. The file variables enable you to access various parts of the file pathname, almost as if you were invoking the methods of the **java.io.File** class (in fact, the file language extracts the information it needs from message headers that have been set by the File or FTP endpoint).

Starting directory

Some of file variables return paths that are defined relative to a *starting directory*, which is just the directory that is specified in the File or FTP endpoint. For example, the following File consumer endpoint has the starting directory, **./filetransfer** (a relative path):

> file:filetransfer

The following FTP consumer endpoint has the starting directory, **./ftptransfer** (a relative path):

> ftp://myhost:2100/ftptransfer

Naming convention of file variables

In general, the file variables are named after corresponding methods on the **java.io.File** class. For example, the **file:absolute** variable gives the value that would be returned by the **java.io.File.getAbsolute()** method.

> **NOTE**
>
> This naming convention is not strictly followed, however. For example, there is *no* such method as **java.io.File.getSize()**.

Table of variables

Table 16.1, "Variables for the File Language" shows all of the variable supported by the file language.

Table 16.1. Variables for the File Language

Variable	Type	Description
file:name	**String**	The pathname relative to the starting directory.
file:name.ext	**String**	The file extension (characters following the last **.** character in the pathname). Supports file extensions with multiple dots, for example, .tar.gz.
file:name.ext.single	**String**	The file extension (characters following the last **.** character in the pathname). If the file extension has mutiple dots, then this expression only returns the last part.
file:name.noext	**String**	The pathname relative to the starting directory, omitting the file extension.
file:name.noext.single	**String**	The pathname relative to the starting directory, omitting the file extension. If the file extension has multiple dots, then this expression strips only the last part, and keep the others.
file:onlyname	**String**	The final segment of the pathname. That is, the file name without the parent directory path.

Variable	Type	Description
file:onlyname.noext	**String**	The final segment of the pathname, omitting the file extension.
file:onlyname.noext.single	**String**	The final segment of the pathname, omitting the file extension. If the file extension has multiple dots, then this expression strips only the last part, and keep the others.
file:ext	**String**	The file extension (same as **file:name.ext**).
file:parent	**String**	The pathname of the parent directory, including the starting directory in the path.
file:path	**String**	The file pathname, including the starting directory in the path.
file:absolute	**Boolean**	**true**, if the starting directory was specified as an absolute path; **false**, otherwise.
file:absolute.path	**String**	The absolute pathname of the file.
file:length	**Long**	The size of the referenced file.
file:size	**Long**	*Same as **file:length**.*
file:modified	**java.util.Date**	Date last modified.

16.3. EXAMPLES

Relative pathname

Consider a File consumer endpoint, where the starting directory is specified as a *relative pathname*. For example, the following File endpoint has the starting directory, **./filelanguage**:

> file://filelanguage

Now, while scanning the **filelanguage** directory, suppose that the endpoint has just consumed the following file:

> ./filelanguage/test/hello.txt

And, finally, assume that the **filelanguage** directory itself has the following absolute location:

> /workspace/camel/camel-core/target/filelanguage

Given the preceding scenario, the file language variables return the following values, when applied to the current exchange:

Expression	Result
file:name	**test/hello.txt**
file:name.ext	**txt**
file:name.noext	**test/hello**
file:onlyname	**hello.txt**
file:onlyname.noext	**hello**
file:ext	**txt**
file:parent	**filelanguage/test**
file:path	**filelanguage/test/hello.txt**
file:absolute	**false**
file:absolute.path	**/workspace/camel/camel-core/target/filelanguage/test/hello.txt**

Absolute pathname

Consider a File consumer endpoint, where the starting directory is specified as an *absolute pathname*. For example, the following File endpoint has the starting directory, **/workspace/camel/camel-core/target/filelanguage**:

> file:///workspace/camel/camel-core/target/filelanguage

Now, while scanning the **filelanguage** directory, suppose that the endpoint has just consumed the following file:

> ./filelanguage/test/hello.txt

Given the preceding scenario, the file language variables return the following values, when applied to the current exchange:

Expression	Result
file:name	test/hello.txt
file:name.ext	txt
file:name.noext	test/hello
file:onlyname	hello.txt
file:onlyname.noext	hello
file:ext	txt
file:parent	/workspace/camel/camel-core/target/filelanguage/test
file:path	/workspace/camel/camel-core/target/filelanguage/test/hello.txt
file:absolute	true
file:absolute.path	/workspace/camel/camel-core/target/filelanguage/test/hello.txt

CHAPTER 17. GROOVY

OVERVIEW

Groovy is a Java-based scripting language that allows quick parsing of object. The Groovy support is part of the **camel-groovy** module.

ADDING THE SCRIPT MODULE

To use Groovy in your routes you need to add a dependency on **camel-groovy** to your project as shown in Example 17.1, "Adding the camel-groovy dependency".

Example 17.1. Adding the camel-groovy dependency

```xml
<!-- Maven POM File -->
<properties>
  <camel-version>2.17.0.redhat-630xxx</camel-version>
  ...
</properties>

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

STATIC IMPORT

To use the **groovy()** static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

BUILT-IN ATTRIBUTES

Table 17.1, "Groovy attributes" lists the built-in attributes that are accessible when using Groovy.

Table 17.1. Groovy attributes

Attribute	Type	Value
context	**org.apache.camel.CamelContext**	The Camel Context
exchange	**org.apache.camel.Exchange**	The current Exchange

Attribute	Type	Value
request	**org.apache.camel.Message**	The IN message
response	**org.apache.camel.Message**	The OUT message
properties	**org.apache.camel.builder.script.PropertiesFunction**	Function with a **resolve** method to make it easier to use the properties component inside scripts.

The attributes all set at **ENGINE_SCOPE**.

EXAMPLE

Example 17.2, "Routes using Groovy" shows two routes that use Groovy scripts.

Example 17.2. Routes using Groovy

```xml
<camelContext>
  <route>
    <from uri="direct:items" />
    <filter>
      <language language="groovy">request.lineItems.any { i -> i.value > 100 }</language>
      <to uri="mock:mock1" />
    </filter>
  </route>
  <route>
    <from uri="direct:in"/>
    <setHeader headerName="firstName">
      <language language="groovy">$user.firstName $user.lastName</language>
    </setHeader>
    <to uri="seda:users"/>
  </route>
</camelContext>
```

USING THE PROPERTIES COMPONENT

To access a property value from the properties component, invoke the **resolve** method on the built-in **properties** attribute, as follows:

```
.setHeader("myHeader").groovy("properties.resolve(PropKey)")
```

Where *PropKey* is the key of the property you want to resolve, where the key value is of **String** type.

For more details about the properties component, see *Properties* in the *Apache Camel Component Reference Guide*.

CUSTOMIZING GROOVY SHELL

Sometimes, you might need to use the custom **GroovyShell** instance, in your Groovy expressions. To provide custom **GroovyShell**, add an implementation of the **org.apache.camel.language.groovy.GroovyShellFactory** SPI interface to your Camel registry.

For example, when you add the following bean to your Spring context, Apache Camel will use the custom **GroovyShell** instance that includes the custom static imports, instead of the default one.

```
public class CustomGroovyShellFactory implements GroovyShellFactory {

  public GroovyShell createGroovyShell(Exchange exchange) {
    ImportCustomizer importCustomizer = new ImportCustomizer();
    importCustomizer.addStaticStars("com.example.Utils");
    CompilerConfiguration configuration = new CompilerConfiguration();
    configuration.addCompilationCustomizers(importCustomizer);
    return new GroovyShell(configuration);
  }
}
```

CHAPTER 18. HEADER

OVERVIEW

The header language provides a convenient way of accessing header values in the current message. When you supply a header name, the header language performs a case-insensitive lookup and returns the corresponding header value.

The header language is part of **camel-core**.

XML EXAMPLE

For example, to resequence incoming exchanges according to the value of a **SequenceNumber** header (where the sequence number must be a positive integer), you can define a route as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <resequence>
      <language language="header">SequenceNumber</language>
    </resequence>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

JAVA EXAMPLE

The same route can be defined in Java, as follows:

```
from("SourceURL")
  .resequence(header("SequenceNumber"))
  .to("TargetURL");
```

CHAPTER 19. JAVASCRIPT

OVERVIEW

JavaScript, also known as ECMAScript is a Java-based scripting language that allows quick parsing of object. The JavaScript support is part of the **camel-script** module.

ADDING THE SCRIPT MODULE

To use JavaScript in your routes you need to add a dependency on **camel-script** to your project as shown in Example 19.1, "Adding the camel-script dependency".

Example 19.1. Adding the camel-script dependency

```xml
<!-- Maven POM File -->
<properties>
  <camel-version>2.17.0.redhat-630xxx</camel-version>
  ...
</properties>

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

STATIC IMPORT

To use the **javaScript()** static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

BUILT-IN ATTRIBUTES

Table 19.1, "JavaScript attributes" lists the built-in attributes that are accessible when using JavaScript.

Table 19.1. JavaScript attributes

Attribute	Type	Value
context	**org.apache.camel.CamelContext**	The Camel Context
exchange	**org.apache.camel.Exchange**	The current Exchange

Attribute	Type	Value
request	**org.apache.camel.Message**	The IN message
response	**org.apache.camel.Message**	The OUT message
properties	**org.apache.camel.builder.scr ipt.PropertiesFunction**	Function with a **resolve** method to make it easier to use the properties component inside scripts.

The attributes all set at **ENGINE_SCOPE**.

EXAMPLE

Example 19.2, "Route using JavaScript" shows a route that uses JavaScript.

Example 19.2. Route using JavaScript

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <langauge langauge="javaScript">request.headers.get('user') == 'admin'</langauge>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

USING THE PROPERTIES COMPONENT

To access a property value from the properties component, invoke the **resolve** method on the built-in **properties** attribute, as follows:

```
.setHeader("myHeader").javaScript("properties.resolve(PropKey)")
```

Where *PropKey* is the key of the property you want to resolve, where the key value is of **String** type.

For more details about the properties component, see *Properties* in the *Apache Camel Component Reference Guide*.

CHAPTER 20. JOSQL

OVERVIEW

The JoSQL (SQL for Java objects) language enables you to evaluate predicates and expressions in Apache Camel. JoSQL employs a SQL-like query syntax to perform selection and ordering operations on data from in-memory Java objects—however, JoSQL is *not* a database. In the JoSQL syntax, each Java object instance is treated like a table row and each object method is treated like a column name. Using this syntax, it is possible to construct powerful statements for extracting and compiling data from collections of Java objects. For details, see http://josql.sourceforge.net/.

ADDING THE JOSQL MODULE

To use JoSQL in your routes you need to add a dependency on **camel-josql** to your project as shown in Example 20.1, "Adding the camel-josql dependency".

Example 20.1. Adding the camel-josql dependency

```xml
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-josql</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

STATIC IMPORT

To use the **sql()** static method in your application code, include the following import statement in your Java source files:

```java
import static org.apache.camel.builder.sql.SqlBuilder.sql;
```

VARIABLES

Table 20.1, "SQL variables" lists the variables that are accessible when using JoSQL.

Table 20.1. SQL variables

Name	Type	Description
exchange	**org.apache.camel.Exchange**	The current Exchange
in	**org.apache.camel.Message**	The IN message

Name	Type	Description
out	**org.apache.camel.Message**	The OUT message
property	**Object**	the Exchange property whose key is *property*
header	**Object**	the IN message header whose key is *header*
variable	**Object**	the variable whose key is *variable*

EXAMPLE

Example 20.2, "Route using JoSQL" shows a route that uses JoSQL.

Example 20.2. Route using JoSQL

```xml
<camelContext>
  <route>
    <from uri="direct:start"/>
    <setBody>
      <language language="sql">select * from MyType</language>
    </setBody>
    <to uri="seda:regularQueue"/>
  </route>
</camelContext>
```

CHAPTER 21. JSONPATH

OVERVIEW

The JSonPath language provides a convenient syntax for extracting portions of a JSon message. The syntax of JSon is similar to XPath, but it is used to extract JSon objects from a JSon message, instead of acting on XML. The **jsonpath** DSL command can be used either as an expression or as a predicate (where an empty result gets interpreted as boolean **false**).

ADDING THE JSONPATH PACKAGE

To use JSonPath in your Camel routes, you need to add a dependency on **camel-jsonpath** to your project, as follows:

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

JAVA EXAMPLE

The following Java example shows how to use the **jsonpath()** DSL command to select items in a certain price range:

```
from("queue:books.new")
  .choice()
    .when().jsonpath("$.store.book[?(@.price < 10)]")
      .to("jms:queue:book.cheap")
    .when().jsonpath("$.store.book[?(@.price < 30)]")
      .to("jms:queue:book.average")
    .otherwise()
      .to("jms:queue:book.expensive")
```

If the JSonPath query returns an empty set, the result is interpreted as **false**. In this way, you can use a JSonPath query as a predicate.

XML EXAMPLE

The following XML example shows how to use the **jsonpath** DSL element to define predicates in a route:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <jsonpath>$.store.book[?(@.price &lt; 10)]</jsonpath>
        <to uri="mock:cheap"/>
      </when>
      <when>
        <jsonpath>$.store.book[?(@.price &lt; 30)]</jsonpath>
```

```
      <to uri="mock:average"/>
    </when>
    <otherwise>
     <to uri="mock:expensive"/>
    </otherwise>
   </choice>
  </route>
</camelContext>
```

SUPPRESS EXCEPTIONS

JSonPath will throw an exception if the path configured by the **jsonpath** expression is not found. The exception can be ignored by setting the **SuppressExceptions** option to true. For example, in the code below, adding the true option as part of the **jsonpath** parameters:

```
from("direct:start")
   .choice()
      // use true to suppress exceptions
      .when().jsonpath("person.middlename", true)
         .to("mock:middle")
      .otherwise()
         .to("mock:other");
```

In XML DSL use the following syntax:

```
<route>
  <from uri="direct:start"/>
  <choice>
    <when>
      <jsonpath suppressExceptions="true">person.middlename</jsonpath>
      <to uri="mock:middle"/>
    </when>
    <otherwise>
     <to uri="mock:other"/>
    </otherwise>
  </choice>
</route>
```

JSONPATH INJECTION

When using bean integration to invoke a bean method, you can use JSonPath to extract a value from the message and bind it to a method parameter. For example:

```
// Java
public class Foo {

   @Consume(uri = "activemq:queue:books.new")
   public void doSomething(@JsonPath("$.store.book[*].author") String author, @Body String json) {
     // process the inbound message here
   }
}
```

INTEGRATING JACKSON WITH CAMEL TYPECONVERTERS

The **camel-jackson** module allows to integrate Jackson as a Type Converter in the Camel Registry. You need to explicitly enable the **camel-jackson** by setting the CamelContext properties, as shown below:

```
// enable Jackson json type converter
getContext().getProperties().put("CamelJacksonEnableTypeConverter", "true");
// allow Jackson json to convert to pojo types also (by default jackson only converts to String and other simple types)
getContext().getProperties().put("CamelJacksonTypeConverterToPojo", "true");
```

The **camel-jackson** type converter integrates with JAXB. You can annotate POJO class with JAXB annotations that Jackson can leverage.

REFERENCE

For more details about JSonPath, see the [JSonPath project page]().

CHAPTER 22. JXPATH

OVERVIEW

The JXPath language enables you to invoke Java beans using the Apache Commons JXPath language. The JXPath language has a similar syntax to XPath, but instead of selecting element or attribute nodes from an XML document, it invokes methods on an object graph of Java beans. If one of the bean attributes returns an XML document (a DOM/JDOM instance), however, the remaining portion of the path is interpreted as an XPath expression and is used to extract an XML node from the document. In other words, the JXPath language provides a hybrid of object graph navigation and XML node selection.

ADDING JXPATH PACKAGE

To use JXPath in your routes you need to add a dependency on **camel-jxpath** to your project as shown in Example 22.1, "Adding the camel-jxpath dependency".

Example 22.1. Adding the camel-jxpath dependency

```xml
<!-- Maven POM File -->
<properties>
  <camel-version>2.17.0.redhat-630xxx</camel-version>
  ...
</properties>

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

VARIABLES

Table 22.1, "JXPath variables" lists the variables that are accessible when using JXPath.

Table 22.1. JXPath variables

Variable	Type	Value
this	**org.apache.camel.Exchange**	The current Exchange
in	**org.apache.camel.Message**	The IN message
out	**org.apache.camel.Message**	The OUT message

EXAMPLE

Example 22.2, "Routes using JXPath" shows a route that use JXPath.

Example 22.2. Routes using JXPath

```
<camelContext>
 <route>
   <from uri="activemq:MyQueue"/>
   <filter>
     <jxpath>in/body/name = 'James'</xpath>
     <to uri="mqseries:SomeOtherQueue"/>
   </filter>
 </route>
</camelContext>
```

CHAPTER 23. MVEL

OVERVIEW

MVEL is a Java-based dynamic language that is similar to OGNL, but is reported to be much faster. The MVEL support is in the **camel-mvel** module.

SYNTAX

You use the MVEL dot syntax to invoke Java methods, for example:

> getRequest().getBody().getFamilyName()

Because MVEL is dynamically typed, it is unnecessary to cast the message body instance (of **Object** type) before invoking the **getFamilyName()** method. You can also use an abbreviated syntax for invoking bean attributes, for example:

> request.body.familyName

ADDING THE MVEL MODULE

To use MVEL in your routes you need to add a dependency on **camel-mvel** to your project as shown in Example 23.1, "Adding the camel-mvel dependency".

> **Example 23.1. Adding the camel-mvel dependency**
>
> ```xml
> <!-- Maven POM File -->
> <properties>
>   <camel-version>2.17.0.redhat-630xxx</camel-version>
>   ...
> </properties>
>
> <dependencies>
>   ...
>   <dependency>
>     <groupId>org.apache.camel</groupId>
>     <artifactId>camel-mvel</artifactId>
>     <version>${camel-version}</version>
>   </dependency>
>   ...
> </dependencies>
> ```

BUILT-IN VARIABLES

Table 23.1, "MVEL variables" lists the built-in variables that are accessible when using MVEL.

Table 23.1. MVEL variables

Name	Type	Description
this	**org.apache.camel.Exchange**	The current Exchange
exchange	**org.apache.camel.Exchange**	The current Exchange
exception	**Throwable**	the Exchange exception (if any)
exchangeID	**String**	the Exchange ID
fault	**org.apache.camel.Message**	The Fault message(if any)
request	**org.apache.camel.Message**	The IN message
response	**org.apache.camel.Message**	The OUT message
properties	**Map**	The Exchange properties
property(*name*)	**Object**	The value of the named Exchange property
property(*name*, *type*)	***Type***	The typed value of the named Exchange property

EXAMPLE

Example 23.2, "Route using MVEL" shows a route that uses MVEL.

> **Example 23.2. Route using MVEL**
>
> ```xml
> <camelContext>
>   <route>
>     <from uri="seda:foo"/>
>     <filter>
>       <language langauge="mvel">request.headers.foo == 'bar'</language>
>       <to uri="seda:bar"/>
>     </filter>
>   </route>
> </camelContext>
> ```

CHAPTER 24. THE OBJECT-GRAPH NAVIGATION LANGUAGE(OGNL)

OVERVIEW

OGNL is an expression language for getting and setting properties of Java objects. You use the same expression for both getting and setting the value of a property. The OGNL support is in the **camel-ognl** module.

CAMEL ON EAP DEPLOYMENT

This component is supported by the Camel on EAP (Wildfly Camel) framework, which offers a simplified deployment model on the Red Hat JBoss Enterprise Application Platform (JBoss EAP) container. For details of this model, see chapter "Apache Camel on JBoss EAP" in "Deploying into a Web Server" .

ADDING THE OGNL MODULE

To use OGNL in your routes you need to add a dependency on **camel-ognl** to your project as shown in Example 24.1, "Adding the camel-ognl dependency".

Example 24.1. Adding the camel-ognl dependency

```
<!-- Maven POM File -->
...
<dependencies>
 ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ognl</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

STATIC IMPORT

To use the **ognl()** static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.language.ognl.OgnlExpression.ognl;
```

BUILT-IN VARIABLES

Table 24.1, "OGNL variables" lists the built-in variables that are accessible when using OGNL.

Table 24.1. OGNL variables

Name	Type	Description
this	**org.apache.camel.Exchange**	The current Exchange
exchange	**org.apache.camel.Exchange**	The current Exchange
exception	**Throwable**	the Exchange exception (if any)
exchangeID	**String**	the Exchange ID
fault	**org.apache.camel.Message**	The Fault message(if any)
request	**org.apache.camel.Message**	The IN message
response	**org.apache.camel.Message**	The OUT message
properties	**Map**	The Exchange properties
property(*name*)	**Object**	The value of the named Exchange property
property(*name*, *type*)	***Type***	The typed value of the named Exchange property

EXAMPLE

Example 24.2, "Route using OGNL" shows a route that uses OGNL.

> **Example 24.2. Route using OGNL**
>
> ```
> <camelContext>
>   <route>
>     <from uri="seda:foo"/>
>     <filter>
>       <language langauge="ognl">request.headers.foo == 'bar'</language>
>       <to uri="seda:bar"/>
>     </filter>
>   </route>
> </camelContext>
> ```

CHAPTER 25. PHP

OVERVIEW

PHP is a widely-used general-purpose scripting language that is especially suited for Web development. The PHP support is part of the **camel-script** module.

ADDING THE SCRIPT MODULE

To use PHP in your routes you need to add a dependency on **camel-script** to your project as shown in Example 25.1, "Adding the camel-script dependency".

Example 25.1. Adding the camel-script dependency

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

STATIC IMPORT

To use the **php()** static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

BUILT-IN ATTRIBUTES

Table 25.1, "PHP attributes" lists the built-in attributes that are accessible when using PHP.

Table 25.1. PHP attributes

Attribute	Type	Value
context	**org.apache.camel.CamelContext**	The Camel Context
exchange	**org.apache.camel.Exchange**	The current Exchange
request	**org.apache.camel.Message**	The IN message

Attribute	Type	Value
response	**org.apache.camel.Message**	The OUT message
properties	**org.apache.camel.builder.script.PropertiesFunction**	Function with a **resolve** method to make it easier to use the properties component inside scripts.

The attributes all set at **ENGINE_SCOPE**.

EXAMPLE

Example 25.2, "Route using PHP" shows a route that uses PHP.

Example 25.2. Route using PHP

```xml
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <language language="php">strpos(request.headers.get('user'), 'admin')!==
FALSE</language>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

USING THE PROPERTIES COMPONENT

To access a property value from the properties component, invoke the **resolve** method on the built-in **properties** attribute, as follows:

```
.setHeader("myHeader").php("properties.resolve(PropKey)")
```

Where *PropKey* is the key of the property you want to resolve, where the key value is of **String** type.

For more details about the properties component, see chapter "Properties" in "Apache Camel Component Reference".

CHAPTER 26. EXCHANGE PROPERTY

OVERVIEW

The exchange property language provides a convenient way of accessing *exchange properties*. When you supply a key that matches one of the exchange property names, the exchange property language returns the corresponding value.

The exchange property language is part of **camel-core**.

XML EXAMPLE

For example, to implement the recipient list pattern when the **listOfEndpoints** exchange property contains the recipient list, you could define a route as follows:

```xml
<camelContext>
  <route>
    <from uri="direct:a"/>
    <recipientList>
      <exchangeProperty>listOfEndpoints</exchangeProperty>
    </recipientList>
  </route>
</camelContext>
```

JAVA EXAMPLE

The same recipient list example can be implemented in Java as follows:

```java
from("direct:a").recipientList(exchangeProperty("listOfEndpoints"));
```

CHAPTER 27. PYTHON

OVERVIEW

Python is a remarkably powerful dynamic programming language that is used in a wide variety of application domains. Python is often compared to Tcl, Perl, Ruby, Scheme or Java. The Python support is part of the **camel-script** module.

ADDING THE SCRIPT MODULE

To use Python in your routes you need to add a dependency on **camel-script** to your project as shown in Example 27.1, "Adding the camel-script dependency".

Example 27.1. Adding the camel-script dependency

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

STATIC IMPORT

To use the **python()** static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

BUILT-IN ATTRIBUTES

Table 27.1, "Python attributes" lists the built-in attributes that are accessible when using Python.

Table 27.1. Python attributes

Attribute	Type	Value
context	**org.apache.camel.CamelContext**	The Camel Context
exchange	**org.apache.camel.Exchange**	The current Exchange
request	**org.apache.camel.Message**	The IN message

Attribute	Type	Value
response	**org.apache.camel.Message**	The OUT message
properties	**org.apache.camel.builder.script.PropertiesFunction**	Function with a **resolve** method to make it easier to use the properties component inside scripts.

The attributes all set at **ENGINE_SCOPE**.

EXAMPLE

Example 27.2, "Route using Python" shows a route that uses Python.

Example 27.2. Route using Python

```xml
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <langauge langauge="python">if request.headers.get('user') = 'admin'</langauge>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

USING THE PROPERTIES COMPONENT

To access a property value from the properties component, invoke the **resolve** method on the built-in **properties** attribute, as follows:

```
.setHeader("myHeader").python("properties.resolve(PropKey)")
```

Where *PropKey* is the key of the property you want to resolve, where the key value is of **String** type.

For more details about the properties component, see chapter "Properties" in "Apache Camel Component Reference".

CHAPTER 28. REF

OVERVIEW

The Ref expression language is really just a way to look up a custom Expression from the Registry. This is particular convenient to use in the XML DSL.

The Ref language is part of **camel-core**.

STATIC IMPORT

To use the Ref language in your Java application code, include the following import statement in your Java source files:

```
import static org.apache.camel.language.ref.RefLanguage.ref;
```

XML EXAMPLE

For example, the splitter pattern can reference a custom expression using the Ref language, as follows:

```
<beans ...>
  <bean id="myExpression" class="com.mycompany.MyCustomExpression"/>
  ...
  <camelContext>
   <route>
     <from uri="seda:a"/>
     <split>
      <ref>myExpression</ref>
      <to uri="mock:b"/>
     </split>
   </route>
  </camelContext>
</beans>
```

JAVA EXAMPLE

The preceding route can also be implemented in the Java DSL, as follows:

```
from("seda:a")
  .split().ref("myExpression")
  .to("seda:b");
```

CHAPTER 29. RUBY

OVERVIEW

Ruby is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write. The Ruby support is part of the **camel-script** module.

ADDING THE SCRIPT MODULE

To use Ruby in your routes you need to add a dependency on **camel-script** to your project as shown in Example 29.1, "Adding the camel-script dependency".

Example 29.1. Adding the camel-script dependency

```
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-script</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

STATIC IMPORT

To use the **ruby()** static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.script.ScriptBuilder.*;
```

BUILT-IN ATTRIBUTES

Table 29.1, "Ruby attributes" lists the built-in attributes that are accessible when using Ruby.

Table 29.1. Ruby attributes

Attribute	Type	Value
context	**org.apache.camel.CamelContext**	The Camel Context
exchange	**org.apache.camel.Exchange**	The current Exchange
request	**org.apache.camel.Message**	The IN message

Attribute	Type	Value
response	**org.apache.camel.Message**	The OUT message
properties	**org.apache.camel.builder.script.PropertiesFunction**	Function with a **resolve** method to make it easier to use the properties component inside scripts.

The attributes all set at **ENGINE_SCOPE**.

EXAMPLE

Example 29.2, "Route using Ruby" shows a route that uses Ruby.

Example 29.2. Route using Ruby

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <langauge langauge="ruby">$request.headers['user'] == 'admin'</langauge>
        <to uri="seda:adminQueue"/>
      </when>
      <otherwise>
        <to uri="seda:regularQueue"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

USING THE PROPERTIES COMPONENT

To access a property value from the properties component, invoke the **resolve** method on the built-in **properties** attribute, as follows:

```
.setHeader("myHeader").ruby("properties.resolve(PropKey)")
```

Where *PropKey* is the key of the property you want to resolve, where the key value is of **String** type.

For more details about the properties component, see chapter "Properties" in "Apache Camel Component Reference".

CHAPTER 30. THE SIMPLE LANGUAGE

Abstract

The simple language is a language that was developed in Apache Camel specifically for the purpose of accessing and manipulating the various parts of an exchange object. The language is not quite as simple as when it was originally created and it now features a comprehensive set of logical operators and conjunctions.

30.1. JAVA DSL

Simple expressions in Java DSL

In the Java DSL, there are two styles for using the **simple()** command in a route. You can either pass the **simple()** command as an argument to a processor, as follows:

```
from("seda:order")
  .filter(simple("${in.header.foo}"))
  .to("mock:fooOrders");
```

Or you can call the **simple()** command as a sub-clause on the processor, for example:

```
from("seda:order")
  .filter()
  .simple("${in.header.foo}")
  .to("mock:fooOrders");
```

Embedding in a string

If you are embedding a simple expression inside a plain text string, you must use the placeholder syntax, **${*Expression*}**. For example, to embed the **in.header.name** expression in a string:

```
simple("Hello ${in.header.name}, how are you?")
```

Customizing the start and end tokens

From Java, you can customize the start and end tokens (**{** and **}**, by default) by calling the **changeFunctionStartToken** static method and the **changeFunctionEndToken** static method on the **SimpleLanguage** object.

For example, you can change the start and end tokens to **[** and **]** in Java, as follows:

```
// Java
import org.apache.camel.language.simple.SimpleLanguage;
...
SimpleLanguage.changeFunctionStartToken("[");
SimpleLanguage.changeFunctionEndToken("]");
```
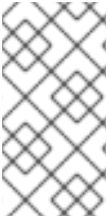
NOTE

Customizing the start and end tokens affects all Apache Camel applications that share the same **camel-core** library on their classpath. For example, in an OSGi server this might affect many applications; whereas in a Web application (WAR file) it would affect only the Web application itself.

30.2. XML DSL

Simple expressions in XML DSL

In the XML DSL, you can use a simple expression by putting the expression inside a **simple** element. For example, to define a route that performs filtering based on the contents of the **foo** header:

```
<route id="simpleExample">
  <from uri="seda:orders"/>
  <filter>
    <simple>${in.header.foo}</simple>
    <to uri="mock:fooOrders"/>
  </filter>
</route>
```

Alternative placeholder syntax

Sometimes—for example, if you have enabled Spring property placeholders or OSGi blueprint property placeholders—you might find that the **${*Expression*}** syntax clashes with another property placeholder syntax. In this case, you can disambiguate the placeholder using the alternative syntax, **$simple{*Expression*}**, for the simple expression. For example:

```
<simple>Hello $simple{in.header.name}, how are you?</simple>
```

Customizing the start and end tokens

From XML configuration, you can customize the start and end tokens (**{** and **}**, by default) by overriding the **SimpleLanguage** instance. For example, to change the start and end tokens to **[** and **]**, define a new **SimpleLanguage** bean in your XML configuration file, as follows:

```
<bean id="simple" class="org.apache.camel.language.simple.SimpleLanguage">
  <constructor-arg name="functionStartToken" value="["/>
  <constructor-arg name="functionEndToken" value="]"/>
</bean>
```

NOTE

Customizing the start and end tokens affects all Apache Camel applications that share the same **camel-core** library on their classpath. For example, in an OSGi server this might affect many applications; whereas in a Web application (WAR file) it would affect only the Web application itself.

Whitespace and auto-trim in XML DSL

By default, whitespace preceding and following a simple expression is automatically trimmed in XML DSL. So this expression with surrounding whitespace:

```
<transform>
  <simple>
    data=${body}
  </simple>
</transform>
```

is automatically trimmed, so that it is equivalent to this expression (no surrounding whitespace):

```
<transform>
  <simple>data=${body}</simple>
</transform>
```

If you want to include newlines before or after the expression, you can either explicitly add a newline character, as follows:

```
<transform>
  <simple>data=${body}\n</simple>
</transform>
```

or you can switch off auto-trimming, by setting the **trim** attribute to **false**, as follows:

```
<transform trim="false">
  <simple>data=${body}
  </simple>
</transform>
```

30.3. INVOKING AN EXTERNAL SCRIPT

Overview

It is possible to execute Simple scripts that are stored in an external resource, as described here.

Syntax for script resource

Use the following syntax to access a Simple script that is stored as an external resource:

```
resource:Scheme:Location
```

Where the **Scheme:** can be either **classpath:**, **file:**, or **http:**.

For example, to read the **mysimple.txt** script from the classpath,

```
simple("resource:classpath:mysimple.txt")
```

30.4. EXPRESSIONS

Overview

The simple language provides various elementary expressions that return different parts of a message exchange. For example, the expression, **simple("${header.timeOfDay}")**, would return the contents of a header called **timeOfDay** from the incoming message.
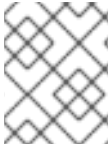
> ### NOTE
>
> Since Apache Camel 2.9, you must *always* use the placeholder syntax, **${*Expression*}**, to return a variable value. It is never permissible to omit the enclosing tokens (**${** and **}**).

Contents of a single variable

You can use the simple language to define string expressions, based on the variables provided. For example, you can use a variable of the form, **in.header.***HeaderName*, to obtain the value of the *HeaderName* header, as follows:

```
simple("${in.header.foo}")
```

Variables embedded in a string

You can embed simple variables in a string expression—for example:

```
simple("Received a message from ${in.header.user} on ${date:in.header.date:yyyyMMdd}.")
```

date and bean variables

As well as providing variables that access all of the different parts of an exchange (see Table 30.1, "Variables for the Simple Language"), the simple language also provides special variables for formatting dates, **date:***command***:***pattern*, and for calling bean methods, **bean:***beanRef*. For example, you can use the date and the bean variables as follows:

```
simple("Todays date is ${date:now:yyyyMMdd}")
simple("The order type is ${bean:orderService?method=getOrderType}")
```

Specifying the result type

You can specify the result type of an expression explicitly. This is mainly useful for converting the result type to a boolean or numerical type.

In the Java DSL, specify the result type as an extra argument to **simple()**. For example, to return an integer result, you could evaluate a simple expression as follows:

```
...
.setHeader("five", simple("5", Integer.class))
```

In the XML DSL, specify the result type using the **resultType** attribute. For example:

```
<setHeader headerName="five">
  <!-- use resultType to indicate that the type should be a java.lang.Integer -->
  <simple resultType="java.lang.Integer">5</simple>
</setHeader>
```

Dynamic Header Key

From Camel 2.17, the setHeaderand setExchange properties allows to use a dynamic header key using the Simple language, if the name of the key is a Simple language expression.

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <setHeader
headerName="$simple{type:org.apache.camel.spring.processor.SpringSetPropertyNameDynamicTest$
TestConstans.EXCHANGE_PROP_TX_FAILED}">
      <simple>${type:java.lang.Boolean.TRUE}</simple>
    </setHeader>
    <to uri="mock:end"/>
  </route>
</camelContext>
```

Nested expressions

Simple expressions can be nested—for example:

```
simple("${header.${bean:headerChooser?method=whichHeader}}")
```

Accessing constants or enums

You can access a bean's constant or enum fields using the following syntax:

```
type:ClassName.Field
```

For example, consider the following Java **enum** type:

```
package org.apache.camel.processor;
...
public enum Customer {
    GOLD, SILVER, BRONZE
}
```

You can access the **Customer** enum fields, as follows:

```
 from("direct:start")
   .choice()
     .when().simple("${header.customer} ==
      ${type:org.apache.camel.processor.Customer.GOLD}")
        .to("mock:gold")
     .when().simple("${header.customer} ==
      ${type:org.apache.camel.processor.Customer.SILVER}")
        .to("mock:silver")
     .otherwise()
        .to("mock:other");
```

OGNL expressions

The Object Graph Navigation Language (OGNL) is a notation for invoking bean methods in a chain-like fashion. If a message body contains a Java bean, you can easily access its bean properties using OGNL notation. For example, if the message body is a Java object with a **getAddress()** accessor, you can access the **Address** object and the **Address** object's properties as follows:

```
simple("${body.address}")
simple("${body.address.street}")
simple("${body.address.zip}")
simple("${body.address.city}")
```

Where the notation, **${body.address.street}**, is shorthand for **${body.getAddress.getStreet}**.

OGNL null-safe operator

You can use the null-safe operator, **?.**, to avoid encountering null-pointer exceptions, in case the body does *not* have an address. For example:

```
simple("${body?.address?.street}")
```

If the body is a **java.util.Map** type, you can look up a value in the map with the key, **foo**, using the following notation:

```
simple("${body[foo]?.name}")
```

OGNL list element access

You can also use square brackets notation, **[k]**, to access the elements of a list. For example:

```
simple("${body.address.lines[0]}")
simple("${body.address.lines[1]}")
simple("${body.address.lines[2]}")
```

The **last** keyword returns the index of the last element of a list. For example, you can access the *second last* element of a list, as follows:

```
simple("${body.address.lines[last-1]}")
```

You can use the **size** method to query the size of a list, as follows:

```
simple("${body.address.lines.size}")
```

OGNL array length access

You can access the length of a Java array through the **length** method, as follows:

```
String[] lines = new String[]{"foo", "bar", "cat"};
exchange.getIn().setBody(lines);

simple("There are ${body.length} lines")
```

30.5. PREDICATES

Overview

You can construct predicates by testing expressions for equality. For example, the predicate, **simple("${header.timeOfDay} == '14:30'")**, tests whether the **timeOfDay** header in the incoming message is equal to **14:30**.

In addition, whenever the **resultType** is specified as a Boolean the expression is evaluated as a predicate instead of an expression. This allows the predicate syntax to be used for these expressions.

Syntax

You can also test various parts of an exchange (headers, message body, and so on) using simple predicates. Simple predicates have the following general syntax:

> ${*LHSVariable*} *Op RHSValue*

Where the variable on the left hand side, *LHSVariable*, is one of the variables shown in Table 30.1, "Variables for the Simple Language" and the value on the right hand side, *RHSValue*, is one of the following:

- Another variable, **${*RHSVariable*}**.

- A string literal, enclosed in single quotes, **' '**.

- A numeric constant, enclosed in single quotes, **' '**.

- The null object, **null**.

The simple language always attempts to convert the RHS value to the type of the LHS value.

> **NOTE**
>
> While the simple language will attempt to convert the RHS, depending on the operator the LHS may need to be cast into the appropriate Type before the comparison is made.

Examples

For example, you can perform simple string comparisons and numerical comparisons as follows:

> simple("${in.header.user} == 'john'")
>
> simple("${in.header.number} > '100'") // String literal can be converted to integer

You can test whether the left hand side is a member of a comma-separated list, as follows:

> simple("${in.header.type} in 'gold,silver'")

You can test whether the left hand side matches a regular expression, as follows:

> simple("${in.header.number} regex '\d{4}'")

You can test the type of the left hand side using the **is** operator, as follows:

> simple("${in.header.type} is 'java.lang.String'")
> simple("${in.header.type} is 'String'") // You can abbreviate java.lang. types

You can test whether the left hand side lies in a specified numerical range (where the range is inclusive), as follows:

> simple("${in.header.number} range '100..199'")

Conjunctions

You can also combine predicates using the logical conjunctions, **&&** and ||.

For example, here is an expression using the **&&** conjunction (logical and):

> simple("${in.header.title} contains 'Camel' && ${in.header.type} == 'gold'")

And here is an expression using the || conjunction (logical inclusive or):

> simple("${in.header.title} contains 'Camel' || ${in.header.type} == 'gold'")

30.6. VARIABLE REFERENCE

Table of variables

Table 30.1, "Variables for the Simple Language" shows all of the variables supported by the simple language.

Table 30.1. Variables for the Simple Language

Variable	Type	Description
camelContext	**Object**	The Camel context. *Supports OGNL expressions.*
camelId	**String**	The Camel context's ID value.
exchangeId	**String**	The exchange's ID value.
id	**String**	The *In* message ID value.
body	**Object**	The *In* message body. *Supports OGNL expressions.*
in.body	**Object**	The *In* message body. *Supports OGNL expressions.*
out.body	**Object**	The *Out* message body.

Variable	Type	Description
bodyAs(*Type*)	*Type*	The *In* message body, converted to the specified type. All types, *Type*, must be specified using their fully-qualified Java name, except for the types: **byte[]**, **String**, **Integer**, and **Long**. The converted body can be null.
mandatoryBodyAs(*Type*)	*Type*	The *In* message body, converted to the specified type. All types, *Type*, must be specified using their fully-qualified Java name, except for the types: **byte[]**, **String**, **Integer**, and **Long**. The converted body is expected to be non-null.
header.*HeaderName*	**Object**	The *In* message's *HeaderName* header. *Supports OGNL expressions.*
header[*HeaderName***]**	**Object**	The *In* message's *HeaderName* header (alternative syntax).
headers.*HeaderName*	**Object**	The *In* message's *HeaderName* header.
headers[*HeaderName***]**	**Object**	The *In* message's *HeaderName* header (alternative syntax).
in.header.*HeaderName*	**Object**	The *In* message's *HeaderName* header. *Supports OGNL expressions.*
in.header[*HeaderName***]**	**Object**	The *In* message's *HeaderName* header (alternative syntax).
in.headers.*HeaderName*	**Object**	The *In* message's *HeaderName* header. *Supports OGNL expressions.*
in.headers[*HeaderName***]**	**Object**	The *In* message's *HeaderName* header (alternative syntax).
out.header.*HeaderName*	**Object**	The *Out* message's *HeaderName* header.
out.header[*HeaderName***]**	**Object**	The *Out* message's *HeaderName* header (alternative syntax).

Variable	Type	Description
out.headers.*HeaderName*	**Object**	The *Out* message's *HeaderName* header.
out.headers[*HeaderName***]**	**Object**	The *Out* message's *HeaderName* header (alternative syntax).
headerAs(*Key***,***Type***)**	*Type*	The *Key* header, converted to the specified type. All types, *Type*, must be specified using their fully-qualified Java name, except for the types: **byte[]**, **String**, **Integer**, and **Long**. The converted value can be null.
headers	**Map**	All of the *In* headers (as a **java.util.Map** type).
in.headers	**Map**	All of the *In* headers (as a **java.util.Map** type).
property.*PropertyName*	**Object**	The *PropertyName* property on the exchange.
property[*PropertyName***]**	**Object**	The *PropertyName* property on the exchange (alternative syntax).
sys.*SysPropertyName*	**String**	The *SysPropertyName* Java system property.
sysenv.*SysEnvVar*	**String**	The *SysEnvVar* system environment variable.
exception	**String**	Either the exception object from **Exchange.getException()** or, if this value is null, the caught exception from the **Exchange.EXCEPTION_CAUGHT** property; otherwise null. *Supports OGNL expressions.*
exception.message	**String**	If an exception is set on the exchange, returns the value of **Exception.getMessage()**; otherwise, returns **null**.

Variable	Type	Description
exception.stacktrace	**String**	If an exception is set on the exchange, returns the value of **Exception.getStackTrace()**; otherwise, returns **null**. **Note:** The simple language first tries to retrieve an exception from **Exchange.getException()**. If that property is not set, it checks for a caught exception, by calling **Exchange.getProperty(Exchange.CAUGHT_EXCEPTION)**.
date:*command***:***pattern*	**String**	A date formatted using a java.text.SimpleDateFormat pattern. The following commands are supported: **now**, for the current date and time; **header.***HeaderName*, or **in.header.***HeaderName* to use a java.util.Date object in the *HeaderName* header from the *In* message; **out.header.***HeaderName* to use a java.util.Date object in the *HeaderName* header from the *Out* message;
bean:*beanID.Method*	**Object**	Invokes a method on the referenced bean and *returns the result of the method invocation*. To specify a method name, you can either use the *beanID.Method* syntax; or you can use the *beanID***? method=***methodName* syntax.
ref:*beanID*	**Object**	Looks up the bean with the ID, *beanID*, in the registry and *returns a reference to the bean itself*. For example, if you are using the splitter EIP, you could use this variable to reference the bean that implements the splitting algorithm.
properties:*Key*	**String**	The value of the *Key* property placeholder (see Section 2.7, "Property Placeholders").

Variable	Type	Description
properties:*Location***:***Key*	**String**	The value of the *Key* property placeholder, where the location of the properties file is given by *Location* (see Section 2.7, "Property Placeholders").
threadName	**String**	The name of the current thread.
routeId	**String**	Returns the ID of the current route through which the **Exchange** is being routed.
type:*Name*[**.***Field*]	**Object**	References a type or field by its Fully-Qualified-Name (FQN). To refer to a field, append **.***Field*. For example, you can refer to the **FILE_NAME** constant field from the **Exchange** class as **type:org.apache.camel.Exchange.FILE_NAME**
collate(group)	**List**	From Camel 2.17, the collate function iterates the message body and groups the data into the sub lists of specific size. You can use with the Splitter EIP to split a message body and group or batch the submessages into a group of N sublists.

30.7. OPERATOR REFERENCE

Binary operators

The binary operators for simple language predicates are shown in Table 30.2, "Binary Operators for the Simple Language".

Table 30.2. Binary Operators for the Simple Language

Operator	Description
==	Equals.
=~	Equals ignore case. Ignore the case when comparing string values.
>	Greater than.

Operator	Description
>=	Greater than or equals.
<	Less than.
<=	Less than or equals.
!=	Not equal to.
contains	Test if LHS string contains RHS string.
not contains	Test if LHS string does *not* contain RHS string.
regex	Test if LHS string matches RHS regular expression.
not regex	Test if LHS string does *not* match RHS regular expression.
in	Test if LHS string appears in the RHS comma-separated list.
not in	Test if LHS string does *not* appear in the RHS comma-separated list.
is	Test if LHS is an instance of RHS Java type (using Java **instanceof** operator).
not is	Test if LHS is *not* an instance of RHS Java type (using Java **instanceof** operator).
range	Test if LHS number lies in the RHS range (where range has the format, **'***min***...***max***'**).
not range	Test if LHS number does *not* lie in the RHS range (where range has the format, **'***min***...***max***'**).

Unary operators and character escapes

The binary operators for simple language predicates are shown in Table 30.3, "Unary Operators for the Simple Language".

Table 30.3. Unary Operators for the Simple Language

Operator	Description
++	Increment a number by 1.

Operator	Description
--	Decrement a number by 1.
\n	The newline character.
\r	The carriage return character.
\t	The tab character.
\	*(Obsolete)* Since Camel version 2.11, the backslash escape character is not supported.

Combining predicates

The conjunctions shown in Table 30.4, "Conjunctions for Simple Language Predicates" can be used to combine two or more simple language predicates.

Table 30.4. Conjunctions for Simple Language Predicates

Operator	Description
&&	Combine two predicates with logical *and*.
||	Combine two predicates with logical *inclusive or*.
and	*Deprecated*. Use **&&** instead.
or	*Deprecated*. Use **||** instead.

CHAPTER 31. SPEL

OVERVIEW

The Spring Expression Language (SpEL) is an object graph navigation language provided with Spring 3, which can be used to construct predicates and expressions in a route. A notable feature of SpEL is the ease with which you can access beans from the registry.

SYNTAX

The SpEL expressions must use the placeholder syntax, **#{*SpelExpression*}**, so that they can be embedded in a plain text string (in other words, SpEL has expression templating enabled).

SpEL can also look up beans in the registry (typically, the Spring registry), using the **@*BeanID*** syntax. For example, given a bean with the ID, **headerUtils**, and the method, **count()** (which counts the number of headers on the current message), you could use the **headerUtils** bean in an SpEL predicate, as follows:

```
#{@headerUtils.count > 4}
```

ADDING SPEL PACKAGE

To use SpEL in your routes you need to add a dependency on **camel-spring** to your project as shown in Example 31.1, "Adding the camel-spring dependency".

Example 31.1. Adding the camel-spring dependency

```xml
<!-- Maven POM File -->
<properties>
  <camel-version>2.17.0.redhat-630xxx</camel-version>
  ...
</properties>

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

VARIABLES

Table 31.1, "SpEL variables" lists the variables that are accessible when using SpEL.

Table 31.1. SpEL variables

Variable	Type	Description
this	**Exchange**	The current exchange is the root object.
exchange	**Exchange**	The current exchange.
exchangeId	**String**	The current exchange's ID.
exception	**Throwable**	The exchange exception (if any).
fault	**Message**	The fault message (if any).
request	**Message**	The exchange's *In* message.
response	**Message**	The exchange's *Out* message (if any).
properties	**Map**	The exchange properties.
property(*Name*)	**Object**	The exchange property keyed by *Name*.
property(*Name*, *Type*)	**Type**	The exchange property keyed by *Name*, converted to the type, *Type*.

XML EXAMPLE

For example, to select only those messages whose **Country** header has the value **USA**, you can use the following SpEL expression:

```
<route>
  <from uri="SourceURL"/>
  <filter>
    <spel>#{request.headers['Country'] == 'USA'}}</spel>
    <to uri="TargetURL"/>
  </filter>
</route>
```

JAVA EXAMPLE

You can define the same route in the Java DSL, as follows:

```
from("SourceURL")
  .filter().spel("#{request.headers['Country'] == 'USA'}")
  .to("TargetURL");
```

The following example shows how to embed SpEL expressions within a plain text string:

```
from("SourceURL")
  .setBody(spel("Hello #{request.body}! What a beautiful #{request.headers['dayOrNight']}"))
  .to("TargetURL");
```

CHAPTER 32. THE XPATH LANGUAGE

Abstract

When processing XML messages, the XPath language enables you to select part of a message, by specifying an XPath expression that acts on the message's Document Object Model (DOM). You can also define XPath predicates to test the contents of an element or an attribute.

32.1. JAVA DSL

Basic expressions

You can use **xpath("*Expression*")** to evaluate an XPath expression on the current exchange (where the XPath expression is applied to the body of the current *In* message). The result of the **xpath()** expression is an XML node (or node set, if more than one node matches).

For example, to extract the contents of the /**person/name** element from the current *In* message body and use it to set a header named **user**, you could define a route like the following:

```
from("queue:foo")
    .setHeader("user", xpath("/person/name/text()"))
    .to("direct:tie");
```

Instead of specifying **xpath()** as an argument to **setHeader()**, you can use the fluent builder **xpath()** command—for example:

```
from("queue:foo")
    .setHeader("user").xpath("/person/name/text()")
    .to("direct:tie");
```

If you want to convert the result to a specific type, specify the result type as the second argument of **xpath()**. For example, to specify explicitly that the result type is **String**:

```
xpath("/person/name/text()", String.class)
```

Namespaces

Typically, XML elements belong to a schema, which is identified by a namespace URI. When processing documents like this, it is necessary to associate namespace URIs with prefixes, so that you can identify element names unambiguously in your XPath expressions. Apache Camel provides the helper class, **org.apache.camel.builder.xml.Namespaces**, which enables you to define associations between namespaces and prefixes.

For example, to associate the prefix, **cust**, with the namespace, **http://acme.com/customer/record**, and then extract the contents of the element, /**cust:person/cust:name**, you could define a route like the following:

```
import org.apache.camel.builder.xml.Namespaces;
...
Namespaces ns = new Namespaces("cust", "http://acme.com/customer/record");
```

```
from("queue:foo")
    .setHeader("user", xpath("/cust:person/cust:name/text()", ns))
    .to("direct:tie");
```

Where you make the namespace definitions available to the **xpath()** expression builder by passing the **Namespaces** object, **ns**, as an additional argument. If you need to define multiple namespaces, use the **Namespace.add()** method, as follows:

```
import org.apache.camel.builder.xml.Namespaces;
...
Namespaces ns = new Namespaces("cust", "http://acme.com/customer/record");
ns.add("inv", "http://acme.com/invoice");
ns.add("xsi", "http://www.w3.org/2001/XMLSchema-instance");
```

If you need to specify the result type *and* define namespaces, you can use the three-argument form of **xpath()**, as follows:

```
xpath("/person/name/text()", String.class, ns)
```

Auditing namespaces

One of the most frequent problems that can occur when using XPath expressions is that there is a mismatch between the namespaces appearing in the incoming messages and the namespaces used in the XPath expression. To help you troubleshoot this kind of problem, the XPath language supports an option to dump all of the namespaces from all of the incoming messages into the system log.

To enable namespace logging at the **INFO** log level, enable the **logNamespaces** option in the Java DSL, as follows:

```
xpath("/foo:person/@id", String.class).logNamespaces()
```

Alternatively, you could configure your logging system to enable **TRACE** level logging on the **org.apache.camel.builder.xml.XPathBuilder** logger.

When namespace logging is enabled, you will see log messages like the following for each processed message:

```
2012-01-16 13:23:45,878 [stSaxonWithFlag] INFO  XPathBuilder  -
Namespaces discovered in message: {xmlns:a=[http://apache.org/camel],
DEFAULT=[http://apache.org/default],
xmlns:b=[http://apache.org/camelA, http://apache.org/camelB]}
```

32.2. XML DSL

Basic expressions

To evaluate an XPath expression in the XML DSL, put the XPath expression inside an **xpath** element. The XPath expression is applied to the body of the current *In* message and returns an XML node (or node set). Typically, the returned XML node is automatically converted to a string.

For example, to extract the contents of the **/person/name** element from the current *In* message body and use it to set a header named **user**, you could define a route like the following:

```
<beans ...>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
   <route>
    <from uri="queue:foo"/>
    <setHeader headerName="user">
      <xpath>/person/name/text()</xpath>
    </setHeader>
    <to uri="direct:tie"/>
   </route>
  </camelContext>

</beans>
```

If you want to convert the result to a specific type, specify the result type by setting the **resultType** attribute to a Java type name (where you must specify the fully-qualified type name). For example, to specify explicitly that the result type is **java.lang.String** (you can omit the **java.lang.** prefix here):

```
<xpath resultType="String">/person/name/text()</xpath>
```

Namespaces

When processing documents whose elements belong to one or more XML schemas, it is typically necessary to associate namespace URIs with prefixes, so that you can identify element names unambiguously in your XPath expressions. It is possible to use the standard XML mechanism for associating prefixes with namespace URIs. That is, you can set an attribute like this: **xmlns:*Prefix*="*NamespaceURI*"**.

For example, to associate the prefix, **cust**, with the namespace, **http://acme.com/customer/record**, and then extract the contents of the element, **/cust:person/cust:name**, you could define a route like the following:

```
<beans ...>

  <camelContext xmlns="http://camel.apache.org/schema/spring"
        xmlns:cust="http://acme.com/customer/record" >
   <route>
    <from uri="queue:foo"/>
    <setHeader headerName="user">
      <xpath>/cust:person/cust:name/text()</xpath>
    </setHeader>
    <to uri="direct:tie"/>
   </route>
  </camelContext>

</beans>
```

Auditing namespaces

One of the most frequent problems that can occur when using XPath expressions is that there is a mismatch between the namespaces appearing in the incoming messages and the namespaces used in the XPath expression. To help you troubleshoot this kind of problem, the XPath language supports an option to dump all of the namespaces from all of the incoming messages into the system log.

To enable namespace logging at the **INFO** log level, enable the **logNamespaces** option in the XML DSL, as follows:

```
<xpath logNamespaces="true" resultType="String">/foo:person/@id</xpath>
```

Alternatively, you could configure your logging system to enable **TRACE** level logging on the **org.apache.camel.builder.xml.XPathBuilder** logger.

When namespace logging is enabled, you will see log messages like the following for each processed message:

```
2012-01-16 13:23:45,878 [stSaxonWithFlag] INFO  XPathBuilder  -
Namespaces discovered in message: {xmlns:a=[http://apache.org/camel],
DEFAULT=[http://apache.org/default],
xmlns:b=[http://apache.org/camelA, http://apache.org/camelB]}
```

32.3. XPATH INJECTION

Parameter binding annotation

When using Apache Camel bean integration to invoke a method on a Java bean, you can use the **@XPath** annotation to extract a value from the exchange and bind it to a method parameter.

For example, consider the following route fragment, which invokes the **credit** method on an **AccountService** object:

```
from("queue:payments")
    .beanRef("accountService","credit")
    ...
```

The **credit** method uses parameter binding annotations to extract relevant data from the message body and inject it into its parameters, as follows:

```
public class AccountService {
    ...
    public void credit(
            @XPath("/transaction/transfer/receiver/text()") String name,
            @XPath("/transaction/transfer/amount/text()") String amount
            )
    {
        ...
    }
    ...
}
```

For more information about bean integration, see Section 2.4, "Bean Integration".

Namespaces

Table 32.1, "Predefined Namespaces for @XPath" shows the namespaces that are predefined for XPath. You can use these namespace prefixes in the **XPath** expression that appears in the **@XPath** annotation.

Table 32.1. Predefined Namespaces for @XPath

Namespace URI	Prefix
http://www.w3.org/2001/XMLSchema	**xsd**
http://www.w3.org/2003/05/soap-envelope	**soap**

Custom namespaces

You can use the **@NamespacePrefix** annotation to define custom XML namespaces. Invoke the **@NamespacePrefix** annotation to initialize the **namespaces** argument of the **@XPath** annotation. The namespaces defined by **@NamespacePrefix** can then be used in the **@XPath** annotation's expression value.

For example, to associate the prefix, **ex**, with the custom namespace, **http://fusesource.com/examples**, invoke the **@XPath** annotation as follows:

```
public class AccountService {
  ...
  public void credit(
    @XPath(
      value = "/ex:transaction/ex:transfer/ex:receiver/text()",
      namespaces = @NamespacePrefix(
        prefix = "ex",
        uri = "http://fusesource.com/examples"
      )
    ) String name,
    @XPath(
      value = "/ex:transaction/ex:transfer/ex:amount/text()",
      namespaces = @NamespacePrefix(
        prefix = "ex",
        uri = "http://fusesource.com/examples"
      )
    ) String amount,
  )
  {
    ...
  }
  ...
}
```

32.4. XPATH BUILDER

Overview

The **org.apache.camel.builder.xml.XPathBuilder** class enables you to evaluate XPath expressions independently of an exchange. That is, if you have an XML fragment from any source, you can use **XPathBuilder** to evaluate an XPath expression on the XML fragment.

Matching expressions

Use the **matches()** method to check whether one or more XML nodes can be found that match the given XPath expression. The basic syntax for matching an XPath expression using **XPathBuilder** is as follows:

```
boolean matches = XPathBuilder
        .xpath("Expression")
        .matches(CamelContext, "XMLString");
```

Where the given expression, *Expression*, is evaluated against the XML fragment, *XMLString*, and the result is true, if at least one node is found that matches the expression. For example, the following example returns **true**, because the XPath expression finds a match in the **xyz** attribute.

```
boolean matches = XPathBuilder
        .xpath("/foo/bar/@xyz")
        .matches(getContext(), "<foo><bar xyz='cheese'/></foo>"));
```

Evaluating expressions

Use the **evaluate()** method to return the contents of the first node that matches the given XPath expression. The basic syntax for evaluating an XPath expression using **XPathBuilder** is as follows:

```
String nodeValue = XPathBuilder
        .xpath("Expression")
        .evaluate(CamelContext, "XMLString");
```

You can also specify the result type by passing the required type as the second argument to **evaluate()**—for example:

```
String name = XPathBuilder
        .xpath("foo/bar")
        .evaluate(context, "<foo><bar>cheese</bar></foo>", String.class);
Integer number = XPathBuilder
        .xpath("foo/bar")
        .evaluate(context, "<foo><bar>123</bar></foo>", Integer.class);
Boolean bool = XPathBuilder
        .xpath("foo/bar")
        .evaluate(context, "<foo><bar>true</bar></foo>", Boolean.class);
```

32.5. ENABLING SAXON

Prerequisites

A prerequisite for using the Saxon parser is that you add a dependency on the **camel-saxon** artifact (either adding this dependency to your Maven POM, if you use Maven, or adding the **camel-saxon-6.3.0.redhat-xxx.jar** file to your classpath, otherwise).

Using the Saxon parser in Java DSL

In Java DSL, the simplest way to enable the Saxon parser is to call the **saxon()** fluent builder method. For example, you could invoke the Saxon parser as shown in the following example:

```
// Java
```

```
// create a builder to evaluate the xpath using saxon
XPathBuilder builder = XPathBuilder.xpath("tokenize(/foo/bar, '_')[2]").saxon();

// evaluate as a String result
String result = builder.evaluate(context, "<foo><bar>abc_def_ghi</bar></foo>");
```

Using the Saxon parser in XML DSL

In XML DSL, the simplest way to enable the Saxon parser is to set the **saxon** attribute to true in the **xpath** element. For example, you could invoke the Saxon parser as shown in the following example:

```
<xpath saxon="true" resultType="java.lang.String">current-dateTime()</xpath>
```

Programming with Saxon

If you want to use the Saxon XML parser in your application code, you can create an instance of the Saxon transformer factory explicitly using the following code:

```
// Java
import javax.xml.transform.TransformerFactory;
import net.sf.saxon.TransformerFactoryImpl;
...
TransformerFactory saxonFactory = new net.sf.saxon.TransformerFactoryImpl();
```

On the other hand, if you prefer to use the generic JAXP API to create a transformer factory instance, you *must* first set the **javax.xml.transform.TransformerFactory** property in the **ESBInstall/etc/system.properties** file, as follows:

```
javax.xml.transform.TransformerFactory=net.sf.saxon.TransformerFactoryImpl
```

You can then instantiate the Saxon factory using the generic JAXP API, as follows:

```
// Java
import javax.xml.transform.TransformerFactory;
...
TransformerFactory factory = TransformerFactory.newInstance();
```

If your application depends on any third-party libraries that use Saxon, it might be necessary to use the second, generic approach.

> **NOTE**
>
> The Saxon library must be installed in the container as the OSGi bundle, **net.sf.saxon/saxon9he** (normally installed by default). In versions of Fuse ESB prior to 7.1, it is not possible to load Saxon using the generic JAXP API.

32.6. EXPRESSIONS

Result type

By default, an XPath expression returns a list of one or more XML nodes, of **org.w3c.dom.NodeList** type. You can use the type converter mechanism to convert the result to a different type, however. In

the Java DSL, you can specify the result type in the second argument of the **xpath()** command. For example, to return the result of an XPath expression as a **String**:

```
xpath("/person/name/text()", String.class)
```

In the XML DSL, you can specify the result type in the **resultType** attribute, as follows:

```
<xpath resultType="java.lang.String">/person/name/text()</xpath>
```

Patterns in location paths

You can use the following patterns in XPath location paths:

/people/person

The basic location path specifies the nested location of a particular element. That is, the preceding location path would match the person element in the following XML fragment:

```
<people>
  <person>...</person>
</people>
```

Note that this basic pattern can match *multiple* nodes—for example, if there is more than one **person** element inside the **people** element.

/name/text()

If you just want to access the *text* inside by the element, append **/text()** to the location path, otherwise the node includes the element's start and end tags (and these tags would be included when you convert the node to a string).

/person/telephone/@isDayTime

To select the value of an attribute, *AttributeName*, use the syntax *@AttributeName*. For example, the preceding location path returns **true** when applied to the following XML fragment:

```
<person>
  <telephone isDayTime="true">1234567890</telephone>
</person>
```

*

A wildcard that matches all elements in the specified scope. For example, **/people/person/*** matches all the child elements of **person**.

@*

A wildcard that matches all attributes of the matched elements. For example, **/person/name/@*** matches all attributes of every matched **name** element.

//

Match the location path at every nesting level. For example, the **//name** pattern matches every **name** element highlighted in the following XML fragment:

```
<invoice>
```

```
  <person>
    <name .../>
  </person>
</invoice>
<person>
  <name .../>
</person>
<name .../>
```

..

Selects the parent of the current context node. Not normally useful in the Apache Camel XPath language, because the current context node is the document root, which has no parent.

node()

Match any kind of node.

text()

Match a text node.

comment()

Match a comment node.

processing-instruction()

Match a processing-instruction node.

Predicate filters

You can filter the set of nodes matching a location path by appending a predicate in square brackets, **[*Predicate*]**. For example, you can select the Nth node from the list of matches by appending **[*N*]** to a location path. The following expression selects the first matching **person** element:

```
/people/person[1]
```

The following expression selects the second-last **person** element:

```
/people/person[last()-1]
```

You can test the value of attributes in order to select elements with particular attribute values. The following expression selects the **name** elements, whose **surname** attribute is either Strachan or Davies:

```
/person/name[@surname="Strachan" or @surname="Davies"]
```

You can combine predicate expressions using any of the conjunctions **and**, **or**, **not()**, and you can compare expressions using the comparators, **=, !=, >, >=, <, <=** (in practice, the less-than symbol must be replaced by the **<** entity). You can also use XPath functions in the predicate filter.

Axes

When you consider the structure of an XML document, the root element contains a sequence of children, and some of those child elements contain further children, and so on. Looked at in this way,

where nested elements are linked together by the *child-of* relationship, the whole XML document has the structure of a *tree*. Now, if you choose a particular node in this element tree (call it the *context node*), you might want to refer to different parts of the tree relative to the chosen node. For example, you might want to refer to the children of the context node, to the parent of the context node, or to all of the nodes that share the same parent as the context node (*sibling nodes*).

An *XPath axis* is used to specify the scope of a node match, restricting the search to a particular part of the node tree, relative to the current context node. The axis is attached as a prefix to the node name that you want to match, using the syntax, ***AxisType::MatchingNode***. For example, you can use the **child::** axis to search the children of the current context node, as follows:

> /invoice/items/child::item

The context node of **child::item** is the **items** element that is selected by the path, **/invoice/items**. The **child::** axis restricts the search to the children of the context node, **items**, so that **child::item** matches the children of **items** that are named **item**. As a matter of fact, the **child::** axis is the default axis, so the preceding example can be written equivalently as:

> /invoice/items/item

But there several other axes (13 in all), some of which you have already seen in abbreviated form: **@** is an abbreviation of **attribute::**, and **//** is an abbreviation of **descendant-or-self::**. The full list of axes is as follows (for details consult the reference below):

- **ancestor**
- **ancestor-or-self**
- **attribute**
- **child**
- **descendant**
- **descendant-or-self**
- **following**
- **following-sibling**
- **namespace**
- **parent**
- **preceding**
- **preceding-sibling**
- **self**

Functions

XPath provides a small set of standard functions, which can be useful when evaluating predicates. For example, to select the last matching node from a node set, you can use the last() function, which returns the index of the last node in a node set, as follows:

> /people/person[last()]

Where the preceding example selects the last **person** element in a sequence (in document order).

For full details of all the functions that XPath provides, consult the reference below.

Reference

For full details of the XPath grammar, see the XML Path Language, Version 1.0 specification.

32.7. PREDICATES

Basic predicates

You can use **xpath** in the Java DSL or the XML DSL in a context where a predicate is expected—for example, as the argument to a **filter()** processor or as the argument to a **when()** clause.

For example, the following route filters incoming messages, allowing a message to pass, only if the /**person**/**city** element contains the value, **London**:

```
from("direct:tie")
    .filter().xpath("/person/city = 'London'").to("file:target/messages/uk");
```

The following route evaluates the XPath predicate in a **when()** clause:

```
from("direct:tie")
    .choice()
        .when(xpath("/person/city = 'London'")).to("file:target/messages/uk")
        .otherwise().to("file:target/messages/others");
```

XPath predicate operators

The XPath language supports the standard XPath predicate operators, as shown in Table 32.2, "Operators for the XPath Language".

Table 32.2. Operators for the XPath Language

Operator	Description
=	Equals.
!=	Not equal to.
>	Greater than.
>=	Greater than or equals.
<	Less than.
<=	Less than or equals.

Operator	Description
or	Combine two predicates with logical *and*.
and	Combine two predicates with logical *inclusive or*.
not()	Negate predicate argument.

32.8. USING VARIABLES AND FUNCTIONS

Evaluating variables in a route

When evaluating XPath expressions inside a route, you can use XPath variables to access the contents of the current exchange, as well as O/S environment variables and Java system properties. The syntax to access a variable value is **$*VarName*** or **$*Prefix*:*VarName***, if the variable is accessed through an XML namespace.

For example, you can access the *In* message's body as **$in:body** and the *In* message's header value as **$in:*HeaderName***. O/S environment variables can be accessed as **$env:*EnvVar*** and Java system properties can be accessed as **$system:*SysVar***.

In the following example, the first route extracts the value of the **/person/city** element and inserts it into the **city** header. The second route filters exchanges using the XPath expression, **$in:city = 'London'**, where the **$in:city** variable is replaced by the value of the **city** header.

```
from("file:src/data?noop=true")
    .setHeader("city").xpath("/person/city/text()")
    .to("direct:tie");

from("direct:tie")
    .filter().xpath("$in:city = 'London'").to("file:target/messages/uk");
```

Evaluating functions in a route

In addition to the standard XPath functions, the XPath language defines additional functions. These additional functions (which are listed in Table 32.4, "XPath Custom Functions") can be used to access the underlying exchange, to evaluate a simple expression or to look up a property in the Apache Camel property placeholder component.

For example, the following example uses the **in:header()** function and the **in:body()** function to access a head and the body from the underlying exchange:

```
from("direct:start").choice()
    .when().xpath("in:header('foo') = 'bar'").to("mock:x")
    .when().xpath("in:body() = '<two/>'").to("mock:y")
    .otherwise().to("mock:z");
```

Notice the similarity between theses functions and the corresponding **in:*HeaderName*** or **in:body** variables. The functions have a slightly different syntax however: **in:header(*'HeaderName'*)** instead of **in:*HeaderName***; and **in:body()** instead of **in:body**.

Evaluating variables in XPathBuilder

You can also use variables in expressions that are evaluated using the **XPathBuilder** class. In this case, you cannot use variables such as **$in:body** or **$in:***HeaderName*, because there is no exchange object to evaluate against. But you *can* use variables that are defined inline using the **variable(***Name***, ***Value***)** fluent builder method.

For example, the following XPathBuilder construction evaluates the **$test** variable, which is defined to have the value, **London**:

```
String var = XPathBuilder.xpath("$test")
        .variable("test", "London")
        .evaluate(getContext(), "<name>foo</name>");
```

Note that variables defined in this way are automatically entered into the global namespace (for example, the variable, **$test**, uses no prefix).

32.9. VARIABLE NAMESPACES

Table of namespaces

Table 32.3, "XPath Variable Namespaces" shows the namespace URIs that are associated with the various namespace prefixes.

Table 32.3. XPath Variable Namespaces

Namespace URI	Prefix	Description
http://camel.apache.org/schema/spring	*None*	Default namespace (associated with variables that have no namespace prefix).
http://camel.apache.org/xml/in/	**in**	Used to reference header or body of the current exchange's *In* message.
http://camel.apache.org/xml/out/	**out**	Used to reference header or body of the current exchange's *Out* message.
http://camel.apache.org/xml/functions/	**functions**	Used to reference some custom functions.
http://camel.apache.org/xml/variables/environment-variables	**env**	Used to reference O/S environment variables.
http://camel.apache.org/xml/variables/system-properties	**system**	Used to reference Java system properties.

Namespace URI	Prefix	Description
http://camel.apache.org/xml/ variables/exchange-property	*Undefined*	Used to reference exchange properties. You must define your own prefix for this namespace.

32.10. FUNCTION REFERENCE

Table of custom functions

Table 32.4, "XPath Custom Functions" shows the custom functions that you can use in Apache Camel XPath expressions. These functions can be used in addition to the standard XPath functions.

Table 32.4. XPath Custom Functions

Function	Description
in:body()	Returns the *In* message body.
in:header(*HeaderName*)	Returns the *In* message header with name, *HeaderName*.
out:body()	Returns the *Out* message body.
out:header(*HeaderName*)	Returns the *Out* message header with name, *HeaderName*.
function:properties(*PropKey*)	Looks up a property with the key,*PropKey* (see Section 2.7, "Property Placeholders").
function:simple(*SimpleExp*)	Evaluates the specified simple expression,*SimpleExp*.

CHAPTER 33. XQUERY

OVERVIEW

XQuery was originally devised as a query language for data stored in XML form in a database. The XQuery language enables you to select parts of the current message, when the message is in XML format. XQuery is a superset of the XPath language; hence, any valid XPath expression is also a valid XQuery expression.

JAVA SYNTAX

You can pass an XQuery expression to **xquery()** in several ways. For simple expressions, you can pass the XQuery expressions as a string (**java.lang.String**). For longer XQuery expressions, you might prefer to store the expression in a file, which you can then reference by passing a **java.io.File** argument or a **java.net.URL** argument to the overloaded **xquery()** method. The XQuery expression implicitly acts on the message content and returns a node set as the result. Depending on the context, the return value is interpreted either as a predicate (where an empty node set is interpreted as false) or as an expression.

ADDING THE SAXON MODULE

To use XQuery in your routes you need to add a dependency on **camel-saxon** to your project as shown in Example 33.1, "Adding the camel-saxon dependency".

Example 33.1. Adding the camel-saxon dependency

```xml
<!-- Maven POM File -->
...
<dependencies>
  ...
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-saxon</artifactId>
    <version>${camel-version}</version>
  </dependency>
  ...
</dependencies>
```

CAMEL ON EAP DEPLOYMENT

The **camel-saxon** component is supported by the Camel on EAP (Wildfly Camel) framework, which offers a simplified deployment model on the Red Hat JBoss Enterprise Application Platform (JBoss EAP) container. For details of this model, see chapter "Apache Camel on JBoss EAP" in "Deploying into a Web Server".

STATIC IMPORT

To use the **xquery()** static method in your application code, include the following import statement in your Java source files:

```java
import static org.apache.camel.component.xquery.XQueryBuilder.xquery;
```

VARIABLES

Table 33.1, "XQuery variables" lists the variables that are accessible when using XQuery.

Table 33.1. XQuery variables

Variable	Type	Description
exchange	**Exchange**	The current Exchange
in.body	**Object**	The body of the IN message
out.body	**Object**	The body of the OUT message
in.headers.*key*	**Object**	The IN message header whose key is *key*
out.headers.*key*	**Object**	The OUT message header whose key is *key*
key	**Object**	The Exchange property whose key is *key*

EXAMPLE

Example 33.2, "Route using XQuery" shows a route that uses XQuery.

Example 33.2. Route using XQuery

```
<camelContext>
  <route>
    <from uri="activemq:MyQueue"/>
    <filter>
      <language langauge="xquery">/foo:person[@name='James']</language>
      <to uri="mqseries:SomeOtherQueue"/>
    </filter>
  </route>
</camelContext>
```

PART III. WEB SERVICES AND ROUTING WITH CAMEL CXF

Abstract

This guide describes how to use Apache Camel's CXF component to create Web services or wrap existing functionality in Web service facades.

CHAPTER 34. DEMONSTRATION CODE FOR CAMEL/CXF

Abstract

This chapter explains how to install, build, and run the demonstrations that accompany this guide.

34.1. DOWNLOADING AND INSTALLING THE DEMONSTRATIONS

Overview

Most of the examples discussed in this guide are based on working demonstrations, which you can download and try out for yourself. The examples can easily be run by deploying them into a Red Hat JBoss Fuse container, as described here.

Prerequisites

For building and running the demonstration code, you must have the following prerequisites installed:

- *Java platform*—the demonstrations can run on Java 7 or Java 8.

- *Apache Maven build tool*—to build the demonstration, you require a recent version of Apache Maven.

- *Internet connection*—Maven requires an Internet connection in order to download required dependencies from remote repositories while performing a build.

- *Red Hat JBoss Fuse*—the demonstrations are deployed into the Apache Karaf container.

> **NOTE**
>
> For more details of the requirements for installing and working with JBoss Fuse, see "Installation on Apache Karaf".

Downloading the demonstration package

The source code for the demonstrations is packaged as a Zip file, **cxf-webinars-jboss-fuse-6.3.zip**, and is available from the following location:

- https://github.com/FuseByExample/cxf-webinars/archive/jboss-fuse-6.3.zip

34.2. RUNNING THE DEMONSTRATIONS

Building the demonstrations

Use Apache Maven to build the demonstrations. Open a new command prompt, change directory to **cxf-webinars-jboss-fuse-6.3**, and enter the following commands:

```
mvn install
```

This command builds all of the demonstrations under the **cxf-webinars-jboss-fuse-6.3** directory (where the demonstrations are defined to be submodules of the **cxf-webinars-jboss-fuse-6.3**/**pom.xml**

project). While Maven is building the demonstration code, it downloads whatever dependencies it needs from the Internet and installs them in the local Maven repository.

Starting and configuring the Red Hat JBoss Fuse container

Start and configure the Red Hat JBoss Fuse container as follows:

1. *(Optional)* If your local Maven repository is in a non-standard location, you might need to edit the JBoss Fuse configuration to specify your custom location. Edit the **InstallDir/etc/org.ops4j.pax.url.mvn.cfg** file and set the **org.ops4j.pax.url.mvn.localRepository** property to the location of your local Maven repository:

   ```
   #
   # Path to the local maven repository which is used to avoid downloading
   # artifacts when they already exist locally.
   # The value of this property will be extracted from the settings.xml file
   # above, or defaulted to:
   #     System.getProperty( "user.home" ) + "/.m2/repository"
   #
   #org.ops4j.pax.url.mvn.localRepository=
   org.ops4j.pax.url.mvn.localRepository=file:E:/Data/.m2/repository
   ```

2. Launch the JBoss Fuse container. Open a new command prompt, change directory to **InstallDir/bin**, and enter the following command:

   ```
   ./fuse
   ```

Running the customer-ws-osgi-bundle demonstration

It is now a relatively straightforward task to run each of the demonstrations by installing the relevant OSGi bundles.

For example, to start up the WSDL-first Web service (discussed in Chapter 36, *WSDL-First Service Implementation*), enter the following console commands:

```
JBossFuse:karaf@root> install -s mvn:com.fusesource.byexample.cxf-webinars/customer-ws-osgi-bundle/1.0-SNAPSHOT
```

To see the Web service in action, start up the sample Web service client (discussed in Chapter 37, *Implementing a WS Client*), by entering the following console command:

```
JBossFuse:karaf@root> install -s mvn:com.fusesource.byexample.cxf-webinars/customer-ws-client/1.0-SNAPSHOT
```

The bundle creates a thread that invokes the Web service once a second and logs the response. View the log by entering the following console command:

```
JBossFuse:karaf@root> log:tail -n 4
```

You should see log output like the following:

```
2015-08-20 16:10:16,271 | INFO  | #0 - timer://foo | timerToLog                | ?
```

```
? | 198 - org.apache.camel.camel-core - 2.15.1.redhat-620133 | The message contains Hi from
Camel at 2015-08-20 16:10:16
2015-08-20 16:10:16,367 | INFO  | qtp432302853-183 | CustomerServiceImpl          | ?
? | 283 - com.fusesource.byexample.cxf-webinars.customer-ws-osgi-bundle - 1.0.0.SNAPSHOT |
Getting status for customer 1234
2015-08-20 16:10:16,370 | INFO  |   invoker thread. | ClientInvoker              | ?
? | 284 - com.fusesource.byexample.cxf-webinars.customer-ws-client - 1.0.0.SNAPSHOT | Got back:
status = Active, statusMessage = In the park, playing with my frisbee.
2015-08-20 16:10:18,373 | INFO  | qtp432302853-182 | CustomerServiceImpl          | ?
? | 283 - com.fusesource.byexample.cxf-webinars.customer-ws-osgi-bundle - 1.0.0.SNAPSHOT |
Getting status for customer 1234
2015-08-20 16:10:18,376 | INFO  |   invoker thread. | ClientInvoker              | ?
? | 284 - com.fusesource.byexample.cxf-webinars.customer-ws-client - 1.0.0.SNAPSHOT | Got back:
status = Active, statusMessage = In the park, playing with my frisbee.
```

To stop viewing the log, type the interrupt character (usually Ctrl–C).

To stop the client, first discover the client's bundle ID using the **osgi:list** console command. For example:

```
JBossFuse:karaf@root> list | grep customer-ws-client
[ 284] [Active     ] [           ] [Started] [   80] customer-ws-client (1.0.0.SNAPSHOT)
```

You can then stop the client using the **osgi:stop** console command. For example:

```
JBossFuse:karaf@root> stop 284
```

To shut down the container completely, enter the following console command:

```
JBossFuse:karaf@root> shutdown
```

Running the other demonstrations

The remaining demonstrations are all based on the Camel CXF component. You can only run *one* of these demonstrations at a time, because they all use the same Web service port and would clash, if started at the same time:

- **customer-ws-camel-cxf-pojo**

- **customer-ws-camel-cxf-payload**

- **customer-ws-camel-cxf-provider**

The preceding demonstrations all require the Camel CXF component and some of them require the Camel Velocity component as well. Before you run the demonstrations, you must install the requisite features for these Camel components, as follows:

```
JBossFuse:karaf@root> features:install camel-cxf
JBossFuse:karaf@root> features:install camel-velocity
```

You can test these demonstrations using the provided **customer-ws-client** client or using the third–party SoapUI utility.

CHAPTER 35. JAVA-FIRST SERVICE IMPLEMENTATION

35.1. JAVA-FIRST OVERVIEW

Overview

The Java-first approach is a convenient way to get started with Web services, if you are unfamiliar with WSDL syntax. Using this approach, you can define the Web service interface using an ordinary Java interface and then use the provided Apache CXF utilities to generate the corresponding WSDL contract from the Java interface.

> **NOTE**
>
> There is no demonstration code to accompany this example.

Service Endpoint Interface (SEI)

An SEI is an ordinary Java interface. In order to use the standard JAX-WS frontend, the SEI must be annotated with the **@WebService** annotation.[1]

In the Java-first approach, the SEI is the starting point for implementing the Web service and it plays a central role in the development of the Web service implementation. The SEI is used in the following ways:

- *Base type of the Web service implementation (server side)* —you define the Web service by implementing the SEI.

- *Proxy type (client side)* —on the client side, you use the SEI to invoke operations on the client proxy object.

- *Basis for generating the WSDL contract* —in the Java-first approach, you generate the WSDL contract by converting the SEI to WSDL.

WSDL contract

The WSDL contract is a platform-neutral and language-neutral description of the Web service interface. When you want to make the Web service available to third-party clients, you should publish the WSDL contract to some well-known location. The WSDL contract contains all of the metadata required by WS clients.

The CustomerService demonstration

Figure 35.1, "Building a Java-First Web Service" shows an overview of the files required to implement and build the **CustomerService** Web service using the Java-first approach.

Figure 35.1. Building a Java-First Web Service
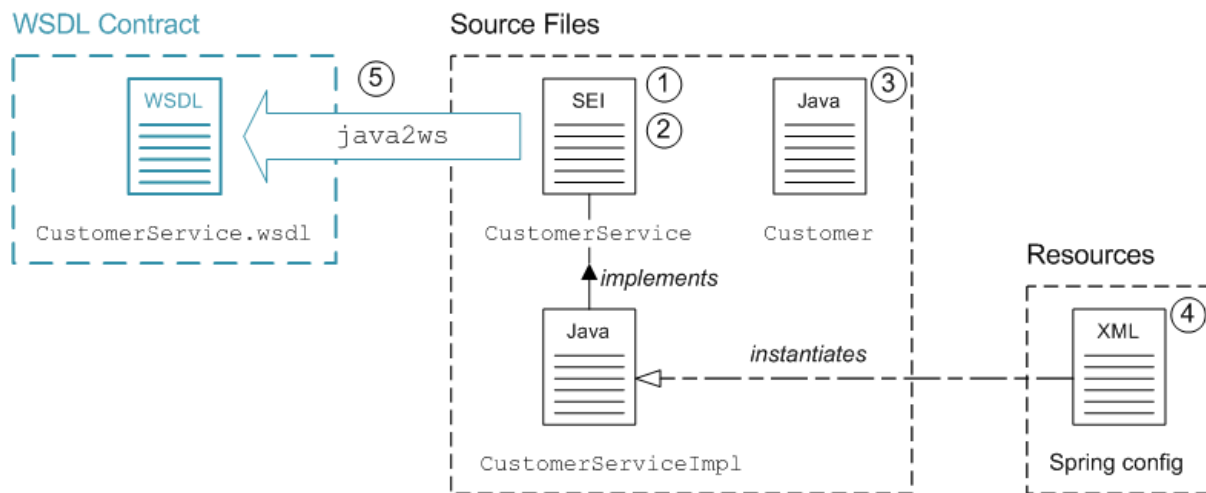
Implementing and building the service

To implement and build the Java-first example shown in Figure 35.1, "Building a Java-First Web Service" , you would perform the following steps:

1. Implement the SEI, which constitutes the basic definition of the Web service's interface.

2. Annotate the SEI (you can use the annotations to influence the ultimate form of the generated WSDL contract).

3. Implement any other requisite Java classes. In particular, implement the following:

    - Any data types referenced by the SEI—for example, the **Customer** class.

    - The implementation of the SEI, **CustomerServiceImpl**.

4. Instantiate the Web service endpoint, by adding the appropriate code to a Spring XML file.

5. Generate the WSDL contract using a Java-to-WSDL converter.

35.2. DEFINE SEI AND RELATED CLASSES

Overview

The Service Endpoint Interface (SEI) is the starting point for implementing a Web service in the Java-first approach. The SEI represents the Web service in Java and it is ultimately used as the basis for generating the WSDL contract. This section describes how to create a sample SEI, the **CustomerService** interface, which enables you to access the details of a customer's account.

The CustomerService SEI

A JAX-WS service endpoint interface (SEI) is essentially an ordinary Java interface, augmented by certain annotations (which are discussed in the next section). For example, consider the following **CustomerService** interface, which defines methods for accessing the **Customer** data type:

```
// Java
package com.fusesource.demo.wsdl.customerservice;
```

```
// NOT YET ANNOTATED!
public interface CustomerService {

    public com.fusesource.demo.customer.Customer lookupCustomer(
        java.lang.String customerId
    );

    public void updateCustomer(
        com.fusesource.demo.customer.Customer cust
    );

    public void getCustomerStatus(
        java.lang.String customerId,
        javax.xml.ws.Holder<java.lang.String> status,
        javax.xml.ws.Holder<java.lang.String> statusMessage
    );
}
```

After adding the requisite annotations to the **CustomerService** interface, this interface provides the basis for defining the **CustomerService** Web service.

javax.xml.ws.Holder<?> types

The **getCustomerStatus** method from the **CustomerService** interface has parameters declared to be of **javax.xml.ws.Holder<String>** type. These so-called *holder types* are needed in order to declare the **OUT** or **INOUT** parameters of a WSDL operation.

The syntax of WSDL operations allows you to define any number of OUT or INOUT parameters, which means that the parameters are used to *return* a value to the caller. This kind of parameter passing is *not* natively supported by the Java language. Normally, the only way that Java allows you to return a value is by declaring it as the return value of a method. You can work around this language limitation, however, by declaring parameters to be *holder* types.

For example, consider the definition of the following method, **getStringValues()**, which takes a holder type as its second parameter:

```
// Java
public void getStringValues(
    String wrongWay,
    javax.xml.ws.Holder<String> rightWay
) {
    wrongWay = "Caller will never see this string!";
    rightWay.value = "But the caller *can* see this string.";
}
```

The caller can access the value of the returned **rightWay** string as **rightWay.value**. For example:

```
// Java
String wrongWay = "This string never changes";
javax.xml.ws.Holder<String> rightWay.value = "This value *can* change.";

sampleObject.getStringValues(wrongWay, rightWay);

System.out.println("Unchanged string: " + wrongWay);
System.out.println("Changed string: " + rightWay.value);
```

It is, perhaps, slightly unnatural to use **Holder<>** types in a Java-first example, because this is not a normal Java idiom. But it is interesting to include OUT parameters in the example, so that you can see how a Web service processes this kind of parameter.

Related classes

When you run the Java-to-WSDL compiler on the SEI, it converts not only the SEI, but also the classes referenced as parameters or return values. The parameter types must be convertible to XML, otherwise it would not be possible for WSDL operations to send or to receive those data types. In fact, when you run the Java-to-WSDL compiler, it is typically necessary to convert an entire tree of related classes to XML using the standard JAXB encoding.

Normally, as long as the related classes do not require any exotic language features, the JAXB encoding should be quite straightforward.

Default constructor for related classes

There is one simple rule, however, that you need to keep in mind when implementing related classes: each related class *must* have a default constructor (that is, a constructor without arguments). If you do not define any constructor for a class, the Java language automatically adds a default constructor. But if you define a class's constructors explicitly, you must ensure that one of them is a default constructor.

The Customer class

For example, the **Customer** class appears as a related class in the definition of the **CustomerService** SEI (the section called "The CustomerService SEI"). The **Customer** class consists of a collection of **String** fields and the only special condition it needs to satisfy is that it includes a default constructor:

```Java
// Java
package com.fusesource.demo.customer;

public class Customer {
    protected String firstName;
    protected String lastName;
    protected String phoneNumber;
    protected String id;

    // Default constructor, required by JAX-WS
  public Customer() { }

    public Customer(String firstName, String lastName, String phoneNumber,
        String id) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.phoneNumber = phoneNumber;
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String value) {
        this.firstName = value;
```

```
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String value) {
        this.lastName = value;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String value) {
        this.phoneNumber = value;
    }

    public String getId() {
        return id;
    }

    public void setId(String value) {
        this.id = value;
    }
}
```

35.3. ANNOTATE SEI FOR JAX-WS

Overview

To use the JAX-WS frontend, an SEI must be annotated using standardised JAX-WS annotations. The annotations signal to the Web services tooling that the SEI uses JAX-WS and the annotations are also used to customize the mapping from Java to WSDL. Here we only cover the most basic annotations—for complete details of JAX-WS annotations, see *Developing Applications Using JAX-WS* from the Apache CXF library.

> **NOTE**
>
> It sometimes makes sense also to annotate the service implementation class (the class that implements the SEI)—for example, if you want to associate an implementation class with a specific WSDL **serviceName** and **portName** (there can be more than one implementation of a given SEI).

Minimal annotation

The minimal annotation required for an SEI using the JAX-WS frontend is to prefix the interface declaration with **@WebService**. For example, the **CustomerService** SEI could be minimally annotated as follows:

```
// Java
package com.fusesource.demo.wsdl.customerservice;

import javax.jws.WebService;
```

```
@WebService
public interface CustomerService {
  ...
}
```

If you run the Java-to-WSDL utility on this interface, it will generate a complete WSDL contract using the standard default style of code conversion.

@WebService annotation

Although it is sufficient to specify the **@WebService** annotation without any attributes, it is usually better to specify some attributes to provide a more descriptive WSDL service name and WSDL port name. You will also usually want to specify the XML target namespace. For this, you can specify the following optional attributes of the **@WebService** annotation:

name

> Specifies the name of the WSDL contract (appearing in the **wsdl:definitions** element).

serviceName

> Specifies the name of the WSDL service (a SOAP service is defined by default in the generated contract).

portName

> Specifies the name of the WSDL port (a SOAP/HTTP port is defined by default in the generated contract).

targetNamespace

> The XML schema namespace that is used, by default, to qualify the elements and types defined in the contract.

@WebParam annotation

You can add the **@WebParam** annotation to method arguments in the SEI. The **@WebParam** annotation is optional, but there are a couple of good reasons for adding it:

- By default, JAX-WS maps Java arguments to parameters with names like **arg0**, ..., **arg$N$**. Messages are much easier to read, however, when the parameters have meaningful names.

- It is a good idea to define parameter elements without a namespace. This makes the XML encoding of requests and responses more compact.

- To enable support for WSDL OUT and INOUT parameters.

You can add **@WebParam** annotations with the following attributes:

name

> Specifies the mapped name of the parameter.

targetNamespace

> Specifies the namespace of the mapped parameter. Set this to a blank string for a more compact XML encoding.

mode

Can have one of the following values:

- **WebParam.Mode.IN**—*(default)* parameter is passed from client to service (in request).

- **WebParam.Mode.INOUT**—parameter is passed from client to service (request) and from the service back to the client (in reply).

- **WebParam.Mode.OUT**—parameter is passed from service back to the client (in reply).

OUT and INOUT parameters

In WSDL, OUT and INOUT parameters represent values that can be sent from the service back to the client (where the INOUT parameter is sent in both directions).

In Java syntax, the only value that can ordinarily be returned from a method is the method's return value. In order to support OUT or INOUT parameters in Java (which are effectively like additional return values), you must:

- Declare the corresponding Java argument using a **javax.xml.ws.Holder<*ParamType*>** type, where ***ParamType*** is the type of the parameter you want to send.

- Annotate the Java argument with **@WebParam**, setting either **mode = WebParam.Mode.OUT** or **mode = WebParam.Mode.INOUT**.

Annotated CustomerService SEI

The following example shows the **CustomerService** SEI after it has been annotated. Many other annotations are possible, but this level of annotation is usually adequate for a WSDL-first project.

```java
// Java
package com.fusesource.demo.wsdl.customerservice;

import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(
  targetNamespace = "http://demo.fusesource.com/wsdl/CustomerService/",
  name = "CustomerService",
  serviceName = "CustomerService",
  portName = "SOAPOverHTTP"
)
public interface CustomerService {

    public com.fusesource.demo.customer.Customer lookupCustomer(
  @WebParam(name = "customerId", targetNamespace = "")
      java.lang.String customerId
    );

    public void updateCustomer(
  @WebParam(name = "cust", targetNamespace = "")
      com.fusesource.demo.customer.Customer cust
    );
```

```
    public void getCustomerStatus(
  @WebParam(name = "customerId", targetNamespace = "")
      java.lang.String customerId,
  @WebParam(mode = WebParam.Mode.OUT, name = "status", targetNamespace = "")
      javax.xml.ws.Holder<java.lang.String> status,
  @WebParam(mode = WebParam.Mode.OUT, name = "statusMessage", targetNamespace = "")
      javax.xml.ws.Holder<java.lang.String> statusMessage
  );
}
```

35.4. INSTANTIATE THE WS ENDPOINT

Overview

In Apache CXF, you create a WS endpoint by defining a **jaxws:endpoint** element in XML. The WS endpoint is effectively the runtime representation of the Web service: it opens an IP port to listen for SOAP/HTTP requests, is responsible for marshalling and unmarshalling messages (making use of the generated Java stub code), and routes incoming requests to the relevant methods on the implementor class.

In other words, creating a Web service in Spring XML consists essentially of the following two steps:

1. Create an instance of the implementor class, using the Spring **bean** element.

2. Create a WS endpoint, using the **jaxws:endpoint** element.

The jaxws:endpoint element

You can instantiate a WS endpoint using the **jaxws:endpoint** element in a Spring file, where the **jaxws:** prefix is associated with the **http://cxf.apache.org/jaxws** namespace.

> **NOTE**
>
> Take care not to confuse the **jaxws:endpoint** element with the **cxf:cxfEndpoint** element, which you meet later in this guide: the **jaxws:endpoint** element is used to integrate a WS endpoint with a Java implementation class; whereas the **cxf:cxfEndpoint** is used to integrate a WS endpoint with a Camel route.

Define JAX-WS endpoint in XML

The following sample Spring file shows how to define a JAX-WS endpoint in XML, using the **jaxws:endpoint** element.

```
<?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:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
```

```
<jaxws:endpoint
  xmlns:customer="http://demo.fusesource.com/wsdl/CustomerService/"
  id="customerService"
  address="/Customer"
  serviceName="customer:CustomerService"
  endpointName="customer:SOAPOverHTTP"
  implementor="#customerServiceImpl">
</jaxws:endpoint>

<bean id="customerServiceImpl"
    class="com.fusesource.customer.ws.CustomerServiceImpl"/>

</beans>
```

Address for the Jetty container

Apache CXF deploys the WS endpoint into a Jetty servlet container instance and the **address** attribute of **jaxws:endpoint** is therefore used to configure the addressing information for the endpoint in the Jetty container.

Apache CXF supports the notion of a *default servlet container* instance. The way the default servlet container is initialized and configured depends on the particular mode of deployment that you choose. For example the Red Hat JBoss Fuse container and Web containers (such as Tomcat) provide a default servlet container.

There are two different syntaxes you can use for the endpoint address, where the syntax that you use effectively determines whether or not the endpoint is deployed into the default servlet container, as follows:

- *Address syntax for default servlet container* —to use the default servlet container, specify only the servlet context for this endpoint. Do *not* specify the protocol, host, and IP port in the address. For example, to deploy the endpoint to the /**Customers** servlet context in the default servlet container:

  ```
  address="/Customers"
  ```

- *Address syntax for custom servlet container* —to instantiate a custom Jetty container for the endpoint, specify a complete HTTP URL, including the host and IP port (the value of the IP port effectively identifies the target Jetty container). Typically, for a Jetty container, you specify the host as **0.0.0.0**, which is interpreted as a wildcard that matches every IP network interface on the local machine (that is, if deployed on a multi-homed host, Jetty opens a listening port on every network card). For example, to deploy the endpoint to the custom Jetty container listening on IP port, **8083**:

  ```
  address="http://0.0.0.0:8083/Customers"
  ```

  > **NOTE**
  >
  > If you want to configure a secure endpoint (secured by SSL), you would specify the **https:** scheme in the address.

Referencing the service implementation

The **implementor** attribute of the **jaxws:endpoint** element references the implementation of the WS

service. The value of this attribute can either be the name of the implementation class or (as in this example) a bean reference in the format, **#BeanID**, where the **#** character indicates that the following identifier is the name of a bean in the bean registry.

35.5. JAVA-TO-WSDL MAVEN PLUG-IN

Overview

To generate a WSDL contract from your SEI, you can use either the **java2ws** command-line utility or the **cxf-java2ws-plugin** Maven plug-in. The plug-in approach is ideal for Maven-based projects: after you paste the requisite plug-in configuration into your POM file, the WSDL code generation step is integrated into your build.

Configure the Java-to-WSDL Maven plug-in

Configuring the Java-to-WSDL Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **plugin** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.

- *SEI class name*—specify the fully-qualified class name of the SEI in the **configuration/className** element.

- *Location of output*—specify the location of the generated WSDL file in the **configuration/outputFile** element.

For example, the following POM fragment shows how to configure the **cxf-java2ws-plugin** plug-in to generate WSDL from the CustomerService SEI:

```xml
<project ...>
  ...
  <properties>
    <cxf.version>3.1.5.redhat-630xxx</cxf.version>
  </properties>

  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-java2ws-plugin</artifactId>
        <version>${cxf.version}</version>
        <dependencies>
          <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-jaxws</artifactId>
            <version>${cxf.version}</version>
          </dependency>
          <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-simple</artifactId>
            <version>${cxf.version}</version>
```

```
      </dependency>
    </dependencies>
    <executions>
      <execution>
        <id>process-classes</id>
        <phase>process-classes</phase>
        <configuration>
          <className>org.fusesource.demo.camelcxf.ws.server.CustomerService</className>
          <outputFile>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</outputFile>
          <genWsdl>true</genWsdl>
          <verbose>true</verbose>
        </configuration>
        <goals>
          <goal>java2ws</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  </plugins>
</build>

</project>
```

Generated WSDL

When using the Java-first approach to defining a Web service, there are typically other parts of your application (for example, WS clients) that depend on the generated WSDL file. For this reason, it is generally a good idea to output the generated WSDL file to a common location, which is accessible to other projects in your application, using the **outputFile** configuration element.

If you do not specify the **outputFile** configuration element, the generated WSDL is sent to the following location, by default:

BaseDir/target/generated/wsdl/*SEIClassName*.wsdl

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven Java2WS plug-in reference page.

---

[1] If the SEI is left without annotations, Apache CXF defaults to using the simple frontend. This is a non-standard frontend, which is *not* recommended for most applications.

CHAPTER 36. WSDL-FIRST SERVICE IMPLEMENTATION

36.1. WSDL-FIRST OVERVIEW

Overview

If you are familiar with the syntax of WSDL and you want to have ultimate control over the layout and conventions applied to the WSDL contract, you will probably prefer to develop your Web service using the WSDL-first approach. In this approach, you start with the WSDL contract and then use the provided Apache CXF utilities to generate the requisite Java stub files from the WSDL contract.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

> cxf-webinars-jboss-fuse-6.3/**customer-ws-osgi-bundle**

For details of how to download and install the demonstration code, see Chapter 34, *Demonstration Code for Camel/CXF*

WSDL contract

The WSDL contract is a platform-neutral and language-neutral description of the Web service interface. In the WSDL-first approach, the WSDL contract is the starting point for implementing the Web service. You can use it to generate Java stub code, which provides the basis for implementing the Web service on the server side.

Service Endpoint Interface (SEI)

The most important piece of the generated stub code is the SEI, which is an ordinary Java interface that represents the Web service interface in the Java language.

The SEI is used in the following ways:

- *Base type of the Web service implementation (server side)* —you define the Web service by implementing the SEI.

- *Proxy type (client side)* —on the client side, you use the SEI to invoke operations on the client proxy object.

The CustomerService demonstration

Figure 36.1, "Building a WSDL-First Web Service" shows an overview of the files required to implement and build the **CustomerService** Web service using the WSDL-first approach.

Figure 36.1. Building a WSDL-First Web Service
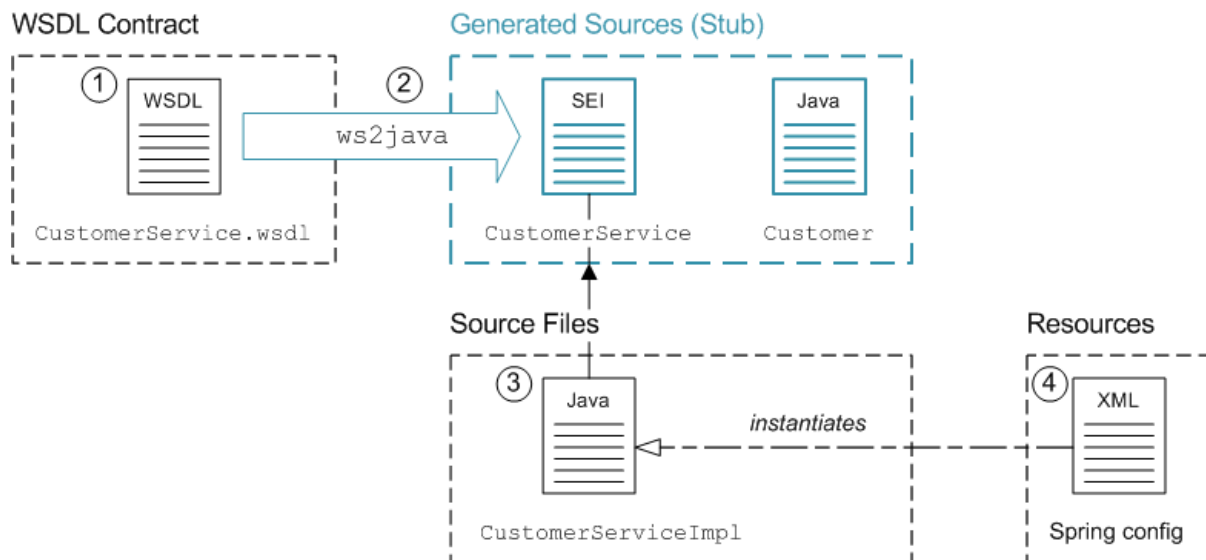
Implementing and building the service

To implement and build the WSDL-first example shown in Figure 36.1, "Building a WSDL-First Web Service", starting from scratch, you would perform the following steps:

1. Create the WSDL contract.

2. Generate the Java stub code from the WSDL contract using a WSDL-to-Java converter, **ws2java**. This gives you the SEI, **CustomerService**, and its related classes, such as **Customer**.

3. Write the implementation of the SEI, **CustomerServiceImpl**.

4. Instantiate the Web service endpoint, by adding the appropriate code to a Spring XML file.

36.2. CUSTOMERSERVICE WSDL CONTRACT

Sample WSDL contract

The WSDL contract used in this demonstration is the **CustomerService** WSDL contract, which is available in the following location:

> cxf-webinars-jboss-fuse-6.3/src/main/resources/wsdl

Because the WSDL contract is a fairly verbose format, it is not shown in here in full. The main point you need to be aware of is that the **CustomerSerivice** WSDL contract exposes the following operations:

lookupCustomer

Given a customer ID, the operation returns the corresponding **Customer** data object.

updateCustomer

Stores the given **Customer** data object against the given customer ID.

getCustomerStatus

Returns the status of the customer with the given customer ID.

Parts of the WSDL contract

A WSDL contract has the following main parts:

- the section called "Port type".

- the section called "WSDL binding".

- the section called "WSDL port".

Port type

The port type is defined in the WSDL contract by the **wsdl:portType** element. It is analogous to an interface and it defines the operations that can be invoked on the Web service.

For example, the following WSDL fragment shows the **wsdl:portType** definition from the **CustomerService** WSDL contract:

```
<wsdl:definitions name="CustomerService"
 targetNamespace="http://demo.fusesource.com/wsdl/CustomerService/"
   ...>

   ...
   <wsdl:portType name="CustomerService">
      <wsdl:operation name="lookupCustomer">
         <wsdl:input message="tns:lookupCustomer"></wsdl:input>
         <wsdl:output message="tns:lookupCustomerResponse"></wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="updateCustomer">
         <wsdl:input message="tns:updateCustomer"></wsdl:input>
         <wsdl:output message="tns:updateCustomerResponse"></wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="getCustomerStatus">
         <wsdl:input message="tns:getCustomerStatus"></wsdl:input>
         <wsdl:output message="tns:getCustomerStatusResponse"></wsdl:output>
      </wsdl:operation>
   </wsdl:portType>
   ...
</wsdl:definitions>
```

WSDL binding

A WSDL binding describes how to *encode* all of the operations and data types associated with a particular port type. A binding is specific to a particular protocol—for example, SOAP or JMS.

WSDL port

A WSDL port *specifies the transport protocol* and contains addressing data that enables clients to locate and connect to a remote server endpoint.

For example, the **CustomerService** WSDL contract defines the following WSDL port:

```
<wsdl:definitions ...>
   ...
   <wsdl:service name="CustomerService">
```

```
    <wsdl:port name="SOAPOverHTTP" binding="tns:CustomerServiceSOAP">
        <soap:address location="http://0.0.0.0:8183/CustomerService" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

The address specified by the **soap:address** element's **location** attribute in the original WSDL contract is typically overridden at run time, however.

The getCustomerStatus operation

Because a WSDL contract is fairly verbose, it can be a bit difficult to see what the parameters of an operation are. Typically, for each operation, you can find data types in the XML schema section that represent the operation request and the operation response. For example, the **getCustomerStatus** operation has its request parameters (IN parameters) encoded by the **getCustomerStatus** element and its response parameters (OUT parameters) encoded by the **getCustomerStatusResponse** element, as follows:

```
<wsdl:definitions name="CustomerService"
 targetNamespace="http://demo.fusesource.com/wsdl/CustomerService/"
   ...>
  <wsdl:types>
    <xsd:schema ...>
      ...
      <xsd:element name="getCustomerStatus">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="customerId" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getCustomerStatusResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="status" type="xsd:string"/>
            <xsd:element name="statusMessage" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  ...
</wsdl:definitions>
```

References

For more details about the format of WSDL contracts and how to create your own WSDL contracts, see *Writing WSDL Contracts* and the Eclipse JAX-WS Tools Component .

36.3. WSDL-TO-JAVA MAVEN PLUG-IN

Overview

In contrast to the Java-first approach, which starts with a Java interface and then generates the WSDL contract, the WSDL-first approach needs to generate Java stub code from the WSDL contract.

To generate Java stub code from the WSDL contract, you can use either the **ws2java** command-line utility or the **cxf-codegen-plugin** Maven plug-in. The plug-in approach is ideal for Maven-based projects: after you paste the requisite plug-in configuration into your POM file, the WSDL-to-Java code generation step is integrated into your build.

Configure the WSDL-to-Java Maven plug-in

Configuring the WSDL-to-Java Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **plugin** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.

- *WSDL file location* —specify the WSDL file location in the **configuration/wsdlOptions/wsdlOption/wsdl** element.

- *Location of output* —specify the root directory of the generated Java source files in the **configuration/sourceRoot** element.

For example, the following POM fragment shows how to configure the **cxf-codegen-plugin** plug-in to generate Java stub code from the **CustomerService.wsdl** WSDL file:

```xml
<project ...>
 ...
 <parent>
   <groupId>com.fusesource.byexample.cxf-webinars</groupId>
   <artifactId>cxf-webinars</artifactId>
   <version>1.0-SNAPSHOT</version>
 </parent>

 <build>
   <defaultGoal>install</defaultGoal>
   <plugins>
    ...
    <plugin>
      <groupId>org.apache.cxf</groupId>
      <artifactId>cxf-codegen-plugin</artifactId>
      <version>${cxf-version}</version>
      <executions>
       <execution>
         <id>generate-sources</id>
         <phase>generate-sources</phase>
         <configuration>
           <!-- Maven auto-compiles any source files under target/generated-sources/ -->
           <sourceRoot>${basedir}/target/generated-sources/jaxws</sourceRoot>
           <wsdlOptions>
             <wsdlOption>
               <wsdl>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</wsdl>
             </wsdlOption>
           </wsdlOptions>
         </configuration>
         <goals>
```

CHAPTER 36. WSDL-FIRST SERVICE IMPLEMENTATION

```
      <goal>wsdl2java</goal>
    </goals>
  </execution>
</executions>
</plugin>


</plugins>
</build>


</project>
```

Generated Java source code

With the sample configuration shown here, the generated Java source code is written under the **target/generated-sources/jaxws** directory. Note that the Web service implementation is dependent on this generated stub code—for example, the service implementation class must implement the generated **CustomerService** SEI.

Adding the generated source to an IDE

If you are using an IDE such as Eclipse or Intellij's IDEA, you need to make sure that the IDE is aware of the generated Java code. For example, in Eclipse it is necessary to add the **target/generated-sources/jaxws** directory to the project as a source code directory.

Compiling the generated code

You must ensure that the generated Java code is compiled and added to the deployment package. By convention, Maven automatically compiles any source files that it finds under the following directory:

> *BaseDir*/target/generated-sources/

Hence, if you configure the output directory as shown in the preceding POM fragment, the generated code is automatically compiled by Maven.

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven cxf-codegen-plugin reference page.

36.4. INSTANTIATE THE WS ENDPOINT

Overview

In Apache CXF, you create a WS endpoint by defining a **jaxws:endpoint** element in XML. The WS endpoint is effectively the runtime representation of the Web service: it opens an IP port to listen for SOAP/HTTP requests, is responsible for marshalling and unmarshalling messages (making use of the generated Java stub code), and routes incoming requests to the relevant methods on the implementor class.

In other words, creating a Web service in Spring XML consists essentially of the following two steps:

1. Create an instance of the implementor class, using the Spring **bean** element.

2. Create a WS endpoint, using the **jaxws:endpoint** element.

Define JAX-WS endpoint in XML

The following sample Spring file shows how to define a JAX-WS endpoint in XML, using the **jaxws:endpoint** element.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:endpoint
    xmlns:customer="http://demo.fusesource.com/wsdl/CustomerService/"
    id="customerService"
    address="/Customer"
    serviceName="customer:CustomerService"
    endpointName="customer:SOAPOverHTTP"
    implementor="#customerServiceImpl">
  </jaxws:endpoint>

  <bean id="customerServiceImpl"
      class="com.fusesource.customer.ws.CustomerServiceImpl"/>

</beans>
```

Address for the Jetty container

In the preceding example, the **address** attribute of the **jaxws:endpoint** element specifies the servlet context for this endpoint, relative to the Jetty container in which it is deployed.

For more details about the options for specifying the endpoint address, see the section called "Address for the Jetty container".

Referencing the service implementation

The **implementor** attribute of the **jaxws:endpoint** element is used to reference the implementation of the WS service. The value of this attribute can either be the name of the implementation class or (as in this example) a bean reference in the format, *#BeanID*, where the **#** character indicates that the following identifier is the name of a bean in the bean registry.

36.5. DEPLOY TO AN OSGI CONTAINER

Overview

One of the options for deploying the Web service is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).

- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **pom.xml** file:

1. Change the packaging type to **bundle** (by editing the value of the **project**/**packaging** element in the POM).

2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```xml
<?xml version="1.0"?>
<project ...>
 ...
 <groupId>com.fusesource.byexample.cxf-webinars</groupId>
 <artifactId>customer-ws-osgi-bundle</artifactId>
 <name>customer-ws-osgi-bundle</name>
 <url>http://www.fusesource.com</url>
 <packaging>bundle</packaging>
 ...
 <build>
   <plugins>
    ...
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>${version.maven-bundle-plugin}</version>
      <extensions>true</extensions>
      <configuration>
       <instructions>
        <Export-Package>
          !com.fusesource.customer.ws,
          !com.fusesource.demo.customer,
          !com.fusesource.demo.wsdl.customerservice
        </Export-Package>
        <Import-Package>
          *
        </Import-Package>
        <DynamicImport-Package>
          org.apache.cxf.*,
```

```
        org.springframework.beans.*
      </DynamicImport-Package>
    </instructions>
  </configuration>
</plugin>
  ...
</plugins>
</build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **DynamicImport-Package** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **DynamicImport-Package** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

> **NOTE**
>
> In general, using **DynamicImport-Package** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **Import-Package** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build-time versions listed in the **Import-Package** header. With dynamic imports, this version checking cannot be performed.

Build and deploy the service bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the service bundle, enter the following command at the command console:

```
karaf@root> install -s mvn:com.fusesource.byexample.cxf-webinars/customer-ws-osgi-bundle/1.0-SNAPSHOT
```

> **NOTE**
>
> If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the *EsbInstallDir*/etc/org.ops4j.pax.url.mvn.cfg file, before you can use the **mvn:** scheme to access Maven artifacts.

Red Hat JBoss Fuse default servlet container

Red Hat JBoss Fuse has a default Jetty container which, by default, listens for HTTP requests on port 8181. Moreover, WS endpoints in this container are implicitly deployed under the servlet context **cxf**/. Hence, any WS endpoint whose **address** attribute is configured in the **jaxws:endpoint** element as

/*EndpointContext* will have the following effective address:

> http://*Hostname*:8181/cxf/*EndpointContext*

You can optionally customize the default servlet container by editing settings in the following file:

> *InstallDir*/etc/org.ops4j.pax.web.cfg

Full details of the properties you can set in this file are given in the Ops4j Pax Web configuration reference.

Check that the service is running

A simple way of checking that the service is running is to point your browser at the following URL:

> http://localhost:8181/cxf/Customer?wsdl

This query should return a copy of the WS endpoint's WSDL contract.

CHAPTER 37. IMPLEMENTING A WS CLIENT

37.1. WS CLIENT OVERVIEW

Overview

The key object in a WS client is the WS client proxy object, which enables you to access the remote Web service by invoking methods on the SEI. The proxy object itself can easily be instantiated using the **jaxws:client** element in Spring XML.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

> cxf-webinars-jboss-fuse-6.3/**customer-ws-client**

For details of how to download and install the demonstration code, see Chapter 34, *Demonstration Code for Camel/CXF*

WSDL contract

The WSDL contract is a platform-neutral and language-neutral description of the Web service interface. It contains all of the metadata that a client needs to find a Web service and invoke its operations. You can generate Java stub code from the WSDL contract, which provides an API that makes it easy to invoke the remote WSDL operations.

Service Endpoint Interface (SEI)

The most important piece of the generated stub code is the SEI, which is an ordinary Java interface that represents the Web service interface in the Java language.

WS client proxy

The WS client proxy is an object that converts Java method invocations to remote procedure calls, sending and receiving messages to a remote instance of the Web service across the network. The methods of the proxy are exposed through the SEI.

> **NOTE**
>
> The proxy type is generated *dynamically* by Apache CXF at run time. That is, there is no class in the stub code that corresponds to the implementation of the proxy (the only relevant entity is the SEI, which defines the proxy's interface).

The CustomerService client

To take a specific example, consider the **customer-ws-client** demonstration, which is available from the following location:

> cxf-webinars-jboss-fuse-6.3/customer-ws-client

Figure 37.1, "Building a WS Client" shows an overview of the files required to implement and build the WS client.

Figure 37.1. Building a WS Client
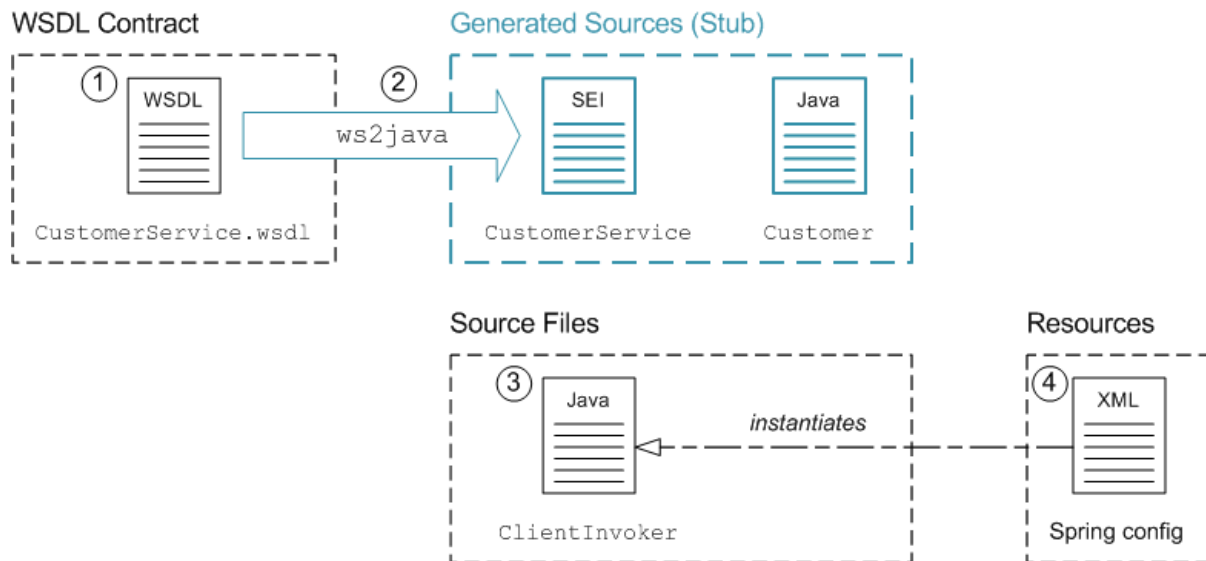
Implementing and building the WS client

To implement and build the sample WS client shown in Figure 37.1, "Building a WS Client" , starting from scratch, you would perform the following steps:

1. Obtain a copy of the WSDL contract.

2. Generate the Java stub code from the WSDL contract using a WSDL-to-Java converter, **ws2java**. This gives you the SEI, **CustomerService**, and its related classes, such as **Customer**.

3. Implement the main client class, **ClientInvoker**, which invokes the Web service operations. In this class define a bean property of type, **CustomerService**, so that the client class can receive a reference to the WS client proxy by property injection.

4. In a Spring XML file, instantiate the WS client proxy and inject it into the main client class, **ClientInvoker**.

37.2. WSDL-TO-JAVA MAVEN PLUG-IN

Overview

To generate Java stub code from the WSDL contract, you can use either the **ws2java** command-line utility or the **cxf-codegen-plugin** Maven plug-in. When using Maven, the plug-in approach is ideal: after you paste the requisite plug-in configuration into your POM file, the WSDL-to-Java code generation step is integrated into your build.

Configure the WSDL-to-Java Maven plug-in

Configuring the WSDL-to-Java Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **plugin** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.

- *WSDL file location*—specify the WSDL file location in the **configuration/wsdlOptions/wsdlOption/wsdl** element.

- *Location of output*—specify the root directory of the generated Java source files in the **configuration/sourceRoot** element.

For example, the following POM fragment shows how to configure the **cxf-codegen-plugin** plug-in to generate Java stub code from the **CustomerService.wsdl** WSDL file:

```
<project ...>
 ...
 <parent>
  <groupId>com.fusesource.byexample.cxf-webinars</groupId>
  <artifactId>cxf-webinars</artifactId>
  <version>1.0-SNAPSHOT</version>
 </parent>

 <build>
  <defaultGoal>install</defaultGoal>
  <plugins>

   ...
   <plugin>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-codegen-plugin</artifactId>
    <version>${cxf-version}</version>
    <executions>
     <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
       <sourceRoot>${basedir}/target/generated-sources/jaxws</sourceRoot>
       <wsdlOptions>
        <wsdlOption>
         <wsdl>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</wsdl>
        </wsdlOption>
       </wsdlOptions>
      </configuration>
      <goals>
       <goal>wsdl2java</goal>
      </goals>
     </execution>
    </executions>
   </plugin>

  </plugins>
 </build>

</project>
```

Generated Java source code

With the sample configuration shown here, the generated Java source code is written under the

target/generated-sources/jaxws directory. Note that the client implementation is dependent on this generated stub code—for example, the client invokes the proxy using the generated **CustomerService** SEI.

Add generated source to IDE

If you are using an IDE such as Eclipse or Intellij's IDEA, you need to make sure that the IDE is aware of the generated Java code. For example, in Eclipse it is necessary to add the **target/generated-sources/jaxws** directory to the project as a source code directory.

Compiling the generated code

You must ensure that the generated Java code is compiled and added to the deployment package. By convention, Maven automatically compiles any source files that it finds under the following directory:

> *BaseDir*/target/generated-sources/

Hence, if you configure the output directory as shown in the preceding POM fragment, the generated code is automatically compiled by Maven.

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven cxf-codegen-plugin reference page.

37.3. INSTANTIATE THE WS CLIENT PROXY

Overview

The WS client proxy is the most important kind of object in a WS client, because it provides a simple way of invoking operations on a remote Web service. The proxy enables you to access a Web service by invoking methods locally on a Java interface. The methods invoked on the proxy object are then translated into remote procedure calls on the Web service.

You can instantiate a WS client proxy straightforwardly using the **jaxws:client** element.

Define the WS client in XML

The following Spring XML fragment shows how to instantiate a client proxy bean using the **jaxws:client** element.

```
<?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:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:client
```

```
      id="customerServiceProxy"
      address="http://localhost:8181/cxf/Customer"
      serviceClass="com.fusesource.demo.wsdl.customerservice.CustomerService"
      />

  <bean id="customerServiceClient"
     class="com.fusesource.customer.client.ClientInvoker"
     init-method="init" destroy-method="destroy">
      <property name="customerService" ref="customerServiceProxy"/>
  </bean>

</beans>
```

The jaxws:client element

The **jaxws:client** element creates a client proxy *dynamically* (that is, there is no dedicated class that represents a proxy implementation in the Java stub code). The following attributes are used to define the proxy:

id

The ID that you specify here is entered in the bean registry and can be used to reference the proxy instance from other beans.

address

The full address of the remote Web service that this proxy connects to.

serviceClass

The fully-qualified class name of the Web service's SEI (you invoke methods on the proxy through the SEI).

Injecting with the proxy reference

To access the proxy instance, simply inject the proxy into one or more other beans defined in XML. Given that the proxy ID has the value, **customerServiceProxy**, you can inject it into a bean property using the Spring **property** element, as follows:

```
<bean ...>
  <property name="customerService" ref="customerServiceProxy"/>
</bean>
```

The bean class that is being injected must have a corresponding **setCustomerService** setter method—for example:

```
// Java
...
public class ClientInvoker implements Runnable {
  ...
  public void setCustomerService(CustomerService customerService) {
    this.customerService = customerService;
  }

}
```

37.4. INVOKE WS OPERATIONS

Proxy interface is SEI interface

The proxy implements the SEI. Hence, to make remote procedure calls on the Web service, simply invoke the SEI methods on the proxy instance.

Invoking the lookupCustomer operation

For example, the **CustomerService** SEI exposes the **lookupCustomer** method, which takes a customer ID as its argument and returns a **Customer** data object. Using the proxy instance, **customerService**, you can invoke the **lookupCustomer** operation as follows:

```
// Java
com.fusesource.demo.customer.Customer response
 = customerService.lookupCustomer("1234");

log.info("Got back " + response.getFirstName() + " "
    + response.getLastName()
    + ", ph:" + response.getPhoneNumber() );
```

The ClientInvoker class

In the **cxf-webinars-jboss-fuse-6.3/customer-ws-client** project, there is a **ClientInvoker** class (located in **src/main/java/com/fusesource/customer/client**), which defines a continuous loop that invokes the **lookupCustomer** operation.

When you are experimenting with the demonstration code in the latter chapters of this guide, you might need to modify the **ClientInvoker** class, possibly adding operation invocations.

37.5. DEPLOY TO AN OSGI CONTAINER

Overview

One of the options for deploying the WS client is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).

- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **pom.xml** file:

1. Change the packaging type to **bundle** (by editing the value of the **project/packaging** element in the POM).

2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```xml
<?xml version="1.0"?>
<project ...>
  ...
  <groupId>com.fusesource.byexample.cxf-webinars</groupId>
  <artifactId>customer-ws-client</artifactId>
  <name>customer-ws-client</name>
 <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Export-Package>
              !com.fusesource.customer.client,
              !com.fusesource.demo.customer,
              !com.fusesource.demo.wsdl.customerservice
            </Export-Package>
            <Import-Package>
              *
            </Import-Package>
            <DynamicImport-Package>
              org.apache.cxf.*,
              org.springframework.beans.*
            </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **DynamicImport-Package** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **DynamicImport-Package** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

NOTE

In general, using **DynamicImport-Package** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **Import-Package** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **Import-Package** header. With dynamic imports, this version checking cannot be performed.

Build and deploy the client bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the client bundle, enter the following command at the containers command console:

```
karaf@root> install -s mvn:com.fusesource.byexample.cxf-webinars/customer-ws-client/1.0-SNAPSHOT
```

NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the *EsbInstallDir*/etc/**org.ops4j.pax.url.mvn.cfg** file, before you can use the **mvn:** scheme to access Maven artifacts.

Check that the client is running

Assuming that you have already deployed the corresponding Web service into the OSGi container, you can verify that the client is successfully invoking WSDL operations by checking the log, as follows:

```
karaf@root> log:display -n 10
```

The client invokes an operation on the Web service once every second.

CHAPTER 38. POJO-BASED ROUTE

38.1. PROCESSING MESSAGES IN POJO FORMAT

Overview

By default, the Camel CXF component marshals incoming Web service requests into the POJO data form, where the *In* message body is encoded as a list of Java objects (one for each operation parameter). The POJO data format has advantages and disadvantages, as follows:

- The big advantage of the POJO data format is that the operation parameters are encoded using the JAXB standard, which makes them easy to manipulate in Java.

- The downside of the POJO data format, on the other hand, is that it requires that the WSDL metadata is converted to Java in advance (as defined by the JAX-WS and JAXB mappings) and compiled into your application. This means that a POJO-based route is not very dynamic.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

> cxf-webinars-jboss-fuse-6.3/**customer-ws-camel-cxf-pojo**

For details of how to download and install the demonstration code, see Chapter 34, *Demonstration Code for Camel/CXF*

Camel CXF component

The Camel CXF component is an Apache CXF component that integrates Web services with routes. You can use it either to instantiate *consumer endpoints* (at the start of a route), which behave like Web service instances, or to instantiate *producer endpoints* (at any other points in the route), which behave like WS clients.

> **NOTE**
>
> Camel CXF endpoints—which are instantiated using the **cxf:cxfEndpoint** XML element and are implemented by the Apache Camel project—are not to be confused with the Apache CXF JAX-WS endpoints—which are instantiated using the **jaxws:endpoint** XML element and are implemented by the Apache CXF project.

POJO data format

POJO data format is the *default* data format used by the Camel CXF component and it has the following characteristics:

- JAX-WS and JAXB stub code (as generated from the WSDL contract) *must* be provided.

- The SOAP body is marshalled into a list of Java objects.

  - One Java object for each part or parameter of the corresponding WSDL operation.

  - The type of the message body is **org.apache.cxf.message.MessageContentsList**.

- The SOAP headers are converted into headers in the exchange's *In* message.

Implementing and building a POJO route

To implement and build the demonstration POJO-based route, starting from scratch, you would perform the following steps:

1. Obtain a copy of the WSDL contract that is to be integrated into the route.

2. Generate the Java stub code from the WSDL contract using a WSDL-to-Java converter. This gives you the SEI, **CustomerService**, and its related classes, such as **Customer**.

3. Instantiate the Camel CXF endpoint in Spring, using the **cxf:cxfEndpoint** element.

4. Implement the route in XML, where you can use the content-based router to sort requests by operation name.

5. Implement the operation processor beans, which are responsible for processing each operation. When implementing these beans, the message contents must be accessed in POJO data format.

Sample POJO route

Figure 38.1, "Sample POJO Route" shows an outline of the route that is used to process the operations of the **CustomerService** Web service using the POJO data format. After sorting the request messages by operation name, an operation-specific processor bean reads the incoming request parameters and then generates a response in the POJO data format.

Figure 38.1. Sample POJO Route

38.2. WSDL-TO-JAVA MAVEN PLUG-IN

Overview

To generate Java stub code from the WSDL contract, you can use either the **ws2java** command-line utility or the **cxf-codegen-plugin** Maven plug-in. When using Maven, the plug-in approach is ideal: after you paste the requisite plug-in configuration into your POM file, the WSDL-to-Java code generation step is integrated into your build.

Configure the WSDL-to-Java Maven plug-in

Configuring the WSDL-to-Java Maven plug-in is relatively easy, because most of the default configuration settings can be left as they are. After copying and pasting the sample **plugin** element into your project's POM file, there are just a few basic settings that need to be customized, as follows:

- *CXF version*—make sure that the plug-in's dependencies are using the latest version of Apache CXF.

- *WSDL file location* —specify the WSDL file location in the **configuration**/**wsdlOptions**/**wsdlOption**/**wsdl** element.

- *Location of output* —specify the root directory of the generated Java source files in the **configuration**/**sourceRoot** element.

For example, the following POM fragment shows how to configure the **cxf-codegen-plugin** plug-in to generate Java stub code from the **CustomerService.wsdl** WSDL file:

```
<project ...>
 ...
 <parent>
  <groupId>com.fusesource.byexample.cxf-webinars</groupId>
  <artifactId>cxf-webinars</artifactId>
  <version>1.0-SNAPSHOT</version>
 </parent>

 <build>
  <defaultGoal>install</defaultGoal>
  <plugins>

   ...
   <plugin>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-codegen-plugin</artifactId>
    <version>${cxf-version}</version>
    <executions>
     <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
       <sourceRoot>${basedir}/target/generated-sources/jaxws</sourceRoot>
       <wsdlOptions>
        <wsdlOption>
         <wsdl>${basedir}/../src/main/resources/wsdl/CustomerService.wsdl</wsdl>
        </wsdlOption>
       </wsdlOptions>
      </configuration>
      <goals>
       <goal>wsdl2java</goal>
      </goals>
     </execution>
    </executions>
   </plugin>

  </plugins>
```

```
</build>

</project>
```

Generated Java source code

With the sample configuration shown here, the generated Java source code is written under the **target/generated-sources/jaxws** directory. Note that the route is dependent on this generated stub code—for example, when processing the POJO parameters, the parameter processor uses the **Customer** data type from the stub code.

Add generated code to IDE

If you are using an IDE such as Eclipse or Intellij's IDEA, you need to make sure that the IDE is aware of the generated Java code. For example, in Eclipse it is necessary to add the **target/generated-sources/jaxws** directory to the project as a source code directory.

Compiling the generated code

You must ensure that the generated Java code is compiled and added to the deployment package. By convention, Maven automatically compiles any source files that it finds under the following directory:

> *BaseDir*/target/generated-sources/

Hence, if you configure the output directory as shown in the preceding POM fragment, the generated code is automatically compiled by Maven.

Reference

For full details of how to configure the Java-to-WSDL plug-in, see the Maven cxf-codegen-plugin reference page.

38.3. INSTANTIATE THE WS ENDPOINT

Overview

In Apache Camel, the Camel CXF component is the key to integrating routes with Web services. You can use the Camel CXF component to create a CXF endpoint, which can be used in either of the following ways:

- *Consumer*—(at the start of a route) represents a Web service instance, which integrates with the route. The type of payload injected into the route depends on the value of the endpoint's **dataFormat** option.

- *Producer*—(at other points in the route) represents a WS client proxy, which converts the current exchange object into an operation invocation on a remote Web service. The format of the current exchange must match the endpoint's **dataFormat** setting.

In the current demonstration, we are interested in creating a Camel CXF consumer endpoint, with the **dataFormat** option set to POJO.

Maven dependency

The Camel CXF component requires you to add a dependency on the **camel-cxf** component in your Maven POM. For example, the **pom.xml** file from the **customer-ws-camel-cxf-pojo** demonstration project includes the following dependency:

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

The cxf:bean: URI syntax

The **cxf:bean:** URI is used to bind an Apache CXF endpoint to a route and has the following general syntax:

```
cxf:bean:CxfEndpointID[?Options]
```

Where **CxfEndpointID** is the ID of a bean created using the **cxf:cxfEndpoint** element, which configures the details of the WS endpoint. You can append options to this URI (where the options are described in detail in chapter "CXF" in "Apache Camel Component Reference"). If you do not specify any additional options, *the endpoint uses the POJO data format by default* .

For example, to start a route with a Apache CXF endpoint that is configured by the bean with ID, **customer-ws**, define the route as follows:

```
<route>
   <from uri="cxf:bean:customer-ws"/>
   ...
</route>
```

> **NOTE**
>
> There is an alternative URI syntax, **cxf://***WsAddress***[?*Options*]**, which enables you to specify *all* of the WS endpoint details in the URI (so there is no need to reference a bean instance). This typically results in a long and cumbersome URI, but is useful in some cases.

The cxf:cxfEndpoint element

The **cxf:cxfEndpoint** element is used to define a WS endpoint that binds either to the start (consumer endpoint) or the end (producer endpoint) of a route. For example, to define the **customer-ws** WS endpoint referenced in the preceding route, you would define a **cxf:cxfEndpoint** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...
  xmlns:cxf="http://camel.apache.org/schema/cxf" ...>
  ...
  <cxf:cxfEndpoint id="customer-ws"
     address="/Customer"
     endpointName="c:SOAPOverHTTP"
     serviceName="c:CustomerService"
     serviceClass="com.fusesource.demo.wsdl.customerservice.CustomerService"
```

```
        xmlns:c="http://demo.fusesource.com/wsdl/CustomerService/"/>
    ...
</beans>
```

> **IMPORTANT**
>
> Remember that the **cxf:cxfEndpoint** element and the **jaxws:endpoint** element use *different* XML schemas (although the syntax looks superficially similar). These elements bind a WS endpoint in different ways: the **cxf:cxfEndpoint** element instantiates and binds a WS endpoint to an Apache Camel route, whereas the **jaxws:endpoint** element instantiates and binds a WS endpoint to a Java class using the JAX-WS mapping.

Address for the Jetty container

Apache CXF deploys the WS endpoint into a Jetty servlet container instance and the **address** attribute of **cxf:cxfEndpoint** is therefore used to configure the addressing information for the endpoint in the Jetty container.

Apache CXF supports the notion of a *default servlet container* instance. The way the default servlet container is initialized and configured depends on the particular mode of deployment that you choose. For example the Red Hat JBoss Fuse container and Web containers (such as Tomcat) provide a default servlet container.

There are two different syntaxes you can use for the endpoint address, where the syntax that you use effectively determines whether or not the endpoint is deployed into the default servlet container, as follows:

- *Address syntax for default servlet container* —to use the default servlet container, specify only the servlet context for this endpoint. Do *not* specify the protocol, host, and IP port in the address. For example, to deploy the endpoint to the /**Customer** servlet context in the default servlet container:

  ```
  address="/Customer"
  ```

- *Address syntax for custom servlet container* —to instantiate a custom Jetty container for this endpoint, specify a complete HTTP URL, including the host and IP port (the value of the IP port effectively identifies the target Jetty container). Typically, for a Jetty container, you specify the host as **0.0.0.0**, which is interpreted as a wildcard that matches every IP network interface on the local machine (that is, if deployed on a multi-homed host, Jetty opens a listening port on every network card). For example, to deploy the endpoint to the custom Jetty container listening on IP port, **8083**:

  ```
  address="http://0.0.0.0:8083/Customer"
  ```

  > **NOTE**
  >
  > If you want to configure a secure endpoint (secured by SSL), you would specify the **https:** scheme in the address.

Referencing the SEI

The **serviceClass** attribute of the **cxf:cxfEndpoint** element references the SEI of the Web service, which in this case is the **CustomerService** interface.

38.4. SORT MESSAGES BY OPERATION NAME

The operationName header

When the WS endpoint parses an incoming operation invocation in POJO mode, it automatically sets the **operationName** header to the name of the invoked operation. You can then use this header to sort messages by operation name.

Sorting by operation name

For example, the **customer-ws-camel-cxf-pojo** demonstration defines the following route, which uses the content-based router pattern to sort incoming messages, based on the operation name. The **when** predicates check the value of the **operationName** header using simple language expressions, sorting messages into invocations on the **updateCustomer** operation, the **lookupCustomer** operation, or the **getCustomerStatus** operation.

```
<beans ...>
    ...
    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxf:bean:customer-ws"/>
            <choice>
                <when>
                    <simple>${in.header.operationName} == 'updateCustomer'</simple>
                    <to uri="updateCustomer"/>
                </when>
                <when>
                    <simple>${in.header.operationName} == 'lookupCustomer'</simple>
                    <to uri="lookupCustomer"/>
                </when>
                <when>
                    <simple>${in.header.operationName} == 'getCustomerStatus'</simple>
                    <to uri="getCustomerStatus"/>
                </when>
            </choice>
        </route>
    </camelContext>

    <bean id="updateCustomer"
        class="com.fusesource.customerwscamelcxfpojo.UpdateCustomerProcessor"/>

    <bean id="getCustomerStatus"
        class="com.fusesource.customerwscamelcxfpojo.GetCustomerStatusProcessor"/>

    <bean id="lookupCustomer"
        class="com.fusesource.customerwscamelcxfpojo.LookupCustomerProcessor"/>

</beans>
```

Beans as endpoints

Note how the preceding route uses a convenient shortcut to divert each branch of the **choice** DSL to a different processor bean. The DSL for sending exchanges to producer endpoints (for example, **<to uri="*Destination*"/>**) is integrated with the bean registry: if the *Destination* does not resolve to an

endpoint or a component, the *Destination* is used as a bean ID to look up the bean registry. In this example, the exchange is routed to processor beans (which implement the **org.apache.camel.Processor** interface).

38.5. PROCESS OPERATION PARAMETERS

Overview

The most important characteristic of using Camel CXF in POJO mode is that the exchange's message body contains a list of Java objects, representing the parameters of the WSDL operation. The types of the Java objects are defined by the standard JAXB mapping and the implementations of these parameter types are provided by the Java stub code.

Contents of request message body

In POJO mode, the body of the request message is an **org.apache.cxf.message.MessageContentsList** object. You can also obtain the message body as an **Object[]** array (where type conversion is automatic).

When the body is obtained as an **Object[]** array, the array contains the list of all the operation's IN, INOUT, and OUT parameters *in exactly the same order as defined in the WSDL contract* (and in the same order as the corresponding operation signature of the SEI). The parameter mode affects the content as follows:

IN

> Contains a parameter value from the client.

INOUT

> Contains a **Holder** object containing a parameter value from the client.

OUT

> Contains an *empty* **Holder** object, which is a placeholder for the response.

> **NOTE**
>
> Unlike OUT parameters, there is no placeholder in the request's **Object[]** array to represent a return value.

Contents of response message body

In POJO mode, the body of the response message can be either an **org.apache.cxf.message.MessageContentsList** object or an **Object[]** array.

When setting the response body as an **Object[]** array, the array should contain *only* the operation's INOUT and OUT parameters in the same order as defined in the WSDL contract, omitting the IN parameters. The parameter mode affects the content as follows:

INOUT

> Contains a **Holder** object, which you must set to a response value. The **Holder** object used here *must be exactly the* **Holder** *object for the corresponding parameter that was extracted from the request* **Object[]** *array.* Creating and inserting a new **Holder** object into the **Object[]** array does *not* work.

OUT

Contains a **Holder** object, which you must initialize with a response value. The **Holder** object used here *must be exactly the **Holder** object for the corresponding parameter that was extracted from the request **Object[]** array*. Creating and inserting a new **Holder** object into the **Object[]** array does *not* work.

> **NOTE**
>
> If you defined the Web service interface using the Java-first approach, note that the return value (if any) must be set as the *first* element in the response's **Object[]** array. The return type is set as a plain object: it does *not* use a **Holder** object.

Example: getCustomerStatus operation

For example, the **getCustomerStatus** operation takes three parameters: IN, OUT, and OUT, respectively. The corresponding method signature in the SEI is, as follows:

```Java
// Java
public void getCustomerStatus(
    @WebParam(name = "customerId", targetNamespace = "")
    java.lang.String customerId,

    @WebParam(mode = WebParam.Mode.OUT, name = "status", targetNamespace = "")
    javax.xml.ws.Holder<java.lang.String> status,

    @WebParam(mode = WebParam.Mode.OUT, name = "statusMessage", targetNamespace = "")
    javax.xml.ws.Holder<java.lang.String> statusMessage
);
```

Example: request and response bodies

For the **getCustomerStatus** operation, the bodies of the request message and the response message have the following contents:

- *Request message*—as an **Object[]** array type, the contents are: **{ String customerId, Holder<String> status, Holder<String> statusMessage }**.

- *Response message*—as an **Object[]** array type, the contents are: **{Holder<String> status, Holder<String> statusMessage }**

Example: processing getCustomerStatus

The **GetCustomerStatusProcessor** class is responsible for processing incoming **getCustomerStatus** invocations. The following sample implementation for POJO mode shows how to read the request parameters from the *In* message body and then set the response parameters in the *Out* message body.

```Java
// Java
package com.fusesource.customerwscamelcxfpojo;

import javax.xml.ws.Holder;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;


public class GetCustomerStatusProcessor implements Processor {
    public static final Logger log = LoggerFactory.getLogger(GetCustomerStatusProcessor.class);

    public void process(Exchange exchng) throws Exception {
        Object[] args =  exchng.getIn().getBody(Object[].class);

        String id = (String) args[0];
        Holder<String> status = (Holder<String>) args[1];
        Holder<String> statusMsg = (Holder<String>) args[2];

        log.debug("Getting status for customer '" + id + "'");

        // This is where you'd actually do the work! Setting
        // the holder values to constants for the sake of brevity.
        //
        status.value = "Offline";
        statusMsg.value = "Going to sleep now!";

        exchng.getOut().setBody(new Object[] {status , statusMsg});
    }

}
```

38.6. DEPLOY TO OSGI

Overview

One of the options for deploying the POJO-based route is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).

- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **pom.xml** file:

1. Change the packaging type to **bundle** (by editing the value of the **project/packaging** element in the POM).

2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```xml
<?xml version="1.0"?>
<project ...>
  ...
  <groupId>com.fusesource.byexample.cxf-webinars</groupId>
  <artifactId>customer-ws-camel-cxf-pojo</artifactId>

  <name>customer-ws-camel-cxf-pojo</name>
  <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
         <instructions>
           <Import-Package>
             *
           </Import-Package>
           <DynamicImport-Package>
             org.apache.cxf.*,
             org.springframework.beans.*
           </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **DynamicImport-Package** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **DynamicImport-Package** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

NOTE

In general, using **DynamicImport-Package** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **Import-Package** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **Import-Package** header. With dynamic imports, this version checking cannot be performed.

Build and deploy the POJO route bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the route bundle, enter the following command at the JBoss Fuse console:

```
karaf@root> install -s mvn:com.fusesource.byexample.cxf-webinars/customer-ws-camel-cxf-pojo/1.0-
SNAPSHOT
```

NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the *EsbInstallDir*/etc/org.ops4j.pax.url.mvn.cfg file, before you can use the **mvn:** scheme to access Maven artifacts.

CHAPTER 39. PAYLOAD-BASED ROUTE

39.1. PROCESSING MESSAGES IN PAYLOAD FORMAT

Overview

Select the PAYLOAD format, if you want to access the SOAP message body in XML format, encoded as a DOM object (that is, of **org.w3c.dom.Node** type). One of the advantages of the PAYLOAD format is that no JAX-WS and JAXB stub code is required, which allows your application to be dynamic, potentially handling many different WSDL interfaces.

Having a message body in XML format enables you to parse the request using XML languages such as XPath and to generate responses using templating languages, such as Velocity.

> **NOTE**
>
> The DOM format is not the optimal type to use for *large* XML message bodies. For large messages, consider using the techniques described in Chapter 40, *Provider-Based Route*.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

> cxf-webinars-jboss-fuse-6.3/**customer-ws-camel-cxf-payload**

For details of how to download and install the demonstration code, see Chapter 34, *Demonstration Code for Camel/CXF*

Camel CXF component

The Camel CXF component is an Apache CXF component that integrates Web services with routes. You can use it either to instantiate *consumer endpoints* (at the start of a route), which behave like Web service instances, or to instantiate *producer endpoints* (at any other points in the route), which behave like WS clients.

> **NOTE**
>
> Came CXF endpoints—which are instantiated using the **cxf:cxfEndpoint** XML element and are implemented by the Apache Camel project—are not to be confused with the Apache CXF JAX-WS endpoints—which are instantiated using the **jaxws:endpoint** XML element and are implemented by the Apache CXF project.

PAYLOAD data format

The PAYLOAD data format is selected by setting the **dataFormat=PAYLOAD** option on a Camel CXF endpoint URI and it has the following characteristics:

- Enables you to access the message body as a DOM object (XML payload).

- No JAX-WS or JAXB stub code required.

- The SOAP body is marshalled as follows:

- The message body is effectively an XML payload of **org.w3c.dom.Node** type (wrapped in a **CxfPayload** object).

- The type of the message body is **org.apache.camel.component.cxf.CxfPayload**.

- The SOAP headers are converted into headers in the exchange's *In* message, of **org.apache.cxf.binding.soap.SoapHeader** type.

Implementing and building a PAYLOAD route

To implement and build the demonstration PAYLOAD-based route, starting from scratch, you would perform the following steps:

1. Instantiate the Camel CXF endpoint in Spring, using the **cxf:cxfEndpoint** element.

2. Implement the route in XML, where you can use the content-based router to sort requests by operation name.

3. For each operation, define a processor bean to process the request.

4. Define velocity templates for generating the reponse messages.

Sample PAYLOAD route

Figure 39.1, "Sample PAYLOAD Route" shows an outline of the route that is used to process the operations of the **CustomerService** Web service using the PAYLOAD data format. After sorting the request messages by operation name, an operation-specific processor bean reads the incoming request parameters. Finally, the response messages are generated using Velocity templates.

Figure 39.1. Sample PAYLOAD Route
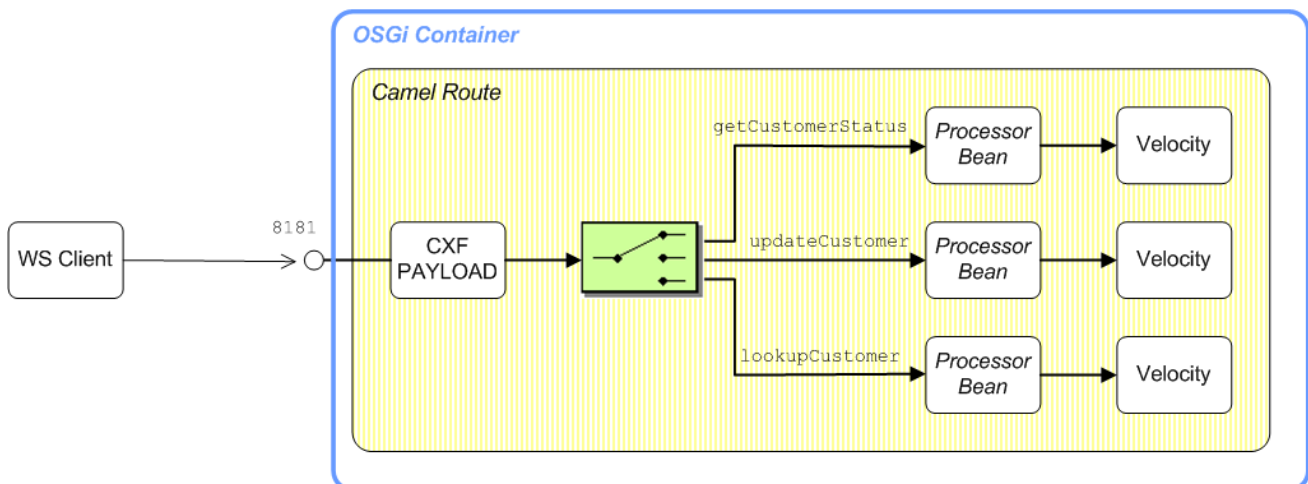
39.2. STREAM CACHING

In Apache Camel, different stream types like StreamSource, InputStream and Reader are commonly used in messaging for performance reasons and can be read only once. From Apache Camel 2.16 release, you can explicitly enable stream caching by invoking the *StreamCaching()* DSL method:

```
from("jbi:service:http://foo.bar.org/MyService")
    .streamCaching()
    .to("jbi:service:http://foo.bar.org/MyOtherService");
```

In Spring XML, you can set the attribute **streamCache= 'true'** on the route tag.

```
<route streamCache="true">
  <from uri="jbi:service:http://foo.bar.org/MyService"/>
  <to uri="jbi:service:http://foo.bar.org/MyOtherService"/>
</route>
```

39.3. INSTANTIATE THE WS ENDPOINT

Overview

In Apache Camel, the CXF component is the key to integrating routes with Web services. You can use the CXF component to create two different kinds of endpoint:

- *Consumer endpoint*—(at the start of a route) represents a Web service instance, which integrates with the route. The type of payload injected into the route depends on the value of the endpoint's **dataFormat** option.

- *Producer endpoint*—represents a special kind of WS client proxy, which converts the current exchange object into an operation invocation on a remote Web service. The format of the current exchange must match the endpoint's **dataFormat** setting.

The cxf:bean: URI syntax

The **cxf:bean:** URI is used to bind an Apache CXF endpoint to a route and has the following general syntax:

```
cxf:bean:CxfEndpointID[?Options]
```

Where **CxfEndpointID** is the ID of a bean created using the **cxf:cxfEndpoint** element, which configures the details of the WS endpoint. You can append options to this URI (where the options are described in detail in *CXF* in the *Apache Camel Component Reference Guide*). To enable payload mode, you must set the URI option, **dataFormat=PAYLOAD**.

For example, to start a route with an endpoint in PAYLOAD mode, where the endpoint is configured by the **customer-ws** bean, define the route as follows:

```
<route>
  <from uri="cxf:bean:customer-ws?dataFormat=PAYLOAD"/>
  ...
</route>
```

The cxf:cxfEndpoint element

The **cxf:cxfEndpoint** element is used to define a WS endpoint that binds either to the start (consumer endpoint) or the end (producer endpoint) of a route. For example, to define the **customer-ws** WS endpoint in PAYLOAD mode, you define a **cxf:cxfEndpoint** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <cxf:cxfEndpoint id="customer-ws"
    address="/Customer"
```

```
    endpointName="c:SOAPOverHTTP"
    serviceName="c:CustomerService"
    wsdlURL="wsdl/CustomerService.wsdl"
    xmlns:c="http://demo.fusesource.com/wsdl/CustomerService/"/>
  ...
</beans>
```

> **NOTE**
>
> In the case of PAYLOAD mode, you do *not* need to reference the SEI and you *must* specify the WSDL location instead. In fact, in PAYLOAD mode, you do not require any Java stub code at all.

Address for the Jetty container

Apache CXF deploys the WS endpoint into a Jetty servlet container instance and the **address** attribute of **cxf:cxfEndpoint** is therefore used to configure the addressing information for the endpoint in the Jetty container.

Apache CXF supports the notion of a *default servlet container* instance. The way the default servlet container is initialized and configured depends on the particular mode of deployment that you choose. For example the OSGi container and Web containers (such as Tomcat) provide a default servlet container.

There are two different syntaxes you can use for the endpoint address, where the syntax that you use effectively determines whether or not the endpoint is deployed into the default servlet container, as follows:

- *Address syntax for default servlet container* —to use the default servlet container, specify only the servlet context for this endpoint. Do *not* specify the protocol, host, and IP port in the address. For example, to deploy the endpoint to the /**Customer** servlet context in the default servlet container:

  ```
  address="/Customer"
  ```

- *Address syntax for custom servlet container* —to instantiate a custom Jetty container for this endpoint, specify a complete HTTP URL, including the host and IP port (the value of the IP port effectively identifies the target Jetty container). Typically, for a Jetty container, you specify the host as **0.0.0.0**, which is interpreted as a wildcard that matches every IP network interface on the local machine (that is, if deployed on a multi-homed host, Jetty opens a listening port on every network card). For example, to deploy the endpoint to the custom Jetty container listening on IP port, **8083**:

  ```
  address="http://0.0.0.0:8083/Customer"
  ```

  > **NOTE**
  >
  > If you want to configure a secure endpoint (secured by SSL), you would specify the **https:** scheme in the address.

Specifying the WSDL location

The **wsdlURL** attribute of the **cxf:cxfEndpoint** element is used to specify the location of the WSDL contract for this endpoint. The WSDL contract is used exclusively as the source of metadata for this endpoint: there is need to specify an SEI in PAYLOAD mode.

39.4. SORT MESSAGES BY OPERATION NAME

The operationName header

When the WS endpoint parses an incoming operation invocation in PAYLOAD mode, it automatically sets the **operationName** header to the name of the invoked operation. You can then use this header to sort messages by operation name.

Sorting by operation name

For example, the **customer-ws-camel-cxf-payload** demonstration defines the following route, which uses the content-based router pattern to sort incoming messages, based on the operation name. The **when** predicates check the value of the **operationName** header using simple language expressions, sorting messages into invocations on the **updateCustomer** operation, the **lookupCustomer** operation, or the **getCustomerStatus** operation.

```
<beans ...>
  ...
  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="cxf:bean:customer-ws?dataFormat=PAYLOAD"/>
      <choice>
        <when>
          <simple>${in.header.operationName} == 'updateCustomer'</simple>
          ...
        </when>
        <when>
          <simple>${in.header.operationName} == 'lookupCustomer'</simple>
          ...
        </when>
        <when>
          <simple>${in.header.operationName} == 'getCustomerStatus'</simple>
          ...
        </when>
      </choice>
    </route>
  </camelContext>

</beans>
```

39.5. SOAP/HTTP-TO-JMS BRIDGE USE CASE

Overview

In this section, we consider a SOAP/HTTP-to-JMS bridge use case: that is, you want to create a route that transforms a synchronous operation invocation (over SOAP/HTTP) into an asynchronous message delivery (by pushing the message onto a JMS queue). In this way, it becomes possible to process the incoming operation invocations at a later time, by pulling messages off the JMS queue.

Of course, an alternative solution would be to modify the WSDL contract directly to declare the operation as *OneWay*, thus making the operation asynchronous. Unfortunately, it is often impractical to modify existing WSDL contracts in the real world, because this can have an impact on third-party applications.

Figure 39.2, "SOAP/HTTP-to-JMS Bridge" shows the general outline of a bridge that can transform synchronous SOAP/HTTP invocations into asynchronous JMS message deliveries.

Figure 39.2. SOAP/HTTP-to-JMS Bridge
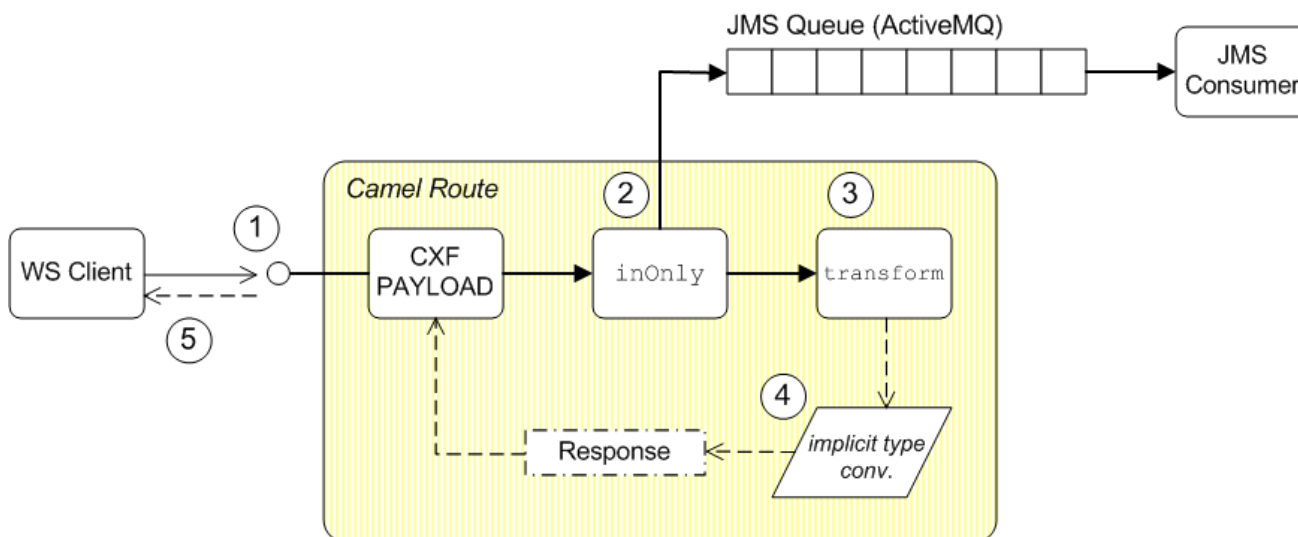
Transforming RPC operations to One Way

As shown in Figure 39.2, "SOAP/HTTP-to-JMS Bridge", the route for transforming synchronous SOAP/HTTP to asynchronous JMS works as follows:

1. The WS client invokes a synchronous operation on the Camel CXF endpoint at the start of the route. The Camel CXF endpoint then creates an initial *InOut* exchange at the start of the route, where the body of the exchange message contains a payload in XML format.

2. The **inOnly** DSL command pushes a copy of the XML payload onto a JMS queue, so that it can be processed offline at some later time.

3. The **transform** DSL command constructs an immediate response to send back to the client, where the response has the form of an XML string.

4. The Camel CXF component supports implicit type conversion of the XML string to payload format.

5. The response is sent back to the WS client, thus completing the synchronous operation invocation.

Evidently, this transformation can only work, if the original operation invocation has no return value. Otherwise, it would be impossible to generate a response message before the request has been processed.

Creating a broker instance

You can use Apache ActiveMQ as the JMS implementation. A convenient approach to use in this demonstration is to embed the Apache ActiveMQ broker in the bridge bundle. Simply define an **amq:broker** element in the Spring XML file, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    ...
    xmlns:amq="http://activemq.apache.org/schema/core"
    ...>

  <amq:broker brokerName="CxfPayloadDemo" persistent="false">
    <amq:transportConnectors>
      <amq:transportConnector name="openwire" uri="tcp://localhost:51616"/>
      <amq:transportConnector name="vm" uri="vm:local"/>
    </amq:transportConnectors>
  </amq:broker>
  ...
</beans>
```

> **NOTE**
>
> This broker instance is created with the **persistent** attribute set to **false**, so that the messages are stored *only* in memory.

Configuring the JMS component

Because the broker is co-located with the bridge route (in the same JVM), the most efficient way to connect to the broker is to use the VM (Virtual Machine) transport. Configure the Apache ActiveMQ component as follows, to connect to the co-located broker using the VM protocol:

```
<beans ...>
    ...
    <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
      <property name="brokerURL" value="vm:local"/>
    </bean>
    ...
</beans>
```

> **NOTE**
>
> By defining the bean with an **id** value of **activemq**, you are implicitly overriding the component associated with the endpoint URI prefix, **activemq:**. In other words, your custom **ActiveMQComponent** instance is used instead of the default **ActiveMQComponent** instance from the **camel-activemq** JAR file.

Sample SOAP/HTTP-to-JMS route

For example, you could define a route that implements the SOAP/HTTP-to-JMS bridge specifically for the **updateCustomer** operation from the **CustomerService** SEI, as follows:

```
<when>
    <simple>${in.header.operationName} == 'updateCustomer'</simple>
    <log message="Placing update customer message onto queue."/>
    <inOnly uri="activemq:queue:CustomerUpdates?jmsMessageType=Text"/>
    <transform>
      <constant>
      <![CDATA[
<ns2:updateCustomerResponse xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/"/>
```

```
        ]]>
      </constant>
    </transform>
  </when>
```

Sending to the JMS endpoint in inOnly mode

Note how the message payload is sent to the JMS queue using the **inOnly** DSL command instead of the **to** DSL command. When you send a message using the **to** DSL command, the default behavior is to use the same invocation mode as the current exchange. But the current exchange has an *InOut* MEP, which means that the **to** DSL command would wait forever for a response message from JMS.

The invocation mode we want to use when sending the payload to the JMS queue is *InOnly* (asynchronous), and we can force this mode by inserting the **inOnly** DSL command into the route.

> **NOTE**
>
> By specifying the option, **jmsMessageType=Text**, Camel CXF implicitly converts the message payload to an XML string before pushing it onto the JMS queue.

Returning a literal response value

The **transform** DSL command uses an expression to set the body of the exchange's *Out* message and this message is then used as the response to the client. Your first impulse when defining a response in XML format might be to use a DOM API, but in this example, the response is specified as a string literal. This approach has the advantage of being both efficient and very easy to program.

The final step of processing, which consists of converting the XML string to a DOM object, is performed by Apache Camel's implicit type conversion mechanism.

39.6. GENERATING RESPONSES USING TEMPLATES

Overview

One of the simplest and quickest approaches to generating a response message is to use a velocity template. Figure 39.3, "Response Generated by Velocity" shows the outline of a general template-based route. At the start of the route is a Camel CXF endpoint in PAYLOAD mode, which is the appropriate mode to use for processing the message as an XML document. After doing the work required to process the message and stashing some intermediate results in message headers, the route generates the response message using a Velocity template.

Figure 39.3. Response Generated by Velocity
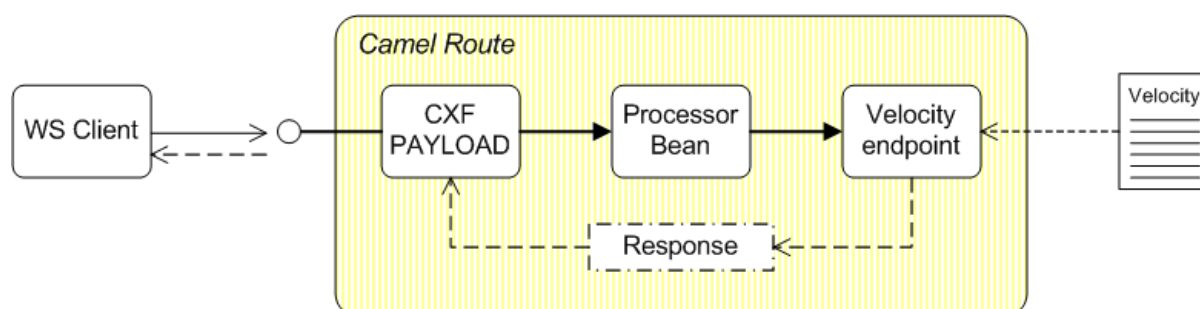
Sample template-based route

For example, you could define a template-based route specifically for the **getCustoemrStatus** operation, as follows:

```
...
  <when>
    <simple>${in.header.operationName} == 'getCustomerStatus'</simple>
    <convertBodyTo type="org.w3c.dom.Node"/>
    <setHeader headerName="customerId">
      <xpath>/cus:getCustomerStatus/customerId/text()</xpath>
    </setHeader>
    <to uri="getCustomerStatus"/>
    <to uri="velocity:getCustomerStatusResponse.vm"/>
  </when>
 </choice>
</route>
</camelContext>
...
<bean id="getCustomerStatus"
  class="com.fusesource.customerwscamelcxfpayload.GetCustomerStatus"/>
```

Route processing steps

Given the preceding route definition, any message whose operation name matches **getCustomerStatus** would be processed as follows:

1. To facilitate processing the payload body, the first step uses **convertBodyTo** to convert the body type from **org.apache.camel.component.cxf.CxfPayload** (the default payload type) to **org.w3c.dom.Node**.

2. The route then applies an XPath expression to the message in order to extract the customer ID value and then stashes it in the **customerId** header.

3. The next step sends the message to the **getCustomerStatus** bean, which does whatever processing is required to get the customer status for the specified customer ID. The results from this step are stashed in message headers.

4. Finally, a response is generated using a velocity template.

> **NOTE**
>
> A common pattern when implementing Apache Camel routes is to use message headers as a temporary stash to hold intermediate results (you could also use exchange properties in the same way).

Converting XPath result to a string

Because the default return type of XPath is a node list, you must explicitly convert the result to a string, if you want to obtain the string contents of an element. There are two alternative ways of obtaining the string value of an element:

- Specify the result type explicitly using the **resultType** attribute, as follows:

```
<xpath resultType="java.lang.String">/cus:getCustomerStatus/customerId</xpath>
```

- Modify the expression so that it returns a **text()** node, which automatically converts to string:

```
<xpath>/cus:getCustomerStatus/customerId/text()</xpath>
```

getCustomerStatus processor bean

The **getCustomerStatus** processor bean is an instance of the **GetCustomerStatus** processor class, which is defined as follows:

```java
// Java
package com.fusesource.customerwscamelcxfpayload;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class GetCustomerStatus implements Processor
{
    public void process(Exchange exchng) throws Exception {
        String id = exchng.getIn().getHeader("customerId", String.class);

        // Maybe do some kind of lookup here!
        //

        exchng.getIn().setHeader("status", "Away");
        exchng.getIn().setHeader("statusMessage", "Going to sleep.");
    }
}
```

The implementation shown here is just a placeholder. In a realistic application you would perform some sort of checks or database lookup to obtain the customer status. In the demonstration code, however, the **status** and **statusMessage** are simply set to constant values and stashed in message headers.

In the preceding code, we make the modifications directly to the *In* message. When the exchange's *Out* message is **null**, the next processor in the route gets a copy of the current *In* message instead

> **NOTE**
>
> An exceptional case occurs when the message exchange pattern is *inOnly*, in which case the *Out* message value is *always* copied into the In message, even if it is **null**.

getCustomerStatusResponse.vm Velocity template

You can generate a response message very simply using a Velocity template. The Velocity template consists of a message in plain text, where specific pieces of data can be inserted using expressions—for example, the expression **${header.*HeaderName*}** substitutes the value of a named header.

The Velocity template for generating the **getCustomerStatus** reponse is located in the **customer-ws-camel-cxf-payload/src/main/resources** directory and it contains the following template script:

```xml
<ns2:getCustomerStatusResponse
xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/">
  <status>${headers.status}</status>
  <statusMessage>${headers.statusMessage}</statusMessage>
</ns2:getCustomerStatusResponse>
```

39.7. TYPECONVERTER FOR CXFPAYLOAD

Overview

Apache Camel supports a type converter mechanism, which is used to perform implicit and explicit type conversions of message bodies and message headers. The payload demonstration requires a customer type converter that can convert **String** objects to **CXFPayload** objects. This type converter automatically gets invoked at the end of the Camel route, when the generated response message (which is a **String** type) gets converted into a **CXFPayload** object.

String to CXFPayload type converter

The **String** to **CXFPayload** type converter is implemented in the **AdditionalCxfPayloadConverters** class, as follows:

```java
// Java
package com.fusesource.customerwscamelcxfpayload;

import java.io.ByteArrayInputStream;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.List;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.apache.camel.Converter;
import org.apache.camel.component.cxf.CxfPayload;
import org.apache.cxf.binding.soap.SoapHeader;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

@Converter
public class AdditionalCxfPayloadConverters {
    ...
    @Converter
    public static CxfPayload<SoapHeader> toCxfPayload(String xml) {
        // System.out.println("To CxfPayload " + xml);
        List<Element> elements = new ArrayList<Element>();
        try {
            Document doc = b.newDocumentBuilder().parse(new ByteArrayInputStream(xml.getBytes()));
            elements.add(doc.getDocumentElement());
        } catch (Exception ex) {
            log.warn("Exception while converting String payload to CxfPayload; resulting payload will be empty.");
        }
        // The CxfPayload is changed to use Source object under layer, the elements API only work if we already setup the list before creating the CxfPayload
        CxfPayload<SoapHeader> ret = new CxfPayload<SoapHeader>(null, elements);
```

```
        return ret;
    }
    ...
}
```

Reference

For full details of the type converter mechanism in Apache Camel, see Section 43.3, "Built-In Type Converters" and Chapter 45, *Type Converters*.

39.8. DEPLOY TO OSGI

Overview

One of the options for deploying the payload-based route is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).

- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **pom.xml** file:

1. Change the packaging type to **bundle** (by editing the value of the **project/packaging** element in the POM).

2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```xml
<?xml version="1.0"?>
<project ...>
  ...
  <groupId>com.fusesource.byexample.cxf-webinars</groupId>
  <artifactId>customer-ws-camel-cxf-payload</artifactId>
  <name>customer-ws-camel-cxf-payload</name>
 <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
```

```
    ...
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <extensions>true</extensions>
      <configuration>
       <instructions>
         <Import-Package>
           org.apache.camel.component.velocity,
           javax.jws,
           javax.wsdl,
           javax.xml.bind,
           javax.xml.bind.annotation,
           javax.xml.namespace,
           javax.xml.ws,
           org.w3c.dom,
           *
         </Import-Package>
         <DynamicImport-Package>
           org.apache.cxf.*,
           org.springframework.beans.*
         </DynamicImport-Package>
       </instructions>
      </configuration>
    </plugin>
    ...
    </plugins>
   </build>
 </project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **DynamicImport-Package** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **DynamicImport-Package** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

> **NOTE**
>
> In general, using **DynamicImport-Package** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **Import-Package** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **Import-Package** header. With dynamic imports, this version checking cannot be performed.

Build and deploy the client bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

Install the **camel-velocity** feature, which is needed for this example:

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

To deploy the route bundle, enter the following command at the console:

```
karaf@root> install -s mvn:com.fusesource.byexample.cxf-webinars/customer-ws-camel-cxf-
payload/1.0-SNAPSHOT
```

NOTE

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the *InstallDir*/**etc/org.ops4j.pax.url.mvn.cfg** file, before you can use the **mvn:** scheme to access Maven artifacts.

CHAPTER 40. PROVIDER-BASED ROUTE

40.1. PROVIDER-BASED JAX-WS ENDPOINT

Overview

Use the provider-based approach, if you need to process *very* large Web services messages. The provider-based approach is a variant of the PAYLOAD data format that enables you to encode the message body as an XML streaming type, such as **SAXSource**. Since the XMLstreaming types are more efficient than DOM objects, the provider-based approach is ideal for large XML messages.

Demonstration location

The code presented in this chapter is taken from the following demonstration:

> cxf-webinars-jboss-fuse-6.3/**customer-ws-camel-cxf-provider**

For details of how to download and install the demonstration code, see Chapter 34, *Demonstration Code for Camel/CXF*

Camel CXF component

The Camel CXF component is an Apache CXF component that integrates Web services with routes. You can use it either to instantiate *consumer endpoints* (at the start of a route), which behave like Web service instances, or to instantiate *producer endpoints* (at any other points in the route), which behave like WS clients.

> **NOTE**
>
> Came CXF endpoints—which are instantiated using the **cxf:cxfEndpoint** XML element and are implemented by the Apache Camel project—are not to be confused with the Apache CXF JAX-WS endpoints—which are instantiated using the **jaxws:endpoint** XML element and are implemented by the Apache CXF project.

Provider-based approach and the PAYLOAD data format

The provider-based approach is a variant of the PAYLOAD data format, which is enabled as follows:

- Define a custom **javax.xml.ws.Provider<*StreamType*>** class, where the *StreamType* type is an XML streaming type, such as **SAXSource**.

- The PAYLOAD data format is selected by an annotation on the custom **Provider<?>** class (see the section called "The SAXSourceService provider class").

- The custom **Provider<?>** class is referenced by setting the **serviceClass** attribute of the **cxf:cxfEndpoint** element in XML configuration.

The provider-based approach has the following characteristics:

- Enables you to access the message body as a streamed XML type—for example, **javax.xml.transform.sax.SAXSource**.

- No JAX-WS or JAXB stub code required.

- The SOAP body is marshalled into a stream-based **SAXSource** type.

- The SOAP headers are converted into headers in the exchange's *In* message, of **org.apache.cxf.binding.soap.SoapHeader** type.

Implementing and building a provider-based route

To implement and build the demonstration provider-based route, starting from scratch, you would perform the following steps:

1. Define a custom **javax.xml.ws.Provider<*StreamType*>** class (the current demonstration uses **SAXSource** as the *StreamType* type).

2. Instantiate the Camel CXF endpoint in Spring, using the **cxf:cxfEndpoint** element and reference the custom provider class (using the **serviceClass** attribute).

3. Implement the route in XML, where you can use the content-based router to sort requests by operation name.

4. For each operation, define a processor bean to process the request.

5. Define velocity templates for generating the response messages.

6. Define a custom type converter, to support converting a **String** message body to a **SAXSource** message body.

Sample provider-based route

Figure 40.1, "Sample Provider-Based Route" shows an outline of the route that is used to process the operations of the **CustomerService** Web service using the provider-based approach. After sorting the request messages by operation name, an operation-specific processor bean reads the incoming request parameters. Finally, the response messages are generated using Velocity templates.

Figure 40.1. Sample Provider-Based Route
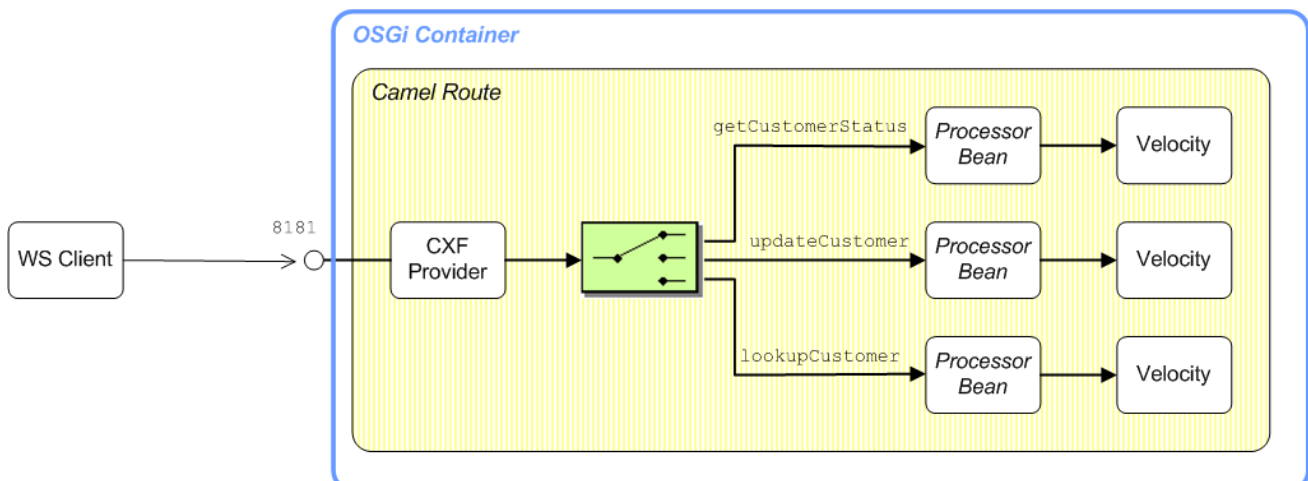
40.2. CREATE A PROVIDER<?> IMPLEMENTATION CLASS

Overview

The fundamental prerequisite for using provider mode is to define a custom **Provider<>** class that implements the **invoke()** method. In fact, the sole purpose of this class is to provide runtime type information for Apache CXF: *the **invoke()** method never gets called!*

By implementing the provider class in the way shown here, you are merely indicating to the Apache CXF runtime that the WS endpoint should operate in in PAYLOAD mode and the type of the message PAYLOAD should be **SAXSource**.

The SAXSourceService provider class

The definition of the provider class is relatively short and the complete definition of the customer provider class, **SAXSourceService**, is as follows:

```java
// Java
package com.fusesource.customerwscamelcxfprovider;

import javax.xml.transform.sax.SAXSource;
import javax.xml.ws.Provider;
import javax.xml.ws.Service.Mode;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider()
@ServiceMode(Mode.PAYLOAD)
public class SAXSourceService implements Provider<SAXSource>
{
  public SAXSource invoke(SAXSource t) {
    throw new UnsupportedOperationException("Not supported yet.");
  }
}
```

The customer provider class, **SAXSourceService**, must be annotated by the **@WebServiceProvider** annotation to mark it as a provider class and can be optionally annotated by the **@ServiceMode** annotation to select PAYLOAD mode.

40.3. INSTANTIATE THE WS ENDPOINT

Overview

In Apache Camel, the CXF component is the key to integrating routes with Web services. You can use the CXF component to create two different kinds of endpoint:

- *Consumer endpoint*—(at the start of a route) represents a Web service instance, which integrates with the route. The type of payload injected into the route depends on the value of the endpoint's **dataFormat** option.

- *Producer endpoint*—represents a special kind of WS client proxy, which converts the current exchange object into an operation invocation on a remote Web service. The format of the current exchange must match the endpoint's **dataFormat** setting.

The cxf:bean: URI syntax

The **cxf:bean:** URI is used to bind an Apache CXF endpoint to a route and has the following general syntax:

> cxf:bean:*CxfEndpointID[?Options]*

Where ***CxfEndpointID*** is the ID of a bean created using the **cxf:cxfEndpoint** element, which configures the details of the WS endpoint. You can append options to this URI (where the options are described in detail in *CXF* in the *Apache Camel Component Reference Guide*). Provider mode is essentially a variant of PAYLOAD mode: you could specify this mode on the URI (by setting **dataFormat=PAYLOAD**), but this is not necessary, because PAYLOAD mode is already selected by the **@ServiceMode** annotation on the custom **Provider** class.

For example, to start a route with an endpoint in provider mode, where the endpoint is configured by the **customer-ws** bean, define the route as follows:

```
<route>
    <from uri="cxf:bean:customer-ws"/>
    ...
</route>
```

The cxf:cxfEndpoint element

The **cxf:cxfEndpoint** element is used to define a WS endpoint that binds either to the start (consumer endpoint) or the end (producer endpoint) of a route. For example, to define the **customer-ws** WS endpoint in provider mode, you define a **cxf:cxfEndpoint** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    ...
    <cxf:cxfEndpoint id="customer-ws"
        address="/Customer"
        endpointName="c:SOAPOverHTTP"
        serviceName="c:CustomerService"
        wsdlURL="wsdl/CustomerService.wsdl"
        serviceClass="com.fusesource.customerwscamelcxfprovider.SAXSourceService"
        xmlns:c="http://demo.fusesource.com/wsdl/CustomerService/"/>
    ...
</beans>
```

Specifying the WSDL location

The **wsdlURL** attribute of the **cxf:cxfEndpoint** element is used to specify the location of the WSDL contract for this endpoint. The WSDL contract is used as the source of metadata for this endpoint.

Specifying the service class

A key difference between provider mode and ordinary PAYLOAD mode is that the **serviceClass** attribute must be set to the provider class, **SAXSourceService**.

40.4. SORT MESSAGES BY OPERATION NAME

The operationName header

When the WS endpoint parses an incoming operation invocation in PROVIDER mode, it automatically sets the **operationName** header to the name of the invoked operation. You can then use this header to sort messages by operation name.

Sorting by operation name

For example, the **customer-ws-camel-cxf-provider** demonstration defines the following route, which uses the content-based router pattern to sort incoming messages, based on the operation name. The **when** predicates check the value of the **operationName** header using simple language expressions, sorting messages into invocations on the **updateCustomer** operation, the **lookupCustomer** operation, or the **getCustomerStatus** operation.

```
<beans ...>
    ...
    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxf:bean:customer-ws"/>
            <choice>
                <when>
                    <simple>${in.header.operationName} == 'updateCustomer'</simple>
                    ...
                </when>
                <when>
                    <simple>${in.header.operationName} == 'lookupCustomer'</simple>
                    ...
                </when>
                <when>
                    <simple>${in.header.operationName} == 'getCustomerStatus'</simple>
                    ...
                </when>
            </choice>
        </route>
    </camelContext>
    ...
</beans>
```

40.5. SOAP/HTTP-TO-JMS BRIDGE USE CASE

Overview

In this section, we consider a SOAP/HTTP-to-JMS bridge use case: that is, you want to create a route that transforms a synchronous operation invocation (over SOAP/HTTP) into an asynchronous message delivery (by pushing the message onto a JMS queue). In this way, it becomes possible to process the incoming operation invocations at a later time, by pulling messages off the JMS queue.

Figure 40.2, "SOAP/HTTP-to-JMS Bridge" shows the general outline of a bridge that can transform synchronous SOAP/HTTP invocations into asynchronous JMS message deliveries.

Figure 40.2. SOAP/HTTP-to-JMS Bridge
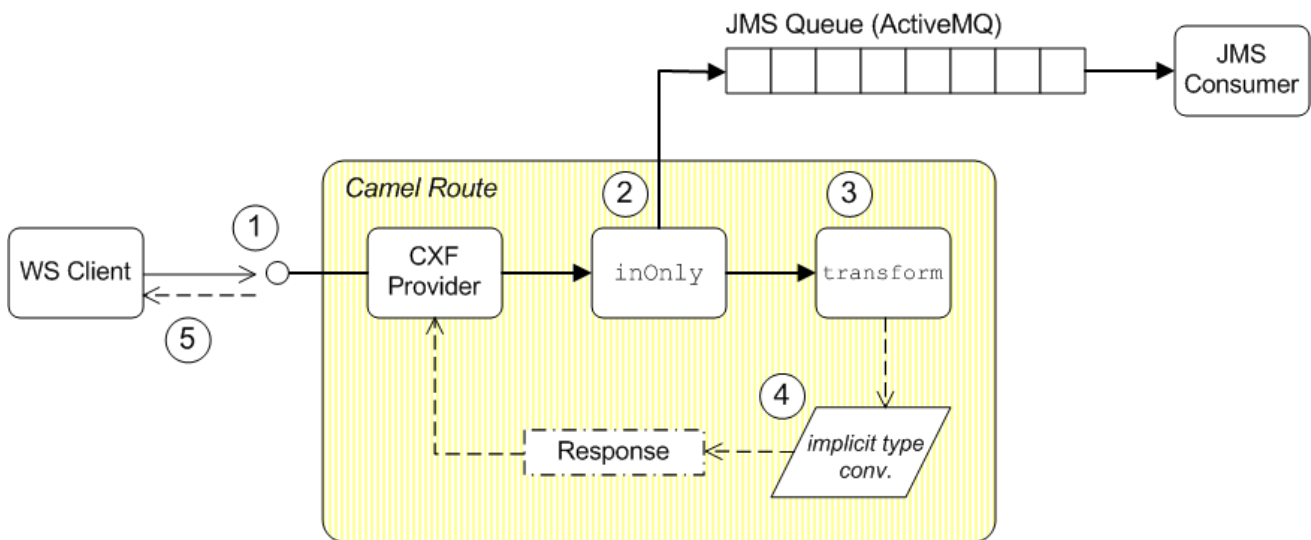
Transforming RPC operations to One Way

As shown in Figure 40.2, "SOAP/HTTP-to-JMS Bridge", the route for transforming synchronous SOAP/HTTP to asynchronous JMS works as follows:

1. The WS client invokes a synchronous operation on the Camel CXF endpoint at the start of the route. The Camel CXF endpoint then creates an initial *InOut* exchange at the start of the route, where the body of the exchange message contains a payload in XML format.

2. The **inOnly** DSL command pushes a copy of the XML payload onto a JMS queue, so that it can be processed offline at some later time.

3. The **transform** DSL command constructs an immediate response to send back to the client, where the response has the form of an XML string.

4. The route explicitly converts the XML string to the **javax.xml.transform.sax.SAXSource** type.

5. The response is sent back to the WS client, thus completing the synchronous operation invocation.

Evidently, this transformation can only work, if the original operation invocation has no return value. Otherwise, it would be impossible to generate a response message before the request has been processed.

Creating a broker instance

You can use Apache ActiveMQ as the JMS implementation. A convenient approach to use in this demonstration is to embed the Apache ActiveMQ broker in the bridge bundle. Simply define an **amq:broker** element in the Spring XML file, as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    ...
    xmlns:amq="http://activemq.apache.org/schema/core"
    ...>

<amq:broker brokerName="CxfPayloadDemo" persistent="false">
    <amq:transportConnectors>
        <amq:transportConnector name="openwire" uri="tcp://localhost:51616"/>
```

```
      <amq:transportConnector name="vm" uri="vm:local"/>
    </amq:transportConnectors>
  </amq:broker>
  ...
</beans>
```

NOTE

This broker instance is created with the **persistent** attribute set to **false**, so that the messages are stored *only* in memory.

Configuring the JMS component

Because the broker is co-located with the bridge route (in the same JVM), the most efficient way to connect to the broker is to use the VM (Virtual Machine) transport. Configure the Apache ActiveMQ component as follows, to connect to the co-located broker using the VM protocol:

```
<beans ...>
   ...
   <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
      <property name="brokerURL" value="vm:local"/>
   </bean>
   ...
</beans>
```

NOTE

By defining the bean with an **id** value of **activemq**, you are implicitly overriding the component associated with the endpoint URI prefix, **activemq:**. In other words, your custom **ActiveMQComponent** instance is used instead of the default **ActiveMQComponent** instance from the **camel-activemq** JAR file.

Sample SOAP/HTTP-to-JMS route

For example, you could define a route that implements the SOAP/HTTP-to-JMS bridge specifically for the **updateCustomer** operation from the **CustomerService** SEI, as follows:

```
<when>
   <simple>${in.header.operationName} == 'updateCustomer'</simple>
   <log message="Placing update customer message onto queue."/>
   <inOnly uri="activemq:queue:CustomerUpdates?jmsMessageType=Text"/>
   <transform>
      <constant>
      <![CDATA[
<ns2:updateCustomerResponse xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/"/>
      ]]>
      </constant>
   </transform>
   <convertBodyTo type="javax.xml.transform.sax.SAXSource"/>
</when>
```

Sending to the JMS endpoint in inOnly mode

Note how the message payload is sent to the JMS queue using the **inOnly** DSL command instead of the **to** DSL command. When you send a message using the **to** DSL command, the default behavior is to use the same invocation mode as the current exchange. But the current exchange has an *InOut* MEP, which means that the **to** DSL command would wait forever for a response message from JMS.

The invocation mode we want to use when sending the payload to the JMS queue is *InOnly* (asynchronous), and we can force this mode by inserting the **inOnly** DSL command into the route.

> **NOTE**
>
> By specifying the option, **jmsMessageType=Text**, Camel CXF implicitly converts the message payload to an XML string before pushing it onto the JMS queue.

Returning a literal response value

The **transform** DSL command uses an expression to set the body of the exchange's *Out* message and this message is then used as the response to the client. Your first impulse when defining a response in XML format might be to use a DOM API, but in this example, the response is specified as a string literal. This approach has the advantage of being both efficient and very easy to program.

Type conversion of the response message

In this example, the reply message (like the request message) is required to be of type, **javax.xml.transform.sax.SAXSource**. In the last step of the route, therefore, you must convert the message body from **String** type to **javax.xml.transform.sax.SAXSource** type, by invoking the **convertBodyTo** DSL command.

The implementation of the String to **SAXSource** conversion is provided by a custom type converter, as described in Section 40.7, "TypeConverter for SAXSource".

40.6. GENERATING RESPONSES USING TEMPLATES

Overview

One of the simples and quickest approaches to generating a response message is to use a velocity template. Figure 40.3, "Response Generated by Velocity" shows the outline of a general template-based route. At the start of the route is a Camel CXF endpoint in provider mode, which is the appropriate mode to use for processing the message as an XML document. After doing the work required to process the message and stashing some intermediate results in message headers, the route generates the response message using a Velocity template.

Figure 40.3. Response Generated by Velocity
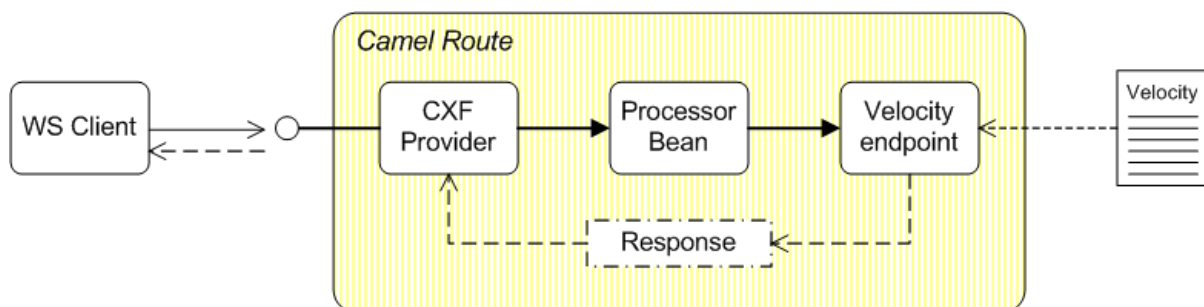
Sample template-based route

For example, you could define a template-based route specifically for the **getCustoemrStatus** operation, as follows:

```
    ...
    <when>
     <simple>${in.header.operationName} == 'getCustomerStatus'</simple>
     <setHeader headerName="customerId">
       <xpath resultType="java.lang.String">/cus:getCustomerStatus/customerId</xpath>
     </setHeader>
     <to uri="getCustomerStatus"/>
     <to uri="velocity:getCustomerStatusResponse.vm"/>
     <convertBodyTo type="javax.xml.transform.sax.SAXSource"/>
    </when>
   </choice>
  </route>
</camelContext>
...
<bean id="getCustomerStatus"
  class="com.fusesource.customerwscamelcxfpayload.GetCustomerStatus"/>
```

Route processing steps

Given the preceding route definition, any message whose operation name matches **getCustomerStatus** would be processed as follows:

1. The route applies an XPath expression to the message in order to extract the customer ID value and then stashes it in the **customerId** header.

2. The next step sends the message to the **getCustomerStatus** bean, which does whatever processing is required to get the customer status for the specified customer ID. The results from this step are stashed in message headers.

3. A response is generated using a Velocity template.

4. Finally, the XML string generated by the Velocity template must be explicitly converted to the **javax.xml.transform.sax.SAXSource** type using **convertBodyTo** (which implicitly relies on a type converter).

> **NOTE**
>
> A common pattern when implementing Apache Camel routes is to use message headers as a temporary stash to hold intermediate results (you could also use exchange properties in the same way).

XPath expressions and SAXSource

XPath expressions can be applied directly to SAXSource objects. The XPath implementation has a pluggable architecture that supports a variety of XML parsers and when XPath encounters a SAXSource object, it automatically loads the plug-in required to support SAXSource parsing.

getCustomerStatus processor bean

The **getCustomerStatus** processor bean is an instance of the **GetCustomerStatus** processor class, which is defined as follows:

```java
// Java
package com.fusesource.customerwscamelcxfpayload;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class GetCustomerStatus implements Processor
{
    public void process(Exchange exchng) throws Exception {
        String id = exchng.getIn().getHeader("customerId", String.class);

        // Maybe do some kind of lookup here!
        //

        exchng.getIn().setHeader("status", "Away");
        exchng.getIn().setHeader("statusMessage", "Going to sleep.");
    }
}
```

The implementation shown here is just a placeholder. In a realistic application you would perform some sort of checks or database lookup to obtain the customer status. In the demonstration code, however, the **status** and **statusMessage** are simply set to constant values and stashed in message headers.

getCustomerStatusResponse.vm Velocity template

You can generate a response message very simply using a Velocity template. The Velocity template consists of a message in plain text, where specific pieces of data can be inserted using expressions—for example, the expression **${header.*HeaderName*}** substitutes the value of a named header.

The Velocity template for generating the **getCustomerStatus** response is located in the **customer-ws-camel-cxf-provider/src/main/resources** directory and it contains the following template script:

```
<ns2:getCustomerStatusResponse
xmlns:ns2="http://demo.fusesource.com/wsdl/CustomerService/">
  <status>${headers.status}</status>
  <statusMessage>${headers.statusMessage}</statusMessage>
</ns2:getCustomerStatusResponse>
```

40.7. TYPECONVERTER FOR SAXSOURCE

Overview

Apache Camel supports a type converter mechanism, which is used to perform implicit and explicit type conversions of message bodies and message headers. The type converter mechanism is extensible and it so happens that the provider demonstration requires a custom type converter that can convert **String** objects to **SAXSource** objects.

String to SAXSource type converter

The **String** to **SAXSource** type converter is implemented in the **AdditionalConverters** class, as follows:

```java
// Java
package com.fusesource.customerwscamelcxfprovider;
```

```
import java.io.ByteArrayInputStream;
import javax.xml.transform.sax.SAXSource;
import org.apache.camel.Converter;
import org.xml.sax.InputSource;

@Converter
public class AdditionalConverters {

  @Converter
  public static SAXSource toSAXSource(String xml) {
      return new SAXSource(new InputSource(new ByteArrayInputStream(xml.getBytes())));
  }
}
```

Reference

For full details of the type converter mechanism in Apache Camel, see Section 43.3, "Built-In Type Converters" and Chapter 45, *Type Converters*.

40.8. DEPLOY TO OSGI

Overview

One of the options for deploying the provider-based route is to package it as an OSGi bundle and deploy it into an OSGi container such as Red Hat JBoss Fuse. Some of the advantages of an OSGi deployment include:

- Bundles are a relatively lightweight deployment option (because dependencies can be shared between deployed bundles).

- OSGi provides sophisticated dependency management, ensuring that only version-consistent dependencies are added to the bundle's classpath.

Using the Maven bundle plug-in

The Maven bundle plug-in is used to package your project as an OSGi bundle, in preparation for deployment into the OSGi container. There are two essential modifications to make to your project's **pom.xml** file:

1. Change the packaging type to **bundle** (by editing the value of the **project/packaging** element in the POM).

2. Add the Maven bundle plug-in to your POM file and configure it as appropriate.

Configuring the Maven bundle plug-in is quite a technical task (although the default settings are often adequate). For full details of how to customize the plug-in configuration, consult *Deploying into the OSGi Container* and *Managing OSGi Dependencies*.

Sample bundle plug-in configuration

The following POM fragment shows a sample configuration of the Maven bundle plug-in, which is appropriate for the current example.

```xml
<?xml version="1.0"?>
<project  ...>
  ...
  <groupId>com.fusesource.byexample.cxf-webinars</groupId>
  <artifactId>customer-ws-camel-cxf-provider</artifactId>
  <name>customer-ws-camel-cxf-provider</name>
  <packaging>bundle</packaging>
  ...
  <build>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
         <instructions>
            <Import-Package>
              org.apache.camel.component.velocity,
              javax.jws,
              javax.wsdl,
              javax.xml.bind,
              javax.xml.bind.annotation,
              javax.xml.namespace,
              javax.xml.ws,
              org.w3c.dom,
              *
            </Import-Package>
            <DynamicImport-Package>
              org.apache.cxf.*,
              org.springframework.beans.*
            </DynamicImport-Package>
          </instructions>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

Dynamic imports

The Java packages from Apache CXF and the Spring API are imported using dynamic imports (specified using the **DynamicImport-Package** element). This is a pragmatic way of dealing with the fact that Spring XML files are not terribly well integrated with the Maven bundle plug-in. At build time, the Maven bundle plug-in is *not* able to figure out which Java classes are required by the Spring XML code. By listing wildcarded package names in the **DynamicImport-Package** element, however, you allow the OSGi container to figure out which Java classes are needed by the Spring XML code at *run* time.

> **NOTE**
>
> In general, using **DynamicImport-Package** headers is not recommended in OSGi, because it short-circuits OSGi version checking. Normally, what should happen is that the Maven bundle plug-in lists the Java packages used at *build* time, along with their versions, in the **Import-Package** header. At *deploy* time, the OSGi container then checks that the available Java packages are compatible with the build time versions listed in the **Import-Package** header. With dynamic imports, this version checking cannot be performed.

Build and deploy the client bundle

After you have configured the POM file, you can build the Maven project and install it in your local repository by entering the following command:

```
mvn install
```

To deploy the route bundle, enter the following command at the container console:

```
karaf@root> install -s mvn:com.fusesource.byexample.cxf-webinars/customer-ws-camel-cxf-
provider/1.0-SNAPSHOT
```

> **NOTE**
>
> If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the *EsbInstallDir*/etc/org.ops4j.pax.url.mvn.cfg file, before you can use the **mvn:** scheme to access Maven artifacts.

CHAPTER 41. PROXYING A WEB SERVICE

Abstract

A common use case for the Camel CXF component is to use a route as a *proxy* for a Web service. That is, in order to perform additional processing of WS request and response messages, you interpose a route between the WS client and the original Web service.

41.1. PROXYING WITH HTTP

Overview

The simplest way to proxy a SOAP/HTTP Web service is to treat the request and reply messages as HTTP packets. This type of proxying can be used where there is no requirement to read or modify the messages passing through the route. For example, you could use this kind of proxying to apply various patterns of flow control on the WS messges.

Figure 41.1, "Proxy Route with Message in HTTP Format" shows an overview of how to proxy a Web service using an Apache Camel route, where the route treats the messages as HTTP packets. The key feature of this route is that both the consumer endpoint (at the start of the route) and the producer endpoint (at the end of the route) must be compatible with the HTTP packet format.

Figure 41.1. Proxy Route with Message in HTTP Format
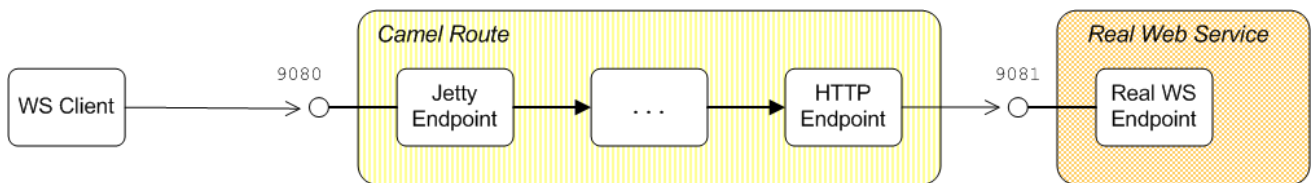
Alternatives for the consumer endpoint

The following Apache Camel endpoints can be used as consumer endpoints for HTTP format messages:

- *Jetty endpoint*—is a lightweight Web server. You can use Jetty to handle messages for *any* HTTP-based protocol, including the commonly-used Web service SOAP/HTTP protocol.

- *Camel CXF endpoint in MESSAGE mode* —when a Camel CXF endpoint is used in MESSAGE mode, the body of the exchange message is the raw message received from the transport layer (which is HTTP). In other words, the Camel CXF endpoint in MESSAGE mode is equivalent to a Jetty endpoint in the case of HTTP-based protocols.

Consumer endpoint for HTTP

A Jetty endpoint has the general form, **jetty:*HttpAddress***. To configure the Jetty endpoint to be a proxy for a Web service, use a ***HttpAddress*** value that is almost identical to the HTTP address the client connects to, except that Jetty's version of ***HttpAddress*** uses the special hostname, **0.0.0.0** (which matches *all* of the network interfaces on the current machine).

```
<route>
   <from uri="jetty:http://0.0.0.0:9093/Customers?matchOnUriPrefix=true"/>
   ...
</route>
```

matchOnUriPrefix option

Normally, a Jetty consumer endpoint accepts only an *exact* match on the context path. For example, a request that is sent to the address **http://localhost:9093/Customers** would be accepted, but a request sent to **http://localhost:9093/Customers/Foo** would be rejected. By setting **matchOnUriPrefix** to **true**, however, you enable a kind of wildcarding on the context path, so that any context path prefixed by **/Customers** is accepted.

Alternatives for the producer endpoint

The following Apache Camel endpoints can be used as producer endpoints for HTTP format messages:

- *Jetty HTTP client endpoint—(recommended)* the Jetty library implements a HTTP client. In particular, the Jetty HTTP client features support for **HttpClient** thread pools, which means that the Jetty implementation scales particularly well.

- *HTTP endpoint*—the HTTP endpoint implements a HTTP client based on the **HttpClient** 3.x API.

- *HTTP4 endpoint*—the HTTP endpoint implements a HTTP client based on the **HttpClient** 4.x API.

Producer endpoint for HTTP

To configure a Jetty HTTP endpoint to send HTTP requests to a remote SOAP/HTTP Web service, set the **uri** attribute of the **to** element at the end of the route to be the address of the remote Web service, as follows:

```
<route>
    ...
    <to uri="jetty:http://localhost:8083/Customers?
bridgeEndpoint=true&amp;throwExceptionOnFailure=false"/>
</route>
```

bridgeEndpoint option

The HTTP component supports a **bridgeEndpoint** option, which you can enable on a HTTP producer endpoint to configure the endpoint appropriately for operating in a HTTP-to-HTTP bridge (as is the case in this demonstration). In particular, when **bridgeEndpoint=true**, the HTTP endpoint ignores the value of the **Exchange.HTTP_URI** header, using the HTTP address from the endpoint URI instead.

throwExceptionOnFailure option

Setting **throwExceptionOnFailure** to **false** ensures that any HTTP exceptions are relayed back to the original WS client, instead of being thrown within the route.

Handling message headers

When defining a HTTP bridge application, the **CamelHttp*** headers set by the consumer endpoint at the start of the route can affect the behavior of the producer endpoint. For this reason, in a bridge application it is advisable to remove the **CamelHttp*** headers before the message reaches the producer endpoint, as follows:

```
<route>
    <from uri="jetty:http:..."/>
```

```
...
<removeHeaders pattern="CamelHttp*"/>
<to uri="jetty:http:..."/>
</route>
```

Outgoing HTTP headers

By default, any headers in the exchange that are *not* prefixed by **Camel** will be converted into HTTP headers and sent out over the wire by the HTTP producer endpoint. This could have adverse consequences on the behavior of your application, *so it is important to be aware of any headers that are set in the exchange object and to remove them, if necessary.*

For more details about dealing with headers, see Section 41.4, "Handling HTTP Headers" .

41.2. PROXYING WITH POJO FORMAT

Overview

If you want to access the content of the Web services messages that pass throught the route, you might prefer to process the messages in POJO format: that is, where the body of the exchange consists of a list of Java objects representing the WS operation parameters. The key advantate of using POJO format is that you can easily *process the contents of a message* , by accessing the operation parameters as Java objects.

Figure 41.2, "Proxy Route with Message in POJO Format" shows an overview of how to proxy a Web service using an Apache Camel route, where the route processes the messages in POJO format. The key feature of this route is that both the consumer endpoint (at the start of the route) and the producer endpoint (at the end of the route) must be compatible with the POJO data format.

Figure 41.2. Proxy Route with Message in POJO Format
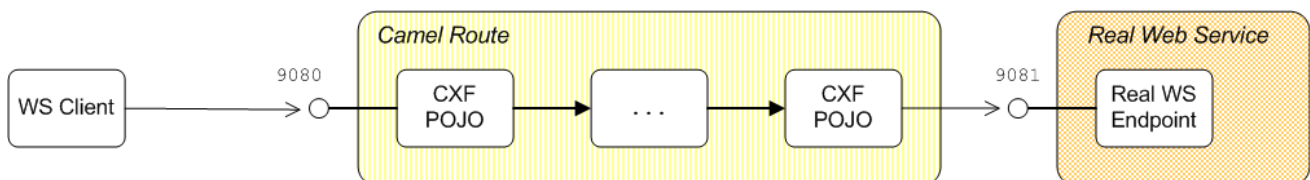
Consumer endpoint for CXF/POJO

To parse incoming messages into POJO data format, the consumer endpoint at the start of the route must be a Camel CXF endpoint that is configured to use POJO mode. Use the **cxf:bean:***BeanID* URI format to reference the Camel CXF endpoint as follows (where the **dataFormat** option defaults to POJO):

```
<route>
  <from uri="cxf:bean:customerServiceProxy"/>
  ...
</route>
```

The bean with the ID, **customerServiceProxy**, is a Camel CXF/POJO endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
```

```
...
<cxf:cxfEndpoint
    id="customerServiceProxy"
    xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
    address="/Customers"
    endpointName="c:SOAPOverHTTP"
    serviceName="c:CustomerService"
    wsdlURL="wsdl/CustomerService.wsdl"
    serviceClass="org.fusesource.demo.wsdl.camelcxf.CustomerService"
 />
 ...
</beans>
```

Producer endpoint for CXF/POJO

To convert the exchange body from POJO data format to a SOAP/HTTP message, the producer endpoint at the end of the route must be a Camel CXF endpoint configured to use POJO mode. Use the **cxf:bean:***BeanID* URI format to reference the Camel CXF endpoint as follows (where the **dataFormat** option defaults to POJO):

```
<route>
   ...
   <to uri="cxf:bean:customerServiceReal"/>
</route>
```

The bean with the ID, **customerServiceReal**, is a Camel CXF/POJO endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
 ...
 <cxf:cxfEndpoint
    id="customerServiceReal"
    xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
    address="http://localhost:8083/Customers"
    endpointName="c:SOAPOverHTTP"
    serviceName="c:CustomerService"
    wsdlURL="wsdl/CustomerService.wsdl"
    serviceClass="org.fusesource.demo.wsdl.camelcxf.CustomerService"
 />
 ...
</beans>
```

41.3. PROXYING WITH PAYLOAD FORMAT

Overview

If you want to access the content of the Web services messages that pass throught the route, you might prefer to process the messages in the normal PAYLOAD format: that is, where the body of the exchange is accessible as an XML document (essentially, an **org.w3c.dom.Node** object). The key advantate of using PAYLOAD format is that you can easily *process the contents of a message* , by accessing the message body as an XML document.

Figure 41.3, "Proxy Route with Message in PAYLOAD Format" shows an overview of how to proxy a Web service using an Apache Camel route, where the route processes the messages in PAYLOAD format. The key feature of this route is that both the consumer endpoint (at the start of the route) and the producer endpoint (at the end of the route) must be compatible with the PAYLOAD data format.

Figure 41.3. Proxy Route with Message in PAYLOAD Format
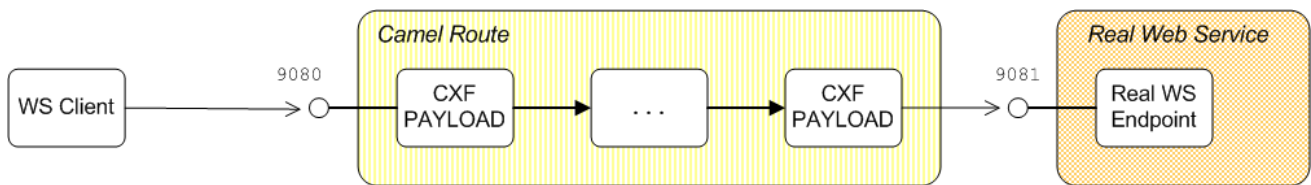
Consumer endpoint for CXF/PAYLOAD

To parse incoming messages into PAYLOAD data format, the consumer endpoint at the start of the route must be a Camel CXF endpoint that is configured to use PAYLOAD mode. Use the **cxf:bean:***BeanID* URI format to reference the Camel CXF endpoint as follows, where you *must* set the **dataFormat** option to PAYLOAD:

```
<route>
  <from uri="cxf:bean:customerServiceProxy?dataFormat=PAYLOAD"/>
  ...
</route>
```

The bean with the ID, **customerServiceProxy**, is a Camel CXF/PAYLOAD endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
 ...
 <cxf:cxfEndpoint
    id="customerServiceProxy"
    xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
    address="/Customers"
    endpointName="c:SOAPOverHTTP"
    serviceName="c:CustomerService"
    wsdlURL="wsdl/CustomerService.wsdl"
 />
 ...
</beans>
```

Producer endpoint for CXF/PAYLOAD

To convert the exchange body from PAYLOAD data format to a SOAP/HTTP message, the producer endpoint at the end of the route must be a Camel CXF endpoint configured to use PAYLOAD mode. Use the **cxf:bean:***BeanID* URI format to reference the Camel CXF endpoint as follows, where you *must* set the **dataFormat** option to PAYLOAD:

```
<route>
  ...
  <to uri="cxf:bean:customerServiceReal?dataFormat=PAYLOAD"/>
</route>
```

The bean with the ID, **customerServiceReal**, is a Camel CXF/PAYLOAD endpoint, which is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <cxf:cxfEndpoint
      id="customerServiceReal"
      xmlns:c="http://demo.fusesource.org/wsdl/camelcxf"
      address="http://localhost:8083/Customers"
      endpointName="c:SOAPOverHTTP"
      serviceName="c:CustomerService"
      wsdlURL="wsdl/CustomerService.wsdl"
  />
  ...
</beans>
```

Outgoing HTTP headers

By default, any headers in the exchange that are *not* prefixed by **Camel** will be converted into HTTP headers and sent out over the wire by the Camel CXF producer endpoint. This could have adverse consequences on the behavior of your application, *so it is important to be aware of any headers that are set in the exchange object and to remove them, if necessary.*

For more details about dealing with headers, see Section 41.4, "Handling HTTP Headers" .

41.4. HANDLING HTTP HEADERS

Overview

When building bridge applications using HTTP or HTTP-based components, it is important to be aware of how the HTTP-based endpoints process headers. In many cases, internal headers (prefixed by Camel) or other headers can cause unwanted side-effects on your application. It is often necessary to remove or filter out certain headings or classes of headings in your route, in order to ensure that your application behaves as expected.

HTTP-based components

The behavior described in this section affects not just the Camel HTTP component (**camel-http**), but also a number of other HTTP-based components, including:

```
camel-http
camel-http4
camel-jetty
camel-restlet
camel-cxf
camel-cxfrs
```

HTTP headers in Camel CXF

The Camel CXF component copies HTTP headers into message headers for *all* of the supported data formats:

- **POJO**

- **PAYLOAD**

- **MESSAGE**

HTTP consumer endpoint

When a HTTP consumer endpoint receives an incoming message, it creates an *In* message with the following headers:

CamelHttp* headers

Several headers with the **CamelHttp** prefix are created, which record the status of the incoming message. For details of these internal headers, see HTTP.

HTTP headers

All of the HTTP headers from the original incoming message are mapped to headers on the exchange's *In* message.

URL options *(Jetty only)*

The URL options from the original HTTP request URL are mapped to headers on the exchange's *In* message. For example, given the client request with the URL, **http://myserver/myserver?orderid=123**, a Jetty consumer endpoint creates the **orderid** header with value **123**.

HTTP producer endpoint

When a HTTP producer endpoint receives an exchange and converts it to the target message format, it handles the *In* message headers as follows:

CamelHttp*

Headers prefixed by **CamelHttp** are used to control the behaviour of the HTTP producer endpoint. Any headers of this kind are consumed by the HTTP producer endpoint and the endpoint behaves as directed.
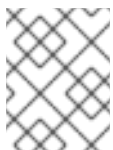
> **NOTE**
>
> However, **CamelHttp** message headers are *ignored* by Camel CXF producer endpoints (but not by Camel CXF-RS producer endpoints).

Camel*

All other headers prefixed by **Camel** are presumed to be meant for internal use and are *not* mapped to HTTP headers in the target message (in other words, these headers are ignored).

*

All other headers are converted to HTTP headers in the target message, with the exception of the following headers, which are blocked (based on a case-insensitive match):

```
content-length
content-type
cache-control
```

> connection
> date
> pragma
> trailer
> transfer-encoding
> upgrade
> via
> warning

Implications for HTTP bridge applications

When defining a HTTP bridge application (that is, a route starting with a HTTP consumer endpoint and ending with a HTTP producer endpoint), the **CamelHttp*** headers set by the consumer endpoint at the start of the route can affect the behavior of the producer endpoint. For this reason, in a bridge application it is advisable to remove the **CamelHttp*** headers, as follows:

```
from("http://0.0.0.0/context/path")
  .removeHeaders("CamelHttp*)
  ...
  .to("http://remoteHost/context/path");
```

Setting a custom header filter

If you want to customize the way that a HTTP producer endpoint processes headers, you can define your own customer header filter by defining the **headerFilterStrategy** option on the endpoint URI. For example, to configure a producer endpoint with the **myHeaderFilterStrategy** filter, you could use a URI like the following:

```
http://remoteHost/context/path?headerFilterStrategy=#myHeaderFilterStrategy
```

Where **myHeaderFilterStrategy** is the bean ID of your custom filter instance.

CHAPTER 42. FILTERING SOAP MESSAGE HEADERS

Abstract

The Camel CXF component supports a flexible header filtering mechanism, which enables you to process SOAP headers, applying different filters according to the header's XML namespace.

42.1. BASIC CONFIGURATION

Overview

When more than one CXF endpoint appears in a route, you need to decide whether or not to allow headers to propagate between the endpoints. By default, the headers are relayed back and forth between the endpoints, but in many cases it might be necessary to filter the headers or to block them altogether. You can control header propagation by applying filters to producer endpoints.

CxfHeaderFilterStrategy

Header filtering is controlled by the **CxfHeaderFilterStrategy** class. Basic configuration of the **CxfHeaderFilterStrategy** class involves setting one or more of the following options:

- the section called "relayHeaders option".

- the section called "relayAllMessageHeaders option".

relayHeaders option

The semantics of the **relayHeaders** option can be summarized as follows:

	In-band headers	Out-of-band headers
relayHeaders=true, **dataFormat=PAYLOAD**	*Filter*	*Filter*
relayHeaders=true, **dataFormat=POJO**	*Relay all*	*Filter*
relayHeaders=false	*Block*	*Block*

In-band headers

An *in-band header* is a header that is explicitly defined as part of the WSDL binding contract for an endpoint.

Out-of-band headers

An *out-of-band header* is a header that is serialized over the wire, but is not explicitly part of the WSDL binding contract. In particular, the SOAP binding permits out-of-band headers, because the SOAP specification does *not* require headers to be defined in the WSDL contract.

Payload format

The CXF endpoint's payload format affects the filter behavior as follows:

POJO

(Default) Only out-of-band headers are available for filtering, because the in-band headers have already been processed and removed from the list by CXF. The in-band headers are incorporated into the **MessageContentList** in POJO mode. If you require access to headers in POJO mode, you have the option of implementing a custom CXF interceptor or JAX-WS handler.

PAYLOAD

In this mode, both in-band and out-of-band headers are available for filtering.

MESSAGE

Not applicable. (In this mode, the message remains in a raw format and the headers are not processed at all.)

Default filter

The default filter is of type, **SoapMessageHeaderFilter**, which removes only the SOAP headers that the SOAP specification expects an intermediate Web service to consume. For more details, see the section called "SoapMessageHeaderFilter".

Overriding the default filter

You can override the default **CxfHeaderFilterStrategy** instance by defining a new **CxfHeaderFilterStrategy** bean and associating it with a CXF endpoint.

Sample relayHeaders configuration

The following example shows how you can use the **relayHeaders** option to create a **CxfHeaderFilterStrategy** bean that blocks all message headers. The CXF endpoints in the route use the **headerFilterStrategy** option to install the filter strategy in the endpoint, where the **headerFilterStrategy** setting has the syntax, **headerFilterStrategy=#*BeanID***.

```xml
<beans ...>
  ...
  <bean id="dropAllMessageHeadersStrategy"
class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrategy">
      <!--  Set relayHeaders to false to drop all SOAP headers -->
      <property name="relayHeaders" value="false"/>
  </bean>

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
      <route>
        <from uri="cxf:bean:routerNoRelayEndpoint?
headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
        <to uri="cxf:bean:serviceNoRelayEndpoint?
headerFilterStrategy=#dropAllMessageHeadersStrategy"/>
      </route>
```

```
    </camelContext>
    ...
  </beans>
```

relayAllMessageHeaders option

The **relayAllMessageHeaders** option is used to propagate *all* SOAP headers, without applying any filtering (any installed filters would be bypassed). In order to enable this feature, you must set *both* **relayHeaders** and **relayAllMessageHeaders** to **true**.

Sample relayAllMessageHeaders configuration

The following example shows how to configure CXF endpoints to propagate *all* SOAP message headers. The **propagateAllMessages** filter strategy sets both **relayHeaders** and **relayAllMessageHeaders** to **true**.

```
<beans ...>
  ...
  <bean id="propagateAllMessages"
class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrategy">
    <!-- Set both properties to true to propagate *all* SOAP headers -->
    <property name="relayHeaders" value="true"/>
    <property name="relayAllMessageHeaders" value="true"/>
  </bean>

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="cxf:bean:routerNoRelayEndpoint?headerFilterStrategy=#propagateAllMessages"/>
      <to uri="cxf:bean:serviceNoRelayEndpoint?headerFilterStrategy=#propagateAllMessages"/>
    </route>
  </camelContext>
  ...
</beans>
```

42.2. HEADER FILTERING

Overview

You can optionally install multiple headers in a **CxfHeaderFilterStrategy** instance. The filtering mechanism then uses the header's XML namespace to lookup a particular filter, which it then applies to the header.

Filter map

Figure 42.1, "Filter Map" shows an overview of the filter map that is contained within a **CxfHeaderFilterStrategy** instance. For each filter that you install in **CxfHeaderFilterStrategy**, corresponding entries are made in the filter map, where one or more XML schema namespaces are associated with each filter.

Figure 42.1. Filter Map
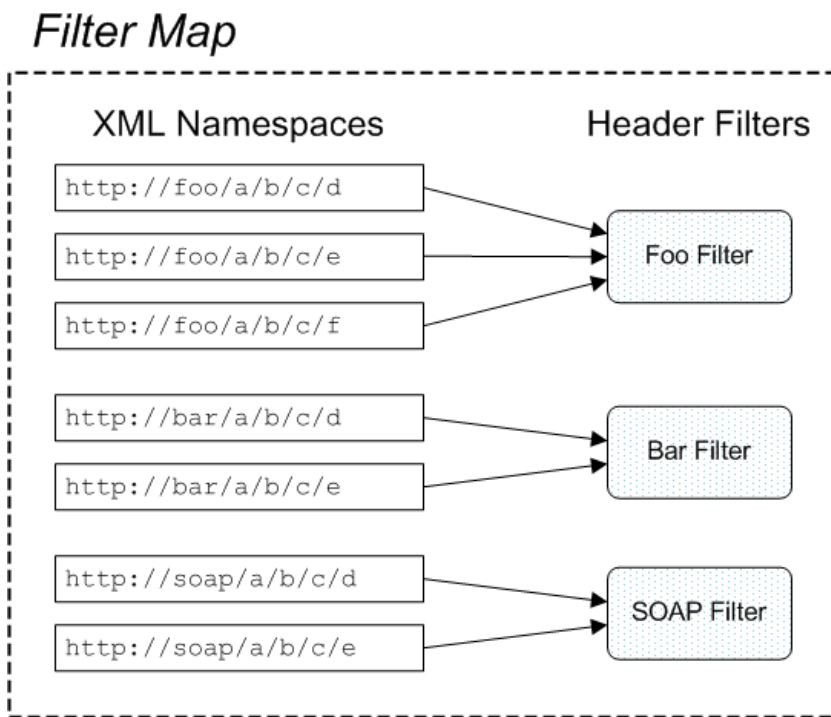
Filter behavior

When a header is filtered, the filter mechanism peeks at the header to discover the header's XML namespace. The filter then looks up the XML namespace in the filter map to find the corresponding filter implementation. This filter is then applied to the header.

PAYLOAD mode

In PAYLOAD mode, both in-band and out-of-band messages pass through the installed filters.

POJO mode

In POJO mode, only out-of-band messages pass through the installed filters. In-band messages bypass the filters and are propagated by default.

42.3. IMPLEMENTING A CUSTOM FILTER

Overview

You can implement your own customer message header filters by implementing the **MessageHeaderFilter** Java interface. You must associate a filter with one or more XML schema namespaces (representing the header's namespace) and it is possible to differentiate between request message headers and response message headers.

MessageHeaderFilter interface

The **MessageHeaderFilter** interface is defined in the **org.apache.camel.component.cxf.common.header** package, as follows:

```
// Java
package org.apache.camel.component.cxf.common.header;
```

```
import java.util.List;

import org.apache.camel.spi.HeaderFilterStrategy.Direction;
import org.apache.cxf.headers.Header;

public interface MessageHeaderFilter {
    List<String> getActivationNamespaces();

    void filter(Direction direction, List<Header> headers);
}
```

Implementing the filter() method

The **MessageHeaderFilter.filter()** method is reponsible for applying header filtering. Filtering is applied both before and after an operation is invoked on an endpoint. Hence, there are two directions to which filtering is applied, as follows:

Direction.OUT

When the **direction** parameter equals **Direction.OUT**, the filter is being applied to a request either leaving a consumer endpoint or entering a producer endpoint (that is, it applies to a WS request message propagating through a route).

Direction.IN

When the **direction** parameter equals **Direction.IN**, the filter is being applied to a response either leaving a producer endpoint or entering a consumer endpoint (that is, it applies to a WS response message being sent back).

Filtering can be applied by removing elements from the list of headers, **headers**. Any headers left in the list are propagated.

Binding filters to XML namespaces

It is possible to register multiple header filters against a given CXF endpoint. The CXF endpoint selects the appropriate filter to use based on the XML namespace of the WSDL binding protocol (for example, the namespace for the SOAP 1.1 binding or for the SOAP 1.2 binding). If a header's namespace is unknown, the header is propagated by default.

To bind a filter to one or more namespaces, implement the **getActivationNamespaces()** method, which returns the list of bound XML namespaces.

Identifying the namespace to bind to

Example 42.1, "Sample Binding Namespaces" illustrates how to identify the namespaces to which you can bind a filter. This example shows the WSDL file for a Bank server that exposes SOAP endpoints.

Example 42.1. Sample Binding Namespaces

```
<wsdl:definitions targetNamespace="http://cxf.apache.org/schemas/cxf/idl/bank"
    xmlns:tns="http://cxf.apache.org/schemas/cxf/idl/bank"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

```
...
    <wsdl:binding name="BankSOAPBinding" type="tns:Bank">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="getAccount">
          ...
      </wsdl:operation>
      ...
    </wsdl:binding>
    ...
</wsdl>
```

From the **soap:binding** tag, you can infer that namespace associated with the SOAP binding is
http://schemas.xmlsoap.org/wsdl/soap/.

Implementing a custom filter

If you want to implement your own custom filter, define a class that inherits from the
MessageHeaderFilter interface and implement its methods as described in this section. For example,
Example 42.2, "Sample Header Filter Implementation" shows an example of a custom filter,
CustomHeaderFilter, that binds to the namespace, **http://cxf.apache.org/bindings/custom**, and relays
all of the headers that pass through it.

Example 42.2. Sample Header Filter Implementation

```java
// Java
package org.apache.camel.component.cxf.soap.headers;

import java.util.Arrays;
import java.util.List;

import org.apache.camel.component.cxf.common.header.MessageHeaderFilter;
import org.apache.camel.spi.HeaderFilterStrategy.Direction;
import org.apache.cxf.headers.Header;

public class CustomHeaderFilter implements MessageHeaderFilter {

    public static final String ACTIVATION_NAMESPACE = "http://cxf.apache.org/bindings/custom";
    public static final List<String> ACTIVATION_NAMESPACES =
Arrays.asList(ACTIVATION_NAMESPACE);

    public List<String> getActivationNamespaces() {
        return ACTIVATION_NAMESPACES;
    }

    public void filter(Direction direction, List<Header> headers) {
    }
}
```

42.4. INSTALLING FILTERS

Overview

To install message header filters, set the **messageHeaderFilters** property of the **CxfHeaderFilterStrategy** object. When you initialize this property with a list of message header filters, the header filter strategy combines the specified filters to make a filter map.

The **messageHeaderFilters** property is of type, **List<MessageHeaderFilter>**.

Installing filters in XML

The following example shows how to create a **CxfHeaderFilterStrategy** instance, specifying a customized list of header filters in the **messageHeaderFilters** property. There are two header filters in this example: **SoapMessageHeaderFilter** and **CustomHeaderFilter**.

```
<bean id="customMessageFilterStrategy"
class="org.apache.camel.component.cxf.common.header.CxfHeaderFilterStrategy">
   <property name="messageHeaderFilters">
      <list>
         <!--  SoapMessageHeaderFilter is the built in filter.  It can be removed by omitting it. -->
         <bean
class="org.apache.camel.component.cxf.common.header.SoapMessageHeaderFilter"/>

         <!--  Add custom filter here -->
         <bean class="org.apache.camel.component.cxf.soap.headers.CustomHeaderFilter"/>
      </list>
   </property>
   <!-- The 'relayHeaders' property is 'true' by default -->
</bean>
```

SoapMessageHeaderFilter

The first header filter in the preceding example is the **SoapMessageHeaderFilter** filter, which is the default header filter. This filter is designed to filter standard SOAP headers and is bound to the following XML namespaces:

```
http://schemas.xmlsoap.org/soap/
http://schemas.xmlsoap.org/wsdl/soap/
http://schemas.xmlsoap.org/wsdl/soap12/
```

This filter peeks at the header element, in order to decide whether or not to block a particular header. If the **soap:actor** attribute (SOAP 1.1) or the **soap:role** attribute (SOAP 1.2) is present and has the value *next*, the header is removed from the message. Otherwise, the header is propagated.

Namespace clashes

Normally, each namespace should be bound to just a single header filter. If a namespace is bound to more than one header filter, this normally causes an error. It is possible, however, to override this policy by setting the **allowFilterNamespaceClash** property to **true** in the **CxfHeaderFilterStrategy** instance. When this policy is set to **true**, the nearest to last filter is selected, in the event of a namespace clash.

PART IV. PROGRAMMING EIP COMPONENTS

Abstract

This guide describes how to use the Apache Camel API.

CHAPTER 43. UNDERSTANDING MESSAGE FORMATS

Abstract

Before you can begin programming with Apache Camel, you should have a clear understanding of how messages and message exchanges are modelled. Because Apache Camel can process many message formats, the basic message type is designed to have an abstract format. Apache Camel provides the APIs needed to access and transform the data formats that underly message bodies and message headers.

43.1. EXCHANGES

Overview

An *exchange object* is a wrapper that encapsulates a received message and stores its associated metadata (including the *exchange properties*). In addition, if the current message is dispatched to a producer endpoint, the exchange provides a temporary slot to hold the reply (the *Out* message).

An important feature of exchanges in Apache Camel is that they support lazy creation of messages. This can provide a significant optimization in the case of routes that do not require explicit access to messages.

Figure 43.1. Exchange Object Passing through a Route
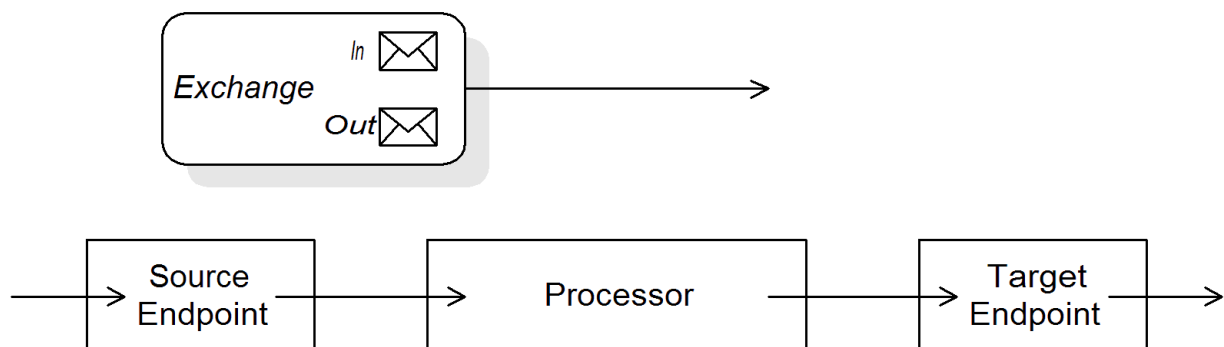
Figure 43.1, "Exchange Object Passing through a Route" shows an exchange object passing through a route. In the context of a route, an exchange object gets passed as the argument of the **Processor.process()** method. This means that the exchange object is directly accessible to the source endpoint, the target endpoint, and all of the processors in between.

The Exchange interface

The **org.apache.camel.Exchange** interface defines methods to access *In* and *Out* messages, as shown in Example 43.1, "Exchange Methods".

Example 43.1. Exchange Methods

```
// Access the In message
Message getIn();
void    setIn(Message in);

// Access the Out message (if any)
Message getOut();
```

```
void   setOut(Message out);
boolean hasOut();

// Access the exchange ID
String  getExchangeId();
void   setExchangeId(String id);
```

For a complete description of the methods in the **Exchange** interface, see Section 52.1, "The Exchange Interface".

Lazy creation of messages

Apache Camel supports lazy creation of *In*, *Out*, and *Fault* messages. This means that message instances are not created until you try to access them (for example, by calling **getIn()** or **getOut()**). The lazy message creation semantics are implemented by the **org.apache.camel.impl.DefaultExchange** class.

If you call one of the no-argument accessors (**getIn()** or **getOut()**), or if you call an accessor with the boolean argument equal to **true** (that is, **getIn(true)** or **getOut(true)**), the default method implementation creates a new message instance, if one does not already exist.

If you call an accessor with the boolean argument equal to **false** (that is, **getIn(false)** or **getOut(false)**), the default method implementation returns the current message value.[2]

Lazy creation of exchange IDs

Apache Camel supports lazy creation of exchange IDs. You can call **getExchangeId()** on any exchange to obtain a unique ID for that exchange instance, but the ID is generated only when you actually call the method. The **DefaultExchange.getExchangeId()** implementation of this method delegates ID generation to the UUID generator that is registered with the **CamelContext**.

For details of how to register UUID generators with the **CamelContext**, see Section 43.4, "Built-In UUID Generators".

43.2. MESSAGES

Overview

Message objects represent messages using the following abstract model:

- *Message body*

- *Message headers*

- *Message attachments*

The message body and the message headers can be of arbitrary type (they are declared as type **Object**) and the message attachments are declared to be of type **javax.activation.DataHandler** , which can contain arbitrary MIME types. If you need to obtain a concrete representation of the message contents, you can convert the body and headers to another type using the type converter mechanism and, possibly, using the marshalling and unmarshalling mechanism.

One important feature of Apache Camel messages is that they support *lazy creation* of message bodies and headers. In some cases, this means that a message can pass through a route without needing to be parsed at all.

The Message interface

The **org.apache.camel.Message** interface defines methods to access the message body, message headers and message attachments, as shown in Example 43.2, "Message Interface".

Example 43.2. Message Interface

```
// Access the message body
Object getBody();
<T> T  getBody(Class<T> type);
void   setBody(Object body);
<T> void setBody(Object body, Class<T> type);

// Access message headers
Object getHeader(String name);
<T> T  getHeader(String name, Class<T> type);
void   setHeader(String name, Object value);
Object removeHeader(String name);
Map<String, Object> getHeaders();
void setHeaders(Map<String, Object> headers);

// Access message attachments
javax.activation.DataHandler getAttachment(String id);
java.util.Map<String, javax.activation.DataHandler> getAttachments();
java.util.Set<String> getAttachmentNames();
void addAttachment(String id, javax.activation.DataHandler content)

// Access the message ID
String getMessageId();
void   setMessageId(String messageId);
```

For a complete description of the methods in the **Message** interface, see Section 53.1, "The Message Interface".

Lazy creation of bodies, headers, and attachments

Apache Camel supports lazy creation of bodies, headers, and attachments. This means that the objects that represent a message body, a message header, or a message attachment are not created until they are needed.

For example, consider the following route that accesses the **foo** message header from the *In* message:

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

In this route, if we assume that the component referenced by *SourceURL* supports lazy creation, the *In* message headers are not actually parsed until the **header("foo")** call is executed. At that point, the

underlying message implementation parses the headers and populates the header map. The message *body* is not parsed until you reach the end of the route, at the **to("*TargetURL*")** call. At that point, the body is converted into the format required for writing it to the target endpoint, *TargetURL*.

By waiting until the last possible moment before populating the bodies, headers, and attachments, you can ensure that unnecessary type conversions are avoided. In some cases, you can completely avoid parsing. For example, if a route contains no explicit references to message headers, a message could traverse the route without ever parsing the headers.

Whether or not lazy creation is implemented in practice depends on the underlying component implementation. In general, lazy creation is valuable for those cases where creating a message body, a message header, or a message attachment is expensive. For details about implementing a message type that supports lazy creation, see Section 53.2, "Implementing the Message Interface" .

Lazy creation of message IDs

Apache Camel supports lazy creation of message IDs. That is, a message ID is generated only when you actually call the **getMessageId()** method. The **DefaultExchange.getExchangeId()** implementation of this method delegates ID generation to the UUID generator that is registered with the **CamelContext**.

Some endpoint implementations would call the **getMessageId()** method implicitly, if the endpoint implements a protocol that requires a unique message ID. In particular, JMS messages normally include a header containing unique message ID, so the JMS component automatically calls **getMessageId()** to obtain the message ID (this is controlled by the **messageIdEnabled** option on the JMS endpoint).

For details of how to register UUID generators with the **CamelContext**, see Section 43.4, "Built-In UUID Generators".

Initial message format

The initial format of an *In* message is determined by the source endpoint, and the initial format of an *Out* message is determined by the target endpoint. If lazy creation is supported by the underlying component, the message remains unparsed until it is accessed explicitly by the application. Most Apache Camel components create the message body in a relatively raw form—for example, representing it using types such as **byte[]**, **ByteBuffer**, **InputStream**, or **OutputStream**. This ensures that the overhead required for creating the initial message is minimal. Where more elaborate message formats are required components usually rely on *type converters* or *marshalling processors*.

Type converters

It does not matter what the initial format of the message is, because you can easily convert a message from one format to another using the built-in type converters (see Section 43.3, "Built-In Type Converters"). There are various methods in the Apache Camel API that expose type conversion functionality. For example, the **convertBodyTo(Class type)** method can be inserted into a route to convert the body of an *In* message, as follows:

```
from("SourceURL").convertBodyTo(String.class).to("TargetURL");
```

Where the body of the *In* message is converted to a **java.lang.String**. The following example shows how to append a string to the end of the *In* message body:

```
from("SourceURL").setBody(bodyAs(String.class).append("My Special Signature")).to("TargetURL");
```

Where the message body is converted to a string format before appending a string to the end. It is not necessary to convert the message body explicitly in this example. You can also use:

```
from("SourceURL").setBody(body().append("My Special Signature")).to("TargetURL");
```

Where the **append()** method automatically converts the message body to a string before appending its argument.

Type conversion methods in Message

The **org.apache.camel.Message** interface exposes some methods that perform type conversion explicitly:

- **getBody(Class<T> type)**—Returns the message body as type, **T**.

- **getHeader(String name, Class<T> type)**—Returns the named header value as type, **T**.

For the complete list of supported conversion types, see Section 43.3, "Built-In Type Converters" .

Converting to XML

In addition to supporting conversion between simple types (such as **byte[]**, **ByteBuffer**, **String**, and so on), the built-in type converter also supports conversion to XML formats. For example, you can convert a message body to the **org.w3c.dom.Document** type. This conversion is more expensive than the simple conversions, because it involves parsing the entire message and then creating a tree of nodes to represent the XML document structure. You can convert to the following XML document types:

- **org.w3c.dom.Document**

- **javax.xml.transform.sax.SAXSource**

XML type conversions have narrower applicability than the simpler conversions. Because not every message body conforms to an XML structure, you have to remember that this type conversion might fail. On the other hand, there are many scenarios where a router deals exclusively with XML message types.

Marshalling and unmarshalling

Marshalling involves converting a high-level format to a low-level format, and *unmarshalling* involves converting a low-level format to a high-level format. The following two processors are used to perform marshalling or unmarshalling in a route:

- **marshal()**

- **unmarshal()**

For example, to read a serialized Java object from a file and unmarshal it into a Java object, you could use the route definition shown in Example 43.3, "Unmarshalling a Java Object" .

Example 43.3. Unmarshalling a Java Object

```
from("file://tmp/appfiles/serialized")
    .unmarshal()
    .serialization()
```

```
.<FurtherProcessing>
.to("TargetURL");
```

Final message format

When an *In* message reaches the end of a route, the target endpoint must be able to convert the message body into a format that can be written to the physical endpoint. The same rule applies to *Out* messages that arrive back at the source endpoint. This conversion is usually performed implicitly, using the Apache Camel type converter. Typically, this involves converting from a low-level format to another low-level format, such as converting from a **byte[]** array to an **InputStream** type.

43.3. BUILT-IN TYPE CONVERTERS

Overview

This section describes the conversions supported by the master type converter. These conversions are built into the Apache Camel core.

Usually, the type converter is called through convenience functions, such as **Message.getBody(Class<T> type)** or **Message.getHeader(String name, Class<T> type)**. It is also possible to invoke the master type converter directly. For example, if you have an exchange object, **exchange**, you could convert a given value to a **String** as shown in Example 43.4, "Converting a Value to a String".

Example 43.4. Converting a Value to a String

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
String str_value = tc.convertTo(String.class, value);
```

Basic type converters

Apache Camel provides built-in type converters that perform conversions to and from the following basic types:

- **java.io.File**

- **String**

- **byte[]** and **java.nio.ByteBuffer**

- **java.io.InputStream** and **java.io.OutputStream**

- **java.io.Reader** and **java.io.Writer**

- **java.io.BufferedReader** and **java.io.BufferedWriter**

- **java.io.StringReader**

However, not all of these types are inter-convertible. The built-in converter is mainly focused on providing conversions from the **File** and **String** types. The **File** type can be converted to any of the preceding types, except **Reader**, **Writer**, and **StringReader**. The **String** type can be converted to **File**,

byte[], **ByteBuffer**, **InputStream**, or **StringReader**. The conversion from **String** to **File** works by interpreting the string as a file name. The trio of **String**, **byte[]**, and **ByteBuffer** are completely inter-convertible.

> **NOTE**
>
> You can explicitly specify which character encoding to use for conversion from **byte[]** to **String** and from **String** to **byte[]** by setting the **Exchange.CHARSET_NAME** exchange property in the current exchange. For example, to perform conversions using the UTF-8 character encoding, call **exchange.setProperty("Exchange.CHARSET_NAME", "UTF-8")**. The supported character sets are described in the **java.nio.charset.Charset** class.

Collection type converters

Apache Camel provides built-in type converters that perform conversions to and from the following collection types:

- **Object[]**

- **java.util.Set**

- **java.util.List**

All permutations of conversions between the preceding collection types are supported.

Map type converters

Apache Camel provides built-in type converters that perform conversions to and from the following map types:

- **java.util.Map**

- **java.util.HashMap**

- **java.util.Hashtable**

- **java.util.Properties**

The preceding map types can also be converted into a set, of **java.util.Set** type, where the set elements are of the **MapEntry<K,V>** type.

DOM type converters

You can perform type conversions to the following Document Object Model (DOM) types:

- **org.w3c.dom.Document**—convertible from **byte[]**, **String**, **java.io.File**, and **java.io.InputStream**.

- **org.w3c.dom.Node**

- **javax.xml.transform.dom.DOMSource**—convertible from **String**.

- **javax.xml.transform.Source**—convertible from **byte[]** and **String**.

All permutations of conversions between the preceding DOM types are supported.

SAX type converters

You can also perform conversions to the **javax.xml.transform.sax.SAXSource** type, which supports the SAX event-driven XML parser (see the SAX Web site for details). You can convert to **SAXSource** from the following types:

- **String**
- **InputStream**
- **Source**
- **StreamSource**
- **DOMSource**

enum type converter

Camel provides a type converter for performing **String** to **enum** type conversions, where the string value is converted to the matching **enum** constant from the specified enumeration class (the matching is *case-insensitive*). This type converter is rarely needed for converting message bodies, but it is frequently used internally by Apache Camel to select particular options.

For example, when setting the logging level option, the following value, **INFO**, is converted into an **enum** constant:

```
<to uri="log:foo?level=INFO"/>
```

Because the **enum** type converter is case-insensitive, any of the following alternatives would also work:

```
<to uri="log:foo?level=info"/>
<to uri="log:foo?level=INfo"/>
<to uri="log:foo?level=InFo"/>
```

Custom type converters

Apache Camel also enables you to implement your own custom type converters. For details on how to implement a custom type converter, see Chapter 45, *Type Converters*.

43.4. BUILT-IN UUID GENERATORS

Overview

Apache Camel enables you to register a UUID generator in the **CamelContext**. This UUID generator is then used whenever Apache Camel needs to generate a unique ID—in particular, the registered UUID generator is called to generate the IDs returned by the **Exchange.getExchangeId()** and the **Message.getMessageId()** methods.

For example, you might prefer to replace the default UUID generator, if part of your application does not support IDs with a length of 36 characters (like Websphere MQ). Also, it can be convenient to generate IDs using a simple counter (see the **SimpleUuidGenerator**) for testing purposes.

Provided UUID generators

You can configure Apache Camel to use one of the following UUID generators, which are provided in the core:

- **org.apache.camel.impl.ActiveMQUuidGenerator**—*(Default)* generates the same style of ID as is used by Apache ActiveMQ. This implementation might not be suitable for all applications, because it uses some JDK APIs that are forbidden in the context of cloud computing (such as the Google App Engine).

- **org.apache.camel.impl.SimpleUuidGenerator**—implements a simple counter ID, starting at **1**. The underlying implementation uses the **java.util.concurrent.atomic.AtomicLong** type, so that it is thread-safe.

- **org.apache.camel.impl.JavaUuidGenerator**—implements an ID based on the **java.util.UUID** type. Because **java.util.UUID** is synchronized, this might affect performance on some highly concurrent systems.

Custom UUID generator

To implement a custom UUID generator, implement the **org.apache.camel.spi.UuidGenerator** interface, which is shown in Example 43.5, "UuidGenerator Interface". The **generateUuid()** must be implemented to return a unique ID string.

Example 43.5. UuidGenerator Interface

```
// Java
package org.apache.camel.spi;

/**
 * Generator to generate UUID strings.
 */
public interface UuidGenerator {
    String generateUuid();
}
```

Specifying the UUID generator using Java

To replace the default UUID generator using Java, call the **setUuidGenerator()** method on the current **CamelContext** object. For example, you can register a **SimpleUuidGenerator** instance with the current **CamelContext**, as follows:

```
// Java
getContext().setUuidGenerator(new org.apache.camel.impl.SimpleUuidGenerator());
```

> **NOTE**
>
> The **setUuidGenerator()** method should be called during startup, *before* any routes are activated.

Specifying the UUID generator using Spring

To replace the default UUID generator using Spring, all you need to do is to create an instance of a UUID generator using the Spring **bean** element. When a **camelContext** instance is created, it automatically

looks up the Spring registry, searching for a bean that implements
org.apache.camel.spi.UuidGenerator. For example, you can register a **SimpleUuidGenerator** instance
with the **CamelContext** as follows:

```
<beans ...>
  <bean id="simpleUuidGenerator"
      class="org.apache.camel.impl.SimpleUuidGenerator" />

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

  ...
</beans>
```

---

[2] If there is no active method the returned value will be **null**.

CHAPTER 44. IMPLEMENTING A PROCESSOR

Abstract

Apache Camel allows you to implement a custom processor. You can then insert the custom processor into a route to perform operations on exchange objects as they pass through the route.

44.1. PROCESSING MODEL

Pipelining model

The *pipelining model* describes the way in which processors are arranged in Section 5.4, "Pipes and Filters". Pipelining is the most common way to process a sequence of endpoints (a producer endpoint is just a special type of processor). When the processors are arranged in this way, the exchange's *In* and *Out* messages are processed as shown in Figure 44.1, "Pipelining Model".

Figure 44.1. Pipelining Model
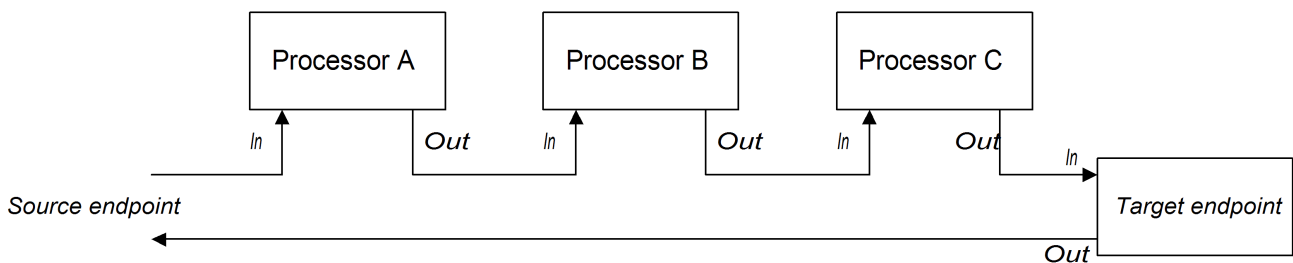
The processors in the pipeline look like services, where the *In* message is analogous to a request, and the *Out* message is analogous to a reply. In fact, in a realistic pipeline, the nodes in the pipeline are often implemented by Web service endpoints, such as the CXF component.

For example, Example 44.1, "Java DSL Pipeline" shows a Java DSL pipeline constructed from a sequence of two processors, **ProcessorA**, **ProcessorB**, and a producer endpoint, *TargetURI*.

> **Example 44.1. Java DSL Pipeline**
>
> ```
> from(SourceURI).pipeline(ProcessorA, ProcessorB, TargetURI);
> ```

44.2. IMPLEMENTING A SIMPLE PROCESSOR

Overview

This section describes how to implement a simple processor that executes message processing logic before delegating the exchange to the next processor in the route.

Processor interface

Simple processors are created by implementing the **org.apache.camel.Processor** interface. As shown in Example 44.2, "Processor Interface", the interface defines a single method, **process()**, which processes an exchange object.

Example 44.2. Processor Interface

```
package org.apache.camel;

public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

Implementing the Processor interface

To create a simple processor you must implement the **Processor** interface and provide the logic for the **process()** method. Example 44.3, "Simple Processor Implementation" shows the outline of a simple processor implementation.

Example 44.3. Simple Processor Implementation

```
import org.apache.camel.Processor;

public class MyProcessor implements Processor {
    public MyProcessor() { }

    public void process(Exchange exchange) throws Exception
    {
        // Insert code that gets executed *before* delegating
        // to the next processor in the chain.

        ...
    }
}
```

All of the code in the **process()** method gets executed *before* the exchange object is delegated to the next processor in the chain.

For examples of how to access the message body and header values inside a simple processor, see Section 44.3, "Accessing Message Content".

Inserting the simple processor into a route

Use the **process()** DSL command to insert a simple processor into a route. Create an instance of your custom processor and then pass this instance as an argument to the **process()** method, as follows:

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL").process(myProc).to("TargetURL");
```

44.3. ACCESSING MESSAGE CONTENT

Accessing message headers

Message headers typically contain the most useful message content from the perspective of a router, because headers are often intended to be processed in a router service. To access header data, you

must first get the message from the exchange object (for example, using **Exchange.getIn()**), and then use the **Message** interface to retrieve the individual headers (for example, using **Message.getHeader()**).

Example 44.4, "Accessing an Authorization Header" shows an example of a custom processor that accesses the value of a header named **Authorization**. This example uses the **ExchangeHelper.getMandatoryHeader()** method, which eliminates the need to test for a null header value.

Example 44.4. Accessing an Authorization Header

```
import org.apache.camel.*;
import org.apache.camel.util.ExchangeHelper;

public class MyProcessor implements Processor {
  public void process(Exchange exchange) {
    String auth = ExchangeHelper.getMandatoryHeader(
              exchange,
              "Authorization",
              String.class
          );
    // process the authorization string...
    // ...
  }
}
```

For full details of the **Message** interface, see Section 43.2, "Messages".

Accessing the message body

You can also access the message body. For example, to append a string to the end of the *In* message, you can use the processor shown in Example 44.5, "Accessing the Message Body".

Example 44.5. Accessing the Message Body

```
import org.apache.camel.*;
import org.apache.camel.util.ExchangeHelper;

public class MyProcessor implements Processor {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}
```

Accessing message attachments

You can access a message's attachments using either the **Message.getAttachment()** method or the **Message.getAttachments()** method. See Example 43.2, "Message Interface" for more details.

44.4. THE EXCHANGEHELPER CLASS

Overview

The **org.apache.camel.util.ExchangeHelper** class is a Apache Camel utility class that provides methods that are useful when implementing a processor.

Resolve an endpoint

The static **resolveEndpoint()** method is one of the most useful methods in the **ExchangeHelper** class. You use it inside a processor to create new **Endpoint** instances on the fly.

> **Example 44.6. The resolveEndpoint() Method**
>
> ```
> public final class ExchangeHelper {
>    ...
>    @SuppressWarnings({"unchecked" })
>    public static Endpoint
>    resolveEndpoint(Exchange exchange, Object value)
>       throws NoSuchEndpointException { ... }
>    ...
> }
> ```

The first argument to **resolveEndpoint()** is an exchange instance, and the second argument is usually an endpoint URI string. Example 44.7, "Creating a File Endpoint" shows how to create a new file endpoint from an exchange instance **exchange**

> **Example 44.7. Creating a File Endpoint**
>
> ```
> Endpoint file_endp = ExchangeHelper.resolveEndpoint(exchange, "file://tmp/messages/in.xml");
> ```

Wrapping the exchange accessors

The **ExchangeHelper** class provides several static methods of the form **getMandatory*BeanProperty*()**, which wrap the corresponding **get*BeanProperty*()** methods on the **Exchange** class. The difference between them is that the original **get*BeanProperty*()** accessors return **null**, if the corresponding property is unavailable, and the **getMandatory*BeanProperty*()** wrapper methods throw a Java exception. The following wrapper methods are implemented in the **ExchangeHelper** class:

```
public final class ExchangeHelper {
   ...
   public static <T> T getMandatoryProperty(Exchange exchange, String propertyName, Class<T>
type)
       throws NoSuchPropertyException { ... }

   public static <T> T getMandatoryHeader(Exchange exchange, String propertyName, Class<T>
type)
       throws NoSuchHeaderException { ... }

   public static Object getMandatoryInBody(Exchange exchange)
       throws InvalidPayloadException { ... }

   public static <T> T getMandatoryInBody(Exchange exchange, Class<T> type)
```

```
      throws InvalidPayloadException { ... }

   public static Object getMandatoryOutBody(Exchange exchange)
      throws InvalidPayloadException { ... }

   public static <T> T getMandatoryOutBody(Exchange exchange, Class<T> type)
      throws InvalidPayloadException { ... }
   ...
}
```

Testing the exchange pattern

Several different exchange patterns are compatible with holding an *In* message. Several different exchange patterns are also compatible with holding an *Out* message. To provide a quick way of checking whether or not an exchange object is capable of holding an *In* message or an *Out* message, the **ExchangeHelper** class provides the following methods:

```
public final class ExchangeHelper {
   ...
   public static boolean isInCapable(Exchange exchange) { ... }

   public static boolean isOutCapable(Exchange exchange) { ... }
   ...
}
```

Get the In message's MIME content type

If you want to find out the MIME content type of the exchange's *In* message, you can access it by calling the **ExchangeHelper.getContentType(exchange)** method. To implement this, the **ExchangeHelper** object looks up the value of the *In* message's **Content-Type** header—this method relies on the underlying component to populate the header value).

CHAPTER 45. TYPE CONVERTERS

Abstract

Apache Camel has a built-in type conversion mechanism, which is used to convert message bodies and message headers to different types. This chapter explains how to extend the type conversion mechanism by adding your own custom converter methods.

45.1. TYPE CONVERTER ARCHITECTURE

Overview

This section describes the overall architecture of the type converter mechanism, which you must understand, if you want to write custom type converters. If you only need to use the built-in type converters, see Chapter 43, *Understanding Message Formats*.

Type converter interface

Example 45.1, "TypeConverter Interface" shows the definition of the **org.apache.camel.TypeConverter** interface, which all type converters must implement.

> **Example 45.1. TypeConverter Interface**
>
> ```
> package org.apache.camel;
>
> public interface TypeConverter {
>     <T> T convertTo(Class<T> type, Object value);
> }
> ```

Master type converter

The Apache Camel type converter mechanism follows a master/slave pattern. There are many *slave* type converters, which are each capable of performing a limited number of type conversions, and a single *master* type converter, which aggregates the type conversions performed by the slaves. The master type converter acts as a front-end for the slave type converters. When you request the master to perform a type conversion, it selects the appropriate slave and delegates the conversion task to that slave.

For users of the type conversion mechanism, the master type converter is the most important because it provides the entry point for accessing the conversion mechanism. During start up, Apache Camel automatically associates a master type converter instance with the **CamelContext** object. To obtain a reference to the master type converter, you call the **CamelContext.getTypeConverter()** method. For example, if you have an exchange object, **exchange**, you can obtain a reference to the master type converter as shown in Example 45.2, "Getting a Master Type Converter".

> **Example 45.2. Getting a Master Type Converter**
>
> ```
> org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
> ```

Type converter loader

The master type converter uses a *type converter loader* to populate the registry of slave type converters. A type converter loader is any class that implements the **TypeConverterLoader** interface. Apache Camel currently uses only one kind of type converter loader—the *annotation type converter loader* (of **AnnotationTypeConverterLoader** type).

Type conversion process

Figure 45.1, "Type Conversion Process" gives an overview of the type conversion process, showing the steps involved in converting a given data value, **value**, to a specified type, **toType**.

Figure 45.1. Type Conversion Process
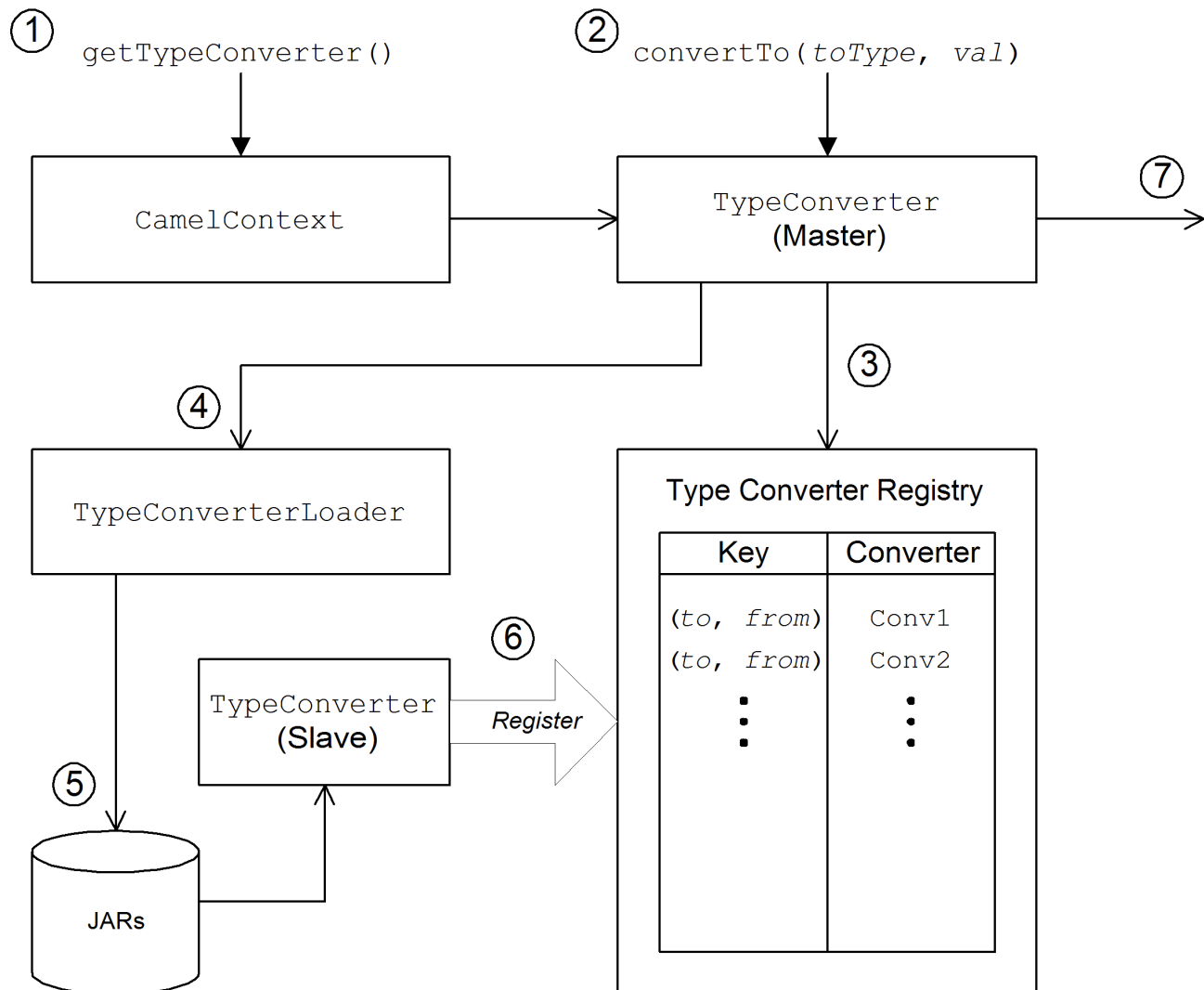
The type conversion mechanism proceeds as follows:

1. The **CamelContext** object holds a reference to the master **TypeConverter** instance. The first step in the conversion process is to retrieve the master type converter by calling **CamelContext.getTypeConverter()**.

2. Type conversion is initiated by calling the **convertTo()** method on the master type converter. This method instructs the type converter to convert the data object, **value**, from its original type to the type specified by the **toType** argument.

3. Because the master type converter is a front end for many different slave type converters, it looks up the appropriate slave type converter by checking a registry of type mappings The

registry of type converters is keyed by a type mapping pair **(*toType*, *fromType*)**. If a suitable type converter is found in the registry, the master type converter calls the slave's **convertTo()** method and returns the result.

4. If a suitable type converter *cannot* be found in the registry, the master type converter loads a new type converter, using the type converter loader.

5. The type converter loader searches the available JAR libraries on the classpath to find a suitable type converter. Currently, the loader strategy that is used is implemented by the annotation type converter loader, which attempts to load a class annotated by the **org.apache.camel.Converter** annotation. See the section called "Create a TypeConverter file" .

6. If the type converter loader is successful, a new slave type converter is loaded and entered into the type converter registry. This type converter is then used to convert the **value** argument to the **toType** type.

7. If the data is successfully converted, the converted data value is returned. If the conversion does not succeed, **null** is returned.

45.2. HANDLING DUPLICATE TYPE CONVERTERS

You can configure what must happen if a duplicate type converter is added.

In the **TypeConverterRegistry** (See Section 45.3, "Implementing Type Converter Using Annotations") you can set the action to *Override*, *Ignore* or *Fail* using the following code:

```
typeconverterregistry = camelContext.getTypeConverter()
// Define the behaviour if the TypeConverter already exists
typeconverterregistry.setTypeConverterExists(TypeConverterExists.Override);
```

Override in this code can be replaced by *Ignore* or *Fail*, depending on your requirements.

TypeConverterExists Class

The TypeConverterExists class consists of the following commands:

```
package org.apache.camel;

import javax.xml.bind.annotation.XmlEnum;

/**
 * What to do if attempting to add a duplicate type converter
 *
 * @version
 */
@XmlEnum
public enum TypeConverterExists {

    Override, Ignore, Fail

}
```

45.3. IMPLEMENTING TYPE CONVERTER USING ANNOTATIONS

Overview

The type conversion mechanism can easily be customized by adding a new slave type converter. This section describes how to implement a slave type converter and how to integrate it with Apache Camel, so that it is automatically loaded by the annotation type converter loader.

How to implement a type converter

To implement a custom type converter, perform the following steps:

1. Implement an annotated converter class .

2. Create a TypeConverter file .

3. Package the type converter.

Implement an annotated converter class

You can implement a custom type converter class using the **@Converter** annotation. You must annotate the class itself and each of the **static** methods intended to perform type conversion. Each converter method takes an argument that defines the *from* type, optionally takes a second **Exchange** argument, and has a non-void return value that defines the *to* type. The type converter loader uses Java reflection to find the annotated methods and integrate them into the type converter mechanism. Example 45.3, "Example of an Annotated Converter Class" shows an example of an annotated converter class that defines a converter method for converting from **java.io.File** to **java.io.InputStream** and another converter method (with an **Exchange** argument) for converting from **byte[]** to **String**.

Example 45.3. Example of an Annotated Converter Class

```java
package com.YourDomain.YourPackageName;

import org.apache.camel.Converter;

import java.io.*;

@Converter
public class IOConverter {
    private IOConverter() {
    }

    @Converter
    public static InputStream toInputStream(File file) throws FileNotFoundException {
        return new BufferedInputStream(new FileInputStream(file));
    }

    @Converter
    public static String toString(byte[] data, Exchange exchange) {
        if (exchange != null) {
            String charsetName = exchange.getProperty(Exchange.CHARSET_NAME, String.class);
            if (charsetName != null) {
                try {
                    return new String(data, charsetName);
                } catch (UnsupportedEncodingException e) {
                    LOG.warn("Can't convert the byte to String with the charset " + charsetName, e);
                }
```

```
            }
        }
        return new String(data);
    }
}
```

The **toInputStream()** method is responsible for performing the conversion from the **File** type to the **InputStream** type and the **toString()** method is responsible for performing the conversion from the **byte[]** type to the **String** type.

> **NOTE**
>
> The method name is unimportant, and can be anything you choose. What is important are the argument type, the return type, and the presence of the **@Converter** annotation.

Create a TypeConverter file

To enable the discovery mechanism (which is implemented by the *annotation type converter loader*) for your custom converter, create a **TypeConverter** file at the following location:

META-INF/services/org/apache/camel/TypeConverter

The **TypeConverter** file must contain a comma-separated list of Fully Qualified Names (FQN) of type converter classes. For example, if you want the type converter loader to search the *YourPackageName*.*YourClassName* package for annotated converter classes, the **TypeConverter** file would have the following contents:

com.*PackageName.FooClass*

An alternative method of enabling the discovery mechanism is to add just package names to the **TypeConverter** file. For example, the **TypeConverter** file would have the following contents:

com.*PackageName*

This would cause the package scanner to scan through the packages for the **@Converter** tag. Using the FQN method is faster and is the preferred method.

Package the type converter

The type converter is packaged as a JAR file containing the compiled classes of your custom type converters and the **META-INF** directory. Put this JAR file on your classpath to make it available to your Apache Camel application.

Fallback converter method

In addition to defining regular converter methods using the **@Converter** annotation, you can optionally define a fallback converter method using the **@FallbackConverter** annotation. The fallback converter method will only be tried, if the master type converter fails to find a regular converter method in the type registry.

The essential difference between a regular converter method and a fallback converter method is that whereas a regular converter is defined to perform conversion between a specific pair of types (for

example, from **byte[]** to **String**), a fallback converter can potentially perform conversion between *any* pair of types. It is up to the code in the body of the fallback converter method to figure out which conversions it is able to perform. At run time, if a conversion cannot be performed by a regular converter, the master type converter iterates through every available fallback converter until it finds one that can perform the conversion.

The method signature of a fallback converter can have either of the following forms:

```
// 1. Non-generic form of signature
@FallbackConverter
public static Object MethodName(
    Class type,
    Exchange exchange,
    Object value,
    TypeConverterRegistry registry
)

// 2. Templating form of signature
@FallbackConverter
public static <T> T MethodName(
    Class<T> type,
    Exchange exchange,
    Object value,
    TypeConverterRegistry registry
)
```

Where *MethodName* is an arbitrary method name for the fallback converter.

For example, the following code extract (taken from the implementation of the File component) shows a fallback converter that can convert the body of a **GenericFile** object, exploiting the type converters already available in the type converter registry:

```
package org.apache.camel.component.file;

import org.apache.camel.Converter;
import org.apache.camel.FallbackConverter;
import org.apache.camel.Exchange;
import org.apache.camel.TypeConverter;
import org.apache.camel.spi.TypeConverterRegistry;

@Converter
public final class GenericFileConverter {

    private GenericFileConverter() {
        // Helper Class
    }

    @FallbackConverter
    public static <T> T convertTo(Class<T> type, Exchange exchange, Object value,
TypeConverterRegistry registry) {
        // use a fallback type converter so we can convert the embedded body if the value is GenericFile
        if (GenericFile.class.isAssignableFrom(value.getClass())) {
            GenericFile file = (GenericFile) value;
            Class from = file.getBody().getClass();
            TypeConverter tc = registry.lookup(type, from);
            if (tc != null) {
```

```
            Object body = file.getBody();
            return tc.convertTo(type, exchange, body);
      }
    }

    return null;
  }
  ...
}
```

45.4. IMPLEMENTING A TYPE CONVERTER DIRECTLY

Overview

Generally, the recommended way to implement a type converter is to use an annotated class, as described in the previous section, Section 45.3, "Implementing Type Converter Using Annotations" . But if you want to have complete control over the registration of your type converter, you can implement a custom slave type converter and add it directly to the type converter registry, as described here.

Implement the TypeConverter interface

To implement your own type converter class, define a class that implements the **TypeConverter** interface. For example, the following **MyOrderTypeConverter** class converts an integer value to a **MyOrder** object, where the integer value is used to initialize the order ID in the **MyOrder** object.

```
import org.apache.camel.TypeConverter

private class MyOrderTypeConverter implements TypeConverter {

  public <T> T convertTo(Class<T> type, Object value) {
    // converter from value to the MyOrder bean
    MyOrder order = new MyOrder();
    order.setId(Integer.parseInt(value.toString()));
    return (T) order;
  }

  public <T> T convertTo(Class<T> type, Exchange exchange, Object value) {
    // this method with the Exchange parameter will be preferd by Camel to invoke
    // this allows you to fetch information from the exchange during convertions
    // such as an encoding parameter or the likes
    return convertTo(type, value);
  }

  public <T> T mandatoryConvertTo(Class<T> type, Object value) {
    return convertTo(type, value);
  }

  public <T> T mandatoryConvertTo(Class<T> type, Exchange exchange, Object value) {
    return convertTo(type, value);
  }
}
```

Add the type converter to the registry

You can add the custom type converter *directly* to the type converter registry using code like the following:

```
// Add the custom type converter to the type converter registry
context.getTypeConverterRegistry().addTypeConverter(MyOrder.class, String.class, new
MyOrderTypeConverter());
```

Where **context** is the current **org.apache.camel.CamelContext** instance. The **addTypeConverter()** method registers the **MyOrderTypeConverter** class against the specific type conversion, from **String.class** to **MyOrder.class**.

You can add custom type converters to your Camel applications without having to use the **META-INF** file. If you are using *Spring* or *Blueprint*, then you can just declare a <bean>. CamelContext discovers the bean automatically and adds the converters.

```
<bean id="myOrderTypeConverters" class="..."/>
 <camelContext>
   ...
</camelContext>
```

You can declare multiple <bean>s if you have more classes.

CHAPTER 46. PRODUCER AND CONSUMER TEMPLATES

Abstract

The producer and consumer templates in Apache Camel are modelled after a feature of the Spring container API, whereby access to a resource is provided through a simplified, easy-to-use API known as a *template*. In the case of Apache Camel, the producer template and consumer template provide simplified interfaces for sending messages to and receiving messages from producer endpoints and consumer endpoints.

46.1. USING THE PRODUCER TEMPLATE

46.1.1. Introduction to the Producer Template

Overview

The producer template supports a variety of different approaches to invoking producer endpoints. There are methods that support different formats for the request message (as an **Exchange** object, as a message body, as a message body with a single header setting, and so on) and there are methods to support both the synchronous and the asynchronous style of invocation. Overall, producer template methods can be grouped into the following categories:

- the section called "Synchronous invocation" .

- the section called "Synchronous invocation with a processor" .

- the section called "Asynchronous invocation".

- the section called "Asynchronous invocation with a callback" .

Synchronous invocation

The methods for invoking endpoints synchronously have names of the form **send***Suffix***()** and **request***Suffix***()**. For example, the methods for invoking an endpoint using either the default message exchange pattern (MEP) or an explicitly specified MEP are named **send()**, **sendBody()**, and **sendBodyAndHeader()** (where these methods respectively send an **Exchange** object, a message body, or a message body and header value). If you want to force the MEP to be *InOut* (request/reply semantics), you can call the **request()**, **requestBody()**, and **requestBodyAndHeader()** methods instead.

The following example shows how to create a **ProducerTemplate** instance and use it to send a message body to the **activemq:MyQueue** endpoint. The example also shows how to send a message body and header value using **sendBodyAndHeader()**.

```
import org.apache.camel.ProducerTemplate
import org.apache.camel.impl.DefaultProducerTemplate
...
ProducerTemplate template = context.createProducerTemplate();

// Send to a specific queue
template.sendBody("activemq:MyQueue", "<hello>world!</hello>");

// Send with a body and header
template.sendBodyAndHeader(
```

```
"activemq:MyQueue",
"<hello>world!</hello>",
"CustomerRating", "Gold" );
```

Synchronous invocation with a processor

A special case of synchronous invocation is where you provide the **send()** method with a **Processor** argument instead of an **Exchange** argument. In this case, the producer template implicitly asks the specified endpoint to create an **Exchange** instance (typically, but not always having the *InOnly* MEP by default). This default exchange is then passed to the processor, which initializes the contents of the exchange object.

The following example shows how to send an exchange initialized by the **MyProcessor** processor to the **activemq:MyQueue** endpoint.

```
import org.apache.camel.ProducerTemplate
import org.apache.camel.impl.DefaultProducerTemplate
...
ProducerTemplate template = context.createProducerTemplate();

// Send to a specific queue, using a processor to initialize
template.send("activemq:MyQueue", new MyProcessor());
```

The **MyProcessor** class is implemented as shown in the following example. In addition to setting the *In* message body (as shown here), you could also initialize message heades and exchange properties.

```
import org.apache.camel.Processor;
import org.apache.camel.Exchange;
...
public class MyProcessor implements Processor {
    public MyProcessor() { }

    public void process(Exchange ex) {
        ex.getIn().setBody("<hello>world!</hello>");
    }
}
```

Asynchronous invocation

The methods for invoking endpoints *asynchronously* have names of the form **asyncSend***Suffix***()** and **asyncRequest***Suffix***()**. For example, the methods for invoking an endpoint using either the default message exchange pattern (MEP) or an explicitly specified MEP are named **asyncSend()** and **asyncSendBody()** (where these methods respectively send an **Exchange** object or a message body). If you want to force the MEP to be *InOut* (request/reply semantics), you can call the **asyncRequestBody()**, **asyncRequestBodyAndHeader()**, and **asyncRequestBodyAndHeaders()** methods instead.

The following example shows how to send an exchange asynchronously to the **direct:start** endpoint. The **asyncSend()** method returns a **java.util.concurrent.Future** object, which is used to retrieve the invocation result at a later time.

```
import java.util.concurrent.Future;

import org.apache.camel.Exchange;
```

```
import org.apache.camel.impl.DefaultExchange;
...
Exchange exchange = new DefaultExchange(context);
exchange.getIn().setBody("Hello");

Future<Exchange> future = template.asyncSend("direct:start", exchange);

// You can do other things, whilst waiting for the invocation to complete
...
// Now, retrieve the resulting exchange from the Future
Exchange result = future.get();
```

The producer template also provides methods to send a message body asynchronously (for example, using **asyncSendBody()** or **asyncRequestBody()**). In this case, you can use one of the following helper methods to extract the returned message body from the **Future** object:

```
<T> T extractFutureBody(Future future, Class<T> type);
<T> T extractFutureBody(Future future, long timeout, TimeUnit unit, Class<T> type) throws
TimeoutException;
```

The first version of the **extractFutureBody()** method blocks until the invocation completes and the reply message is available. The second version of the **extractFutureBody()** method allows you to specify a timeout. Both methods have a type argument, **type**, which casts the returned message body to the specified type using a built-in type converter.

The following example shows how to use the **asyncRequestBody()** method to send a message body to the **direct:start** endpoint. The blocking **extractFutureBody()** method is then used to retrieve the reply message body from the **Future** object.

```
Future<Object> future = template.asyncRequestBody("direct:start", "Hello");

// You can do other things, whilst waiting for the invocation to complete
...
// Now, retrieve the reply message body as a String type
String result = template.extractFutureBody(future, String.class);
```

Asynchronous invocation with a callback

In the preceding asynchronous examples, the request message is dispatched in a sub-thread, while the reply is retrieved and processed by the main thread. The producer template also gives you the option, however, of processing replies in the sub-thread, using one of the **asyncCallback()**, **asyncCallbackSendBody()**, or **asyncCallbackRequestBody()** methods. In this case, you supply a callback object (of **org.apache.camel.impl.SynchronizationAdapter** type), which automatically gets invoked in the sub-thread as soon as a reply message arrives.

The **Synchronization** callback interface is defined as follows:

```
package org.apache.camel.spi;

import org.apache.camel.Exchange;

public interface Synchronization {
    void onComplete(Exchange exchange);
    void onFailure(Exchange exchange);
}
```

Where the **onComplete()** method is called on receipt of a normal reply and the **onFailure()** method is called on receipt of a fault message reply. Only one of these methods gets called back, so you must override both of them to ensure that all types of reply are processed.

The following example shows how to send an exchange to the **direct:start** endpoint, where the reply message is processed in the sub-thread by the **SynchronizationAdapter** callback object.

```
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultExchange;
import org.apache.camel.impl.SynchronizationAdapter;
...
Exchange exchange = context.getEndpoint("direct:start").createExchange();
exchange.getIn().setBody("Hello");

Future<Exchange> future = template.asyncCallback("direct:start", exchange, new
SynchronizationAdapter() {
    @Override
    public void onComplete(Exchange exchange) {
        assertEquals("Hello World", exchange.getIn().getBody());
    }
});
```

Where the **SynchronizationAdapter** class is a default implementation of the **Synchronization** interface, which you can override to provide your own definitions of the **onComplete()** and **onFailure()** callback methods.

You still have the option of accessing the reply from the main thread, because the **asyncCallback()** method also returns a **Future** object—for example:

```
// Retrieve the reply from the main thread, specifying a timeout
Exchange reply = future.get(10, TimeUnit.SECONDS);
```

46.1.2. Synchronous Send

Overview

The *synchronous send* methods are a collection of methods that you can use to invoke a producer endpoint, where the current thread blocks until the method invocation is complete and the reply (if any) has been received. These methods are compatible with any kind of message exchange protocol.

Send an exchange

The basic **send()** method is a general-purpose method that sends the contents of an **Exchange** object to an endpoint, using the message exchange pattern (MEP) of the exchange. The return value is the exchange that you get after it has been processed by the producer endpoint (possibly containing an *Out* message, depending on the MEP).

There are three varieties of **send()** method for sending an exchange that let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object.

```
Exchange send(Exchange exchange);
Exchange send(String endpointUri, Exchange exchange);
Exchange send(Endpoint endpoint, Exchange exchange);
```

Send an exchange populated by a processor

A simple variation of the general **send()** method is to use a processor to populate a default exchange, instead of supplying the exchange object explicitly (see the section called "Synchronous invocation with a processor" for details).

The **send()** methods for sending an exchange populated by a processor let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default.

```
Exchange send(Processor processor);
Exchange send(String endpointUri, Processor processor);
Exchange send(Endpoint endpoint, Processor processor);
Exchange send(
    String endpointUri,
    ExchangePattern pattern,
    Processor processor
);
Exchange send(
    Endpoint endpoint,
    ExchangePattern pattern,
    Processor processor
);
```

Send a message body

If you are only concerned with the contents of the message body that you want to send, you can use the **sendBody()** methods to provide the message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **sendBody()** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default. The methods *without* a **pattern** argument return **void** (even though the invocation might give rise to a reply in some cases); and the methods *with* a **pattern** argument return either the body of the *Out* message (if there is one) or the body of the *In* message (otherwise).

```
void sendBody(Object body);
void sendBody(String endpointUri, Object body);
void sendBody(Endpoint endpoint, Object body);
Object sendBody(
    String endpointUri,
    ExchangePattern pattern,
    Object body
);
Object sendBody(
    Endpoint endpoint,
```

```
    ExchangePattern pattern,
    Object body
);
```

Send a message body and header(s)

For testing purposes, it is often interesting to try out the effect of a *single* header setting and the **sendBodyAndHeader()** methods are useful for this kind of header testing. You supply the message body and header setting as arguments to **sendBodyAndHeader()** and let the producer template take care of inserting the body and header setting into a default exchange object.

The **sendBodyAndHeader()** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default. The methods *without* a **pattern** argument return **void** (even though the invocation might give rise to a reply in some cases); and the methods *with* a **pattern** argument return either the body of the *Out* message (if there is one) or the body of the *In* message (otherwise).

```
void sendBodyAndHeader(
    Object body,
    String header,
    Object headerValue
);
void sendBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
void sendBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
Object sendBodyAndHeader(
    String endpointUri,
    ExchangePattern pattern,
    Object body,
    String header,
    Object headerValue
);
Object sendBodyAndHeader(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body,
    String header,
    Object headerValue
);
```

The **sendBodyAndHeaders()** methods are similar to the **sendBodyAndHeader()** methods, except that instead of supplying just a single header setting, these methods allow you to specify a complete hash map of header settings.

```
void sendBodyAndHeaders(
```

```
      Object body,
      Map<String, Object> headers
   );
   void sendBodyAndHeaders(
      String endpointUri,
      Object body,
      Map<String, Object> headers
   );
   void sendBodyAndHeaders(
      Endpoint endpoint,
      Object body,
      Map<String, Object> headers
   );
   Object sendBodyAndHeaders(
      String endpointUri,
      ExchangePattern pattern,
      Object body,
      Map<String, Object> headers
   );
   Object sendBodyAndHeaders(
      Endpoint endpoint,
      ExchangePattern pattern,
      Object body,
      Map<String, Object> headers
   );
```

Send a message body and exchange property

You can try out the effect of setting a single exchange property using the **sendBodyAndProperty()** methods. You supply the message body and property setting as arguments to **sendBodyAndProperty()** and let the producer template take care of inserting the body and exchange property into a default exchange object.

The **sendBodyAndProperty()** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default. The methods *without* a **pattern** argument return **void** (even though the invocation might give rise to a reply in some cases); and the methods *with* a **pattern** argument return either the body of the *Out* message (if there is one) or the body of the *In* message (otherwise).

```
   void sendBodyAndProperty(
      Object body,
      String property,
      Object propertyValue
   );
   void sendBodyAndProperty(
      String endpointUri,
      Object body,
      String property,
      Object propertyValue
   );
   void sendBodyAndProperty(
      Endpoint endpoint,
      Object body,
      String property,
```

```
        Object propertyValue
    );
    Object sendBodyAndProperty(
        String endpoint,
        ExchangePattern pattern,
        Object body,
        String property,
        Object propertyValue
    );
    Object sendBodyAndProperty(
        Endpoint endpoint,
        ExchangePattern pattern,
        Object body,
        String property,
        Object propertyValue
    );
```

46.1.3. Synchronous Request with InOut Pattern

Overview

The *synchronous request* methods are similar to the synchronous send methods, except that the request methods force the message exchange pattern to be *InOut* (conforming to request/reply semantics). Hence, it is generally convenient to use a synchronous request method, if you expect to receive a reply from the producer endpoint.

Request an exchange populated by a processor

The basic **request()** method is a general-purpose method that uses a processor to populate a default exchange and forces the message exchange pattern to be *InOut* (so that the invocation obeys request/reply semantics). The return value is the exchange that you get after it has been processed by the producer endpoint, where the *Out* message contains the reply message.

The **request()** methods for sending an exchange populated by a processor let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
    Exchange request(String endpointUri, Processor processor);
    Exchange request(Endpoint endpoint, Processor processor);
```

Request a message body

If you are only concerned with the contents of the message body in the request and in the reply, you can use the **requestBody()** methods to provide the request message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **requestBody()** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. The return value is the body of the reply message (*Out* message body), which can either be returned as plain **Object** or converted to a specific type, **T**, using the built-in type converters (see Section 43.3, "Built-In Type Converters").

```
    Object requestBody(Object body);
    <T> T requestBody(Object body, Class<T> type);
    Object requestBody(
        String endpointUri,
```

```
        Object body
);
<T> T requestBody(
    String endpointUri,
    Object body,
    Class<T> type
);
Object requestBody(
    Endpoint endpoint,
    Object body
);
<T> T requestBody(
    Endpoint endpoint,
    Object body,
    Class<T> type
);
```

Request a message body and header(s)

You can try out the effect of setting a single header value using the **requestBodyAndHeader()** methods. You supply the message body and header setting as arguments to **requestBodyAndHeader()** and let the producer template take care of inserting the body and exchange property into a default exchange object.

The **requestBodyAndHeader()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object. The return value is the body of the reply message (*Out* message body), which can either be returned as plain **Object** or converted to a specific type, **T**, using the built-in type converters (see Section 43.3, "Built-In Type Converters").

```
Object requestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
<T> T requestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
Object requestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
<T> T requestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
```

The **requestBodyAndHeaders()** methods are similar to the **requestBodyAndHeader()** methods, except that instead of supplying just a single header setting, these methods allow you to specify a complete hash map of header settings.

```
Object requestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers
);
<T> T requestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
Object requestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers
);
<T> T requestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
```

46.1.4. Asynchronous Send

Overview

The producer template provides a variety of methods for invoking a producer endpoint asynchronously, so that the main thread does not block while waiting for the invocation to complete and the reply message can be retrieved at a later time. The asynchronous send methods described in this section are compatible with any kind of message exchange protocol.

Send an exchange

The basic **asyncSend()** method takes an **Exchange** argument and invokes an endpoint asynchronously, using the message exchange pattern (MEP) of the specified exchange. The return value is a **java.util.concurrent.Future** object, which is a ticket you can use to collect the reply message at a later time—for details of how to obtain the return value from the **Future** object, see the section called "Asynchronous invocation".

The following **asyncSend()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Exchange> asyncSend(String endpointUri, Exchange exchange);
Future<Exchange> asyncSend(Endpoint endpoint, Exchange exchange);
```

Send an exchange populated by a processor

A simple variation of the general **asyncSend()** method is to use a processor to populate a default exchange, instead of supplying the exchange object explicitly.

The following **asyncSend()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Exchange> asyncSend(String endpointUri, Processor processor);
Future<Exchange> asyncSend(Endpoint endpoint, Processor processor);
```

Send a message body

If you are only concerned with the contents of the message body that you want to send, you can use the **asyncSendBody()** methods to send a message body asynchronously and let the producer template take care of inserting the body into a default exchange object.

The **asyncSendBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Object> asyncSendBody(String endpointUri, Object body);
Future<Object> asyncSendBody(Endpoint endpoint, Object body);
```

46.1.5. Asynchronous Request with InOut Pattern

Overview

The *asynchronous request* methods are similar to the asynchronous send methods, except that the request methods force the message exchange pattern to be *InOut* (conforming to request/reply semantics). Hence, it is generally convenient to use an asynchronous request method, if you expect to receive a reply from the producer endpoint.

Request a message body

If you are only concerned with the contents of the message body in the request and in the reply, you can use the **requestBody()** methods to provide the request message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **asyncRequestBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object. The return value that is retrievable from the **Future** object is the body of the reply message (*Out* message body), which can be returned either as a plain **Object** or converted to a specific type, **T**, using a built-in type converter (see the section called "Asynchronous invocation").

```
Future<Object> asyncRequestBody(
    String endpointUri,
    Object body
);
<T> Future<T> asyncRequestBody(
    String endpointUri,
    Object body,
    Class<T> type
);
Future<Object> asyncRequestBody(
    Endpoint endpoint,
    Object body
```

```
);
<T> Future<T> asyncRequestBody(
    Endpoint endpoint,
    Object body,
    Class<T> type
);
```

Request a message body and header(s)

You can try out the effect of setting a single header value using the **asyncRequestBodyAndHeader()** methods. You supply the message body and header setting as arguments to **asyncRequestBodyAndHeader()** and let the producer template take care of inserting the body and exchange property into a default exchange object.

The **asyncRequestBodyAndHeader()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object. The return value that is retrievable from the **Future** object is the body of the reply message (*Out* message body), which can be returned either as a plain **Object** or converted to a specific type, **T**, using a built-in type converter (see the section called "Asynchronous invocation").

```
Future<Object> asyncRequestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
<T> Future<T> asyncRequestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
Future<Object> asyncRequestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
<T> Future<T> asyncRequestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
```

The **asyncRequestBodyAndHeaders()** methods are similar to the **asyncRequestBodyAndHeader()** methods, except that instead of supplying just a single header setting, these methods allow you to specify a complete hash map of header settings.

```
Future<Object> asyncRequestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers
```

```
    );
    <T> Future<T> asyncRequestBodyAndHeaders(
        String endpointUri,
        Object body,
        Map<String, Object> headers,
        Class<T> type
    );
    Future<Object> asyncRequestBodyAndHeaders(
        Endpoint endpoint,
        Object body,
        Map<String, Object> headers
    );
    <T> Future<T> asyncRequestBodyAndHeaders(
        Endpoint endpoint,
        Object body,
        Map<String, Object> headers,
        Class<T> type
    );
```

46.1.6. Asynchronous Send with Callback

Overview

The producer template also provides the option of processing the reply message in the same sub-thread that is used to invoke the producer endpoint. In this case, you provide a callback object, which automatically gets invoked in the sub-thread as soon as the reply message is received. In other words, the *asynchronous send with callback* methods enable you to initiate an invocation in your main thread and then have all of the associated processing—invocation of the producer endpoint, waiting for a reply and processing the reply—occur asynchronously in a sub-thread.

Send an exchange

The basic **asyncCallback()** method takes an **Exchange** argument and invokes an endpoint asynchronously, using the message exchange pattern (MEP) of the specified exchange. This method is similar to the **asyncSend()** method for exchanges, except that it takes an additional **org.apache.camel.spi.Synchronization** argument, which is a callback interface with two methods: **onComplete()** and **onFailure()**. For details of how to use the **Synchronization** callback, see the section called "Asynchronous invocation with a callback".

The following **asyncCallback()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
    Future<Exchange> asyncCallback(
        String endpointUri,
        Exchange exchange,
        Synchronization onCompletion
    );
    Future<Exchange> asyncCallback(
        Endpoint endpoint,
        Exchange exchange,
        Synchronization onCompletion
    );
```

Send an exchange populated by a processor

The **asyncCallback()** method for processors calls a processor to populate a default exchange and forces the message exchange pattern to be *InOut* (so that the invocation obeys request/reply semantics).

The following **asyncCallback()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Exchange> asyncCallback(
    String endpointUri,
    Processor processor,
    Synchronization onCompletion
);
Future<Exchange> asyncCallback(
    Endpoint endpoint,
    Processor processor,
    Synchronization onCompletion
);
```

Send a message body

If you are only concerned with the contents of the message body that you want to send, you can use the **asyncCallbackSendBody()** methods to send a message body asynchronously and let the producer template take care of inserting the body into a default exchange object.

The **asyncCallbackSendBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Object> asyncCallbackSendBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackSendBody(
    Endpoint endpoint,
    Object body,
    Synchronization onCompletion
);
```

Request a message body

If you are only concerned with the contents of the message body in the request and in the reply, you can use the **asyncCallbackRequestBody()** methods to provide the request message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **asyncCallbackRequestBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Object> asyncCallbackRequestBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackRequestBody(
    Endpoint endpoint,
```

```
    Object body,
    Synchronization onCompletion
);
```

46.2. USING THE CONSUMER TEMPLATE

Overview

The consumer template provides methods for polling a consumer endpoint in order to receive incoming messages. You can choose to receive the incoming message either in the form of an exchange object or in the form of a message body (where the message body can be cast to a particular type using a built-in type converter).

Example of polling exchanges

You can use a consumer template to poll a consumer endpoint for exchanges using one of the following polling methods: blocking **receive()**; **receive()** with a timeout; or **receiveNoWait()**, which returns immediately. Because a consumer endpoint represents a service, it is also essential to start the service thread by calling **start()** before you attempt to poll for exchanges.

The following example shows how to poll an exchange from the **seda:foo** consumer endpoint using the blocking **receive()** method:

```
import org.apache.camel.ProducerTemplate;
import org.apache.camel.ConsumerTemplate;
import org.apache.camel.Exchange;
...
ProducerTemplate template = context.createProducerTemplate();
ConsumerTemplate consumer = context.createConsumerTemplate();

// Start the consumer service
consumer.start();
...
template.sendBody("seda:foo", "Hello");
Exchange out = consumer.receive("seda:foo");
...
// Stop the consumer service
consumer.stop();
```

Where the consumer template instance, **consumer**, is instantiated using the **CamelContext.createConsumerTemplate()** method and the consumer service thread is started by calling **ConsumerTemplate.start()**.

Example of polling message bodies

You can also poll a consumer endpoint for incoming message bodies using one of the following methods: blocking **receiveBody()**; **receiveBody()** with a timeout; or **receiveBodyNoWait()**, which returns immediately. As in the previous example, it is also essential to start the service thread by calling **start()** before you attempt to poll for exchanges.

The following example shows how to poll an incoming message body from the **seda:foo** consumer endpoint using the blocking **receiveBody()** method:

```
import org.apache.camel.ProducerTemplate;
import org.apache.camel.ConsumerTemplate;
...
ProducerTemplate template = context.createProducerTemplate();
ConsumerTemplate consumer = context.createConsumerTemplate();

// Start the consumer service
consumer.start();
...
template.sendBody("seda:foo", "Hello");
Object body = consumer.receiveBody("seda:foo");
...
// Stop the consumer service
consumer.stop();
```

Methods for polling exchanges

There are three basic methods for polling *exchanges* from a consumer endpoint: **receive()** without a timeout blocks indefinitely; **receive()** with a timeout blocks for the specified period of milliseconds; and **receiveNoWait()** is non-blocking. You can specify the consumer endpoint either as an endpoint URI or as an **Endpoint** instance.

```
Exchange receive(String endpointUri);
Exchange receive(String endpointUri, long timeout);
Exchange receiveNoWait(String endpointUri);

Exchange receive(Endpoint endpoint);
Exchange receive(Endpoint endpoint, long timeout);
Exchange receiveNoWait(Endpoint endpoint);
```

Methods for polling message bodies

There are three basic methods for polling *message bodies* from a consumer endpoint: **receiveBody()** without a timeout blocks indefinitely; **receiveBody()** with a timeout blocks for the specified period of milliseconds; and **receiveBodyNoWait()** is non-blocking. You can specify the consumer endpoint either as an endpoint URI or as an **Endpoint** instance. Moreover, by calling the templating forms of these methods, you can convert the returned body to a particular type, **T**, using a built-in type converter.

```
Object receiveBody(String endpointUri);
Object receiveBody(String endpointUri, long timeout);
Object receiveBodyNoWait(String endpointUri);

Object receiveBody(Endpoint endpoint);
Object receiveBody(Endpoint endpoint, long timeout);
Object receiveBodyNoWait(Endpoint endpoint);

<T> T receiveBody(String endpointUri, Class<T> type);
<T> T receiveBody(String endpointUri, long timeout, Class<T> type);
<T> T receiveBodyNoWait(String endpointUri, Class<T> type);

<T> T receiveBody(Endpoint endpoint, Class<T> type);
<T> T receiveBody(Endpoint endpoint, long timeout, Class<T> type);
<T> T receiveBodyNoWait(Endpoint endpoint, Class<T> type);
```

CHAPTER 47. IMPLEMENTING A COMPONENT

Abstract

This chapter provides a general overview of the approaches can be used to implement a Apache Camel component.

47.1. COMPONENT ARCHITECTURE

47.1.1. Factory Patterns for a Component

Overview

A Apache Camel component consists of a set of classes that are related to each other through a factory pattern. The primary entry point to a component is the **Component** object itself (an instance of **org.apache.camel.Component** type). You can use the **Component** object as a factory to create **Endpoint** objects, which in turn act as factories for creating **Consumer**, **Producer**, and **Exchange** objects. These relationships are summarized in Figure 47.1, "Component Factory Patterns"

Figure 47.1. Component Factory Patterns
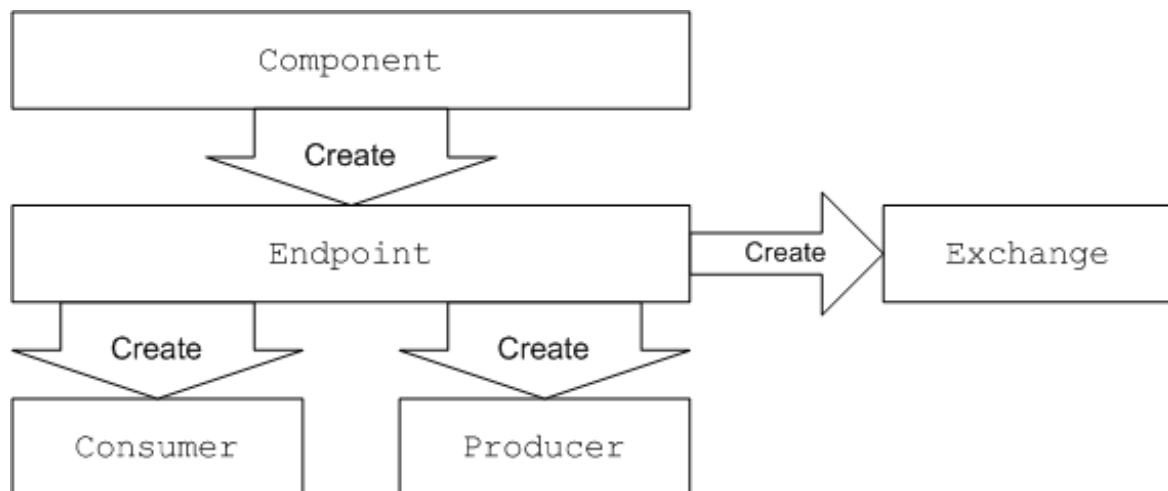
Component

A component implementation is an endpoint factory. The main task of a component implementor is to implement the **Component.createEndpoint()** method, which is responsible for creating new endpoints on demand.

Each kind of component must be associated with a *component prefix* that appears in an endpoint URI. For example, the file component is usually associated with the file prefix, which can be used in an endpoint URI like file://tmp/messages/input. When you install a new component in Apache Camel, you must define the association between a particular component prefix and the name of the class that implements the component.

Endpoint

Each endpoint instance encapsulates a particular endpoint URI. Every time Apache Camel encounters a new endpoint URI, it creates a new endpoint instance. An endpoint object is also a factory for creating consumer endpoints and producer endpoints.

Endpoints must implement the **org.apache.camel.Endpoint** interface. The **Endpoint** interface defines the following factory methods:

- **createConsumer()** and **createPollingConsumer()**—Creates a consumer endpoint, which represents the source endpoint at the beginning of a route.

- **createProducer()**—Creates a producer endpoint, which represents the target endpoint at the end of a route.

- **createExchange()**—Creates an exchange object, which encapsulates the messages passed up and down the route.

Consumer

Consumer endpoints *consume* requests. They always appear at the start of a route and they encapsulate the code responsible for receiving incoming requests and dispatching outgoing replies. From a service-oriented prospective a consumer represents a *service*.

Consumers must implement the **org.apache.camel.Consumer** interface. There are a number of different patterns you can follow when implementing a consumer. These patterns are described in Section 47.1.3, "Consumer Patterns and Threading".

Producer

Producer endpoints *produce* requests. They always appears at the end of a route and they encapsulate the code responsible for dispatching outgoing requests and receiving incoming replies. From a service-oriented prospective a producer represents a *service consumer*.

Producers must implement the **org.apache.camel.Producer** interface. You can optionally implement the producer to support an asynchronous style of processing. See Section 47.1.4, "Asynchronous Processing" for details.

Exchange

Exchange objects encapsulate a related set of messages. For example, one kind of message exchange is a synchronous invocation, which consists of a request message and its related reply.

Exchanges must implement the **org.apache.camel.Exchange** interface. The default implementation, **DefaultExchange**, is sufficient for many component implementations. However, if you want to associated extra data with the exchanges or have the exchanges preform additional processing, it can be useful to customize the exchange implementation.

Message

There are two different message slots in an **Exchange** object:

- *In* message—holds the current message.

- *Out* message—temporarily holds a reply message.

All of the message types are represented by the same Java object, **org.apache.camel.Message**. It is not always necessary to customize the message implementation—the default implementation, **DefaultMessage**, is usually adequate.

47.1.2. Using a Component in a Route

Overview

A Apache Camel route is essentially a pipeline of processors, of **org.apache.camel.Processor** type. Messages are encapsulated in an exchange object, **E**, which gets passed from node to node by invoking the **process()** method. The architecture of the processor pipeline is illustrated in Figure 47.2, "Consumer and Producer Instances in a Route".

Figure 47.2. Consumer and Producer Instances in a Route
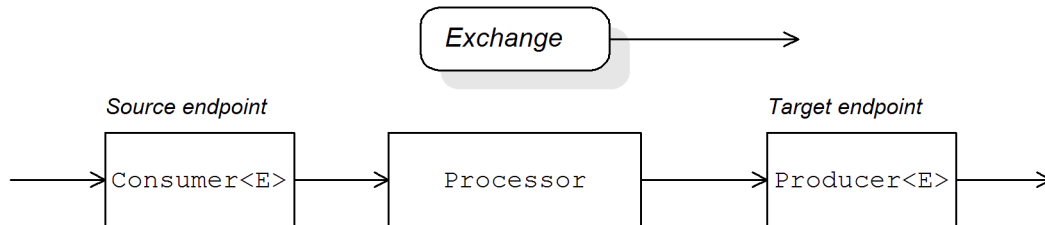
Source endpoint

At the start of the route, you have the source endpoint, which is represented by an **org.apache.camel.Consumer** object. The source endpoint is responsible for accepting incoming request messages and dispatching replies. When constructing the route, Apache Camel creates the appropriate **Consumer** type based on the component prefix from the endpoint URI, as described in Section 47.1.1, "Factory Patterns for a Component".

Processors

Each intermediate node in the pipeline is represented by a processor object (implementing the **org.apache.camel.Processor** interface). You can insert either standard processors (for example, **filter**, **throttler**, or **delayer**) or insert your own custom processor implementations.

Target endpoint

At the end of the route is the target endpoint, which is represented by an **org.apache.camel.Producer** object. Because it comes at the end of a processor pipeline, the producer is also a processor object (implementing the **org.apache.camel.Processor** interface). The target endpoint is responsible for sending outgoing request messages and receiving incoming replies. When constructing the route, Apache Camel creates the appropriate **Producer** type based on the component prefix from the endpoint URI.

47.1.3. Consumer Patterns and Threading

Overview

The pattern used to implement the consumer determines the threading model used in processing the incoming exchanges. Consumers can be implemented using one of the following patterns:

- Event-driven pattern—The consumer is driven by an external thread.

- Scheduled poll pattern —The consumer is driven by a dedicated thread pool.

- Polling pattern—The threading model is left undefined.

Event-driven pattern

In the event-driven pattern, the processing of an incoming request is initiated when another part of the application (typically a third-party library) calls a method implemented by the consumer. A good example of an event-driven consumer is the Apache Camel JMX component, where events are initiated by the JMX library. The JMX library calls the **handleNotification()** method to initiate request processing —see Example 50.4, "JMXConsumer Implementation" for details.

Figure 47.3, "Event-Driven Consumer" shows an outline of the event-driven consumer pattern. In this example, it is assumed that processing is triggered by a call to the ***notify()*** method.

Figure 47.3. Event-Driven Consumer
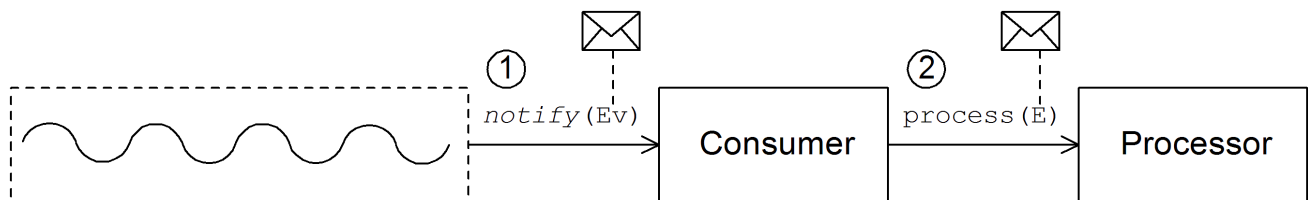
The event-driven consumer processes incoming requests as follows:

1. The consumer must implement a method to receive the incoming event (in Figure 47.3, "Event-Driven Consumer" this is represented by the ***notify()*** method). The thread that calls ***notify()*** is normally a separate part of the application, so the consumer's threading policy is externally driven.

   For example, in the case of the JMX consumer implementation, the consumer implements the **NotificationListener.handleNotification()** method to receive notifications from JMX. The threads that drive the consumer processing are created within the JMX layer.

2. In the body of the ***notify()*** method, the consumer first converts the incoming event into an exchange object, **E**, and then calls **process()** on the next processor in the route, passing the exchange object as its argument.

Scheduled poll pattern

In the scheduled poll pattern, the consumer retrieves incoming requests by checking at regular time intervals whether or not a request has arrived. Checking for requests is scheduled automatically by a built-in timer class, the *scheduled executor service*, which is a standard pattern provided by the java.util.concurrent library. The scheduled executor service executes a particular task at timed intervals and it also manages a pool of threads, which are used to run the task instances.

Figure 47.4, "Scheduled Poll Consumer" shows an outline of the scheduled poll consumer pattern.

Figure 47.4. Scheduled Poll Consumer
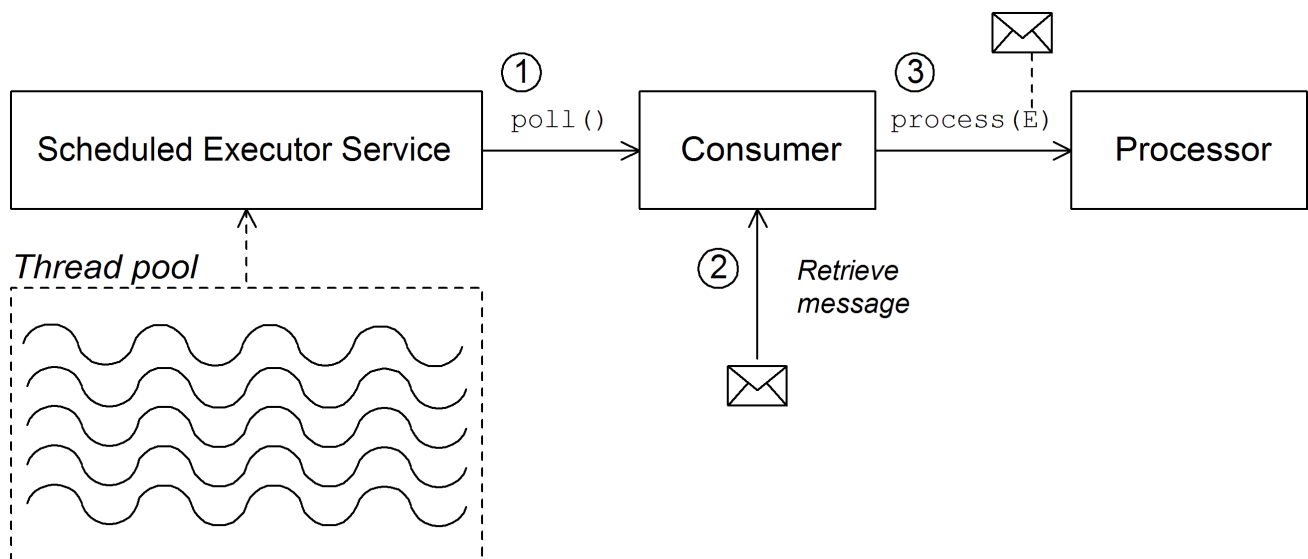
The scheduled poll consumer processes incoming requests as follows:

1. The scheduled executor service has a pool of threads at its disposal, that can be used to initiate consumer processing. After each scheduled time interval has elapsed, the scheduled executor service attempts to grab a free thread from its pool (there are five threads in the pool by default). If a free thread is available, it uses that thread to call the **poll()** method on the consumer.

2. The consumer's **poll()** method is intended to trigger processing of an incoming request. In the body of the **poll()** method, the consumer attempts to retrieve an incoming message. If no request is available, the **poll()** method returns immediately.

3. If a request message is available, the consumer inserts it into an exchange object and then calls **process()** on the next processor in the route, passing the exchange object as its argument.

Polling pattern

In the polling pattern, processing of an incoming request is initiated when a third-party calls one of the consumer's polling methods:

- **receive()**

- **receiveNoWait()**

- **receive(long timeout)**

It is up to the component implementation to define the precise mechanism for initiating calls on the polling methods. This mechanism is not specified by the polling pattern.

Figure 47.5, "Polling Consumer" shows an outline of the polling consumer pattern.

Figure 47.5. Polling Consumer
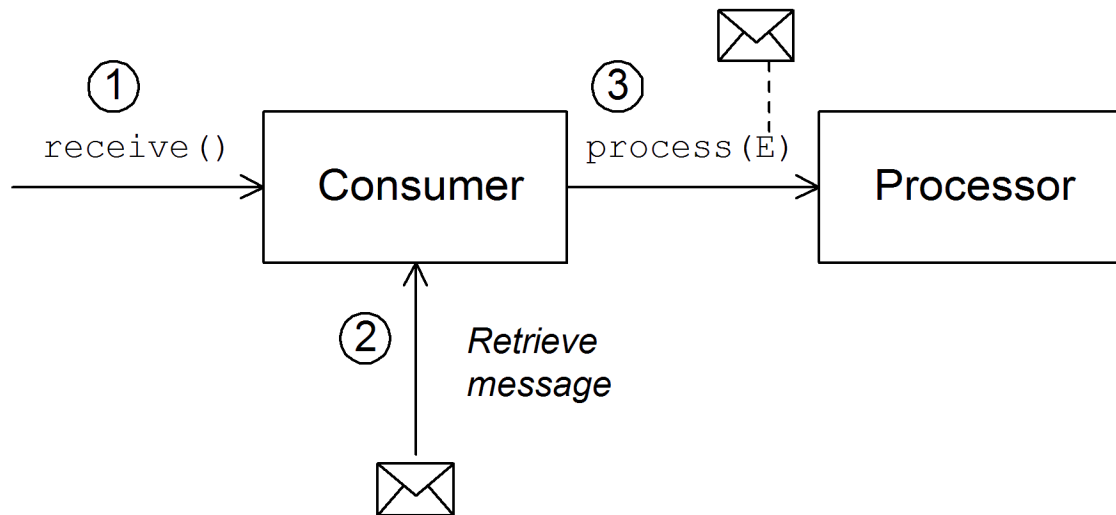
The polling consumer processes incoming requests as follows:

1. Processing of an incoming request is initiated whenever one of the consumer's polling methods is called. The mechanism for calling these polling methods is implementation defined.

2. In the body of the **receive()** method, the consumer attempts to retrieve an incoming request message. If no message is currently available, the behavior depends on which receive method was called.

    -   **receiveNoWait()** returns immediately

    -   **receive(long timeout)** waits for the specified timeout interval [3] before returning

    -   **receive()** waits until a message is received

3. If a request message is available, the consumer inserts it into an exchange object and then calls **process()** on the next processor in the route, passing the exchange object as its argument.

47.1.4. Asynchronous Processing

Overview

Producer endpoints normally follow a *synchronous* pattern when processing an exchange. When the preceding processor in a pipeline calls **process()** on a producer, the **process()** method blocks until a reply is received. In this case, the processor's thread remains blocked until the producer has completed the cycle of sending the request and receiving the reply.

Sometimes, however, you might prefer to decouple the preceding processor from the producer, so that the processor's thread is released immediately and the **process()** call does *not* block. In this case, you should implement the producer using an *asynchronous* pattern, which gives the preceding processor the option of invoking a non-blocking version of the **process()** method.

To give you an overview of the different implementation options, this section describes both the synchronous and the asynchronous patterns for implementing a producer endpoint.

Synchronous producer

Figure 47.6, "Synchronous Producer" shows an outline of a synchronous producer, where the preceding processor blocks until the producer has finished processing the exchange.

Figure 47.6. Synchronous Producer
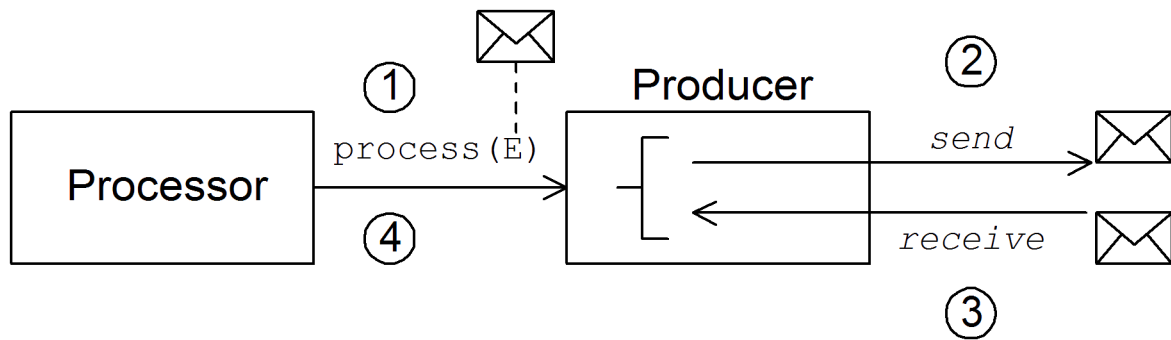
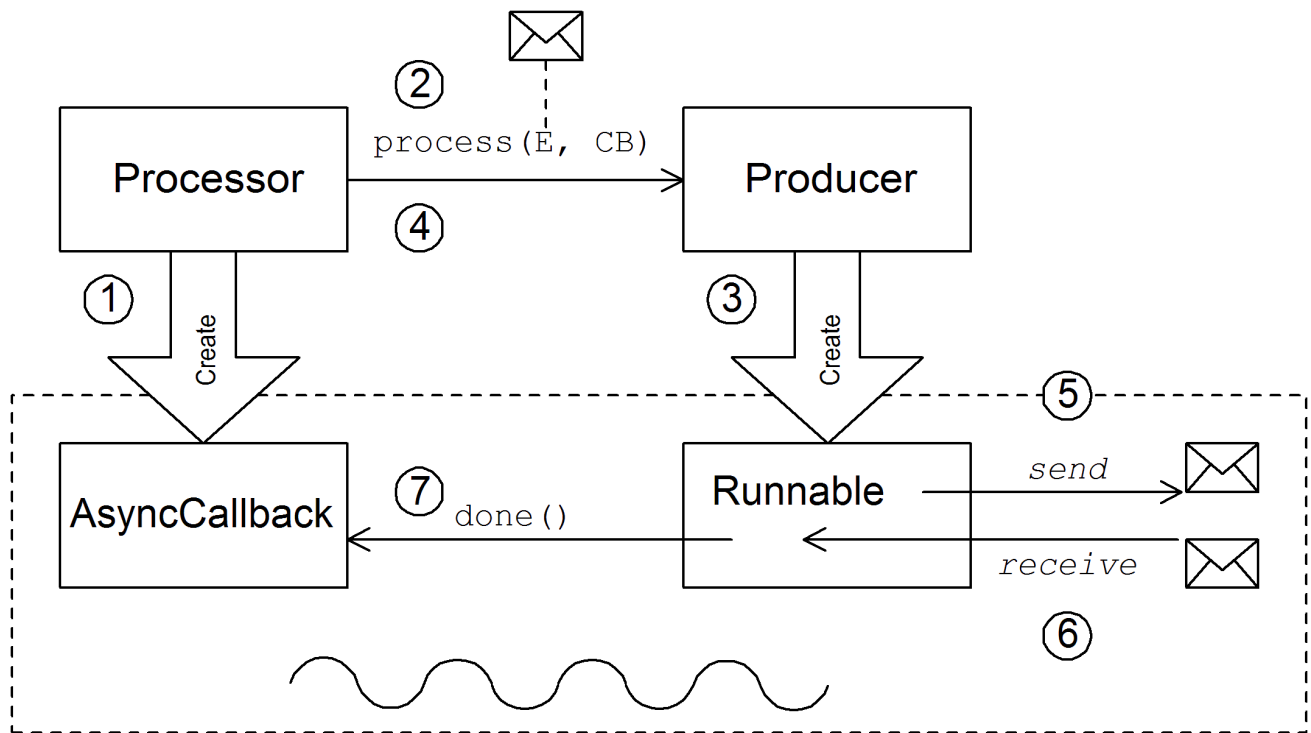The synchronous producer processes an exchange as follows:

1. The preceding processor in the pipeline calls the synchronous **process()** method on the producer to initiate synchronous processing. The synchronous **process()** method takes a single exchange argument.

2. In the body of the **process()** method, the producer sends the request (*In* message) to the endpoint.

3. If required by the exchange pattern, the producer waits for the reply (*Out* message) to arrive from the endpoint. This step can cause the **process()** method to block indefinitely. However, if the exchange pattern does not mandate a reply, the **process()** method can return immediately after sending the request.

4. When the **process()** method returns, the exchange object contains the reply from the synchronous call (an *Out* message message).

Asynchronous producer

Figure 47.7, "Asynchronous Producer" shows an outline of an asynchronous producer, where the producer processes the exchange in a sub-thread, and the preceding processor is not blocked for any significant length of time.

Figure 47.7. Asynchronous Producer

The asynchronous producer processes an exchange as follows:

1. Before the processor can call the asynchronous **process()** method, it must create an *asynchronous callback* object, which is responsible for processing the exchange on the return portion of the route. For the asynchronous callback, the processor must implement a class that inherits from the **AsyncCallback** interface.

2. The processor calls the asynchronous **process()** method on the producer to initiate asynchronous processing. The asynchronous **process()** method takes two arguments:

   - an exchange object

   - a synchronous callback object

3. In the body of the **process()** method, the producer creates a **Runnable** object that encapsulates the processing code. The producer then delegates the execution of this **Runnable** object to a sub-thread.

4. The asynchronous **process()** method returns, thereby freeing up the processor's thread. The exchange processing continues in a separate sub-thread.

5. The **Runnable** object sends the *In* message to the endpoint.

6. If required by the exchange pattern, the **Runnable** object waits for the reply (*Out* or *Fault* message) to arrive from the endpoint. The **Runnable** object remains blocked until the reply is received.

7. After the reply arrives, the **Runnable** object inserts the reply (*Out* message) into the exchange object and then calls **done()** on the asynchronous callback object. The asynchronous callback is then responsible for processing the reply message (executed in the sub-thread).

47.2. HOW TO IMPLEMENT A COMPONENT

Overview

This section gives a brief overview of the steps required to implement a custom Apache Camel component.

Which interfaces do you need to implement?

When implementing a component, it is usually necessary to implement the following Java interfaces:

- **org.apache.camel.Component**

- **org.apache.camel.Endpoint**

- **org.apache.camel.Consumer**

- **org.apache.camel.Producer**

In addition, it can also be necessary to implement the following Java interfaces:

- **org.apache.camel.Exchange**

- **org.apache.camel.Message**

Implementation steps

You typically implement a custom component as follows:

1. *Implement the* **Component** *interface*—A component object acts as an endpoint factory. You extend the **DefaultComponent** class and implement the **createEndpoint()** method.

   See Chapter 48, *Component Interface*.

2. *Implement the* **Endpoint** *interface*—An endpoint represents a resource identified by a specific URI. The approach taken when implementing an endpoint depends on whether the consumers follow an *event-driven* pattern, a *scheduled poll* pattern, or a *polling* pattern.

   For an event-driven pattern, implement the endpoint by extending the **DefaultEndpoint** class and implementing the following methods:

   - **createProducer()**

   - **createConsumer()**

   For a scheduled poll pattern, implement the endpoint by extending the **ScheduledPollEndpoint** class and implementing the following methods:

   - **createProducer()**

   - **createConsumer()**

   For a polling pattern, implement the endpoint by extending the **DefaultPollingEndpoint** class and implementing the following methods:

   - **createProducer()**

   - **createPollConsumer()**

See Chapter 49, *Endpoint Interface* .

3. *Implement the* **Consumer** *interface*—There are several different approaches you can take to implementing a consumer, depending on which pattern you need to implement (event-driven, scheduled poll, or polling). The consumer implementation is also crucially important for determining the threading model used for processing a message exchange.

   See Section 50.2, "Implementing the Consumer Interface" .

4. *Implement the* **Producer** *interface*—To implement a producer, you extend the **DefaultProducer** class and implement the **process()** method.

   See Chapter 51, *Producer Interface* .

5. *Optionally implement the Exchange or the Message interface* —The default implementations of **Exchange** and **Message** can be used directly, but occasionally, you might find it necessary to customize these types.

   See Chapter 52, *Exchange Interface* and Chapter 53, *Message Interface* .

Installing and configuring the component

You can install a custom component in one of the following ways:

- *Add the component directly to the CamelContext* —The **CamelContext.addComponent()** method adds a component programatically.

- *Add the component using Spring configuration* —The standard Spring **bean** element creates a component instance. The bean's **id** attribute implicitly defines the component prefix. For details, see Section 47.3.2, "Configuring a Component".

- *Configure Apache Camel to auto-discover the component*—Auto-discovery, ensures that Apache Camel automatically loads the component on demand. For details, see Section 47.3.1, "Setting Up Auto-Discovery".

47.3. AUTO-DISCOVERY AND CONFIGURATION

47.3.1. Setting Up Auto-Discovery

Overview

Auto-discovery is a mechanism that enables you to dynamically add components to your Apache Camel application. The component URI prefix is used as a key to load components on demand. For example, if Apache Camel encounters the endpoint URI, activemq://MyQName, and the ActiveMQ endpoint is not yet loaded, Apache Camel searches for the component identified by the activemq prefix and dynamically loads the component.

Availability of component classes

Before configuring auto-discovery, you must ensure that your custom component classes are accessible from your current classpath. Typically, you bundle the custom component classes into a JAR file, and add the JAR file to your classpath.

Configuring auto-discovery

To enable auto-discovery of your component, create a Java properties file named after the component prefix, *component-prefix*, and store that file in the following location:

> /META-INF/services/org/apache/camel/component/*component-prefix*

The *component-prefix* properties file must contain the following property setting:

> class=*component-class-name*

Where *component-class-name* is the fully-qualified name of your custom component class. You can also define additional system property settings in this file.

Example

For example, you can enable auto-discovery for the Apache Camel FTP component by creating the following Java properties file:

> /META-INF/services/org/apache/camel/component/ftp

Which contains the following Java property setting:

> class=org.apache.camel.component.file.remote.RemoteFileComponent

> **NOTE**
>
> The Java properties file for the FTP component is already defined in the JAR file, **camel-ftp-*Version*.jar**.

47.3.2. Configuring a Component

Overview

You can add a component by configuring it in the Apache Camel Spring configuration file, **META-INF/spring/camel-context.xml**. To find the component, the component's URI prefix is matched against the ID attribute of a **bean** element in the Spring configuration. If the component prefix matches a bean element ID, Apache Camel instantiates the referenced class and injects the properties specified in the Spring configuration.

> **NOTE**
>
> This mechanism has priority over auto-discovery. If the CamelContext finds a Spring bean with the requisite ID, it will not attempt to find the component using auto-discovery.

Define bean properties on your component class

If there are any properties that you want to inject into your component class, define them as bean properties. For example:

```
public class CustomComponent extends
  DefaultComponent<CustomExchange> {
    ...
```

```
    PropType getProperty() { ... }
    void setProperty(PropType v) { ...  }
}
```

The **get*Property*()** method and the **set*Property*()** method access the value of *property*.

Configure the component in Spring

To configure a component in Spring, edit the configuration file, **META-INF/spring/camel-context.xml**, as shown in .

Example 47.1. Configuring a Component in Spring

```xml
<?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-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">
   <package>RouteBuilderPackage</package>
  </camelContext>

  <bean id="component-prefix" class="component-class-name">
   <property name="property" value="propertyValue"/>
   </bean>
</beans>
```

The **bean** element with ID *component-prefix* configures the *component-class-name* component. You can inject properties into the component instance using **property** elements. For example, the **property** element in the preceding example would inject the value, *propertyValue*, into the *property* property by calling **set*Property*()** on the component.

Examples

shows an example of how to configure the Apache Camel's JMS component by defining a bean element with ID equal to **jms**. These settings are added to the Spring configuration file, **camel-context.xml**.

Example 47.2. JMS Component Spring Configuration

```xml
<?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-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">
    <package>org.apache.camel.example.spring</package>
</camelContext>

<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
            value="vm://localhost?broker.persistent=false&amp;broker.useJmx=false"/>
        </bean>
    </property>
  </bean>
</beans>
```

[1] The **CamelContext** automatically instantiates any **RouteBuilder** classes that it finds in the specified Java package, org.apache.camel.example.spring.

[2] The bean element with ID, **jms**, configures the JMS component. The bean ID corresponds to the component's URI prefix. For example, if a route specifies an endpoint with the URI, jms://MyQName, Apache Camel automatically loads the JMS component using the settings from the **jms** bean element.

[3] JMS is just a wrapper for a messaging service. You must specify the concrete implementation of the messaging system by setting the **connectionFactory** property on the **JmsComponent** class.

[4] In this example, the concrete implementation of the JMS messaging service is Apache ActiveMQ. The **brokerURL** property initializes a connection to an ActiveMQ broker instance, where the message broker is embedded in the local Java virtual machine (JVM). If a broker is not already present in the JVM, ActiveMQ will instantiate it with the options **broker.persistent=false** (the broker does not persist messages) and **broker.useJmx=false** (the broker does not open a JMX port).

---

[3] The timeout interval is typically specified in milliseconds.

CHAPTER 48. COMPONENT INTERFACE

Abstract

This chapter describes how to implement the **Component** interface.

48.1. THE COMPONENT INTERFACE

Overview

To implement a Apache Camel component, you must implement the **org.apache.camel.Component** interface. An instance of **Component** type provides the entry point into a custom component. That is, all of the other objects in a component are ultimately accessible through the **Component** instance. Figure 48.1, "Component Inheritance Hierarchy" shows the relevant Java interfaces and classes that make up the **Component** inheritance hierarchy.

Figure 48.1. Component Inheritance Hierarchy
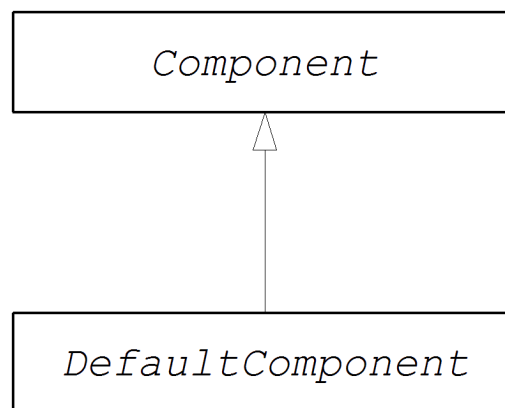
The Component interface

Example 48.1, "Component Interface" shows the definition of the **org.apache.camel.Component** interface.

Example 48.1. Component Interface

```
package org.apache.camel;

public interface Component {
    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    Endpoint createEndpoint(String uri) throws Exception;
}
```

Component methods

The **Component** interface defines the following methods:

- **getCamelContext()** and **setCamelContext()**—References the **CamelContext** to which this **Component** belongs. The **setCamelContext()** method is automatically called when you add the component to a **CamelContext**.

- **createEndpoint()**—The factory method that gets called to create **Endpoint** instances for this component. The **uri** parameter is the endpoint URI, which contains the details required to create the endpoint.

48.2. IMPLEMENTING THE COMPONENT INTERFACE

The DefaultComponent class

You implement a new component by extending the **org.apache.camel.impl.DefaultComponent** class, which provides some standard functionality and default implementations for some of the methods. In particular, the **DefaultComponent** class provides support for URI parsing and for creating a *scheduled executor* (which is used for the scheduled poll pattern).

URI parsing

The **createEndpoint(String uri)** method defined in the base **Component** interface takes a complete, unparsed endpoint URI as its sole argument. The **DefaultComponent** class, on the other hand, defines a three-argument version of the **createEndpoint()** method with the following signature:

```
protected abstract Endpoint createEndpoint(
    String uri,
    String remaining,
    Map parameters
)
throws Exception;
```

uri is the original, unparsed URI; **remaining** is the part of the URI that remains after stripping off the component prefix at the start and cutting off the query options at the end; and **parameters** contains the parsed query options. It is this version of the **createEndpoint()** method that you must override when inheriting from **DefaultComponent**. This has the advantage that the endpoint URI is already parsed for you.

The following sample endpoint URI for the **file** component shows how URI parsing works in practice:

```
file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
```

For this URI, the following arguments are passed to the three-argument version of **createEndpoint()**:

Argument	Sample Value
uri	file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
remaining	**/tmp/messages/foo**

Argument	Sample Value
parameters	Two entries are set in **java.util.Map**: • parameter **delete** is boolean **true** • parameter **moveNamePostfix** has the string value, **.old**.

Parameter injection

By default, the parameters extracted from the URI query options are injected on the endpoint's bean properties. The **DefaultComponent** class automatically injects the parameters for you.

For example, if you want to define a custom endpoint that supports two URI query options: **delete** and **moveNamePostfix**. All you must do is define the corresponding bean methods (getters and setters) in the endpoint class:

```java
public class FileEndpoint extends ScheduledPollEndpoint {
    ...
    public boolean isDelete() {
        return delete;
    }
    public void setDelete(boolean delete) {
        this.delete = delete;
    }
    ...
    public String getMoveNamePostfix() {
        return moveNamePostfix;
    }
    public void setMoveNamePostfix(String moveNamePostfix) {
        this.moveNamePostfix = moveNamePostfix;
    }
}
```

It is also possible to inject URI query options into *consumer* parameters. For details, see the section called "Consumer parameter injection".

Disabling endpoint parameter injection

If there are no parameters defined on your **Endpoint** class, you can optimize the process of endpoint creation by disabling endpoint parameter injection. To disable parameter injection on endpoints, override the **useIntrospectionOnEndpoint()** method and implement it to return **false**, as follows:

```java
protected boolean useIntrospectionOnEndpoint() {
  return false;
}
```

NOTE

The **useIntrospectionOnEndpoint()** method does *not* affect the parameter injection that might be performed on a **Consumer** class. Parameter injection at that level is controlled by the **Endpoint.configureProperties()** method (see Section 49.2, "Implementing the Endpoint Interface").

Scheduled executor service

The scheduled executor is used in the scheduled poll pattern, where it is responsible for driving the periodic polling of a consumer endpoint (a scheduled executor is effectively a thread pool implementation).

To instantiate a scheduled executor service, use the **ExecutorServiceStrategy** object that is returned by the **CamelContext.getExecutorServiceStrategy()** method. For details of the Apache Camel threading model, see Section 2.8, "Threading Model".

NOTE

Prior to Apache Camel 2.3, the **DefaultComponent** class provided a **getExecutorService()** method for creating thread pool instances. Since 2.3, however, the creation of thread pools is now managed centrally by the **ExecutorServiceStrategy** object.

Validating the URI

If you want to validate the URI before creating an endpoint instance, you can override the **validateURI()** method from the **DefaultComponent** class, which has the following signature:

protected void validateURI(String uri,
             String path,
             Map parameters)
  throws ResolveEndpointFailedException;

If the supplied URI does not have the required format, the implementation of **validateURI()** should throw the **org.apache.camel.ResolveEndpointFailedException** exception.

Creating an endpoint

Example 48.2, "Implementation of **createEndpoint()**" outlines how to implement the **DefaultComponent.createEndpoint()** method, which is responsible for creating endpoint instances on demand.

Example 48.2. Implementation of createEndpoint()

```
public class CustomComponent extends DefaultComponent {
    ...
    protected Endpoint createEndpoint(String uri, String remaining, Map parameters) throws
Exception {
        CustomEndpoint result = new CustomEndpoint(uri, this);
        // ...
    return result;
    }
}
```

1 The *CustomComponent* is the name of your custom component class, which is defined by extending the **DefaultComponent** class.

2 When extending **DefaultComponent**, you must implement the **createEndpoint()** method with three arguments (see the section called "URI parsing").

3 Create an instance of your custom endpoint type, *CustomEndpoint*, by calling its constructor. At a minimum, this constructor takes a copy of the original URI string, **uri**, and a reference to this component instance, **this**.

Example

Example 48.3, "FileComponent Implementation" shows a sample implementation of a **FileComponent** class.

Example 48.3. FileComponent Implementation

```
package org.apache.camel.component.file;

import org.apache.camel.CamelContext;
import org.apache.camel.Endpoint;
import org.apache.camel.impl.DefaultComponent;

import java.io.File;
import java.util.Map;

public class FileComponent extends DefaultComponent {
    public static final String HEADER_FILE_NAME = "org.apache.camel.file.name";

    public FileComponent() {
    }

    public FileComponent(CamelContext context) {
        super(context);
    }

    protected Endpoint createEndpoint(String uri, String remaining, Map parameters) throws
    Exception {
        File file = new File(remaining);
        FileEndpoint result = new FileEndpoint(file, uri, this);
        return result;
    }
}
```

1 Always define a no-argument constructor for the component class in order to facilitate automatic instantiation of the class.

2 A constructor that takes the parent **CamelContext** instance as an argument is convenient when creating a component instance by programming.

3 The implementation of the **FileComponent.createEndpoint()** method follows the pattern described in Example 48.2, "Implementation of **createEndpoint()**". The implementation creates a **FileEndpoint** object.

SynchronizationRouteAware Interface

SynchronizationRouteAware interface allows you to have callbacks before and after the exchange has been routed.

- **onBeforeRoute**: Invoked before the exchange has been routed by the given route. However, this callback may not get invoked, if you add the **SynchronizationRouteAware** implementation to the **UnitOfWork**, after starting the route.

- **onAfterRoute**: Invoked after the exchange has been routed by the given route. However, if the exchange is being routed through multiple routes, it would generate call backs for each route.

  This invocation occurs before these callbacks:

  1. The consumer of the route writes any response back to the caller (if in **InOut** mode)

  2. The **UnitOfWork** is done by calling either **Synchronization.onComplete(org.apache.camel.Exchange)** or **Synchronization.onFailure(org.apache.camel.Exchange)**

CHAPTER 49. ENDPOINT INTERFACE

Abstract

This chapter describes how to implement the **Endpoint** interface, which is an essential step in the implementation of a Apache Camel component.

49.1. THE ENDPOINT INTERFACE

Overview

An instance of **org.apache.camel.Endpoint** type encapsulates an endpoint URI, and it also serves as a factory for **Consumer**, **Producer**, and **Exchange** objects. There are three different approaches to implementing an endpoint:

- Event-driven

- scheduled poll

- polling

These endpoint implementation patterns complement the corresponding patterns for implementing a consumer—see Section 50.2, "Implementing the Consumer Interface" .

Figure 49.1, "Endpoint Inheritance Hierarchy" shows the relevant Java interfaces and classes that make up the **Endpoint** inheritance hierarchy.

Figure 49.1. Endpoint Inheritance Hierarchy
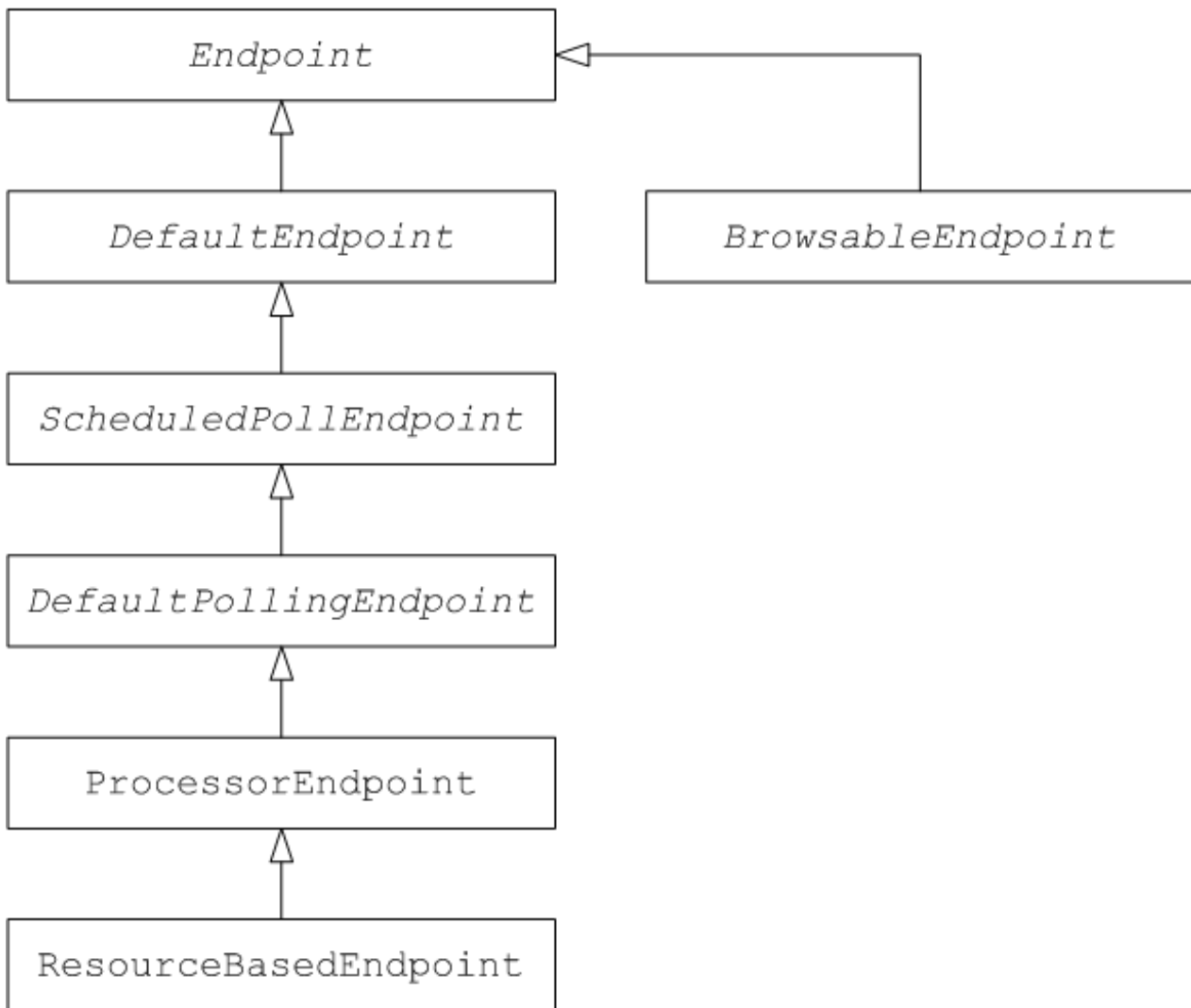
The Endpoint interface

Example 49.1, "Endpoint Interface" shows the definition of the **org.apache.camel.Endpoint** interface.

Example 49.1. Endpoint Interface

```
package org.apache.camel;

public interface Endpoint {
    boolean isSingleton();

    String getEndpointUri();

    String getEndpointKey();

    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    void configureProperties(Map options);

    boolean isLenientProperties();

    Exchange createExchange();
```

```
    Exchange createExchange(ExchangePattern pattern);
    Exchange createExchange(Exchange exchange);

    Producer createProducer() throws Exception;

    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

Endpoint methods

The **Endpoint** interface defines the following methods:

- **isSingleton()**—Returns **true**, if you want to ensure that each URI maps to a single endpoint within a CamelContext. When this property is **true**, multiple references to the identical URI within your routes always refer to a *single* endpoint instance. When this property is **false**, on the other hand, multiple references to the same URI within your routes refer to *distinct* endpoint instances. Each time you refer to the URI in a route, a new endpoint instance is created.

- **getEndpointUri()**—Returns the endpoint URI of this endpoint.

- **getEndpointKey()**—Used by **org.apache.camel.spi.LifecycleStrategy** when registering the endpoint.

- **getCamelContext()**—return a reference to the **CamelContext** instance to which this endpoint belongs.

- **setCamelContext()**—Sets the **CamelContext** instance to which this endpoint belongs.

- **configureProperties()**—Stores a copy of the parameter map that is used to inject parameters when creating a new **Consumer** instance.

- **isLenientProperties()**—Returns **true** to indicate that the URI is allowed to contain unknown parameters (that is, parameters that cannot be injected on the **Endpoint** or the **Consumer** class). Normally, this method should be implemented to return **false**.

- **createExchange()**—An overloaded method with the following variants:

  - **Exchange createExchange()**—Creates a new exchange instance with a default exchange pattern setting.

  - **Exchange createExchange(ExchangePattern pattern)**—Creates a new exchange instance with the specified exchange pattern.

  - **Exchange createExchange(Exchange exchange)**—Converts the given **exchange** argument to the type of exchange needed for this endpoint. If the given exchange is not already of the correct type, this method copies it into a new instance of the correct type. A default implementation of this method is provided in the **DefaultEndpoint** class.

- **createProducer()**—Factory method used to create new **Producer** instances.

- **createConsumer()**—Factory method to create new event-driven consumer instances. The **processor** argument is a reference to the first processor in the route.

- **createPollingConsumer()**—Factory method to create new polling consumer instances.

Endpoint singletons

In order to avoid unnecessary overhead, it is a good idea to create a *single* endpoint instance for all endpoints that have the same URI (within a CamelContext). You can enforce this condition by implementing **isSingleton()** to return **true**.

> **NOTE**
>
> In this context, *same URI* means that two URIs are the same when compared using string equality. In principle, it is possible to have two URIs that are equivalent, though represented by different strings. In that case, the URIs would not be treated as the same.

49.2. IMPLEMENTING THE ENDPOINT INTERFACE

Alternative ways of implementing an endpoint

The following alternative endpoint implementation patterns are supported:

- Event-driven endpoint implementation

- Scheduled poll endpoint implementation

- Polling endpoint implementation

Event-driven endpoint implementation

If your custom endpoint conforms to the event-driven pattern (see Section 47.1.3, "Consumer Patterns and Threading"), it is implemented by extending the abstract class, **org.apache.camel.impl.DefaultEndpoint**, as shown in Example 49.2, "Implementing DefaultEndpoint".

Example 49.2. Implementing DefaultEndpoint

```
import java.util.Map;
import java.util.concurrent.BlockingQueue;

import org.apache.camel.Component;
import org.apache.camel.Consumer;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;
import org.apache.camel.impl.DefaultExchange;

public class CustomEndpoint extends DefaultEndpoint {    ❶

    public CustomEndpoint(String endpointUri, Component component) {    ❷
        super(endpointUri, component);
        // Do any other initialization...
    }

    public Producer createProducer() throws Exception {    ❸
        return new CustomProducer(this);
    }
```

```
    public Consumer createConsumer(Processor processor) throws Exception {
        return new CustomConsumer(this, processor);
    }

    public boolean isSingleton() {
        return true;
    }

    // Implement the following methods, only if you need to set exchange properties.
    //
    public Exchange createExchange() {
        return this.createExchange(getExchangePattern());
    }

    public Exchange createExchange(ExchangePattern pattern) {
        Exchange result = new DefaultExchange(getCamelContext(), pattern);
        // Set exchange properties
        ...
        return result;
    }
}
```

(4) in code — `public Consumer createConsumer(...)`
(5) in code — `public Exchange createExchange() {`

1 Implement an event-driven custom endpoint, *CustomEndpoint*, by extending the **DefaultEndpoint** class.

2 You must have at least one constructor that takes the endpoint URI, **endpointUri**, and the parent component reference, **component**, as arguments.

3 Implement the **createProducer()** factory method to create producer endpoints.

4 Implement the **createConsumer()** factory method to create event-driven consumer instances.
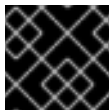
> **IMPORTANT**
>
> Do *not* override the **createPollingConsumer()** method.

5 In general, it is *not* necessary to override the **createExchange()** methods. The implementations inherited from **DefaultEndpoint** create a **DefaultExchange** object by default, which can be used in any Apache Camel component. If you need to initialize some exchange properties in the **DefaultExchange** object, however, it is appropriate to override the **createExchange()** methods here in order to add the exchange property settings.

The **DefaultEndpoint** class provides default implementations of the following methods, which you might find useful when writing your custom endpoint code:

- **getEndpointUri()**—Returns the endpoint URI.

- **getCamelContext()**—Returns a reference to the **CamelContext**.

- **getComponent()**—Returns a reference to the parent component.

- **createPollingConsumer()**—Creates a polling consumer. The created polling consumer's functionality is based on the event-driven consumer. If you override the event-driven consumer method, **createConsumer()**, you get a polling consumer implementation for free.

- **createExchange(Exchange e)**—Converts the given exchange object, **e**, to the type required for this endpoint. This method creates a new endpoint using the overridden **createExchange()** endpoints. This ensures that the method also works for custom exchange types.

Scheduled poll endpoint implementation

If your custom endpoint conforms to the scheduled poll pattern (see Section 47.1.3, "Consumer Patterns and Threading") it is implemented by inheriting from the abstract class, **org.apache.camel.impl.ScheduledPollEndpoint**, as shown in Example 49.3, "ScheduledPollEndpoint Implementation".

Example 49.3. ScheduledPollEndpoint Implementation

```java
import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.ExchangePattern;
import org.apache.camel.Message;
import org.apache.camel.impl.ScheduledPollEndpoint;

public class CustomEndpoint extends ScheduledPollEndpoint {          ❶

    protected CustomEndpoint(String endpointUri, CustomComponent component) {   ❷
        super(endpointUri, component);
        // Do any other initialization...
    }

    public Producer createProducer() throws Exception {             ❸
        Producer result = new CustomProducer(this);
        return result;
    }

    public Consumer createConsumer(Processor processor) throws Exception {   ❹
        Consumer result = new CustomConsumer(this, processor);
        configureConsumer(result);                                  ❺
        return result;
    }

    public boolean isSingleton() {
        return true;
    }

    // Implement the following methods, only if you need to set exchange properties.
    //
    public Exchange createExchange() {                              ❻
        return this.createExchange(getExchangePattern());
    }

    public Exchange createExchange(ExchangePattern pattern) {
        Exchange result = new DefaultExchange(getCamelContext(), pattern);
        // Set exchange properties
        ...
```

```
        return result;
    }
}
```

1 Implement a scheduled poll custom endpoint, *CustomEndpoint*, by extending the **ScheduledPollEndpoint** class.

2 You must to have at least one constructor that takes the endpoint URI, **endpointUri**, and the parent component reference, **component**, as arguments.

3 Implement the **createProducer()** factory method to create a producer endpoint.

4 Implement the **createConsumer()** factory method to create a scheduled poll consumer instance.
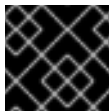
> ⬛ IMPORTANT
>
> Do *not* override the **createPollingConsumer()** method.

5 The **configureConsumer()** method, defined in the **ScheduledPollEndpoint** base class, is responsible for injecting consumer query options into the consumer. See the section called "Consumer parameter injection".

6 In general, it is *not* necessary to override the **createExchange()** methods. The implementations inherited from **DefaultEndpoint** create a **DefaultExchange** object by default, which can be used in any Apache Camel component. If you need to initialize some exchange properties in the **DefaultExchange** object, however, it is appropriate to override the **createExchange()** methods here in order to add the exchange property settings.

Polling endpoint implementation

If your custom endpoint conforms to the polling consumer pattern (see Section 47.1.3, "Consumer Patterns and Threading"), it is implemented by inheriting from the abstract class, **org.apache.camel.impl.DefaultPollingEndpoint**, as shown in Example 49.4, "DefaultPollingEndpoint Implementation".

Example 49.4. DefaultPollingEndpoint Implementation

```
import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.ExchangePattern;
import org.apache.camel.Message;
import org.apache.camel.impl.DefaultPollingEndpoint;

public class CustomEndpoint extends DefaultPollingEndpoint {
  ...
  public PollingConsumer createPollingConsumer() throws Exception {
    PollingConsumer result = new CustomConsumer(this);
    configureConsumer(result);
    return result;
  }
}
```

```
    // Do NOT implement createConsumer(). It is already implemented in DefaultPollingEndpoint.
    ...
}
```

Because this *CustomEndpoint* class is a polling endpoint, you must implement the **createPollingConsumer()** method instead of the **createConsumer()** method. The consumer instance returned from **createPollingConsumer()** must inherit from the **PollingConsumer** interface. For details of how to implement a polling consumer, see the section called "Polling consumer implementation" .

Apart from the implementation of the **createPollingConsumer()** method, the steps for implementing a **DefaultPollingEndpoint** are similar to the steps for implementing a **ScheduledPollEndpoint**. See Example 49.3, "ScheduledPollEndpoint Implementation" for details.

Implementing the BrowsableEndpoint interface

If you want to expose the list of exchange instances that are pending in the current endpoint, you can implement the **org.apache.camel.spi.BrowsableEndpoint** interface, as shown in Example 49.5, "BrowsableEndpoint Interface". It makes sense to implement this interface if the endpoint performs some sort of buffering of incoming events. For example, the Apache Camel SEDA endpoint implements the **BrowsableEndpoint** interface—see Example 49.6, "SedaEndpoint Implementation".

Example 49.5. BrowsableEndpoint Interface

```java
package org.apache.camel.spi;

import java.util.List;

import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;

public interface BrowsableEndpoint extends Endpoint {
    List<Exchange> getExchanges();
}
```

Example

Example 49.6, "SedaEndpoint Implementation" shows a sample implementation of **SedaEndpoint**. The SEDA endpoint is an example of an *event-driven endpoint*. Incoming events are stored in a FIFO queue (an instance of **java.util.concurrent.BlockingQueue**) and a SEDA consumer starts up a thread to read and process the events. The events themselves are represented by **org.apache.camel.Exchange** objects.

Example 49.6. SedaEndpoint Implementation

```java
package org.apache.camel.component.seda;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.BlockingQueue;
```

```
import org.apache.camel.Component;
import org.apache.camel.Consumer;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;
import org.apache.camel.spi.BrowsableEndpoint;
```

1 `public class SedaEndpoint extends DefaultEndpoint implements BrowsableEndpoint {`
```
    private BlockingQueue<Exchange> queue;
```

```
  public SedaEndpoint(String endpointUri, Component component, BlockingQueue<Exchange>
```
2 `queue) {`
```
        super(endpointUri, component);
      this.queue = queue;
  }
```

3   `public SedaEndpoint(String uri, SedaComponent component, Map parameters) {`
```
        this(uri, component, component.createQueue(uri, parameters));
  }
```

4   `public Producer createProducer() throws Exception {`
```
        return new CollectionProducer(this, getQueue());
  }
```

5   `public Consumer createConsumer(Processor processor) throws Exception {`
```
        return new SedaConsumer(this, processor);
  }
```

6   `public BlockingQueue<Exchange> getQueue() {`
```
        return queue;
  }
```

7   `public boolean isSingleton() {`
```
        return true;
  }
```

8   `public List<Exchange> getExchanges() {`
```
        return new ArrayList<Exchange>(getQueue());
  }
}
```

1 The **SedaEndpoint** class follows the pattern for implementing an event-driven endpoint by extending the **DefaultEndpoint** class. The **SedaEndpoint** class also implements the **BrowsableEndpoint** interface, which provides access to the list of exchange objects in the queue.

2 Following the usual pattern for an event-driven consumer, **SedaEndpoint** defines a constructor that takes an endpoint argument, **endpointUri**, and a component reference argument, **component**.

3 Another constructor is provided, which delegates queue creation to the parent component instance.

4 The **createProducer()** factory method creates an instance of **CollectionProducer**, which is a producer implementation that adds events to the queue.

⑤ The **createConsumer()** factory method creates an instance of **SedaConsumer**, which is responsible for pulling events off the queue and processing them.

⑥ The **getQueue()** method returns a reference to the queue.

⑦ The **isSingleton()** method returns **true**, indicating that a single endpoint instance should be created for each unique URI string.

⑧ The **getExchanges()** method implements the corresponding abstract method from **BrowsableEndpoint**.

CHAPTER 50. CONSUMER INTERFACE

Abstract

This chapter describes how to implement the **Consumer** interface, which is an essential step in the implementation of a Apache Camel component.

50.1. THE CONSUMER INTERFACE

Overview

An instance of **org.apache.camel.Consumer** type represents a source endpoint in a route. There are several different ways of implementing a consumer (see Section 47.1.3, "Consumer Patterns and Threading"), and this degree of flexibility is reflected in the inheritance hierarchy (see Figure 50.1, "Consumer Inheritance Hierarchy"), which includes several different base classes for implementing a consumer.

Figure 50.1. Consumer Inheritance Hierarchy
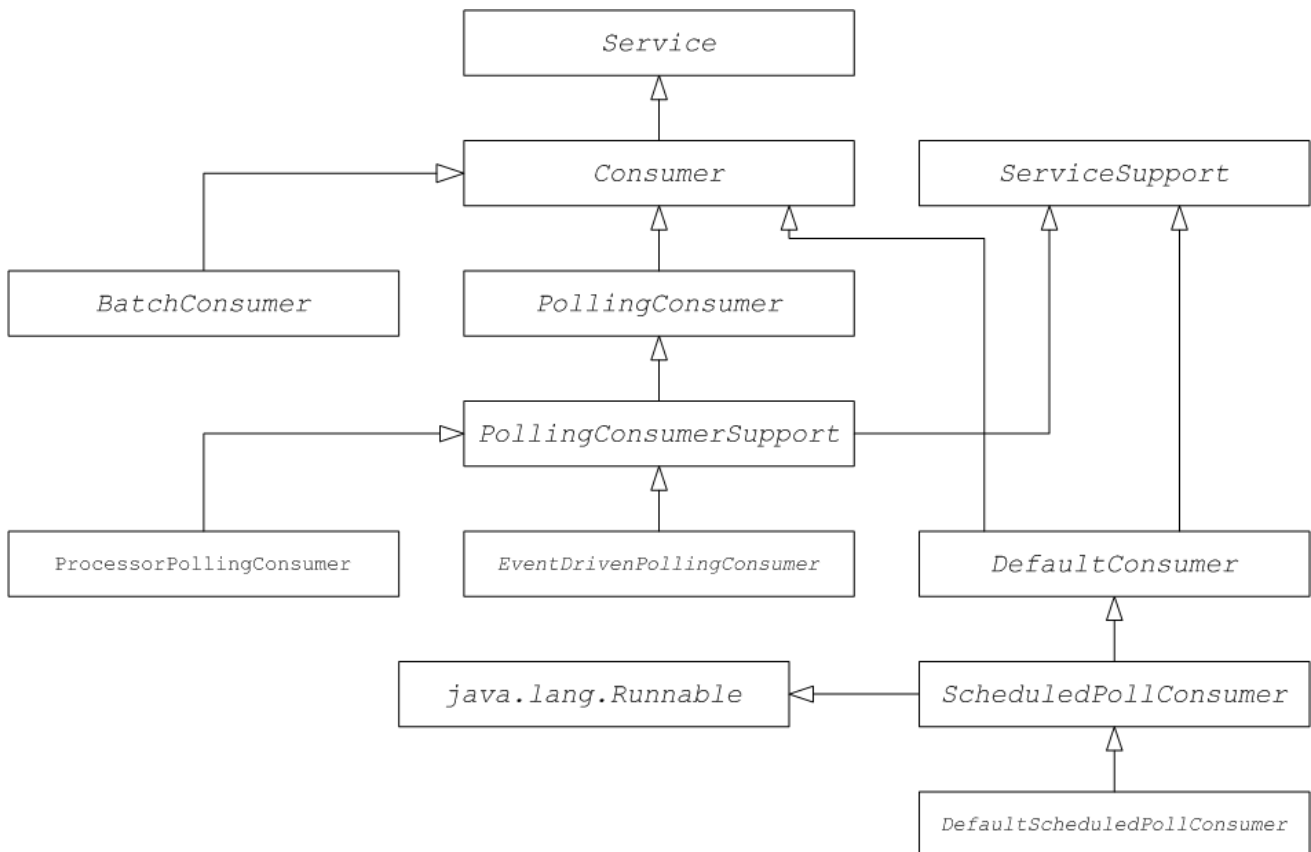
Consumer parameter injection

For consumers that follow the scheduled poll pattern (see the section called "Scheduled poll pattern"), Apache Camel provides support for injecting parameters into consumer instances. For example, consider the following endpoint URI for a component identified by the **custom** prefix:

```
custom:destination?consumer.myConsumerParam
```

Apache Camel provides support for automatically injecting query options of the form **consumer.***. For the **consumer.myConsumerParam** parameter, you need to define corresponding setter and getter methods on the **Consumer** implementation class as follows:

```
public class CustomConsumer extends ScheduledPollConsumer {
    ...
    String getMyConsumerParam() { ... }
    void setMyConsumerParam(String s) { ... }
    ...
}
```

Where the getter and setter methods follow the usual Java bean conventions (including capitalizing the first letter of the property name).

In addition to defining the bean methods in your Consumer implementation, you must also remember to call the **configureConsumer()** method in the implementation of **Endpoint.createConsumer()**. See the section called "Scheduled poll endpoint implementation"). Example 50.1, "FileEndpoint createConsumer() Implementation" shows an example of a **createConsumer()** method implementation, taken from the **FileEndpoint** class in the file component:

Example 50.1. FileEndpoint createConsumer() Implementation

```
...
public class FileEndpoint extends ScheduledPollEndpoint {
    ...
    public Consumer createConsumer(Processor processor) throws Exception {
        Consumer result = new FileConsumer(this, processor);
        configureConsumer(result);
        return result;
    }
    ...
}
```

At run time, consumer parameter injection works as follows:

1. When the endpoint is created, the default implementation of **DefaultComponent.createEndpoint(String uri)** parses the URI to extract the consumer parameters, and stores them in the endpoint instance by calling **ScheduledPollEndpoint.configureProperties()**.

2. When **createConsumer()** is called, the method implementation calls **configureConsumer()** to inject the consumer parameters (see Example 50.1, "FileEndpoint createConsumer() Implementation").

3. The **configureConsumer()** method uses Java reflection to call the setter methods whose names match the relevant options after the **consumer.** prefix has been stripped off.

Scheduled poll parameters

A consumer that follows the scheduled poll pattern automatically supports the consumer parameters shown in Table 50.1, "Scheduled Poll Parameters" (which can appear as query options in the endpoint URI).

Table 50.1. Scheduled Poll Parameters

Name	Default	Description
initialDelay	**1000**	Delay, in milliseconds, before the first poll.
delay	**500**	Depends on the value of the**useFixedDelay** flag (time unit is milliseconds).
useFixedDelay	**false**	If **false**, the **delay** parameter is interpreted as the polling period. Polls will occur at **initialDelay**, **initialDelay+delay**, **initialDelay+2*delay**, and so on.
		If **true**, the **delay** parameter is interpreted as the time elapsed between the previous execution and the next execution. Polls will occur at **initialDelay**, **initialDelay+[*ProcessingTime*]+delay**, and so on. Where *ProcessingTime* is the time taken to process an exchange object in the current thread.

Converting between event-driven and polling consumers

Apache Camel provides two special consumer implementations which can be used to convert back and forth between an event-driven consumer and a polling consumer. The following conversion classes are provided:

- **org.apache.camel.impl.EventDrivenPollingConsumer**—Converts an event-driven consumer into a polling consumer instance.

- **org.apache.camel.impl.DefaultScheduledPollConsumer**—Converts a polling consumer into an event-driven consumer instance.

In practice, these classes are used to simplify the task of implementing an **Endpoint** type. The **Endpoint** interface defines the following two methods for creating a consumer instance:

```
package org.apache.camel;

public interface Endpoint {
    ...
    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

createConsumer() returns an event-driven consumer and **createPollingConsumer()** returns a polling consumer. You would only implement one these methods. For example, if you are following the event-driven pattern for your consumer, you would implement the **createConsumer()** method provide a method implementation for **createPollingConsumer()** that simply raises an exception. With the help of the conversion classes, however, Apache Camel is able to provide a more useful default implementation.

For example, if you want to implement your consumer according to the event-driven pattern, you implement the endpoint by extending **DefaultEndpoint** and implementing the **createConsumer()** method. The implementation of **createPollingConsumer()** is inherited from **DefaultEndpoint**, where it is defined as follows:

```
public PollingConsumer<E> createPollingConsumer() throws Exception {
    return new EventDrivenPollingConsumer<E>(this);
}
```

The **EventDrivenPollingConsumer** constructor takes a reference to the event-driven consumer, **this**, effectively wrapping it and converting it into a polling consumer. To implement the conversion, the **EventDrivenPollingConsumer** instance buffers incoming events and makes them available on demand through the **receive()**, the **receive(long timeout)**, and the **receiveNoWait()** methods.

Analogously, if you are implementing your consumer according to the polling pattern, you implement the endpoint by extending **DefaultPollingEndpoint** and implementing the **createPollingConsumer()** method. In this case, the implementation of the **createConsumer()** method is inherited from **DefaultPollingEndpoint**, and the default implementation returns a **DefaultScheduledPollConsumer** instance (which converts the polling consumer into an event-driven consumer).

ShutdownPrepared interface

Consumer classes can optionally implement the **org.apache.camel.spi.ShutdownPrepared** interface, which enables your custom consumer endpoint to receive shutdown notifications.

Example 50.2, "ShutdownPrepared Interface" shows the definition of the **ShutdownPrepared** interface.

> **Example 50.2. ShutdownPrepared Interface**
>
> ```
> package org.apache.camel.spi;
>
>
> public interface ShutdownPrepared {
>
>     void prepareShutdown(boolean forced);
>
> }
> ```

The **ShutdownPrepared** interface defines the following methods:

prepareShutdown

    Receives notifications to shut down the consumer endpoint in one or two phases, as follows:

    1. *Graceful shutdown*—where the **forced** argument has the value **false**. Attempt to clean up resources gracefully. For example, by stopping threads gracefully.

    2. *Forced shutdown*—where the **forced** argument has the value **true**. This means that the shutdown has timed out, so you must clean up resources more aggressively. This is the last chance to clean up resources before the process exits.

ShutdownAware interface

Consumer classes can optionally implement the **org.apache.camel.spi.ShutdownAware** interface, which interacts with the graceful shutdown mechanism, enabling a consumer to ask for extra time to shut down. This is typically needed for components such as SEDA, which can have pending exchanges stored in an internal queue. Normally, you would want to process all of the exchanges in the queue before shutting down the SEDA consumer.

Example 50.3, "ShutdownAware Interface" shows the definition of the **ShutdownAware** interface.

> **Example 50.3. ShutdownAware Interface**

```Java
// Java
package org.apache.camel.spi;

import org.apache.camel.ShutdownRunningTask;

public interface ShutdownAware extends ShutdownPrepared {

    boolean deferShutdown(ShutdownRunningTask shutdownRunningTask);

    int getPendingExchangesSize();
}
```

The **ShutdownAware** interface defines the following methods:

deferShutdown

Return **true** from this method, if you want to delay shutdown of the consumer. The **shutdownRunningTask** argument is an **enum** which can take either of the following values:

- **ShutdownRunningTask.CompleteCurrentTaskOnly**—finish processing the exchanges that are currently being processed by the consumer's thread pool, but do not attempt to process any more exchanges than that.

- **ShutdownRunningTask.CompleteAllTasks**—process *all* of the pending exchanges. For example, in the case of the SEDA component, the consumer would process all of the exchanges from its incoming queue.

getPendingExchangesSize

Indicates how many exchanges remain to be processed by the consumer. A zero value indicates that processing is finished and the consumer can be shut down.

For an example of how to define the **ShutdownAware** methods, see Example 50.7, "Custom Threading Implementation".

50.2. IMPLEMENTING THE CONSUMER INTERFACE

Alternative ways of implementing a consumer

You can implement a consumer in one of the following ways:

- Event-driven consumer implementation

- Scheduled poll consumer implementation

- Polling consumer implementation

- Custom threading implementation

Event-driven consumer implementation

In an event-driven consumer, processing is driven explicitly by external events. The events are received through an event-listener interface, where the listener interface is specific to the particular event source.

Example 50.4, "JMXConsumer Implementation" shows the implementation of the **JMXConsumer** class, which is taken from the Apache Camel JMX component implementation. The **JMXConsumer** class is an example of an event-driven consumer, which is implemented by inheriting from the **org.apache.camel.impl.DefaultConsumer** class. In the case of the **JMXConsumer** example, events are represented by calls on the **NotificationListener.handleNotification()** method, which is a standard way of receiving JMX events. In order to receive these JMX events, it is necessary to implement the **NotificationListener** interface and override the **handleNotification()** method, as shown in Example 50.4, "JMXConsumer Implementation".

Example 50.4. JMXConsumer Implementation

```
package org.apache.camel.component.jmx;

import javax.management.Notification;
import javax.management.NotificationListener;
import org.apache.camel.Processor;
import org.apache.camel.impl.DefaultConsumer;

public class JMXConsumer extends DefaultConsumer implements NotificationListener {

    JMXEndpoint jmxEndpoint;

    public JMXConsumer(JMXEndpoint endpoint, Processor processor) {
        super(endpoint, processor);
        this.jmxEndpoint = endpoint;
    }

    public void handleNotification(Notification notification, Object handback) {
        try {
            getProcessor().process(jmxEndpoint.createExchange(notification));
        } catch (Throwable e) {
            handleException(e);
        }
    }
}
```

1 The **JMXConsumer** pattern follows the usual pattern for event-driven consumers by extending the **DefaultConsumer** class. Additionally, because this consumer is designed to receive events from JMX (which are represented by JMX notifications), it is necessary to implement the **NotificationListener** interface.

2 You must implement at least one constructor that takes a reference to the parent endpoint, **endpoint**, and a reference to the next processor in the chain, **processor**, as arguments.

3 The **handleNotification()** method (which is defined in **NotificationListener**) is automatically invoked by JMX whenever a JMX notification arrives. The body of this method should contain the code that performs the consumer's event processing. Because the **handleNotification()** call originates from the JMX layer, the consumer's threading model is implicitly controlled by the JMX layer, not by the **JMXConsumer** class.

> **NOTE**
>
> The **handleNotification()** method is specific to the JMX example. When implementing your own event-driven consumer, you must identify an analogous event listener method to implement in your custom consumer.

4 This line of code combines two steps. First, the JMX notification object is converted into an exchange object, which is the generic representation of an event in Apache Camel. Then the newly created exchange object is passed to the next processor in the route (invoked synchronously).

5 The **handleException()** method is implemented by the **DefaultConsumer** base class. By default, it handles exceptions using the **org.apache.camel.impl.LoggingExceptionHandler** class.

Scheduled poll consumer implementation

In a scheduled poll consumer, polling events are automatically generated by a timer class, **java.util.concurrent.ScheduledExecutorService**. To receive the generated polling events, you must implement the **ScheduledPollConsumer.poll()** method (see Section 47.1.3, "Consumer Patterns and Threading").

Example 50.5, "ScheduledPollConsumer Implementation" shows how to implement a consumer that follows the scheduled poll pattern, which is implemented by extending the **ScheduledPollConsumer** class.

Example 50.5. ScheduledPollConsumer Implementation

```
import java.util.concurrent.ScheduledExecutorService;

import org.apache.camel.Consumer;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.PollingConsumer;
import org.apache.camel.Processor;

import org.apache.camel.impl.ScheduledPollConsumer;

public class CustomConsumer extends ScheduledPollConsumer {
    private final CustomEndpoint endpoint;

    public CustomConsumer(CustomEndpoint endpoint, Processor processor) {
        super(endpoint, processor);
        this.endpoint = endpoint;
    }

    protected void poll() throws Exception {
        Exchange exchange = /* Receive exchange object ... */;

        // Example of a synchronous processor.
        getProcessor().process(exchange);
    }

    @Override
    protected void doStart() throws Exception {
```

```
        // Pre-Start:
        // Place code here to execute just before start of processing.
        super.doStart();
        // Post-Start:
        // Place code here to execute just after start of processing.
    }

    @Override
    protected void doStop() throws Exception {
        // Pre-Stop:
        // Place code here to execute just before processing stops.
        super.doStop();
        // Post-Stop:
        // Place code here to execute just after processing stops.
    }
}
```
❻

[1] Implement a scheduled poll consumer class, *CustomConsumer*, by extending the **org.apache.camel.impl.ScheduledPollConsumer** class.

[2] You must implement at least one constructor that takes a reference to the parent endpoint, **endpoint**, and a reference to the next processor in the chain, **processor**, as arguments.

[3] Override the **poll()** method to receive the scheduled polling events. This is where you should put the code that retrieves and processes incoming events (represented by exchange objects).

[4] In this example, the event is processed synchronously. If you want to process events asynchronously, you should use a reference to an asynchronous processor instead, by calling **getAsyncProcessor()**. For details of how to process events asynchronously, see Section 47.1.4, "Asynchronous Processing".

[5] *(Optional)* If you want some lines of code to execute as the consumer is starting up, override the **doStart()** method as shown.

[6] *(Optional)* If you want some lines of code to execute as the consumer is stopping, override the **doStop()** method as shown.

Polling consumer implementation

Example 50.6, "PollingConsumerSupport Implementation" outlines how to implement a consumer that follows the polling pattern, which is implemented by extending the **PollingConsumerSupport** class.

Example 50.6. PollingConsumerSupport Implementation

```
import org.apache.camel.Exchange;
import org.apache.camel.RuntimeCamelException;
import org.apache.camel.impl.PollingConsumerSupport;

public class CustomConsumer extends PollingConsumerSupport {
    private final CustomEndpoint endpoint;

    public CustomConsumer(CustomEndpoint endpoint) {
        super(endpoint);
```
[1]

[2]

```
        this.endpoint = endpoint;
    }

❸   public Exchange receiveNoWait() {
        Exchange exchange = /* Obtain an exchange object. */;
    // Further processing ...
    return exchange;
    }

❹   public Exchange receive() {
        // Blocking poll ...
    }

❺   public Exchange receive(long timeout) {
        // Poll with timeout ...
    }

❻   protected void doStart() throws Exception {
        // Code to execute whilst starting up.
    }

    protected void doStop() throws Exception {
        // Code to execute whilst shutting down.
    }
}
```

❶ Implement your polling consumer class, *CustomConsumer*, by extending the
   org.apache.camel.impl.PollingConsumerSupport class.

❷ You must implement at least one constructor that takes a reference to the parent endpoint,
   endpoint, as an argument. A polling consumer does not need a reference to a processor instance.

❸ The **receiveNoWait()** method should implement a non-blocking algorithm for retrieving an event
   (exchange object). If no event is available, it should return **null**.

❹ The **receive()** method should implement a blocking algorithm for retrieving an event. This method
   can block indefinitely, if events remain unavailable.

❺ The **receive(long timeout)** method implements an algorithm that can block for as long as the
   specified timeout (typically specified in units of milliseconds).

❻ If you want to insert code that executes while a consumer is starting up or shutting down,
   implement the **doStart()** method and the **doStop()** method, respectively.

Custom threading implementation

If the standard consumer patterns are not suitable for your consumer implementation, you can
implement the **Consumer** interface directly and write the threading code yourself. When writing the
threading code, however, it is important that you comply with the standard Apache Camel threading
model, as described in Section 2.8, "Threading Model".

For example, the SEDA component from **camel-core** implements its own consumer threading, which is
consistent with the Apache Camel threading model. Example 50.7, "Custom Threading Implementation"
shows an outline of how the **SedaConsumer** class implements its threading.

Example 50.7. Custom Threading Implementation

```java
package org.apache.camel.component.seda;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

import org.apache.camel.Consumer;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.ShutdownRunningTask;
import org.apache.camel.impl.LoggingExceptionHandler;
import org.apache.camel.impl.ServiceSupport;
import org.apache.camel.util.ServiceHelper;
...
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * A Consumer for the SEDA component.
 *
 * @version $Revision: 922485 $
 */
public class SedaConsumer extends ServiceSupport implements Consumer, Runnable,
ShutdownAware {
    private static final transient Log LOG = LogFactory.getLog(SedaConsumer.class);

    private SedaEndpoint endpoint;
    private Processor processor;
    private ExecutorService executor;
    ...
    public SedaConsumer(SedaEndpoint endpoint, Processor processor) {
        this.endpoint = endpoint;
        this.processor = processor;
    }
    ...

    public void run() {
        BlockingQueue<Exchange> queue = endpoint.getQueue();
    // Poll the queue and process exchanges
    ...
    }


    ...
    protected void doStart() throws Exception {
        int poolSize = endpoint.getConcurrentConsumers();
    executor = endpoint.getCamelContext().getExecutorServiceStrategy()
            .newFixedThreadPool(this, endpoint.getEndpointUri(), poolSize);
            for (int i = 0; i < poolSize; i++) {
                executor.execute(this);
            }
    }
    endpoint.onStarted(this);
```

1

2

3

4

5

```
        }

    6   protected void doStop() throws Exception {
            endpoint.onStopped(this);
        // must shutdown executor on stop to avoid overhead of having them running
    7       endpoint.getCamelContext().getExecutorServiceStrategy().shutdownNow(executor);
            executor = null;

        if (multicast != null) {
            ServiceHelper.stopServices(multicast);
        }
    }
    ...
    //----------
    // Implementation of ShutdownAware interface

    public boolean deferShutdown(ShutdownRunningTask shutdownRunningTask) {
        // deny stopping on shutdown as we want seda consumers to run in case some other queues
        // depend on this consumer to run, so it can complete its exchanges
        return true;
    }

    public int getPendingExchangesSize() {
        // number of pending messages on the queue
        return endpoint.getQueue().size();
    }

 }
```

1 The **SedaConsumer** class is implemented by extending the **org.apache.camel.impl.ServiceSupport** class and implementing the **Consumer**, **Runnable**, and **ShutdownAware** interfaces.

2 Implement the **Runnable.run()** method to define what the consumer does while it is running in a thread. In this case, the consumer runs in a loop, polling the queue for new exchanges and then processing the exchanges in the latter part of the queue.

3 The **doStart()** method is inherited from **ServiceSupport**. You override this method in order to define what the consumer does when it starts up.

4 Instead of creating threads directly, you should create a thread pool using the **ExecutorServiceStrategy** object that is registered with the **CamelContext**. This is important, because it enables Apache Camel to implement centralized management of threads and support such features as graceful shutdown.

For details, see Section 2.8, "Threading Model".

5 Kick off the threads by calling the **ExecutorService.execute()** method **poolSize** times.

6 The **doStop()** method is inherited from **ServiceSupport**. You override this method in order to define what the consumer does when it shuts down.

7 Shut down the thread pool, which is represented by the **executor** instance.
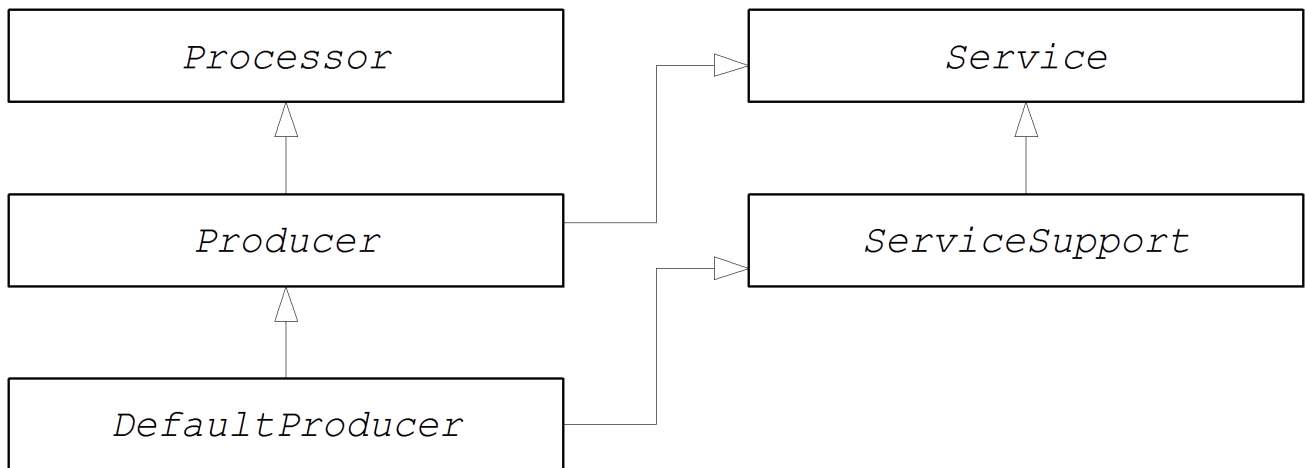
CHAPTER 51. PRODUCER INTERFACE

Abstract

This chapter describes how to implement the **Producer** interface, which is an essential step in the implementation of a Apache Camel component.

51.1. THE PRODUCER INTERFACE

Overview

An instance of **org.apache.camel.Producer** type represents a target endpoint in a route. The role of the producer is to send requests (*In* messages) to a specific physical endpoint and to receive the corresponding response (*Out* or *Fault* message). A **Producer** object is essentially a special kind of **Processor** that appears at the end of a processor chain (equivalent to a route). Figure 51.1, "Producer Inheritance Hierarchy" shows the inheritance hierarchy for producers.

Figure 51.1. Producer Inheritance Hierarchy

The Producer interface

Example 51.1, "Producer Interface" shows the definition of the **org.apache.camel.Producer** interface.

Example 51.1. Producer Interface

```
package org.apache.camel;

public interface Producer extends Processor, Service, IsSingleton {

    Endpoint<E> getEndpoint();

    Exchange createExchange();

    Exchange createExchange(ExchangePattern pattern);

    Exchange createExchange(E exchange);
}
```

Producer methods

The **Producer** interface defines the following methods:

- **process()** *(inherited from Processor)*—The most important method. A producer is essentially a special type of processor that sends a request to an endpoint, instead of forwarding the exchange object to another processor. By overriding the **process()** method, you define how the producer sends and receives messages to and from the relevant endpoint.

- **getEndpoint()**—Returns a reference to the parent endpoint instance.

- **createExchange()**—These overloaded methods are analogous to the corresponding methods defined in the **Endpoint** interface. Normally, these methods delegate to the corresponding methods defined on the parent **Endpoint** instance (this is what the **DefaultEndpoint** class does by default). Occasionally, you might need to override these methods.

Asynchronous processing

Processing an exchange object in a producer—which usually involves sending a message to a remote destination and waiting for a reply—can potentially block for a significant length of time. If you want to avoid blocking the current thread, you can opt to implement the producer as an *asynchronous processor*. The asynchronous processing pattern decouples the preceding processor from the producer, so that the **process()** method returns without delay. See Section 47.1.4, "Asynchronous Processing".

When implementing a producer, you can support the asynchronous processing model by implementing the **org.apache.camel.AsyncProcessor** interface. On its own, this is not enough to ensure that the asynchronous processing model will be used: it is also necessary for the preceding processor in the chain to call the asynchronous version of the **process()** method. The definition of the **AsyncProcessor** interface is shown in Example 51.2, "AsyncProcessor Interface".

> **Example 51.2. AsyncProcessor Interface**
>
> ```
> package org.apache.camel;
>
> public interface AsyncProcessor extends Processor {
>     boolean process(Exchange exchange, AsyncCallback callback);
> }
> ```

The asynchronous version of the **process()** method takes an extra argument, **callback**, of **org.apache.camel.AsyncCallback** type. The corresponding **AsyncCallback** interface is defined as shown in Example 51.3, "AsyncCallback Interface".

> **Example 51.3. AsyncCallback Interface**
>
> ```
> package org.apache.camel;
>
> public interface AsyncCallback {
>     void done(boolean doneSynchronously);
> }
> ```

The caller of **AsyncProcessor.process()** must provide an implementation of **AsyncCallback** to receive the notification that processing has finished. The **AsyncCallback.done()** method takes a boolean

argument that indicates whether the processing was performed synchronously or not. Normally, the flag would be **false**, to indicate asynchronous processing. In some cases, however, it can make sense for the producer *not* to process asynchronously (in spite of being asked to do so). For example, if the producer knows that the processing of the exchange will complete rapidly, it could optimise the processing by doing it synchronously. In this case, the **doneSynchronously** flag should be set to **true**.

ExchangeHelper class

When implementing a producer, you might find it helpful to call some of the methods in the **org.apache.camel.util.ExchangeHelper** utility class. For full details of the **ExchangeHelper** class, see Section 44.4, "The ExchangeHelper Class".

51.2. IMPLEMENTING THE PRODUCER INTERFACE

Alternative ways of implementing a producer

You can implement a producer in one of the following ways:

- How to implement a synchronous producer .

- How to implement an asynchronous producer .

How to implement a synchronous producer

Example 51.4, "DefaultProducer Implementation" outlines how to implement a synchronous producer. In this case, call to **Producer.process()** blocks until a reply is received.

Example 51.4. DefaultProducer Implementation

```
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultProducer;

① public class CustomProducer extends DefaultProducer {

②   public CustomProducer(Endpoint endpoint) {
        super(endpoint);
      // Perform other initialization tasks...
    }

③   public void process(Exchange exchange) throws Exception {
        // Process exchange synchronously.
      // ...
    }
}
```

① Implement a custom synchronous producer class, *CustomProducer*, by extending the **org.apache.camel.impl.DefaultProducer** class.

② Implement a constructor that takes a reference to the parent endpoint.

3 The **process()** method implementation represents the core of the producer code. The implementation of the **process()** method is entirely dependent on the type of component that you are implementing. In outline, the **process()** method is normally implemented as follows:

- If the exchange contains an *In* message, and if this is consistent with the specified exchange pattern, then send the *In* message to the designated endpoint.

- If the exchange pattern anticipates the receipt of an *Out* message, then wait until the *Out* message has been received. This typically causes the **process()** method to block for a significant length of time.

- When a reply is received, call **exchange.setOut()** to attach the reply to the exchange object. If the reply contains a fault message, set the fault flag on the *Out* message using **Message.setFault(true)**.

How to implement an asynchronous producer

Example 51.5, "CollectionProducer Implementation" outlines how to implement an asynchronous producer. In this case, you must implement both a synchronous **process()** method and an asynchronous **process()** method (which takes an additional **AsyncCallback** argument).

Example 51.5. CollectionProducer Implementation

```
import org.apache.camel.AsyncCallback;
import org.apache.camel.AsyncProcessor;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultProducer;
```

1 `public class CustomProducer extends DefaultProducer implements AsyncProcessor {`

2 `    public CustomProducer(Endpoint endpoint) {`
`        super(endpoint);`
`    // ...`
`    }`

3 `    public void process(Exchange exchange) throws Exception {`
`        // Process exchange synchronously.`
`    // ...`
`    }`

4 `    public boolean process(Exchange exchange, AsyncCallback callback) {`
`        // Process exchange asynchronously.`
`    CustomProducerTask task = new CustomProducerTask(exchange, callback);`
`    // Process 'task' in a separate thread...`
`    // ...`
5 `        return false;`
`    }`
`}`

6 `public class CustomProducerTask implements Runnable {`
`        private Exchange exchange;`
`    private AsyncCallback callback;`

```
        public CustomProducerTask(Exchange exchange, AsyncCallback callback) {
            this.exchange = exchange;
            this.callback = callback;
        }

❼      public void run() {
            // Process exchange.
        // ...
            callback.done(false);
        }
    }
```

❶ Implement a custom asynchronous producer class, *CustomProducer*, by extending the **org.apache.camel.impl.DefaultProducer** class, and implementing the **AsyncProcessor** interface.

❷ Implement a constructor that takes a reference to the parent endpoint.

❸ Implement the synchronous **process()** method.

❹ Implement the asynchronous **process()** method. You can implement the asynchronous method in several ways. The approach shown here is to create a **java.lang.Runnable** instance, **task**, that represents the code that runs in a sub-thread. You then use the Java threading API to run the task in a sub-thread (for example, by creating a new thread or by allocating the task to an existing thread pool).

❺ Normally, you return **false** from the asynchronous **process()** method, to indicate that the exchange was processed asynchronously.

❻ The *CustomProducer***Task** class encapsulates the processing code that runs in a sub-thread. This class must store a copy of the **Exchange** object, **exchange**, and the **AsyncCallback** object, **callback**, as private member variables.

❼ The **run()** method contains the code that sends the *In* message to the producer endpoint and waits to receive the reply, if any. After receiving the reply (*Out* message or *Fault* message) and inserting it into the exchange object, you must call **callback.done()** to notify the caller that processing is complete.

CHAPTER 52. EXCHANGE INTERFACE

Abstract

This chapter describes the **Exchange** interface. Since the refactoring of the camel-core module performed in Apache Camel 2.0, there is no longer any necessity to define custom exchange types. The **DefaultExchange** implementation can now be used in all cases.
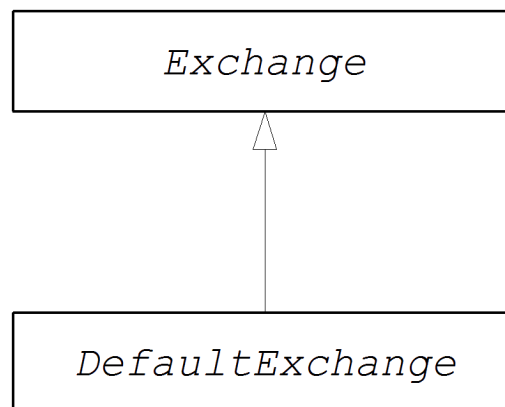
52.1. THE EXCHANGE INTERFACE

Overview

An instance of **org.apache.camel.Exchange** type encapsulates the current message passing through a route, with additional metadata encoded as exchange properties.

Figure 52.1, "Exchange Inheritance Hierarchy" shows the inheritance hierarchy for the exchange type. The default implementation, **DefaultExchange**, is always used.

Figure 52.1. Exchange Inheritance Hierarchy

The Exchange interface

Example 52.1, "Exchange Interface" shows the definition of the **org.apache.camel.Exchange** interface.

Example 52.1. Exchange Interface

```
package org.apache.camel;

import java.util.Map;

import org.apache.camel.spi.Synchronization;
import org.apache.camel.spi.UnitOfWork;

public interface Exchange {
    // Exchange property names (string constants)
    // (Not shown here)
    ...

    ExchangePattern getPattern();
    void setPattern(ExchangePattern pattern);
```

```
    Object getProperty(String name);
    Object getProperty(String name, Object defaultValue);
    <T> T  getProperty(String name, Class<T> type);
    <T> T  getProperty(String name, Object defaultValue, Class<T> type);
    void   setProperty(String name, Object value);
    Object removeProperty(String name);
    Map<String, Object> getProperties();
    boolean hasProperties();

    Message getIn();
    <T> T   getIn(Class<T> type);
    void    setIn(Message in);

    Message getOut();
    <T> T   getOut(Class<T> type);
    void    setOut(Message out);
    boolean hasOut();

    Throwable getException();
    <T> T     getException(Class<T> type);
    void      setException(Throwable e);

    boolean isFailed();

    boolean isTransacted();

    boolean isRollbackOnly();

    CamelContext getContext();

    Exchange copy();

    Endpoint getFromEndpoint();
    void     setFromEndpoint(Endpoint fromEndpoint);

    String getFromRouteId();
    void   setFromRouteId(String fromRouteId);

    UnitOfWork getUnitOfWork();
    void setUnitOfWork(UnitOfWork unitOfWork);

    String getExchangeId();
    void setExchangeId(String id);

    void addOnCompletion(Synchronization onCompletion);
    void handoverCompletions(Exchange target);
}
```

Exchange methods

The **Exchange** interface defines the following methods:

- **getPattern()**, **setPattern()**—The exchange pattern can be one of the values enumerated in **org.apache.camel.ExchangePattern**. The following exchange pattern values are supported:

  - **InOnly**

  - **RobustInOnly**

  - **InOut**

  - **InOptionalOut**

  - **OutOnly**

  - **RobustOutOnly**

  - **OutIn**

  - **OutOptionalIn**

- **setProperty()**, **getProperty()**, **getProperties()**, **removeProperty()**, **hasProperties()**—Use the property setter and getter methods to associate named properties with the exchange instance. The properties consist of miscellaneous metadata that you might need for your component implementation.

- **setIn()**, **getIn()**—Setter and getter methods for the *In* message.

  The **getIn()** implementation provided by the **DefaultExchange** class implements lazy creation semantics: if the *In* message is null when **getIn()** is called, the **DefaultExchange** class creates a default *In* message.

- **setOut()**, **getOut()**, **hasOut()**—Setter and getter methods for the *Out* message.

  The **getOut()** method implicitly supports lazy creation of an *Out* message. That is, if the current *Out* message is **null**, a new message instance is automatically created.

- **setException()**, **getException()**—Getter and setter methods for an exception object (of **Throwable** type).

- **isFailed()**—Returns **true**, if the exchange failed either due to an exception or due to a fault.

- **isTransacted()**—Returns **true**, if the exchange is transacted.

- **isRollback()**—Returns **true**, if the exchange is marked for rollback.

- **getContext()**—Returns a reference to the associated **CamelContext** instance.

- **copy()**—Creates a new, identical (apart from the exchange ID) copy of the current custom exchange object. The body and headers of the *In* message, the *Out* message (if any), and the *Fault* message (if any) are also copied by this operation.

- **setFromEndpoint()**, **getFromEndpoint()**—Getter and setter methods for the consumer endpoint that orginated this message (which is typically the endpoint appearing in the **from()** DSL command at the start of a route).

- **setFromRouteId()**, **getFromRouteId()**—Getters and setters for the route ID that originated this exchange. The **getFromRouteId()** method should only be called internally.

- **setUnitOfWork()**, **getUnitOfWork()**—Getter and setter methods for the **org.apache.camel.spi.UnitOfWork** bean property. This property is only required for exchanges that can participate in a transaction.

- **setExchangeId()**, **getExchangeId()**—Getter and setter methods for the exchange ID. Whether or not a custom component uses and exchange ID is an implementation detail.

- **addOnCompletion()**—Adds an **org.apache.camel.spi.Synchronization** callback object, which gets called when processing of the exchange has completed.

- **handoverCompletions()**—Hands over all of the *OnCompletion* callback objects to the specified exchange object.

CHAPTER 53. MESSAGE INTERFACE

Abstract

This chapter describes how to implement the **Message** interface, which is an optional step in the implementation of a Apache Camel component.

53.1. THE MESSAGE INTERFACE

Overview

An instance of **org.apache.camel.Message** type can represent any kind of message (*In* or *Out*). Figure 53.1, "Message Inheritance Hierarchy" shows the inheritance hierarchy for the message type. You do not always need to implement a custom message type for a component. In many cases, the default implementation, **DefaultMessage**, is adequate.

Figure 53.1. Message Inheritance Hierarchy
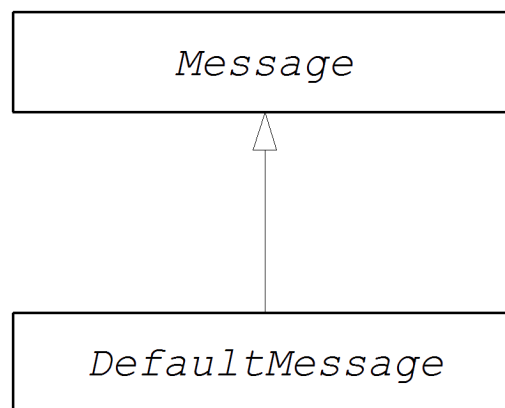
The Message interface

Example 53.1, "Message Interface" shows the definition of the **org.apache.camel.Message** interface.

Example 53.1. Message Interface

```
package org.apache.camel;

import java.util.Map;
import java.util.Set;

import javax.activation.DataHandler;

public interface Message {

    String getMessageId();
    void setMessageId(String messageId);

    Exchange getExchange();

    boolean isFault();
    void    setFault(boolean fault);
```

```
    Object getHeader(String name);
    Object getHeader(String name, Object defaultValue);
    <T> T getHeader(String name, Class<T> type);
    <T> T getHeader(String name, Object defaultValue, Class<T> type);
    Map<String, Object> getHeaders();
    void setHeader(String name, Object value);
    void setHeaders(Map<String, Object> headers);
    Object  removeHeader(String name);
    boolean removeHeaders(String pattern);
    boolean hasHeaders();

    Object getBody();
    Object getMandatoryBody() throws InvalidPayloadException;
    <T> T  getBody(Class<T> type);
    <T> T  getMandatoryBody(Class<T> type) throws InvalidPayloadException;
    void    setBody(Object body);
    <T> void setBody(Object body, Class<T> type);

    DataHandler getAttachment(String id);
    Map<String, DataHandler> getAttachments();
    Set<String> getAttachmentNames();
    void removeAttachment(String id);
    void addAttachment(String id, DataHandler content);
    void setAttachments(Map<String, DataHandler> attachments);
    boolean hasAttachments();

    Message copy();

    void copyFrom(Message message);

    String createExchangeId();
}
```

Message methods

The **Message** interface defines the following methods:

- **setMessageId()**, **getMessageId()**—Getter and setter methods for the message ID. Whether or not you need to use a message ID in your custom component is an implementation detail.

- **getExchange()**—Returns a reference to the parent exchange object.

- **isFault()**, **setFault()**—Getter and setter methods for the fault flag, which indicates whether or not this message is a fault message.

- **getHeader()**, **getHeaders()**, **setHeader()**, **setHeaders()**, **removeHeader()**, **hasHeaders()**—Getter and setter methods for the message headers. In general, these message headers can be used either to store actual header data, or to store miscellaneous metadata.

- **getBody()**, **getMandatoryBody()**, **setBody()**—Getter and setter methods for the message body. The getMandatoryBody() accessor guarantees that the returned body is non-null, otherwise the **InvalidPayloadException** exception is thrown.

- **getAttachment()**, **getAttachments()**, **getAttachmentNames()**, **removeAttachment()**, **addAttachment()**, **setAttachments()**, **hasAttachments()**—Methods to get, set, add, and remove attachments.

- **copy()**—Creates a new, identical (including the message ID) copy of the current custom message object.

- **copyFrom()**—Copies the complete contents (including the message ID) of the specified generic message object, **message**, into the current message instance. Because this method must be able to copy from *any* message type, it copies the generic message properties, but not the custom properties.

- **createExchangeId()**—Returns the unique ID for this exchange, if the message implementation is capable of providing an ID; otherwise, return **null**.

53.2. IMPLEMENTING THE MESSAGE INTERFACE

How to implement a custom message

Example 53.2, "Custom Message Implementation" outlines how to implement a message by extending the **DefaultMessage** class.

Example 53.2. Custom Message Implementation

```
import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultMessage;
```

❶ `public class CustomMessage extends DefaultMessage {`

❷ `    public CustomMessage() {`
`        // Create message with default properties...`
`    }`

`    @Override`
❸ `    public String toString() {`
`        // Return a stringified message...`
`    }`

`    @Override`
❹ `    public CustomMessage newInstance() {`
`        return new CustomMessage( ... );`
`    }`

`    @Override`
❺ `    protected Object createBody() {`
`        // Return message body (lazy creation).`
`    }`

`    @Override`
❻ `    protected void populateInitialHeaders(Map<String, Object> map) {`
`        // Initialize headers from underlying message (lazy creation).`
`    }`

`    @Override`
❼ `    protected void populateInitialAttachments(Map<String, DataHandler> map) {`

```
        // Initialize attachments from underlying message (lazy creation).
    }
}
```

[1] Implements a custom message class, *CustomMessage*, by extending the **org.apache.camel.impl.DefaultMessage** class.

[2] Typically, you need a default constructor that creates a message with default properties.

[3] Override the **toString()** method to customize message stringification.

[4] The **newInstance()** method is called from inside the **MessageSupport.copy()** method. Customization of the **newInstance()** method should focus on copying all of the *custom* properties of the current message instance into the new message instance. The **MessageSupport.copy()** method copies the generic message properties by calling **copyFrom()**.

[5] The **createBody()** method works in conjunction with the **MessageSupport.getBody()** method to implement lazy access to the message body. By default, the message body is **null**. It is only when the application code tries to access the body (by calling **getBody()**), that the body should be created. The **MessageSupport.getBody()** automatically calls **createBody()**, when the message body is accessed for the first time.

[6] The **populateInitialHeaders()** method works in conjunction with the header getter and setter methods to implement lazy access to the message headers. This method parses the message to extract any message headers and inserts them into the hash map, **map**. The **populateInitialHeaders()** method is automatically called when a user attempts to access a header (or headers) for the first time (by calling **getHeader()**, **getHeaders()**, **setHeader()**, or **setHeaders()**).

[7] The **populateInitialAttachments()** method works in conjunction with the attachment getter and setter methods to implement lazy access to the attachments. This method extracts the message attachments and inserts them into the hash map, **map**. The **populateInitialAttachments()** method is automatically called when a user attempts to access an attachment (or attachments) for the first time by calling **getAttachment()**, **getAttachments()**, **getAttachmentNames()**, or **addAttachment()**.

PART V. THE API COMPONENT FRAMEWORK

Abstract

How to create a Camel component that wraps any Java API, using the API Component Framework.

CHAPTER 54. INTRODUCTION TO THE API COMPONENT FRAMEWORK

Abstract

The API component framework helps you with the challenge of implementing complex Camel components based on a large Java API.

54.1. WHAT IS THE API COMPONENT FRAMEWORK?

Motivation

For components with a small number of options, the standard approach to implementing components (Chapter 47, *Implementing a Component*) is quite effective. Where it starts to become problematic, however, is when you need to implement a component with a large number of options. This problem becomes dramatic when it comes to enterprise-level components, which can require you to wrap an API consisting of *hundreds* of operations. Such components require a large effort to create and maintain.

The API component framework was developed precisely to deal with the challenge of implementing such components.

Turning APIs into components

Experience of implementing Camel components based on Java APIs has shown that a lot of the work is routine and mechanical. It consists of taking a particular Java method, mapping it to a particular URI syntax, and enabling the user to set the method parameters through URI options. This type of work is an obvious candidate for automation and code generation.

Generic URI format

The first step in automating the implementation of a Java API is to design a standard way of mapping an API method to a URI. For this we need to define a generic URI format, which can be used to wrap *any* Java API. Hence, the API component framework defines the following syntax for endpoint URIs:

> *scheme://endpoint-prefix/endpoint?Option1=Value1&...&OptionN=ValueN*

Where **scheme** is the default URI scheme defined by the component; **endpoint-prefix** is a short API name, which maps to one of the classes or interfaces from the wrapped Java API; **endpoint** maps to a method name; and the URI options map to method argument names.

URI format for a single API class

In the case where an API consists of just a single Java class, the **endpoint-prefix** part of the URI becomes redundant, and you can specify the URI in the following, shorter format:

> *scheme://endpoint?Option1=Value1&...&OptionN=ValueN*

> **NOTE**
>
> To enable this URI format, it is also necessary for the component implementor to leave the **apiName** element blank in the configuration of the API component Maven plug-in.

Reflection and metadata

In order to map Java method invocations to a URI syntax, it is obvious that some form of reflection mechanism is needed. But the standard Java reflection API suffers from a notable limitation: it does not preserve method argument names. This is a problem, because we need the method argument names in order to generate meaningful URI option names. The solution is to provide metadata in alternative format: either as Javadoc or in method signature files.

Javadoc

Javadoc is an ideal form of metadata for the API component framework, because it preserves the complete method signature, including method argument names. It is also easy to generate (particularly, using **maven-javadoc-plugin**) and, in many cases, is already provided in a third-party library.

Method signature files

If Javadoc is unavailable or unsuitable for some reason, the API component framework also supports an alternative source of metadata: the method signature files. A signature file is a simple text file which consists of a list of Java method signatures. It is relatively easy to create these files manually by copying and pasting from Java code (and lightly editing the resulting files).

What does the framework consist of?

From the perspective of a component developer, the API component framework consists of a number of different elements, as follows:

A Maven archetype

> The **camel-archetype-api-component** Maven archetype is used to generate skeleton code for the component implementation.

A Maven plug-in

> The **camel-api-component-maven-plugin** Maven plug-in is responsible for generating the code that implements the mapping between the Java API and the endpoint URI syntax.

Specialized base classes

> To support the programming model of the API component framework, the Apache Camel core provides a specialized API in the **org.apache.camel.util.component** package. Amongst other things, this API provides specialized base classes for the component, endpoint, consumer, and producer classes.

54.2. HOW TO USE THE FRAMEWORK

Overview

The procedure for implementing a component using the API framework involve a mixture of automated code generation, implementing Java code, and customizing the build, by editing Maven POM files. FIG gives an overview of this development process.

Figure 54.1. Using the API Component Framework
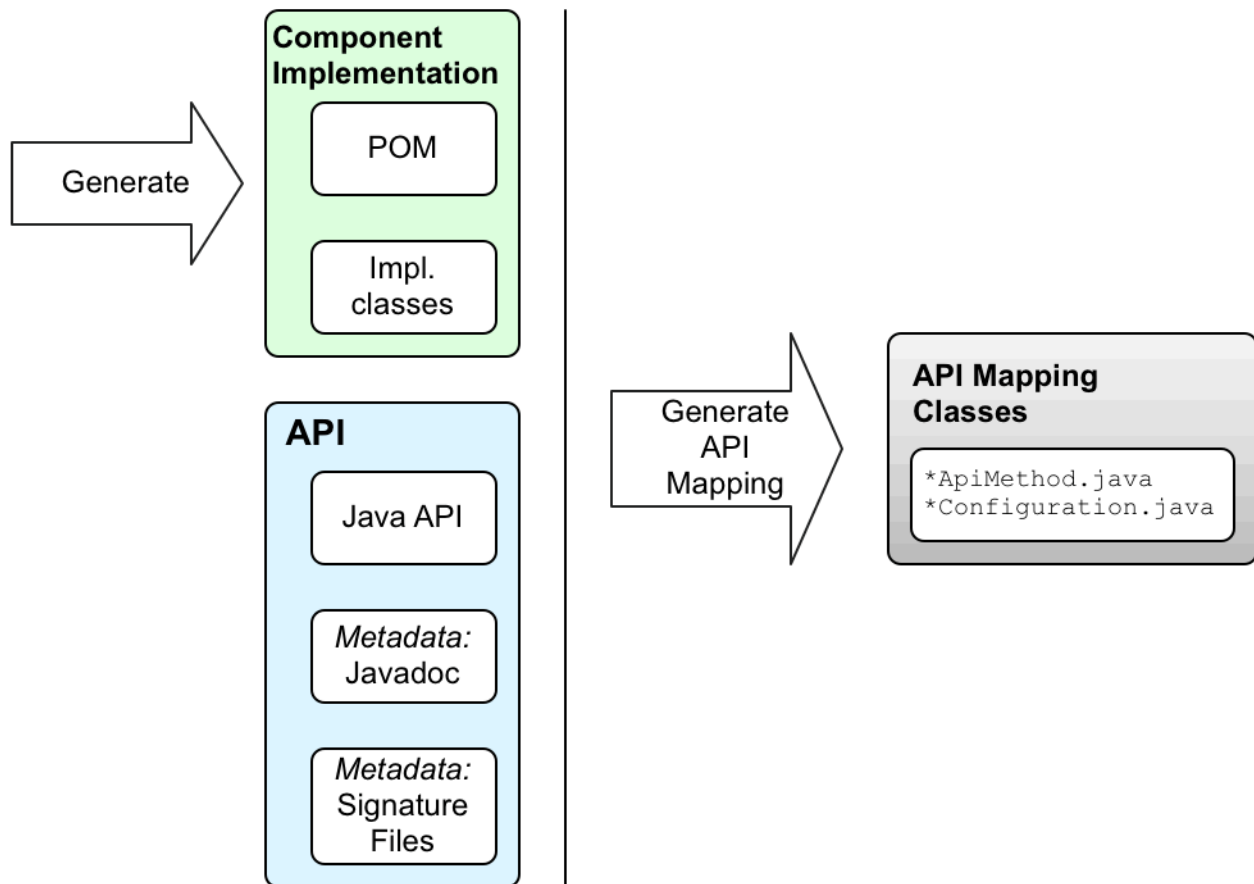
Java API

The starting point for your API component is always a Java API. Generally speaking, in the context of Camel, this usually means a Java client API, which connects to a remote server endpoint. The first question is, where does the Java API come from? Here are a few possibilities:

- Implement the Java API yourself (though this typically would involve a lot of work and is generally not the preferred approach).

- Use a third-party Java API. For example, the Apache Camel Box component is based on the third-party Box Java SDK library.

- Generate the Java API from a language-neutral interface. For example, the Apache Camel LinkedIn component obtains its Java API by converting a WADL description of its REST services to Java (using the Apache CXF **wadl2java** tool).

Javadoc metadata

You have the option of providing metadata for the Java API in the form of Javadoc (which is needed for generating code in the API component framework). If you use a third-party Java API from a Maven repository, you will usually find that the Javadoc is already provided in the Maven artifact. But even in the cases where Javadoc is *not* provided, you can easily generate it, using the **maven-javadoc-plugin** Maven plug-in.

NOTE

Currently, there is a limitation in the processing of Javadoc metadata, such that generic nesting is not supported. For example, **java.util.List<String>** is supported, but **java.util.List<java.util.List<String>>** is not. The workaround is to specify the nested generic type as **java.util.List<java.util.List>** in a signature file.

Signature file metadata

If for some reason it is not convenient to provide Java API metadata in the form of Javadoc, you have the option of providing metadata in the form of *signature files*. The signature files consist of a list of method signatures (one method signature per line). These files can be created manually and are needed only at build time.

Note the following points about signature files:

- You must create one signature file for each proxy class (Java API class).

- The method signatures must *not* include a **raises** clause. All exceptions raised at runtime are wrapped in a **RuntimeCamelException** and returned from the endpoint.

- Class names that specify the type of an argument must be fully-qualified class names (except for the **java.lang.*** types). There is no mechanism for importing package names.

- Currently, there is a limitation in the signature parser, such that generic nesting is not supported. For example, **java.util.List<String>** is supported, whereas **java.util.List<java.util.List<String>>** is not. The workaround is to specify the nested generic type as **java.util.List<java.util.List>**.

The following shows a simple example of the contents of a signature file:

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

Generate starting code with the Maven archetype

The easiest way to get started developing an API component is to generate an initial Maven project using the **camel-archetype-api-component** Maven archetype. For details of how to run the archetype, see Section 55.1, "Generate Code with the Maven Archetype".

After you run the Maven archetype, you will find two sub-projects under the generated *ProjectName* directory:

ProjectName-api

This project contains the Java API, which forms the basis of the API component. When you build this project, it packages up the Java API in a Maven bundle and generates the requisite Javadoc as well. If the Java API and Javadoc are already provided by a third-party, however, you do not need this sub-project.

ProjectName-component

This project contains the skeleton code for the API component.

Edit component classes

You can edit the skeleton code in ***ProjectName*-component** to develop your own component implementation. The following generated classes make up the core of the skeleton implementation:

> *ComponentName*Component
> *ComponentName*Endpoint
> *ComponentName*Consumer
> *ComponentName*Producer
> *ComponentName*Configuration

Customize POM files

You also need to edit the Maven POM files to customize the build, and to configure the **camel-api-component-maven-plugin** Maven plug-in.

Configure the camel-api-component-maven-plugin

The most important aspect of configuring the POM files is the configuration of the **camel-api-component-maven-plugin** Maven plug-in. This plug-in is responsible for generating the mapping between API methods and endpoint URIs, and by editing the plug-in configuration, you can customize the mapping.

For example, the following **camel-api-component-maven-plugin** plug-in configuration shows a minimal configuration for an API class called **ExampleJavadocHello**:

```
<configuration>
  <apis>
   <api>
     <apiName>hello-javadoc</apiName>
     <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
     <fromJavadoc/>
   </api>
  </apis>
</configuration>
```

In this example, the **hello-javadoc** API name is mapped to the **ExampleJavadocHello** class, which means you can invoke methods from this class using URIs of the form, ***scheme*://hello-javadoc/*endpoint***. The presence of the **fromJavadoc** element indicates that the **ExampleJavadocHello** class gets its metadata from Javadoc.

OSGi bundle configuration

The sample POM for the component sub-project, ***ProjectName*-component/pom.xml**, is configured to package the component as an OSGi bundle. The component POM includes a sample configuration of the **maven-bundle-plugin**. You should customize the configuration of the **maven-bundle-plugin** plug-in, to ensure that Maven generates a properly configured OSGi bundle for your component.

Build the component

When you build the component with Maven, the **camel-api-component-maven-plugin** plug-in automatically generates the API mapping classes (which define the mapping between the Java API and the endpoint URI syntax), placing them into the **target/generated-classes** project subdirectory. When

you are dealing with a large and complex Java API, this generated code actually constitutes the bulk of the component source code.

When the Maven build completes, the compiled code and resources are packaged up as an OSGi bundle and stored in your local Maven repository as a Maven artifact.

CHAPTER 55. GETTING STARTED WITH THE FRAMEWORK

Abstract

This chapter explains the basic principles of implementing a Camel component using the API component framework, based on code generated using the **camel-archetype-api-component** Maven archetype.

55.1. GENERATE CODE WITH THE MAVEN ARCHETYPE

Maven archetypes

A Maven archetype is analogous to a code wizard: given a few simple parameters, it generates a complete, working Maven project, populated with sample code. You can then use this project as a template, customizing the implementation to create your own application.

The API component Maven archetype

The API component framework provides a Maven archetype, **camel-archetype-api-component**, that can generate starting point code for your own API component implementation. This is the recommended approach to start creating your own API component.

Prerequisites

The only prerequisites for running the **camel-archetype-api-component** archetype are that Apache Maven is installed and the Maven **settings.xml** file is configured to use the standard JBoss Fuse repositories. For more details, see appendix "Red Hat AMQ Maven Repositories" in "Installation on Apache Karaf".

Invoke the Maven archetype

To create an **Example** component, which uses the **example** URI scheme, invoke the **camel-archetype-api-component** archetype to generate a new Maven project, as follows:

```
mvn archetype:generate \
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-api-component \
-DarchetypeVersion=2.17.0.redhat-630xxx \
-DgroupId=org.jboss.fuse.example \
-DartifactId=camel-api-example \
-Dname=Example \
-Dscheme=example \
-Dversion=1.0-SNAPSHOT \
-DinteractiveMode=false
```

> **NOTE**
>
> The backslash character, \, at the end of each line represents line continuation, which works only on Linux and UNIX platforms. On Windows platforms, remove the backslash and put the arguments all on a single line.

Options

Options are provided to the archetype generation command using the syntax, **-D***Name***=***Value*. Most of the options should be set as shown in the preceding **mvn archetype:generate** command, but a few of the options can be modified, to customize the generated project. The following table shows the options that you can use to customize the generated API component project:

Name	Description
groupId	*(Generic Maven option)* Specifies the group ID of the generated Maven project. By default, this value also defines the Java package name for the generated classes. Hence, it is a good idea to choose this value to match the Java package name that you want.
artifactId	*(Generic Maven option)* Specifies the artifact ID of the generated Maven project.
name	The name of the API component. This value is used for generating class names in the generated code (hence, it is recommended that the name should start with a capital letter).
scheme	The default scheme to use in URIs for this component. You should make sure that this scheme does not conflict with the scheme of any existing Camel components.
archetypeVersion	*(Generic Maven option)* Ideally, this should be the Apache Camel version used by the container where you plan to deploy the component. If necessary, however, you can also modify the versions of Maven dependencies *after* you have generated the project.

Structure of the generated project

Assuming that the code generation step completes successfully, you should see a new directory, **camel-api-example**, which contains the new Maven project. If you look inside the **camel-api-example** directory, you will see that it has the following general structure:

```
camel-api-example/
    pom.xml
    camel-api-example-api/
    camel-api-example-component/
```

At the top level of the project is an aggregate POM, **pom.xml**, which is configured to build two sub-projects, as follows:

camel-api-example-api

The API sub-project (named as ***ArtifactId*-api**) holds the Java API which you are about to turn into a component. If you are basing the API component on a Java API that you wrote yourself, you can put the Java API code directly into this project.

The API sub-project can be used for one or more of the following purposes:

- To package up the Java API code (if it is not already available as a Maven package).

- To generate Javadoc for the Java API (providing the needed metadata for the API component framework).

- To generate the Java API code from an API description (for example, from a WADL description of a REST API).

In some cases, however, you might not need to perform any of these tasks. For example, if the API component is based on a third-party API, which already provides the Java API and Javadoc in a Maven package. In such cases, you can delete the API sub-project.

camel-api-example-component

The component sub-project (named as ***ArtifactId*-component**) holds the implementation of the new API component. This includes the component implementation classes and the configuration of the **camel-api-component-maven** plug-in (which generates the API mapping classes from the Java API).

55.2. GENERATED API SUB-PROJECT

Overview

Assuming that you generated a new Maven project as described in Section 55.1, "Generate Code with the Maven Archetype", you can now find a Maven sub-project for packaging the Java API under the **camel-api-example/camel-api-example-api** project directory. In this section, we take a closer look at the generated example code and describe how it works.

Sample Java API

The generated example code includes a sample Java API, on which the example API component is based. The sample Java API is relatively simple, consisting of just two Hello World classes: **ExampleJavadocHello** and **ExampleFileHello**.

ExampleJavadocHello class

Example 55.1, "ExampleJavadocHello class" shows the **ExampleJavadocHello** class from the sample Java API. As the name of the class suggests, this particular class is used to show how you can supply mapping metadata from Javadoc.

Example 55.1. ExampleJavadocHello class

```java
// Java
package org.jboss.fuse.example.api;

/**
 * Sample API used by Example Component whose method signatures are read from Javadoc.
 */
public class ExampleJavadocHello {

  public String sayHi() {
    return "Hello!";
  }
```

```java
    public String greetMe(String name) {
      return "Hello " + name;
    }

    public String greetUs(String name1, String name2) {
        return "Hello " + name1 + ", " + name2;
    }
}
```

ExampleFileHello class

Example 55.2, "ExampleFileHello class" shows the **ExampleFileHello** class from the sample Java API. As the name of the class suggests, this particular class is used to show how you can supply mapping metadata from a signature file.

Example 55.2. ExampleFileHello class

```java
// Java
package org.jboss.fuse.example.api;

/**
 * Sample API used by Example Component whose method signatures are read from File.
 */
public class ExampleFileHello {

  public String sayHi() {
    return "Hello!";
  }

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

  public String greetUs(String name1, String name2) {
      return "Hello " + name1 + ", " + name2;
  }
}
```

Generating the Javadoc metadata for ExampleJavadocHello

Because the metadata for **ExampleJavadocHello** is provided as Javadoc, it is necessary to generate Javadoc for the sample Java API and install it into the **camel-api-example-api** Maven artifact. The API POM file, **camel-api-example-api/pom.xml**, configures the **maven-javadoc-plugin** to perform this step automatically during the Maven build.

55.3. GENERATED COMPONENT SUB-PROJECT

Overview

The Maven sub-project for building the new component is located under the **camel-api-example/camel-api-example-component** project directory. In this section, we take a closer look at the generated example code and describe how it works.

Providing the Java API in the component POM

The Java API must be provided as a dependency in the component POM. For example, the sample Java API is defined as a dependency in the component POM file, **camel-api-example-component/pom.xml**, as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.fuse.example</groupId>
      <artifactId>camel-api-example-api</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Providing the Javadoc metadata in the component POM

If you are using Javadoc metadata for all or part of the Java API, you must provide the Javadoc as a dependency in the component POM. There are two things to note about this dependency:

- The Maven coordinates for the Javadoc are almost the same as for the Java API, except that you must also specify a **classifier** element, as follows:

  ```xml
  <classifier>javadoc</classifier>
  ```

- You must declare the Javadoc to have **provided** scope, as follows:

  ```xml
  <scope>provided</scope>
  ```

For example, in the component POM, the Javadoc dependency is defined as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  ...
  <dependencies>
    ...
```

```xml
    <!-- Component API javadoc in provided scope to read API signatures -->
    <dependency>
      <groupId>org.jboss.fuse.example</groupId>
      <artifactId>camel-api-example-api</artifactId>
      <version>1.0-SNAPSHOT</version>
      <classifier>javadoc</classifier>
      <scope>provided</scope>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

Defining the file metadata for Example File Hello

The metadata for **ExampleFileHello** is provided in a signature file. In general, this file must be created manually, but it has quite a simple format, which consists of a list of method signatures (one on each line). The example code provides the signature file, **file-sig-api.txt**, in the directory, **camel-api-example-component/signatures**, which has the following contents:

```
public String sayHi();
public String greetMe(String name);
public String greetUs(String name1, String name2);
```

For more details about the signature file format, see the section called "Signature file metadata" .

Configuring the API mapping

One of the key features of the API component framework is that it automatically generates the code to perform *API mapping*. That is, generating stub code that maps endpoint URIs to method invocations on the Java API. The basic inputs to the API mapping are: the Java API, the Javadoc metadata, and/or the signature file metadata.

The component that performs the API mapping is the **camel-api-component-maven-plugin** Maven plug-in, which is configured in the component POM. The following extract from the component POM shows how the **camel-api-component-maven-plugin** plug-in is configured:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  ...
  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>
      ...
      <!-- generate Component source and test source -->
      <plugin>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-api-component-maven-plugin</artifactId>
        <executions>
```

```
      <execution>
        <id>generate-test-component-classes</id>
        <goals>
          <goal>fromApis</goal>
        </goals>
        <configuration>
          <apis>
            <api>
              <apiName>hello-file</apiName>
              <proxyClass>org.jboss.fuse.example.api.ExampleFileHello</proxyClass>
              <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile>
            </api>
            <api>
              <apiName>hello-javadoc</apiName>
              <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
              <fromJavadoc/>
            </api>
          </apis>
        </configuration>
      </execution>
    </executions>
  </plugin>
  ...
</plugins>
  ...
</build>
  ...
</project>
```

The plug-in is configured by the **configuration** element, which contains a single **apis** child element to configure the classes of the Java API. Each API class is configured by an **api** element, as follows:

apiName

The *API name* is a short name for the API class and is used as the ***endpoint-prefix*** part of an endpoint URI.

> **NOTE**
>
> If the API consists of just a single Java class, you can leave the **apiName** element empty, so that the ***endpoint-prefix*** becomes redundant, and you can then specify the endpoint URI using the format shown in the section called "URI format for a single API class".

proxyClass

The proxy class element specifies the fully-qualified name of the API class.

fromJavadoc

If the API class is accompanied by Javadoc metadata, you must indicate this by including the **fromJavadoc** element and the Javadoc itself must also be specified in the Maven file, as a **provided** dependency (see the section called "Providing the Javadoc metadata in the component POM").

fromSignatureFile

If the API class is accompanied by signature file metadata, you must indicate this by including the **fromSignatureFile** element, where the content of this element specifies the location of the signature file.
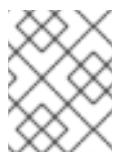
> **NOTE**
>
> The signature files do *not* get included in the final package built by Maven, because these files are needed only at build time, not at run time.

Generated component implementation

The API component consists of the following core classes (which must be implemented for every Camel component), under the **camel-api-example-component/src/main/java** directory:

ExampleComponent

Represents the component itself. This class acts as a factory for endpoint instances (for example, instances of **ExampleEndpoint**).

ExampleEndpoint

Represents an endpoint URI. This class acts as a factory for consumer endpoints (for example, **ExampleConsumer**) and as a factory for producer endpoints (for example, **ExampleProducer**).

ExampleConsumer

Represents a concrete instance of a consumer endpoint, which is capable of consuming messages from the location specified in the endpoint URI.

ExampleProducer

Represents a concrete instance of a producer endpoint, which is capable of sending messages to the location specified in the endpoint URI.

ExampleConfiguration

Can be used to define endpoint URI options. The URI options defined by this configuration class are *not* tied to any specific API class. That is, you can combine these URI options with any of the API classes or methods. This can be useful, for example, if you need to declare username and password credentials in order to connect to the remote service. The primary purpose of the **ExampleConfiguration** class is to provide values for parameters required to instantiate API classes, or classes that implement API interfaces. For example, these could be constructor parameters, or parameter values for a factory method or class.

To implement a URI option, *option*, in this class, all that you need to do is implement the pair of accessor methods, **get***Option* and **set***Option*. The component framework automatically parses the endpoint URI and injects the option values at run time.

ExampleComponent class

The generated **ExampleComponent** class is defined as follows:

```
// Java
package org.jboss.fuse.example;
```

```java
import org.apache.camel.CamelContext;
import org.apache.camel.Endpoint;
import org.apache.camel.spi.UriEndpoint;
import org.apache.camel.util.component.AbstractApiComponent;

import org.jboss.fuse.example.internal.ExampleApiCollection;
import org.jboss.fuse.example.internal.ExampleApiName;

/**
 * Represents the component that manages {@link ExampleEndpoint}.
 */
@UriEndpoint(scheme = "example", consumerClass = ExampleConsumer.class, consumerPrefix = "consumer")
public class ExampleComponent extends AbstractApiComponent<ExampleApiName, ExampleConfiguration, ExampleApiCollection> {

    public ExampleComponent() {
        super(ExampleEndpoint.class, ExampleApiName.class, ExampleApiCollection.getCollection());
    }

    public ExampleComponent(CamelContext context) {
        super(context, ExampleEndpoint.class, ExampleApiName.class,
ExampleApiCollection.getCollection());
    }

    @Override
    protected ExampleApiName getApiName(String apiNameStr) throws IllegalArgumentException {
        return ExampleApiName.fromValue(apiNameStr);
    }

    @Override
    protected Endpoint createEndpoint(String uri, String methodName, ExampleApiName apiName,
                        ExampleConfiguration endpointConfiguration) {
        return new ExampleEndpoint(uri, this, apiName, methodName, endpointConfiguration);
    }
}
```

The important method in this class is **createEndpoint**, which creates new endpoint instances. Typically, you do not need to change any of the default code in the component class. If there are any other objects with the same life cycle as this component, however, you might want to make those objects available from the component class (for example, by adding a methods to create those objects or by injecting those objects into the component).

ExampleEndpoint class

The generated **ExampleEndpoint** class is defined as follows:

```java
// Java
package org.jboss.fuse.example;

import java.util.Map;

import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
```

```java
import org.apache.camel.spi.UriEndpoint;
import org.apache.camel.util.component.AbstractApiEndpoint;
import org.apache.camel.util.component.ApiMethod;
import org.apache.camel.util.component.ApiMethodPropertiesHelper;

import org.jboss.fuse.example.api.ExampleFileHello;
import org.jboss.fuse.example.api.ExampleJavadocHello;
import org.jboss.fuse.example.internal.ExampleApiCollection;
import org.jboss.fuse.example.internal.ExampleApiName;
import org.jboss.fuse.example.internal.ExampleConstants;
import org.jboss.fuse.example.internal.ExamplePropertiesHelper;

/**
 * Represents a Example endpoint.
 */
@UriEndpoint(scheme = "example", consumerClass = ExampleConsumer.class, consumerPrefix =
"consumer")
public class ExampleEndpoint extends AbstractApiEndpoint<ExampleApiName,
ExampleConfiguration> {

    // TODO create and manage API proxy
    private Object apiProxy;

    public ExampleEndpoint(String uri, ExampleComponent component,
                ExampleApiName apiName, String methodName, ExampleConfiguration
endpointConfiguration) {
        super(uri, component, apiName, methodName,
ExampleApiCollection.getCollection().getHelper(apiName), endpointConfiguration);

    }

    public Producer createProducer() throws Exception {
        return new ExampleProducer(this);
    }

    public Consumer createConsumer(Processor processor) throws Exception {
        // make sure inBody is not set for consumers
        if (inBody != null) {
            throw new IllegalArgumentException("Option inBody is not supported for consumer
endpoint");
        }
        final ExampleConsumer consumer = new ExampleConsumer(this, processor);
        // also set consumer.* properties
        configureConsumer(consumer);
        return consumer;
    }

    @Override
    protected ApiMethodPropertiesHelper<ExampleConfiguration> getPropertiesHelper() {
        return ExamplePropertiesHelper.getHelper();
    }

    protected String getThreadProfileName() {
        return ExampleConstants.THREAD_PROFILE_NAME;
    }
```

```java
    @Override
    protected void afterConfigureProperties() {
        // TODO create API proxy, set connection properties, etc.
        switch (apiName) {
            case HELLO_FILE:
                apiProxy = new ExampleFileHello();
                break;
            case HELLO_JAVADOC:
                apiProxy = new ExampleJavadocHello();
                break;
            default:
                throw new IllegalArgumentException("Invalid API name " + apiName);
        }
    }

    @Override
    public Object getApiProxy(ApiMethod method, Map<String, Object> args) {
        return apiProxy;
    }
}
```

In the context of the API component framework, one of the key steps performed by the endpoint class is to create an *API proxy*. The API proxy is an instance from the target Java API, whose methods are invoked by the endpoint. Because a Java API typically consists of many classes, it is necessary to pick the appropriate API class, based on the **endpoint-prefix** appearing in the URI (recall that a URI has the general form, ***scheme**://**endpoint-prefix**/**endpoint***).

ExampleConsumer class

The generated **ExampleConsumer** class is defined as follows:

```java
// Java
package org.jboss.fuse.example;

import org.apache.camel.Processor;
import org.apache.camel.util.component.AbstractApiConsumer;

import org.jboss.fuse.example.internal.ExampleApiName;

/**
 * The Example consumer.
 */
public class ExampleConsumer extends AbstractApiConsumer<ExampleApiName,
ExampleConfiguration> {

    public ExampleConsumer(ExampleEndpoint endpoint, Processor processor) {
        super(endpoint, processor);
    }

}
```

ExampleProducer class

The generated **ExampleProducer** class is defined as follows:

```java
// Java
package org.jboss.fuse.example;

import org.apache.camel.util.component.AbstractApiProducer;

import org.jboss.fuse.example.internal.ExampleApiName;
import org.jboss.fuse.example.internal.ExamplePropertiesHelper;

/**
 * The Example producer.
 */
public class ExampleProducer extends AbstractApiProducer<ExampleApiName,
ExampleConfiguration> {

    public ExampleProducer(ExampleEndpoint endpoint) {
        super(endpoint, ExamplePropertiesHelper.getHelper());
    }
}
```

ExampleConfiguration class

The generated **ExampleConfiguration** class is defined as follows:

```java
// Java
package org.jboss.fuse.example;

import org.apache.camel.spi.UriParams;

/**
 * Component configuration for Example component.
 */
@UriParams
public class ExampleConfiguration {

    // TODO add component configuration properties
}
```

To add a URI option, *option*, to this class, define a field of the appropriate type, and implement a corresponding pair of accessor methods, **get*Option*** and **set*Option***. The component framework automatically parses the endpoint URI and injects the option values at run time.

> **NOTE**
>
> This class is used to define *general* URI options, which can be combined with any API method. To define URI options tied to a specific API method, configure *extra options* in the API component Maven plug-in. See Section 56.7, "Extra Options" for details.

URI format

Recall the general format of an API component URI:

scheme://endpoint-prefix/endpoint?Option1=Value1&...&OptionN=ValueN

In general, a URI maps to a specific method invocation on the Java API. For example, suppose you want to invoke the API method, **ExampleJavadocHello.greetMe("Jane Doe")**, the URI would be constructed, as follows:

[*scheme*]

> The API component scheme, as specified when you generated the code with the Maven archetype. In this case, the scheme is **example**.

[*endpoint-prefix*]

> The API name, which maps to the API class defined by the **camel-api-component-maven-plugin** Maven plug-in configuration. For the **ExampleJavadocHello** class, the relevant configuration is:

```
<configuration>
  <apis>
    <api>
      <apiName>hello-javadoc</apiName>
      <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
      <fromJavadoc/>
    </api>
    ...
  </apis>
</configuration>
```

> Which shows that the required *endpoint-prefix* is **hello-javadoc**.

[*endpoint*]

> The *endpoint* maps to the method name, which is **greetMe**.

[*Option1=Value1*]

> The URI options specify method parameters. The **greetMe(String name)** method takes the single parameter, **name**, which can be specified as **name=Jane%20Doe**. If you want to define default values for options, you can do this by overriding the **interceptProperties** method (see Section 55.4, "Programming Model").

Putting together the pieces of the URI, we see that we can invoke **ExampleJavadocHello.greetMe("Jane Doe")** with the following URI:

```
example://hello-javadoc/greetMe?name=Jane%20Doe
```

Default component instance

In order to map the **example** URI scheme to the default component instance, the Maven archetype creates the following file under the **camel-api-example-component** sub-project:

```
src/main/resources/META-INF/services/org/apache/camel/component/example
```

This resource file is what enables the Camel core to identify the component associated with the **example** URI scheme. Whenever you use an **example://** URI in a route, Camel searches the classpath to look for the corresponding **example** resource file. The **example** file has the following contents:

```
class=org.jboss.fuse.example.ExampleComponent
```

This enables the Camel core to create a default instance of the **ExampleComponent** component. The only time you would need to edit this file is if you refactor the name of the component class.

55.4. PROGRAMMING MODEL

Overview

In the context of the API component framework, the main component implementation classes are derived from base classes in the **org.apache.camel.util.component** package. These base classes define some methods which you can (optionally) override when you are implementing your component. In this section, we provide a brief description of those methods and how you might use them in your own component implementation.

Component methods to implement

In addition to the generated method implementations (which you usually do not need to modify), you can optionally override some of the following methods in the **Component** class:

doStart()

> (Optional) A callback to create resources for the component during a cold start. An alternative approach is to adopt the strategy of *lazy initialization* (creating resources only when they are needed). In fact, lazy initialization is often the best strategy, so the **doStart** method is often not needed.

doStop()

> (Optional) A callback to invoke code while the component is stopping. Stopping a component means that all of its resources are shut down, internal state is deleted, caches are cleared, and so on.

> **NOTE**
>
> Camel guarantees that **doStop** is *always* called when the current **CamelContext** shuts down, even if the corresponding **doStart** was never called.

doShutdown

> (Optional) A callback to invoke code while the **CamelContext** is shutting down. Whereas a stopped component can be restarted (with the semantics of a cold start), a component that gets shut down is completely finished. Hence, this callback represents the last chance to free up any resources belonging to the component.

What else to implement in the Component class?

The **Component** class is the natural place to hold references to objects that have the same (or similar) life cycle to the component object itself. For example, if a component uses OAuth security, it would be natural to hold references to the required OAuth objects in the **Component** class and to define methods in the **Component** class for creating the OAuth objects.

Endpoint methods to implement

You can modify some of the generated methods and, optionally, override some inherited methods in the **Endpoint** class, as follows:

afterConfigureProperties()

The main thing you need to do in this method is to create the appropriate type of proxy class (API class), to match the API name. The API name (which has already been extracted from the endpoint URI) is available either through the inherited **apiName** field or through the **getApiName** accessor. Typically, you would do a switch on the **apiName** field to create the corresponding proxy class. For example:

```java
// Java
private Object apiProxy;
...
@Override
protected void afterConfigureProperties() {
    // TODO create API proxy, set connection properties, etc.
    switch (apiName) {
        case HELLO_FILE:
            apiProxy = new ExampleFileHello();
            break;
        case HELLO_JAVADOC:
            apiProxy = new ExampleJavadocHello();
            break;
        default:
            throw new IllegalArgumentException("Invalid API name " + apiName);
    }
}
```

getApiProxy(ApiMethod method, Map<String, Object> args)

Override this method to return the proxy instance that you created in **afterConfigureProperties**. For example:

```java
@Override
public Object getApiProxy(ApiMethod method, Map<String, Object> args) {
    return apiProxy;
}
```

In special cases, you might want to make the choice of proxy dependent on the API method and arguments. The **getApiProxy** gives you the flexibility to take this approach, if required.

doStart()

(Optional) A callback to create resources during a cold start. Has the same semantics as **Component.doStart()**.

doStop()

(Optional) A callback to invoke code while the component is stopping. Has the same semantics as **Component.doStop()**.

doShutdown

(Optional) A callback to invoke code while the component is shutting down. Has the same semantics as **Component.doShutdown()**.

interceptPropertyNames(Set<String> propertyNames)

(Optional) The API component framework uses the endpoint URI and supplied option values to determine which method to invoke (ambiguity could be due to overloading and aliases). If the

component internally adds options or method parameters, however, the framework might need help in order to determine the right method to invoke. In this case, you must override the **interceptPropertyNames** method and add the extra (hidden or implicit) options to the **propertyNames** set. When the complete list of method parameters are provided in the **propertyNames** set, the framework will be able to identify the right method to invoke.

NOTE

You can override this method at the level of the **Endpoint**, **Producer** or **Consumer** class. The basic rule is, if an option affects *both* producer endpoints and consumer endpoints, override the method in the **Endpoint** class.

interceptProperties(Map<String,Object> properties)

(Optional) By overriding this method, you can modify or set the actual values of the options, before the API method is invoked. For example, you could use this method to set default values for some options, if necessary. In practice, it is often necessary to override both the **interceptPropertyNames** method and the **interceptProperty** method.

NOTE

You can override this method at the level of the **Endpoint**, **Producer** or **Consumer** class. The basic rule is, if an option affects *both* producer endpoints and consumer endpoints, override the method in the **Endpoint** class.

Consumer methods to implement

You can optionally override some inherited methods in the **Consumer** class, as follows:

interceptPropertyNames(Set<String> propertyNames)

(Optional) The semantics of this method are similar to **Endpoint.interceptPropertyNames**

interceptProperties(Map<String,Object> properties)

(Optional) The semantics of this method are similar to **Endpoint.interceptProperties**

doInvokeMethod(Map<String, Object> args)

(Optional) Overriding this method enables you to intercept the invocation of the Java API method. The most common reason for overriding this method is to customize the error handling around the method invocation. For example, a typical approach to overriding **doInvokeMethod** is shown in the following code fragment:

```java
// Java
@Override
protected Object doInvokeMethod(Map<String, Object> args) {
    try {
        return super.doInvokeMethod(args);
    } catch (RuntimeCamelException e) {
        // TODO - Insert custom error handling here!
        ...
    }
}
```

You should invoke **doInvokeMethod** on the super-class, at some point in this implementation, to ensure that the Java API method gets invoked.

interceptResult(Object methodResult, Exchange resultExchange)

(Optional) Do some additional processing on the result of the API method invocation. For example, you could add custom headers to the Camel exchange object, **resultExchange**, at this point.

Object splitResult(Object result)

(Optional) By default, if the result of the method API invocation is a **java.util.Collection** object or a Java array, the API component framework splits the result into *multiple* exchange objects (so that a single invocation result is converted into multiple messages).

If you want to change the default behaviour, you can override the **splitResult** method in the consumer endpoint. The **result** argument contains the result of the API message invocation. If you want to split the result, you should return an array type.
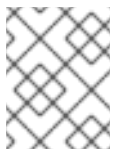
> **NOTE**
>
> You can also switch off the default splitting behaviour by setting **consumer.splitResult=false** on the endpoint URI.

Producer methods to implement

You can optionally override some inherited methods in the **Producer** class, as follows:

interceptPropertyNames(Set<String> propertyNames)

(Optional) The semantics of this method are similar to **Endpoint.interceptPropertyNames**

interceptProperties(Map<String,Object> properties)

(Optional) The semantics of this method are similar to **Endpoint.interceptProperties**

doInvokeMethod(Map<String, Object> args)

(Optional) The semantics of this method are similar to **Consumer.doInvokeMethod**.

interceptResult(Object methodResult, Exchange resultExchange)

(Optional) The semantics of this method are similar to **Consumer.interceptResult**.

> **NOTE**
>
> The **Producer.splitResult()** method is never called, so it is *not* possible to split an API method result in the same way as you can for a consumer endpoint. To get a similar effect for a producer endpoint, you can use Camel's **split()** DSL command (one of the standard enterprise integration patterns) to split **Collection** or array results.

Consumer polling and threading model

The default threading model for consumer endpoints in the API component framework is *scheduled poll consumer*. This implies that the API method in a consumer endpoint is invoked at regular, scheduled time intervals. For more details, see the section called "Scheduled poll consumer implementation" .

55.5. SAMPLE COMPONENT IMPLEMENTATIONS

Overview

Several of the components distributed with Apache Camel have been implemented with the aid of the API component framework. If you want to learn more about the techniques for implementing Camel components using the framework, it is a good idea to study the source code of these component implementations.

Box.com

The Camel Box component shows how to model and invoke the third party Box.com Java SDK using the API component framework. It also demonstrates how the framework can be adapted to customize consumer polling, in order to support Box.com's long polling API.

LinkedIn

The Camel LinkedIn component demonstrates how to wrap a REST API provided in the form of WADL and XML schemas. The implementation of this component exploits the Apache CXF **wadl2java** Maven plug-in to generate a Java API, which can then be wrapped using the API component framework.

This approach can be easily replicated to create a Camel component for any SaaS product or platform.

GoogleDrive

The Camel GoogleDrive component demonstrates how the API component framework can handle even Method Object style Google APIs. In this case, URI options are mapped to a method object, which is then invoked by overriding the **doInvoke** method in the consumer and the producer.

Olingo2

The Camel Olingo2 component demonstrates how a callback-based Asynchronous API can be wrapped using the API component framework. This example shows how asynchronous processing can be pushed into underlying resources, like HTTP NIO connections, to make Camel endpoints more resource efficient.

CHAPTER 56. CONFIGURING THE API COMPONENT MAVEN PLUG-IN

Abstract

This chapter provides a reference for all of the configuration options available on the API component Maven plug-in.

56.1. OVERVIEW OF THE PLUG-IN CONFIGURATION

Overview

The main purpose of the API component Maven plug-in, **camel-api-component-maven-plugin**, is to generate the API mapping classes, which implement the mapping between endpoint URIs and API method invocations. By editing the configuration of the API component Maven plug-in, you can customize various aspects of the API mapping.

Location of the generated code

The API mapping classes generated by the API component Maven plug-in are placed in the following location, by default:

> *ProjectName*-component/target/generated-sources/camel-component

Prerequisites

The main inputs to the API component Maven plug-in are the Java API classes and the Javadoc metadata. These are made available to the plug-in by declaring them as regular Maven dependencies (where the Javadoc Maven dependencies should be declared with **provided** scope).

Setting up the plug-in

The recommended way to set up the API component Maven plug-in is to generate starting point code using the API component archetype. This generates the default plug-in configuration in the *ProjectName*-**component/pom.xml** file, which you can then customize for your project. The main aspects of the plug-in set-up are, as follows:

1. Maven dependencies must be declared for the requisite Java API and for the Javadoc metadata.

2. The plug-in's base configuration is declared in the **pluginManagement** scope (which also defines the version of the plug-in to use).

3. The plug-in instance itself is declared and configured.

4. The **build-helper-maven** plug-in is configured to pick up the generated sources from the **target/generated-sources/camel-component** directory and include them in the Maven build.

Example base configuration

The following POM file excerpt shows the basic configuration of the API component Maven plug-in

The following POM file extract shows the base configuration of the API component Maven plug-in, as defined in the Maven **pluginManagement** scope when the code has been generated using the API component archetype:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <build>
   ...
    <pluginManagement>
     <plugins>
      <plugin>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-api-component-maven-plugin</artifactId>
        <version>2.17.0.redhat-630xxx</version>
        <configuration>
          <scheme>${schemeName}</scheme>
          <componentName>${componentName}</componentName>
          <componentPackage>${componentPackage}</componentPackage>
          <outPackage>${outPackage}</outPackage>
        </configuration>
      </plugin>
     </plugins>
    </pluginManagement>
   ...
  </build>
  ...
</project
```

The configuration specified in the **pluginManagement** scope provides default settings for the plug-in. It does not actually create an instance of a plug-in, but its default settings will be used by any API component plug-in instance.

Base configuration

In addition to specifying the plug-in version (in the **version** element), the preceding base configuration specifies the following configuration properties:

scheme

The URI scheme for this API component.

componentName

The name of this API component (which is also used as a prefix for generated class names).

componentPackage

Specifies the Java package containing the classes generated by the API component Maven archetype. This package is also exported by the default **maven-bundle-plugin** configuration. Hence, if you want a class to be publicly visible, you should place it in this Java package.

outPackage

Specifies the Java package where the generated API mapping classes are placed (when they are generated by the API component Maven plug-in). By default, this has the value of the **componentName** property, with the addition of the **.internal** suffix. This package is declared as

private by the default **maven-bundle-plugin** configuration. Hence, if you want a class to be private, you should place it in this Java package.

Example instance configuration

The following POM file extract shows a sample instance of the API component Maven plug-in, which is configured to generate an API mapping during the Maven build:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  ...
  <build>
    <defaultGoal>install</defaultGoal>

    <plugins>
      ...
      <!-- generate Component source and test source -->
      <plugin>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-api-component-maven-plugin</artifactId>
        <executions>
          <execution>
            <id>generate-test-component-classes</id>
            <goals>
              <goal>fromApis</goal>
            </goals>
            <configuration>
              <apis>
                <api>
                  <apiName>hello-file</apiName>
                  <proxyClass>org.jboss.fuse.example.api.ExampleFileHello</proxyClass>
                  <fromSignatureFile>signatures/file-sig-api.txt</fromSignatureFile>
                </api>
                <api>
                  <apiName>hello-javadoc</apiName>
                  <proxyClass>org.jboss.fuse.example.api.ExampleJavadocHello</proxyClass>
                  <fromJavadoc/>
                </api>
              </apis>
            </configuration>
          </execution>
        </executions>
      </plugin>
      ...
    </plugins>
    ...
  </build>
  ...
</project>
```

Basic mapping configuration

The plug-in is configured by the **configuration** element, which contains a single **apis** child element to configure the classes of the Java API. Each API class is configured by an **api** element, as follows:

apiName

The *API name* is a short name for the API class and is used as the ***endpoint-prefix*** part of an endpoint URI.
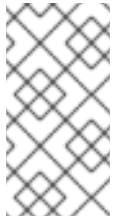
> **NOTE**
>
> If the API consists of just a single Java class, you can leave the **apiName** element empty, so that the ***endpoint-prefix*** becomes redundant, and you can then specify the endpoint URI using the format shown in the section called "URI format for a single API class".

proxyClass

This element specifies the fully-qualified name of the API class.

fromJavadoc

If the API class is accompanied by Javadoc metadata, you must indicate this by including the **fromJavadoc** element and the Javadoc itself must also be specified in the Maven file, as a **provided** dependency.

fromSignatureFile

If the API class is accompanied by signature file metadata, you must indicate this by including the **fromSignatureFile** element, where the content of this element specifies the location of the signature file.

> **NOTE**
>
> The signature files do *not* get included in the final package built by Maven, because these files are needed only at build time, not at run time.

Customizing the API mapping

The following aspects of the API mapping can be customized by configuring the plug-in:

- *Method aliases*—you can define additional names (aliases) for an API method using the **aliases** configuration element. For details, see Section 56.3, "Method Aliases".

- *Nullable options*—you can use the **nullableOptions** configuration element to declare method arguments that default to **null**. For details, see Section 56.4, "Nullable Options".

- *Argument name substitution*—due to the way the API mapping is implemented, the arguments from all of the methods in a particular API class belong to the *same* namespace. If two arguments with the same name are declared to be of different type, this leads to a clash. To avoid such name clashes, you can use the **substitutions** configuration element to rename method arguments (as they would appear in a URI). For details, see Section 56.5, "Argument Name Substitution".

- *Excluding arguments* —when it comes to mapping Java arguments to URI options, you might sometimes want to exclude certain arguments from the mapping. You can filter out unwanted arguments by specifying either the **excludeConfigNames** element or the **excludeConfigTypes** element. For details, see Section 56.6, "Excluded Arguments".

- *Extra options*—sometimes you might want to define extra options, which are not part of the Java API. You can do this using the **extraOptions** configuration element.

Configuring Javadoc metadata

It is possible to filter the Javadoc metadata to ignore or explicitly include certain content. For details of how to do this, see Section 56.2, "Javadoc Options".

Configuring signature file metadata

In cases where no Javadoc is available, you can resort to signature files to supply the needed mapping metadata. The **fromSignatureFile** is used to specify the location of the corresponding signature file. It has no special options.

56.2. JAVADOC OPTIONS

Overview

If the metadata for your Java API is provided by Javadoc, it is generally sufficient to specify the **fromJavadoc** element with no options. But in cases where you do not want to include the entire Java API in your API mapping, you can filter the Javadoc metadata to customize the content. In other words, because the API component Maven plug-in generates the API mapping by iterating over the Javadoc metadata, it is possible to customize the scope of the generated API mapping by filtering out unwanted parts of the Javadoc metadata.

Syntax

The **fromJavadoc** element can be configured with optional child elements, as follows:

```
<fromJavadoc>
 <excludePackages>PackageNamePattern</excludePackages>
 <excludeClasses>ClassNamePattern</excludeClasses>
 <excludeMethods>MethodNamePattern</excludeMethods>
 <includeMethods>MethodNamePattern</includeMethods>
 <includeStaticMethods>[true|false]<includeStaticMethods>
</fromJavadoc>
```

Scope

As shown in the following extract, the **fromJavadoc** element can optionally appear as a child of the **apis** element and/or as a child of **api** elements:

```
<configuration>
 <apis>
  <api>
   <apiName>...</apiName>
   ...
 <fromJavadoc>...</fromJavadoc>
```

```
    </api>
 <fromJavadoc>...</fromJavadoc>
    ...
  </apis>
</configuration>
```

You can define the **fromJavadoc** element at the following scopes:

- *As a child of an* **api** *element*—the **fromJavadoc** options apply only to the API class specified by the **api** element.

- *As a child of the* **apis** *element*—the **fromJavadoc** options apply to all API classes by default, but can be overridden at the **api** level.

Options

The following options can be defined as child elements of **fromJavadoc**:

excludePackages

Specifies a regular expression (**java.util.regex** syntax) for excluding Java packages from the API mapping model. All package names that match the regular expression are excluded; and *all classes derived from the excluded classes are also ignored*. Default value is **javax?\.lang.***.

excludeClasses

Specifies a regular expression (**java.util.regex** syntax) for excluding API base classes from the API mapping. All class names that match the regular expression are excluded; and *all classes derived from the excluded classes are also ignored*.

excludeMethods

Specifies a regular expression (**java.util.regex** syntax) for excluding methods from the API mapping model.

includeMethods

Specifies a regular expression (**java.util.regex** syntax) for including methods from the API mapping model.

includeStaticMethods

If **true**, static methods will also be included in the API mapping model. Default is **false**.

56.3. METHOD ALIASES

Overview

Often it can be useful to define additional names (aliases) for a given method, in addition to the standard method name that appears in the Java API. A particularly common case is where you allow a property name (such as **widget**) to be used as an alias for an accessor method (such as **getWidget** or **setWidget**).

Syntax

The **aliases** element can be defined with one or more **alias** child elements, as follows:

```
<aliases>
  <alias>
    <methodPattern>MethodPattern</methodPattern>
    <methodAlias>Alias</methodAlias>
  </alias>
  ...
</aliases>
```

Where **MethodPattern** is a regular expression (**java.util.regex** syntax) for matching method names from the Java API, and the pattern typically includes capturing groups. The **Alias** is the replacement expression (for use in a URI), which can use the text from the preceding capturing groups (for example, specified as **$1**, **$2**, or **$3** for the text from the first, second, or third capturing group).

Scope

As shown in the following extract, the **aliases** element can optionally appear as a child of the **apis** element and/or as a child of **api** elements:

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
  <aliases>...</aliases>
    </api>
  <aliases>...</aliases>
    ...
  </apis>
</configuration>
```

You can define the **aliases** element at the following scopes:

- *As a child of an **api** element*—the **aliases** mappings apply only to the API class specified by the **api** element.

- *As a child of the **apis** element*—the **aliases** mappings apply to all API classes by default, but can be overridden at the **api** level.

Example

The following example shows how to generate aliases for the common get/set bean method pattern:

```
<aliases>
  <alias>
    <methodPattern>[gs]et(.+)</methodPattern>
    <methodAlias>$1</methodAlias>
  </alias>
</aliases>
```

With the preceding alias definition, you could use **widget** as an alias for either of the methods **getWidget** or **setWidget**. Note the use of a capturing group, **(.+)**, to capture the latter part of the method name (for example, **Widget**).

56.4. NULLABLE OPTIONS

Overview

In some cases, it can make sense to let method arguments default to **null**. But this is not allowed by default. If you want to allow some of your method arguments from the Java API to take **null** values, you must declare this explicitly using the **nullableOptions** element.

Syntax

The **nullableOptions** element can be defined with one or more **nullableOption** child elements, as follows:

```
<nullableOptions>
  <nullableOption>ArgumentName</nullableOption>
  ...
</nullableOptions>
```

Where **ArgumentName** is the name of a method argument from the Java API.

Scope

As shown in the following extract, the **nullableOptions** element can optionally appear as a child of the **apis** element and/or as a child of **api** elements:

```
<configuration>
 <apis>
  <api>
    <apiName>...</apiName>
    ...
   <nullableOptions>...</nullableOptions>
  </api>
  ...
 <nullableOptions>...</nullableOptions>
 </apis>
</configuration>
```

You can define the **nullableOptions** element at the following scopes:

- *As a child of an **api** element*—the **nullableOptions** mappings apply only to the API class specified by the **api** element.

- *As a child of the **apis** element*—the **nullableOptions** mappings apply to all API classes by default, but can be overridden at the **api** level.

Example

The following example shows the nullable options declared for the **CompaniesResource** proxy class from the Apache Camel LinkedIn component:

```
<nullableOptions>
  <nullableOption>companySizes</nullableOption>
  <nullableOption>count</nullableOption>
```

```
    <nullableOption>email_domain</nullableOption>
    <nullableOption>end_timestamp</nullableOption>
    <nullableOption>event_type</nullableOption>
    <nullableOption>geos</nullableOption>
    <nullableOption>industries</nullableOption>
    <nullableOption>is_company_admin</nullableOption>
    <nullableOption>jobFunc</nullableOption>
    <nullableOption>secure_urls</nullableOption>
    <nullableOption>seniorities</nullableOption>
    <nullableOption>start</nullableOption>
    <nullableOption>start_timestamp</nullableOption>
    <nullableOption>statistics_update_key</nullableOption>
    <nullableOption>time_granularity</nullableOption>
</nullableOptions>
```

56.5. ARGUMENT NAME SUBSTITUTION

Overview

The API component framework requires that URI option names are *unique within each proxy class (Java API class)*. This is not always the case for method argument names, however. For example, consider the following Java methods in an API class:

```
public void doSomething(int id, String name);
public void doSomethingElse(int id, String name);
```

When you build your Maven project, the **camel-api-component-maven-plugin** generates the configuration class, ***ProxyClass*EndpointConfiguration**, which contains getter and setter methods for *all* of the arguments in the ***ProxyClass*** class. For example, given the preceding methods, the plug-in would generate the following getter and setter methods in the configuration class:

```
public int  getId();
public void setId(int id);
public String getName();
public void   setName(String name);
```

But what happens, if the **id** argument appears multiple times as different types, as in the following example:

```
public void doSomething(int id, String name);
public void doSomethingElse(int id, String name);
public String lookupByID(String id);
```

In this case, the code generation would fail, because you cannot define a **getId** method that returns **int** and a **getId** method that returns **String** in the same scope. The solution to this problem is to use *argument name substitution* to customize the mapping of argument names to URI option names.

Syntax

The **substitutions** element can be defined with one or more **substitution** child elements, as follows:

```
<substitutions>
 <substitution>
```

```
    <method>MethodPattern</method>
    <argName>ArgumentNamePattern</argName>
    <argType>TypeNamePattern</argType>
    <replacement>SubstituteArgName</replacement>
    <replaceWithType>[true|false]</replaceWithType>
  </substitution>
  ...
</substitutions>
```

Where the **argType** element and the **replaceWithType** element are optional and can be omitted.

Scope

As shown in the following extract, the **substitutions** element can optionally appear as a child of the **apis** element and/or as a child of **api** elements:

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>
      ...
  <substitutions>...</substitutions>
    </api>
  <substitutions>...</substitutions>
    ...
  </apis>
</configuration>
```

You can define the **substitutions** element at the following scopes:

- *As a child of an api element*—the **substitutions** apply only to the API class specified by the **api** element.

- *As a child of the apis element*—the **substitutions** apply to all API classes by default, but can be overridden at the **api** level.

Child elements

Each **substitution** element can be defined with the following child elements:

method

Specifies a regular expression (**java.util.regex** syntax) to match a method name from the Java API.

argName

Specifies a regular expression (**java.util.regex** syntax) to match an argument name from the matched method, where the pattern typically includes capturing groups.

argType

(Optional) Specifies a regular expression (**java.util.regex** syntax) to match the type of the argument. If you set the **replaceWithType** option to **true**, you would typically use capturing groups in this regular expression.

replacement

Given a particular match of the **method** pattern, **argName** pattern, and (optionally) **argType** pattern, the **replacement** element defines the substitute argument name (for use in a URI). The replacement text can be constructed using strings captured from the **argName** regular expression pattern (using the syntax, **$1**, **$2**, **$3** to insert the first, second, or third capturing group, respectively). Alternatively, the replacement text can be constructed using strings captured from the **argType** regular expression pattern, if you set the **replaceWithType** option to **true**.

replaceWithType

When **true**, specifies that the replacement text is constructed using strings captured from the **argType** regular expression. Defaults to **false**.

Example

The following substitution example modifies every argument of **java.lang.String** type, by adding the suffix, **Param** to the argument name:

```
<substitutions>
  <substitution>
    <method>^.+$</method>
    <argName>^.+$</argName>
    <argType>java.lang.String</argType>
    <replacement>$1Param</replacement>
    <replaceWithType>false</replaceWithType>
  </substitution>
</substitutions>
```

For example, given the following method signature:

```
public String greetUs(String name1, String name2);
```

The arguments of this method would be specified through the options, **name1Param** and **name2Param**, in the endpoint URI.

56.6. EXCLUDED ARGUMENTS

Overview

Sometimes, you might need to exclude certain arguments, when it comes to mapping Java arguments to URI options. You can filter out unwanted arguments by specifying either the **excludeConfigNames** element or the **excludeConfigTypes** element in the **camel-api-component-maven-plugin** plug-in configuration.

Syntax

The **excludeConfigNames** element and the **excludeConfigTypes** element are specified as follows:

```
<excludeConfigNames>ArgumentNamePattern</excludeConfigNames>
<excludeConfigTypes>TypeNamePattern</excludeConfigTypes>
```

Where ***ArgumentNamePattern*** and ***TypeNamePattern*** are regular expressions that match the argument name and the argument type, respectively.

Scope

As shown in the following extract, the **excludeConfigNames** element and the **excludeConfigTypes** element can optionally appear as children of the **apis** element and/or as children of **api** elements:

```
<configuration>
 <apis>
  <api>
    <apiName>...</apiName>
    ...
  <excludeConfigNames>...</excludeConfigNames>
  <excludeConfigTypes>...</excludeConfigTypes>
   </api>
  <excludeConfigNames>...</excludeConfigNames>
  <excludeConfigTypes>...</excludeConfigTypes>
   ...
 </apis>
</configuration>
```

You can define the **excludeConfigNames** element and the **excludeConfigTypes** element at the following scopes:

- *As a child of an **api** element*—the exclusions apply only to the API class specified by the **api** element.

- *As a child of the **apis** element*—the exclusions apply to all API classes by default, but can be overridden at the **api** level.

Elements

The following elements can be used to exclude arguments from the API mapping (so that they are unavailable as URI options):

excludeConfigNames

Specifies a regular expression (**java.util.regex** syntax) for excluding arguments, based on matching the argument name.

excludeConfigTypes

Specifies a regular expression (**java.util.regex** syntax) for excluding arguments, based on matching the argument type.

56.7. EXTRA OPTIONS

Overview

The **extraOptions** options are usually used to either compute or hide complex API parameters by providing simpler options instead. For example, the API method might take a POJO option, that could be provided more easily as parts of the POJO in the URI. The component could do this by adding the parts as extra options, and creating the POJO parameter internally. To complete the implementation of these extra options, you also need to override the **interceptProperties** method in the **EndpointConsumer** and/or **EndpointProducer** classes (see Section 55.4, "Programming Model").

Syntax

The **extraOptions** element can be defined with one or more **extraOption** child elements, as follows:

```
<extraOptions>
  <extraOption>
    <type>TypeName</type>
    <name>OptionName</name>
  </extraOption>
</extraOptions>
```

Where **TypeName** is the fully-qualified type name of the extra option and **OptionName** is the name of the extra URI option.

Scope

As shown in the following extract, the **extraOptions** element can optionally appear as a child of the **apis** element and/or as a child of **api** elements:

```
<configuration>
  <apis>
    <api>
      <apiName>...</apiName>

  <extraOptions>...</extraOptions>
    </api>
  <extraOptions>...</extraOptions>

    ...
  </apis>
</configuration>
```

You can define the **extraOptions** element at the following scopes:

- *As a child of an **api** element*—the **extraOptions** apply only to the API class specified by the **api** element.

- *As a child of the **apis** element*—the **extraOptions** apply to all API classes by default, but can be overridden at the **api** level.

Child elements

Each **extraOptions** element can be defined with the following child elements:

type

Specifies the fully-qualified type name of the extra option.

name

Specifies the option name, as it would appear in an endpoint URI.

Example

The following example defines an extra URI option, **customOption**, which is of **java.util.list<String>** type:

```
<extraOptions>
  <extraOption>
    <type>java.util.List&lt;String&gt;</type>
    <name>customOption</name>
  </extraOption>
</extraOptions>
```

INDEX

Symbols

@Converter, Implement an annotated converter class

A

AsyncCallback, Asynchronous processing

asynchronous producer

implementing, How to implement an asynchronous producer

AsyncProcessor, Asynchronous processing

auto-discovery

configuration, Configuring auto-discovery

C

Component

createEndpoint(), URI parsing

definition, The Component interface

methods, Component methods

component prefix, Component

components, Component

bean properties, Define bean properties on your component class

configuring, Installing and configuring the component

implementation steps, Implementation steps

installing, Installing and configuring the component

interfaces to implement, Which interfaces do you need to implement?

parameter injection, Parameter injection

Spring configuration, Configure the component in Spring

Consumer, Consumer

consumers, Consumer

M

Message, Message

getHeader(), Accessing message headers

message headers

accessing, Accessing message headers

messages, Message

P

performer, Overview

pipeline, Pipelining model

Processor, Processor interface

implementing, Implementing the Processor interface

producer, Producer

Producer, Producer

createExchange(), Producer methods

getEndpoint(), Producer methods

process(), Producer methods

producers

asynchronous, Asynchronous producer

synchronous, Synchronous producer

S

ScheduledPollEndpoint, Scheduled poll endpoint implementation

simple processor

implementing, Implementing the Processor interface

synchronous producer

implementing, How to implement a synchronous producer

T

type conversion

runtime process, Type conversion process

type converter

annotating the implementation, Implement an annotated converter class