Red Hat build of Apache Camel 4.14

Camel development guide for Red Hat build of Apache Camel for Spring Boot

Basic building blocks, patterns, basic syntax for routing expression and predicate languages.

Last Updated: 2026-04-01

Red Hat build of Apache Camel 4.14 Camel development guide for Red Hat build of Apache Camel for Spring Boot

Basic building blocks, patterns, basic syntax for routing expression and predicate languages.

Legal Notice

Abstract

Camel development guide

Table of Contents

CHAPTER 1. CAMEL ROUTES

Apache Camel supports three alternative **Domain Specific Languages** (DSL) for defining routes:

- Java DSL
- XML IO DSL
- YAML IO 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. ALL ROUTES EXTEND A ROUTEBUILDER CLASS

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.

1.1.1. 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. 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()**).

1.1.2. Implementing a RouteBuilder

Following example shows a minimal **RouteBuilder** implementation. The **configure()** method body contains a routing rule; each rule is a single Java statement.

Implementation of a RouteBuilder Class

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

public class MyRouteBuilderextends 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 to filter **RouteBuilder** classes, by default Apache Camel will look for singleton beans. You can turn on the old behavior to include prototype scoped with the new option **includeNonSingletons**.

1.2. INTRODUCTION TO THE JAVA DSL FOR DEFINING ROUTES

1.2.1. 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:

```
command01().command02().command03()
```

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

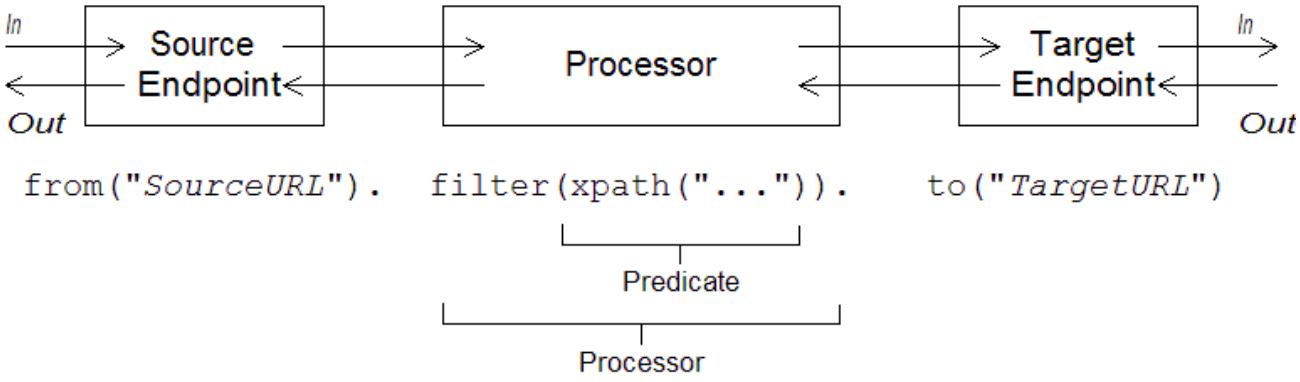
```
command01().startBlock().command02().command03().endBlock()
```

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

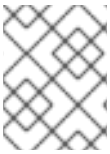
1.2.2. Format of Java DSL rules that define routes

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. The following figure 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()**).

1.2.3. 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**, or an Apache ActiveMQ endpoint, **activemq:queue:MyQName**.

1.2.4. Exchange objects

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:

- **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 Pipeline Processing section.

  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:

  - **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.

- Exchange properties – a list of named properties containing metadata for the current message.

1.2.4.1. 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**
- **InOut**

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

1.2.4.2. 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.

1.2.5. How processors modify behavior according to expressions and predicates

1.2.5.1. 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 the figure "local routing rules" includes a **filter()** processor that takes an **xpath()** predicate as its argument.

1.2.5.2. 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.

1.2.5.3. Router Schema in a Spring XML File

1.2.5.3.1. Namespace

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

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

1.2.5.3.2. 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 the following example.

Example 1.1. 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>
```

1.2.5.3.3. 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.

1.2.5.3.4. 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.

1.3. APACHE CAMEL ENDPOINTS

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.

1.3.1. 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 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
```

1.3.2. Specifying time periods in a URI

Many of the Red Hat build of 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 Red Hat build of Apache Camel also supports a more readable syntax for time periods, which enables you to express the period in hours, minutes, and seconds. Formally, the human-readable time period is a string that conforms to the following syntax:

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

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

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

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

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

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

1.3.3. 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 behavior. 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**.

1.3.4. 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"/>
```

1.3.5. Specifying URI Resources

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.
```

1.3.6. 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**, **Platform-HTTP**, **Direct**, and **Mock**. For the full list of supported components, see the Red Hat build of Apache Camel for Spring Boot Reference.

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>{camel-version-full}</camel-version>
     ...
  </properties>

<dependencies>
...
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-http-starter</artifactId>
     </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

1.3.7. 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:

-

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

    <route id="payments-route">
        <from uri="jms:queue:payments"/>
        <process ref="someProcessorId"/>
        <to uri="TargetURI"/>
    </route>
</routes>
```

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://foo@myserver?assword=%s", 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.

1.3.8. 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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route id="myRoute">
        <from uri="SourceURI" />
        <processref="someProcessorId" />
```

```
        <to uri="jms:queue:orderForms" />
    </route>
</routes>
```

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:

```
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.

1.4. APACHE CAMEL PROCESSORS

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.

1.4.1. Apache Camel processors

Apache Camel provides a wide variety of different processors, as shown below.

Table 1.1. Camel Processors

Java DSL	XML DSL	Description
aggregate()	**aggregate**	Aggregator : Creates an aggregator, which combines multiple incoming exchanges into a single exchange.
aop()	**aop**	Use Aspect Oriented Programming (AOP) to do work before and after a specified sub-route.
bean(), **beanRef()**	**bean**	Process the current exchange by invoking a method on a Java object (or bean).

Java DSL	XML DSL	Description
choice()	**choice**	Content Based Router: 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: 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: Combines the current exchange with data requested from a specified **producer** endpoint URI.
filter()	**filter**	Message Filter: Uses a predicate expression to filter incoming exchanges.
idempotentConsumer()	**idempotentConsumer**	Idempotent Consumer: 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).
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). **Deprecated**: use **setExchangePattern** instead.

Java DSL	XML DSL	Description
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). **Deprecated**: use **setExchangePattern** instead.
loadBalance()	**loadBalance**	LoadBalancer: Implements load balancing over a collection of endpoints.
log()	**log**	Logs a message to the console.
loop()	**loop**	Loop: 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.
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.
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: 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.
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: Sends the exchange to a series of endpoints, where the output of one endpoint becomes the input of the next endpoint.
policy()	**policy**	Apply a policy to the current route (currently only used for transactional policies.
pollEnrich(),**pollEnrichRef()**	**pollEnrich**	Content Enricher: Combines the current exchange with data polled from a specified **consumer** endpoint URI.
process(),**processRef**	**process**	Execute a custom processor on the current exchange.
recipientList()	**recipientList**	Recipient List: 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: 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).
routingSlip()	**routingSlip**	Routing Slip: 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.
setHeader()	**setHeader**	Sets the specified header in the exchange's **In** message.
setOutHeader()	**setOutHeader**	Sets the specified header in the exchange's **Out** message.

Java DSL	XML DSL	Description
setProperty()	**setProperty()**	Sets the specified exchange property.
sort()	**sort**	Sorts the contents of the **In** message body (where a custom comparator can optionally be specified).
split()	**split**	Splitter: 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: 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.
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.
transform()	**transform**	Message Translator: Copy the **In** message headers to the **Out** message headers and set the **Out** message body to the specified value.

Java DSL	XML DSL	Description
unmarshal()	**unmarshal**	Transforms the **In** message body from a low-level or binary format to a high-level format, using the specified data format.
validate()	**validate**	Takes a predicate expression to test whether the current message is valid. If the predicate returns **false**, throws a **PredicateValidationException** exception.
wireTap()	**wireTap**	Sends a copy of the current exchange to the specified wire tap URI, using the **ExchangePattern.InOnly** MEP.

1.4.1.1. Some sample processors

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

- Choice
- Filter
- Throttler
- Custom

1.4.1.1.1. 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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route id="buildSimpleRouteWithChoice">
    <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>
</routes>
```

In the Java DSL, there is a special case where you might need to use the **endChoice()** command. Some of the standard Red Hat build of 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:

```
from("direct:start")
    .choice()
        .when(bodyAs(String.class).contains("Camel"))
            .loadBalance().roundRobin().to("mock:foo").to("mock:bar").endChoice()
        .otherwise()
            .to("mock:result");
```

1.4.1.1.2. 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:

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

1.4.1.1.3. 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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
<route id="throttleRoute">
<from uri="direct:start"/>
<throttle timePeriodMillis="1000">
<constant>100</constant>
</throttle>
<log message="HI XML"/>
</route>
</routes>
```

1.4.1.1.4. 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:

Implementing a Custom Processor Class

```
public class *MyProcessor* implements org.apache.camel.*Processor* {
public void process(org.apache.camel.Exchange exchange) {
  inMessage = exchange.getMessage();
  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 the above example.

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL ").process(myProc).to("TargetURL ");
```

1.4.2. How to route messages to producer endpoints using a choice processor

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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
<route id="buildSimpleRouteWithChoice">
<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>
</routes>
```

In the Java DSL, there is a special case where you might need to use the **endChoice()** command. Some of the standard Red Hat build of 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:

```
from("direct:start")
    .choice()
        .when(bodyAs(String.class).contains("Camel"))
            .loadBalance().roundRobin().to("mock:foo").to("mock:bar").endChoice()
        .otherwise()
            .to("mock:result");
```

1.4.3. How to exclude messages using the filter processor

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:

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

1.4.3.1. 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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
<route id="throttleRoute">
<from uri="direct:start"/>
<throttle timePeriodMillis="1000">
<constant>100</constant>
</throttle>
<log message="HI XML"/>
</route>
</routes>
```

1.4.3.2. 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:

Implementing a Custom Processor Class

```
public class *MyProcessor* implements org.apache.camel.*Processor* {
public void process(org.apache.camel.Exchange exchange) {
  inMessage = exchange.getMessage();
  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 the above example.

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL ").process(myProc).to("TargetURL ");
```

1.4.4. How to limit the number of messages sent to a producer endpoint

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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
<route id="throttleRoute">
<from uri="direct:start"/>
<throttle timePeriodMillis="1000">
<constant>100</constant>
</throttle>
<log message="HI XML"/>
</route>
</routes>
```

1.4.5. How to define your own 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:

Implementing a Custom Processor Class

```
public class *MyProcessor* implements org.apache.camel.*Processor* {
public void process(org.apache.camel.Exchange exchange) {
  inMessage = exchange.getMessage();
  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 the above example.

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL ").process(myProc).to("TargetURL ");
```

CHAPTER 2. BASIC PRINCIPLES FOR BUILDING CAMEL ROUTES

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

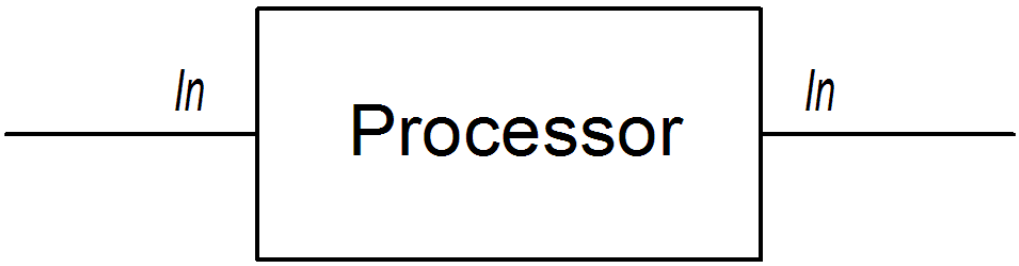
In Apache Camel, pipelining is the dominant paradigm for connecting nodes in a route definition. The pipeline concept is probably most familiar to users of the UNIX operating system, where it is used to join operating system commands. For example, **ls | more** is an example of a command that pipes a 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.

2.1.1. 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 below. The exchange's **Out** message remains **null** in this case.

Figure 2.1. Processor Modifying an In Message

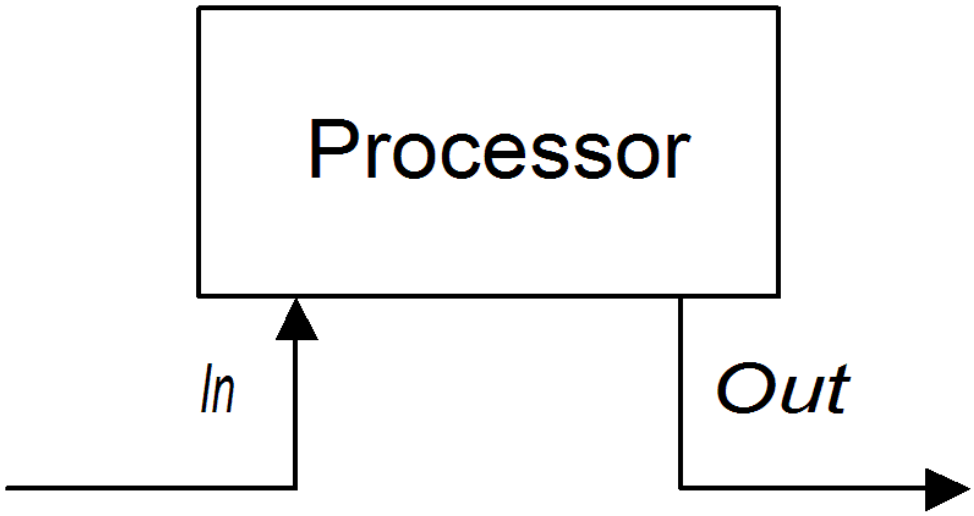


The following route shows a **setHeader()** command that modifies the current **In** message by adding (or modifying) the **BillingSystem** heading:

```
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 below.

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**:

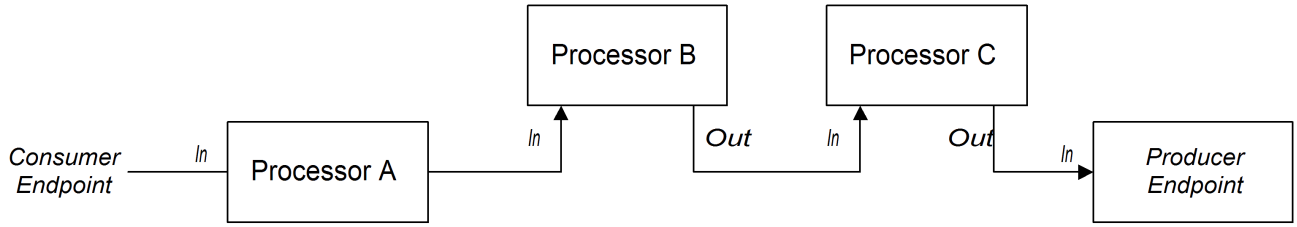
```
from("activemq:orderQueue")
    .transform(constant("DummyBody"))
    .to("activemq:billingQueue");
```

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

2.1.2. Pipeline for InOnly exchanges

The following figure 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. The processors' outputs and inputs are joined into a continuous pipeline.

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, **envelopeAddresses**:

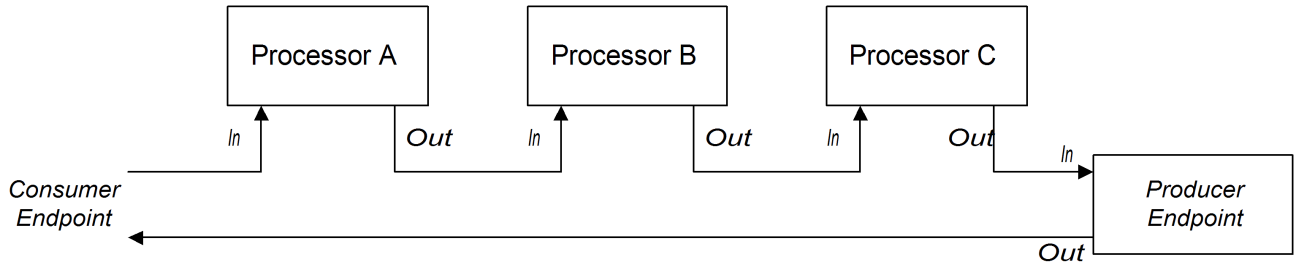
```
from("activemq:userdataQueue")
    .to(ExchangePattern.InOut, "velocity:file:AdressTemplate.vm")
    .to("activemq:envelopeAddresses");
```

Where the Velocity endpoint, **velocity:file:AddressTemplate.vm**, specifies the location of a Velocity template file, **file:AddressTemplate.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.

2.1.3. Pipeline for InOut exchanges

The following figure 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 way back to the consumer endpoint, where it provides the reply to the original request.

Figure 2.4. Sample Pipeline for InOut Exchanges



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

```
from("platform-http: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 endpoint.

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

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

2.1.4. Pipeline for InOptionalOut exchanges

The pipeline for **InOptionalOut** exchanges is essentially the same as the pipeline in the .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 **nullOut** message is copied to the **In** message of the next node in the pipeline. By contrast, in the case of an **InOut** exchange, a **nullOut** message is discarded and the original **In** message from the current node would be copied to the **In** message of the next node instead.

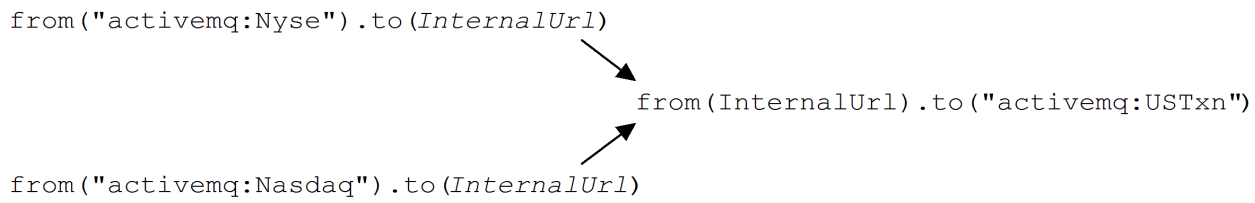
2.2. USING MULTIPLE INPUTS IN ROUTES

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 ContentEnricher pattern).

2.2.1. 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 below.

Figure 2.5. Processing Multiple Inputs with Segmented Routes



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:

- Direct endpoint
- SEDA 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.

2.2.2. 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 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.

2.2.3. 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:

- 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 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.

2.2.4. 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:

```
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.

2.3. EXCEPTION HANDLING IN ROUTES

Apache Camel provides several different mechanisms, which let you handle exceptions at different levels of granularity:

Within the route

Handle exceptions using **doTry**, **doCatch**, and **doFinally**.

Specify an action for each exception type

Apply this rule to all routes in a **RouteBuilder** using **onException**.

Specify an action forall exception types

Apply the rule to all routes in a **RouteBuilder** using **errorHandler**.

For more details about exception handling, see DeadLetter Channel section.

2.3.1. OnException Clause

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.

2.3.1.1. 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.

2.3.1.2. Java DSL example

In the following Java DSL example, the **onException** clause applies to all 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).

```
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");
  }
}
```

2.3.1.3. XML DSL example

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

```
<routes xmlns="http://camel.apache.org/schema/spring"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">
    <route>
       <from uri="seda:inputA"/>
       <onException>
          <exception>com.mycompany.ValidationException</exception>
          <to uri="activemq:validationFailed"/>
       </onException>
       <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>
</routes>
```

2.3.1.4. 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:

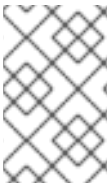
```
onException(ValidationException.class, BuesinessException.class)
  .to("activemq:validationFailed");
```

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 subtype 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,

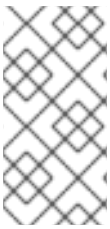
```
<throwException exceptionType="java.lang.IllegalArgumentException" message="${body}"/>
```

2.3.1.5. 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.

2.3.1.5.1. Use original message

By the time an exception is raised in the middle of a route, the message in the exchange could have been modified considerably (and might not even be readable by a human). Often, it is easier for an administrator to decide what corrective actions to take, if the messages visible in the deadletter queue are the **original** messages, as received at the start of the route. The **useOriginalMessage** option is **false** by default, but will be auto-enabled if it is configured on an error handler.



NOTE

The **useOriginalMessage** option can result in unexpected behavior when applied to Camel routes that send messages to multiple endpoints, or split messages into parts. The original message might not be preserved in a **Multicast**, **Splitter**, or **RecipientList** route in which intermediate processing steps modify the original message.

In the Java DSL, you can replace the message in the exchange by the original message. Set the **setAllowUseOriginalMessage()** to **true**, then use 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:

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

NOTE

If the **setAllowUseOriginalMessage()** option is set to **true**, 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()**. However, if the **setAllowUseOriginalMessage()** option is set to **false** (this is the default) on the Camel context, the original message will **not** be accessible and you cannot call **useOriginalMessage()**.

A reason to exploit the default behavior is to optimize performance when processing large messages.

2.3.1.6. 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.

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.

2.3.1.7. 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:

```
// 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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
  <route>
    <onException>
      <exception>com.mycompany.MyUserException</exception>
      <onWhen>
        <simple>${header.user} != null</simple>
      </onWhen>
      <redeliveryPolicy maximumRedeliveries="2"/>
      <to uri="activemq:error_user_queue"/>
    </onException>
    <onException>
      <exception>com.mycompany.MyUserException</exception>
      <redeliveryPolicy maximumRedeliveries="2"/>
      <to uri="activemq:error_queue"/>
    </onException>
    <!-- Main route definition would go here -->
  </route>
</routes>
```

2.3.1.8. 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:

- 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.
- 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.
- 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.

2.3.1.9. 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:

```
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:

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

2.3.1.10. 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:

```
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:

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

2.3.1.11. 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:

```
    // 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>
```

2.3.1.12. 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.

2.3.1.13. 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 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 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.

2.3.1.14. 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:

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

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

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

2.3.2. Error Handler

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.

2.3.2.1. 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 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.

2.3.2.2. 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:

```
<camel xmlns="http://camel.apache.org/schema/spring"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://camel.apache.org/schema/spring
                           http://camel.apache.org/schema/spring/camel-spring.xsd">
    <!-- Error Handler Configuration -->
    <routeConfiguration>
        <errorHandler type="DeadLetterChannel">
            <deadLetterUri>activemq:deadLetter</deadLetterUri>
        </errorHandler>
    </routeConfiguration>
    <!-- Route from ActiveMQ to POP3 -->
    <route>
        <from uri="activemq:orderQueue"/>
        <to uri="pop3://fulfillment@acme.com"/>
    </route>
    <!-- Route from File to File -->
    <route>
        <from uri="file:src/data?noop=true"/>
        <to uri="file:target/messages"/>
    </route>
</camel>
```

2.3.2.3. Types of error handler

The following table 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
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

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.

2.3.3.1. 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.

- Rethrowing exceptions – you can rethrow the current exception from within a **doCatch** clause using constructs.

2.3.3.2. 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 feature is specific to **doCatch()**:

- **Conditional catching** – you can catch an exception conditionally, by appending an **onWhen** sub-clause to the **doCatch** clause.

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:

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <doTry>
            <process ref="processorFail"/>
            <to uri="mock:result"/>
            <doCatch>
                <onException>
                    <exception>java.io.IOException</exception>

                    <handled>
                        <constant>false</constant>
                    </handled>
                </onException>
                <to uri="mock:io"/>
            </doCatch>
            <doCatch>
                <!-- and catch all other exceptions they are handled by default (handled = true) -->
                <exception>java.lang.Exception</exception>
                <to uri="mock:error"/>
            </doCatch>
        </doTry>
    </route>
</routes>
```

2.3.3.3. Rethrowing exceptions in doCatch

It is possible to rethrow an exception in a **doCatch()** clause, using constructs, as follows:

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

```
        .to("mock:io")
        // Rethrow the exception using a construct instead of `handled(false)` which is deprecated in a
`doTry/doCatch` clause.
        .throwException(new IllegalArgumentException("Forced"))
    .doCatch(Exception.class)
        // Catch all other exceptions.
        .to("mock:error")
    .end();
```

You can also re-throw an exception using a processor instead of **handled(false)** which is deprecated in a **doTry**/**doCatch** clause:

```
.process(exchange -> {throw exchange.getProperty(Exchange.EXCEPTION_CAUGHT,
Exception.class);})
```

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>
            <to uri="mock:io"/>
            <throwException message="Forced" exceptionType="java.lang.IllegalArgumentException"/>
        </doCatch>
        <doCatch>
            <!-- Catch all other exceptions. -->
            <exception>java.lang.Exception</exception>
            <to uri="mock:error"/>
        </doCatch>
    </doTry>
</route>
```

2.3.3.4. Conditional exception catching using onWhen

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

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

```
from("direct:start")
    .doTry()
        .process(new ProcessorFail())
        .to("mock:result")
    .doCatch(IOException.class, IllegalStateException.class)
        .*onWhen*(exceptionMessage().contains("Severe"))
```

```
        .to("mock:catch")
    .doCatch(CamelExchangeException.class)
        .to("mock:catchCamel")
    .doFinally()
        .to("mock:finally")
    .end();
```

Or equivalently, in Spring XML:

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

2.3.3.5. 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:

```
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

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.

2.3.4.1. 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** bean:

```
public CxfEndpoint router() {
    CxfEndpoint routerEndpoint = new CxfEndpoint();
    routerEndpoint.setAddress("http://localhost:9002/TestMessage");
    routerEndpoint.setWsdlURL("ship.wsdl");
    QName endpointQName = new QName("http://test", "TestSoapEndpoint");
    routerEndpoint.setEndpointNameAsQName(endpointQName);
    QName serviceQName = new QName("http://test", "TestService");
    routerEndpoint.setServiceNameAsQName(serviceQName);
    Map<String, Object> properties = new HashMap<>();
    // enable sending the stack trace back to client; the default value is false
    properties.put("faultStackTraceEnabled", true);
    routerEndpoint.setProperties(properties);
    routerEndpoint.setDataFormat(DataFormat.PAYLOAD);

    return routerEndpoint;
}
```

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:

```
public CxfEndpoint router() {
    CxfEndpoint routerEndpoint = new CxfEndpoint();
    routerEndpoint.setAddress("http://localhost:9002/TestMessage");
    routerEndpoint.setWsdlURL("ship.wsdl");
    QName endpointQName = new QName("http://test", "TestSoapEndpoint");
    routerEndpoint.setEndpointNameAsQName(endpointQName);
    QName serviceQName = new QName("http://test", "TestService");
    routerEndpoint.setServiceNameAsQName(serviceQName);
    Map<String, Object> properties = new HashMap<>();
// enable to show the cause exception message and the default value is false
    properties.put("faultStackTraceEnabled", true);
// enable to send the stack trace back to client,  the default value is false
    properties.put("exceptionMessageCauseEnabled", true);
    routerEndpoint.setProperties(properties);
```

```
    routerEndpoint.setDataFormat(DataFormat.PAYLOAD);
    return routerEndpoint;
}
```

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

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 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.

2.4.1. 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 or Spring. Registry entries are usually created implicitly (for example, when you instantiate a bean in a Spring XML file).

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.

2.4.2. 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");
```

2.4.3. Accessing overloaded bean methods

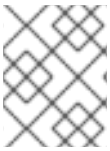
If a bean defines overloaded methods, you can choose which of the overloaded methods to invoke by specifying the method name along with its parameter types. For example, if the **MyBeanBrocessor** class has two overloaded methods, **processBody(String)** and **processBody(String,String)**, you can invoke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

2.4.4. 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.

In addition to the simple type values, you can also specify parameter values using 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");
```

2.4.5. 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:

- Method signature for processing message bodies
- Method signature for processing exchanges

2.4.5.1. 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:

```
package com.acme;

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

-

2.4.5.2. 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:

```
package com.acme;

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

2.4.6. Accessing a bean from XML IO DSL

2.4.6.1. Registry beans

With **xml-io-dsl** you can declare some beans to be bound to the Camel Registry. Beans may be declared in XML and have their properties (also nested) defined. For example:

```
<camel>

 <bean name="beanFromProps" type="com.acme.MyBean">
  <properties>
   <property key="field1" value="f1_p" />
   <property key="field2" value="f2_p" />
   <property key="nested.field1" value="nf1_p" />
   <property key="nested.field2" value="nf2_p" />
  </properties>
 </bean>

</camel>
```

While keeping all the benefits of fast XML parser used by **xml-io-dsl**, Camel can also process XML elements declared in other XML namespaces and process them separately. With this mechanism it is possible to include XML elements using Spring's **http://www.springframework.org/schema/beans** namespace.

This brings the flexibility of Spring Beans into Camel Main without actually running any Spring Application Context (or Spring Boot). When elements from Spring namespace are found, they are used to populate and configure an instance of **org.springframework.beans.factory.support.DefaultListableBeanFactory** and leverage Spring dependency injection to wire the beans together. These beans are then exposed through normal Camel Registry and may be used by Camel routes.

Here's an example **camel.xml** file, which defines both the routes and beans used (referred to) by the route definition:

camel.xml

-

```
<camel>

  <beans xmlns="http://www.springframework.org/schema/beans">
    <bean id="messageString" class="java.lang.String">
      <constructor-arg index="0" value="Hello"/>
    </bean>

    <bean id="greeter" class="org.apache.camel.main.app.Greeter">
      <description>Spring Bean</description>
      <property name="message">
        <bean class="org.apache.camel.main.app.GreeterMessage">
          <property name="msg" ref="messageString"/>
        </bean>
      </property>
    </bean>
  </beans>

  <route id="my-route">
    <from uri="direct:start"/>
    <bean ref="greeter"/>
    <to uri="mock:finish"/>
  </route>

</camel>
```

A **my-route** route is referring to **greeter** bean which is defined using Spring **<bean>** element.

2.4.6.1.1. Using bean with constructors

If you need to create beans with constructor arguments, this can be done:

```
<camel>

 <bean name="beanFromProps" type="com.acme.MyBean">
      <constructors>
        <constructor index="0" value="true"/>
        <constructor index="1" value="Hello World"/>
      </constructors>
      <!-- and you can still have properties -->
  <properties>
   <property key="field1" value="f1_p" />
   <property key="field2" value="f2_p" />
   <property key="nested.field1" value="nf1_p" />
   <property key="nested.field2" value="nf2_p" />
  </properties>
 </bean>

</camel>
```

2.4.6.1.2. Creating beans from factory method

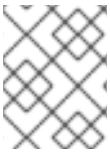
You can also create beans from a (public static) factory method as shown below:

```
 <bean name="myBean" type="com.acme.MyBean" factoryMethod="createMyBean">
```

```
    <constructors>
      <constructor index="0" value="true"/>
      <constructor index="1" value="Hello World"/>
    </constructors>
</bean>
```

When using **factoryMethod** then the arguments to this method is taken from **constructors**. This means that the example class **com.acme.MyBean** should be as follows:

```
public class MyBean {

    public static MyBean createMyBean(boolean important, String message) {
        MyBean answer = ...
        // create and configure the bean
        return answer;
    }
}
```



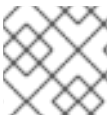
NOTE

The factory method must be **public static** and from the same class as the created class itself.

2.4.6.1.3. Creating beans from builder classes

A bean can also be created from another builder class as shown below:

```
<bean name="myBean" type="com.acme.MyBean"
      builderClass="com.acme.MyBeanBuilder" builderMethod="createMyBean">
    <properties>
      <property key="id" value="123"/>
      <property key="name" value="Acme"/>
    </constructors>
</bean>
```



NOTE

The builder class must be **public** and have a no-arg default constructor.

The builder class is then used to create the actual bean by using fluent builder style configuration. So the properties will be set on the builder class, and the bean is created by invoking the **builderMethod** at the end. The invocation of this method is done via Java reflection.

2.4.6.1.4. Creating beans from factory bean

A bean can also be created from a factory bean as shown below:

```
<bean name="myBean" type="com.acme.MyBean"
      factoryBean="com.acme.MyHelper" factoryMethod="createMyBean">
    <constructors>
      <constructor index="0" value="true"/>
```

```
        <constructor index="1" value="Hello World"/>
      </constructors>
</bean>
```

TIP

factoryBean can also refer to an existing bean by bean id instead of FQN classname.

When using **factoryBean** and **factoryMethod** then the arguments to this method is taken from **constructors**. So in the example above, this means that class **com.acme.MyHelper** should be as follows:

```
public class MyHelper {

    public static MyBean createMyBean(boolean important, String message) {
        MyBean answer = ...
        // create and configure the bean
        return answer;
    }
}
```

NOTE

The factory method must be **public static**.

2.4.6.1.5. Using init and destroy methods on beans

Sometimes beans need to do some initialization and cleanup work before a bean is ready to be used. For this you can use **initMethod** and **destroyMethod** that Camel triggers accordingly.

Those methods must be public void and have no arguments, as shown below:

```
public class MyBean {

    public void initMe() {
        // do init work here
    }

    public void destroyMe() {
        // do cleanup work here
    }

}
```

You then have to declare those methods in XML DSL as follows:

```
<bean name="myBean" type="com.acme.MyBean"
        initMethod="initMe" destroyMethod="destroyMe">
      <constructors>
        <constructor index="0" value="true"/>
        <constructor index="1" value="Hello World"/>
      </constructors>
</bean>
```

The init and destroy methods are optional, so a bean does not have to have both, for example you may only have destroy methods.

2.4.7. Accessing a bean from Spring XML

You can create a bean instance using Spring XML. To define a bean in XML, use the standard **bean** element.

The following example shows how to create an instance of **MyBeanProcessor**:

```
<camel xmlns="http://camel.apache.org/schema/spring">
    ...
    <bean name="myBeanId" type="com.acme.MyBeanProcessor"/>
</camel>
```

It is also possible to pass data to the bean's constructor arguments using XML IO DSL syntax.

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

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

Where the **beanRef()** processor invokes the **MyBeanProcessor.processBody()** method on the specified bean instance.

You can also invoke the bean from within an XML IO DSL route, using the Camel schema's **bean** element. For example:

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

    <bean name="myBeanId" type="com.acme.MyBeanProcessor"/>

    <route id="_route1">
        <from uri="file:data/inbound"/>
        <to uri="bean:myBeanId?method=processBody"/>
        <to uri="file:data/outbound"/>
    </route>
</camel>
```

2.4.8. Accessing a 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:

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

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

```
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;
```

2.4.9. Parameter binding annotations

The basic parameter bindings 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:

- Basic annotations
- Language annotations
- Inherited annotations

2.4.9.1. Basic annotations

The following table 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 2.2. Basic Bean Annotations

Annotation	Meaning	Parameter?
@Body	Binds to an inbound message body.	
@Header	Binds to an inbound message header.	String name of the header.
@Headers	To bind to the Map of the message headers.	
@Variable	To bind to a named variable	String name of the variable
@Variables	To bind to the variables map	

Annotation	Meaning	Parameter?
@ExchangeProperty	Binds to a named exchange property.	String name of the property.
@ExchangeProperties	To bind to the exchange property map on the exchange	
@ExchangeException	To bind to an Exception set on the exchange	

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

```
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.getMessage().setBody(body + "UserName = " + user);
    }
}
```

You can mix annotations with default conventions. The parameter binding automatically injects both the annotated arguments and the exchange object into the **org.apache.camel.Exchange** argument.

2.4.9.2. 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.

The following table 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 2.3. Expression Language Annotations

Annotation	Description
@Bean	Injects a Bean expression.
@Constant	Injects a Constant expression
@CSimple	Injects a CSimple expression

Annotation	Description
@EL	Injects an EL expression.
@exchangeProperty	Injects an exchangeProperty expression.
@Groovy	Injects a Groovy expression.
@Header	Injects a Header expression.
@HL7	Injects an HEl7 Terser expression.
@JavaScript	Injects a JavaScript expression.
@JsonPath	Injects a JsonPath expression
@JQ	Injects a JQ expression.
@Simple	Injects a Simple expression.
@XPath	Injects an XPath 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:

```
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:

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

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

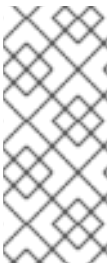
```
        ...
    }
}
```

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:

```
<camel>
...
    <bean name="myCorrIdGenerator" type="com.acme.MyIdGenerator"/>
</camel>
```

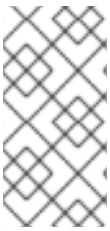
Where the **MyIdGenerator** class could be defined as follows:

```
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;
    }
}
```



NOTE

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, 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.

2.4.9.3. 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:

```
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:

```
import org.apache.camel.*;

public class MyBeanProcessor implements MyBeanProcessorIntf {
    public void processExchange(
        String user,  // Inherits Header annotation
        String body,  // Inherits Body annotation
        Exchange exchange
    ) {
        ...
    }
}
```

2.4.9.4. 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:

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

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

```
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");
```

2.4.9.5. 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()**:

```
...
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:

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

NOTE

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**.

2.5. CREATING EXCHANGE INSTANCES

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.

2.5.1. 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.

2.5.2. 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:

-

```
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();
```

2.5.3. 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

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

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 below appends the text, **World!**, to the end of the incoming message body.

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.

2.6.1.1. 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**

2.6.1.2. 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 shows the **ProcessorDefinition** methods that are relevant to transforming message content:

Table 2.4. 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.

Method	Description
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.

2.6.1.3. 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 below summarizes the static methods available in the **Builder** class.

Table 2.5. 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.

Method	Description
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 sending the exchange to the given endpoint uri.
static <E extends Exchange> ValueBuilder<E> systemProperty(String name)	Returns an expression for the given system property.
static <E extends Exchange> ValueBuilder<E> systemProperty(String name, String defaultValue)	Returns an expression for the given system property.

2.6.1.4. 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. The following table 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 2.6. 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.

Method	Description
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.
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.

Method	Description
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

2.6.2.1. 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.

2.6.2.2. Data formats

Marshalling and unmarshalling support is provided by a number of data formats. The following example uses JAXB.

There are many other possible dataformats, (for example JSON), see the Apache Camel Component reference Data formats section.

2.6.2.3. 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:

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

2.6.3. Endpoint Bindings

2.6.3.1. 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.

2.6.3.2. 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. 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 below 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:

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>
```

2.6.3.3. Associating a binding with an endpoint

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

- Binding URI
- Component

2.6.3.3.1. 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 the above example.

```
<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>
```

2.6.3.3.2. 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>
```

2.6.3.3.3. 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.

2.6.3.3.4. 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.

2.6.3.3.5. Binding interface

Following example shows the definition of the **org.apache.camel.spi.Binding** interface, which you must implement to define a custom binding.

The org.apache.camel.spi.Binding Interface

```
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 it is passed to the endpoint consumer producer.
     */
    Processor createConsumeProcessor();
}
```

2.6.3.3.6. 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

Camel has extensive support for property placeholders, which can be used *almost anywhere* in Camel.

Property placeholders are used to define a *placeholder* instead of the actual value. This is important as you would want to be able to make your applications external configurable, such as values for network addresses, port numbers, authentication credentials, login tokens, and configuration in general.

2.7.1. Properties component

Camel provides the Properties out of the box from the core, which is responsible for handling and resolving the property placeholders.

See the Properties documentation for how to configure Camel to known from which location(a) to load properties.

2.7.2. Property placeholder syntax

The value of a Camel property can be obtained by specifying its key name within a property placeholder, using the following syntax: **{{key}}**

For example:

```
{{file.uri}}
```

where **file.uri** is the property key.

Property placeholders can be used to specify parts, or all, of an endpoint's URI by embedding one or more placeholders in the URI's string definition.

2.7.2.1. Using property placeholder with default value

You can specify a default value to use if a property with the key does not exist, where the default value is the text after the colon:

```
{{file.url:/some/path}}
```

In this case the default value is /**some**/**path**.

2.7.2.2. Using optional property placeholders

Camel's elaborate property placeholder feature supports optional placeholders, which is defined with the **?** (question mark) as prefix in the key name, as shown:

```
{{?myBufferSize}}
```

If a value for the key exists then the value is used, however if the key does not exist, then Camel understands this:

```
file:foo?bufferSize={{?myBufferSize}}
```

Then the **bufferSize** option will only be configured in the endpoint, if a placeholder exists. Otherwise the option will not be set on the endpoint, meaning the endpoint would be *restructued* as:

```
file:foo
```

Then the option **bufferSize** is not in specified at all, and this would allow Camel to use the standard default value for **bufferSize** if any exists.

2.7.2.3. Reverse a boolean value

If a property placeholder is a boolean value, then it is possible to negate (reverse) the value by using **!** as prefix in the key.

```
integration.ftpEnabled=true
```

```
from("ftp:....").autoStartup("{{integration.ftpEnabled}}")
    .to("kafka:cheese")

from("jms:....").autoStartup("{{!integration.ftpEnabled}}")
    .to("kafka:cheese")
```

In the example above then the FTP route or the JMS route should only be started. So if the FTP is enabled then JMS should be disable, and vise-versa. We can do this be negating the **autoStartup** in the JMS route, by using **!integration.ftpEnabled** as the key.

2.7.3. Using property placeholders

When using property placeholders in the endpoint URIs you should use this with the syntax **{{key}}** as shown in this example:

```
cool.end = mock:result
where = cheese
```

And in Java DSL:

```
from("direct:start")
    .to("{{cool.end}}");
```

And in XML DSL:

```
<route>
  <from uri="direct:start"/>
  <to uri="{{cool.end}}"/>
</route>
```

A property placeholder may also just be a one part in the endpoint URI. A common use-case is to use a placeholder for an endpoint option such as the size of the write buffer in the file endpoint:

```
buf = 8192
```

```
from("direct:start")
    .to("file:outbox?bufferSize={{buf}}");
```

And in XML DSL:

```
<route>
  <from uri="direct:start"/>
  <to uri="file:outbox?bufferSize={{buf}}"/>
</route>
```

However the placeholder can be anywhere, so it could also be the name of a mock endpoint

```
from("direct:start")
    .to("mock:{{where}}");
```

In the example above the mock endpoint, is already hardcoded to start with **mock:**, and the **where** placeholder has the value **cheese** so the resolved uri becomes **mock:cheese**.

2.7.3.1. Property placeholders referring to other properties (nested placeholders)

You can also have properties with refer to each other such as:

```
cool.foo=result
cool.concat=mock:{{cool.foo}}
```

Notice how **cool.concat** refer to another property.

And the route in XML:

```
<route>
  <from uri="direct:start"/>
  <to uri="{{cool.concat}}"/>
</route>
```

2.7.3.1.1. Turning off nested placeholders

If the placeholder value contains data that interfere with the property placeholder syntax **{{** and **}}** (such as JSon data), you can be then explicit turn off nested placeholder by **?nested=false** in the key name, such as shown:

```
<route>
  <from uri="direct:start"/>
  <to uri="elasticsearch:foo?query={{myQuery?nested=false}}"/>
</route>
```

In the example above the placeholder *myQuery* placeholder value is as follows

```
{"query":{"match_all":{}}}
```

Notice how the json query ends with **}}** which interfere with the Camel property placeholder syntax.

Nested placeholders can also be turned off globally on the Properties component, such as:

```
CamelContext context = ...
context.getPropertiesComponent().setNestedPlaceholder(false);
```

2.7.3.2. Escape a property placeholder

The property placeholder can be problematic if the double curly brackets are used by a third party library like for example a query in ElasticSearch of type **{"query":{"match_all":{}}}**.

To work around that it is possible to escape the double curly brackets with a backslash character like for example **\{{ property-name \}}**. This way, it won't be interpreted as a property placeholder to resolve and will be resolved as **{{ property-name }}**.

If for some reason, the backslash character before the double curly brackets must not be interpreted as an escape character, it is possible to add another backslash in front of it to escape it, it will then be seen as a backslash.

2.7.3.3. Using property placeholders multiple times

You can of course also use placeholders several times:

```
cool.start=direct:start
cool.showid=true
cool.result=result
```

And in this route we use **cool.start** two times:

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

2.7.3.4. Using property placeholders with producer template

You can also your property placeholders when using a ProducerTemplate, for example:

```
template.sendBody("{{cool.start}}", "Hello World");
```

2.7.3.5. Using property placeholders with consumer template

This can also be done when using ConsumerTemplate, such as:

```
Object body = template.receiveBody("{{cool.start}}");
```

2.7.4. Resolving property placeholders on cloud

When you are running your Camel application on the cloud you may want to automatically scan any Configmap or Secret as it was an application properties. Given the following Secret:

```
apiVersion: v1
data:
  my-property: Q2FtZWwgNC44
kind: Secret
metadata:
  name: my-secret
type: Opaque
```

You can mount it in your Pod container, for instance, under **/etc/camel/conf.d/_secrets/my-secret**. Now, just make your Camel application be aware where to scan your configuration via **camel.main.cloud-properties-location = /etc/camel/conf.d/_secrets/my-secret** application properties. It's a comma separated value, so, you can add as many Secrets/Configmaps you need.

At runtime, you will be able to read the configuration transparently as **{{ my-property }}** as you're doing with the rest of properties.

NOTE

the same configuration works with Configmap.

2.7.5. Resolving property placeholders from Java code

If you need to resolve property placeholder(s) from some Java code, then Camel has two APIs for this:

- You can use the method **resolveProperty** on the **PropertiesComponent** to resolve a single property from Java code.
- Use the method **resolvePropertyPlaceholders** on the **CamelContext** to resolve (one or more) property placeholder(s) in a String.

For example to resolve a placeholder with key foo, you can do:

```
Optional<String> prop = camelContext.getPropertiesComponent().resolveProperty("foo");
if (prop.isPresent()) {
    String value = prop.get();
    ....
}
```

This API is to lookup a single property and returns a **java.util.Optional** type.

The **CamelContext** have another API which is capable of resolving multiple placeholders, and interpolate placeholders from an input String. Lets try with an example to explain this:

```
String msg = camelContext.resolvePropertyPlaceholders("{{greeting}} Camel user, Camel is {{cool}} dont you think?");
```

The input string is a text statement which have two placeholders that will be resolved, for example:

```
greeting = Hi
cool = awesome
```

Will be resolved to:

```
Hi Camel user, Camel is awesome dont you think?
```

2.7.6. Using property placeholders for any kind of attribute in Spring XML files

Previously it was only the **xs:string** type attributes in the XML DSL that support placeholders. For example often a timeout attribute would be a **xs:int** type and thus you cannot set a string value as the placeholder key. This is now possible using a special placeholder namespace.

In the example below we use the **prop** prefix for the namespace **http://camel.apache.org/schema/placeholder**. Now we can use **prop:** as prefix to configure any kind of XML attributes in Spring XML files.

In the example below we want to use a placeholder for the **stopOnException** option in the Multicast EIP. The **stopOnException** is a **xs:boolean** type, so we cannot configure this as:

```
<multicast stopOnException="{{stop}}">
  ...
</multicast>
```

Instead, we must use the **prop:** namespace, so we must add this namespace in the top of the XML file in the **<beans>** tag.

To configure the option we must then use the **prop:optionName** as shown below:

```
<multicast prop:stopOnException="stop">
  ...
</multicast>
```

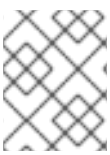
The complete example is below:

```
<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">

   <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:myprop.properties"/>
      <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>
```

In our properties file we have the value defined as:

```
stop = true
```

2.7.7. Bridging Camel property placeholders with Spring XML files



NOTE

If you are using Spring Boot then this does not apply. This is only for legacy Camel and Spring applications which are using Spring XML files.

The Spring Framework does not allow third-party frameworks such as Apache Camel to seamless hook into the Spring property placeholder mechanism. You can bridge Spring and Camel by declaring a Spring bean with the type **org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer**, which is a Spring **org.springframework.beans.factory.config.PropertyPlaceholderConfigurer** type.

To bridge Spring and Camel you must define a single bean as shown below:

```
<!-- bridge spring property placeholder with Camel -->
<!-- you must NOT use the <context:property-placeholder at the same time, only this bridge bean -->
<bean id="bridgePropertyPlaceholder"
class="org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer">
  <property name="location"
value="classpath:org/apache/camel/component/properties/cheese.properties"/>
</bean>
```

You **must not** use the spring **<context:property-placeholder>** namespace at the same time; this is not possible.

After declaring this bean, you can define property placeholders using both the Spring style, and the Camel style within the **<camelContext>** tag as shown below:

```
<!-- 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">
  <!-- in this route we use Camels property placeholder {{ }} style -->
  <route>
    <from uri="direct:{{cool.bar}}"/>
    <bean ref="hello"/>
    <to uri="{{cool.end}}"/>
  </route>
</camelContext>
```

Notice how the hello bean is using pure Spring property placeholders using the **${}** notation. And in the Camel routes we use the Camel placeholder notation with **{{key}}**.

2.7.8. Using property placeholder functions

The Properties component includes the following functions out of the box:

- **env** - A function to lookup the property from OS environment variables
- **sys** - A function to lookup the property from Java JVM system properties
- **bean** - A function to lookup the property from the return value of bean's method (requires **camel-bean** JAR)
- **service** - A function to lookup the property from OS environment variables using the service naming idiom
- **service.name** - A function to lookup the property from OS environment variables using the service naming idiom returning the hostname part only

- **service.port** - A function to lookup the property from OS environment variables using the service naming idiom returning the port part only

These functions are intended to make it easy to lookup values from the environment, as shown in the example below:

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <to uri="{{env:SOMENAME}}"/>
    <to uri="{{sys:MyJvmPropertyName}}"/>
  </route>
</camelContext>
```

You can use default values as well, so if the property does not exist, you can define a default value as shown below, where the default value is a **log:foo** and **log:bar** value.

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <to uri="{{env:SOMENAME:log:foo}}"/>
    <to uri="{{sys:MyJvmPropertyName:log:bar}}"/>
  </route>
</camelContext>
```

The service function is for looking up a service which is defined using OS environment variables using the service naming idiom, to refer to a service location using **hostname : port**

- *NAME*_SERVICE_HOST
- *NAME*_SERVICE_PORT

in other words the service uses **_SERVICE_HOST** and **_SERVICE_PORT** as prefix. So if the service is named FOO, then the OS environment variables should be set as

```
export $FOO_SERVICE_HOST=myserver
export $FOO_SERVICE_PORT=8888
```

For example if the FOO service a remote HTTP service, then we can refer to the service in the Camel endpoint uri, and use the HTTP component to make the HTTP call:

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <to uri="http://{{service:FOO}}/myapp"/>
  </route>
</camelContext>
```

And we can use default values if the service has not been defined, for example to call a service on localhost, maybe for unit testing.

```
<camelContext>
<route>
  <from uri="direct:start"/>
```

```
    <to uri="http://{{service:FOO:localhost:8080}}/myapp"/>
</route>
</camelContext>
```

The bean function (you need to have **camel-bean** JAR on classpath) is for looking up the property from the return value of bean's method.

Assuming we have registered a bean named 'foo' that has a method called 'bar' that returns a directory name, then we can refer to the bean's method in the camel endpoint url, and use the file component to poll a directory:

```
<camelContext>
<route>
    <from uri="file:{{bean:foo.bar}}"/>
    <to uri="direct:result"/>
</route>
</camelContext>
```

IMPORTANT

The method must be a public no-arg method (i.e. no parameters) and return a value such as a String, boolean, int.

2.7.8.1. Using Kubernetes property placeholder functions

The **camel-kubernetes** component include the following functions:

- **configmap** - A function to lookup the string property from Kubernetes ConfigMaps.
- **configmap-binary** - A function to lookup the binary property from Kubernetes ConfigMaps.
- **secret** - A function to lookup the string property from Kubernetes Secrets.
- **secret-binary** - A function to lookup the binary property from Kubernetes Secrets.

The syntax for both functions are:

```
configmap:name/key[:defaultValue]
```

Where the default value is optional, for example the following will lookup **myKey**, and fail if there is no such configmap.

```
configmap:mymap/mykey
```

In this example then it would not fail as a default value is provided:

```
configmap:mymap/mykey:123
```

If the value stored in the configmap is in binary format, so it is stored as **Binary Data**, it will be downloaded in a file, and it returns the absolute path of the file

```
configmap-binary:mymap/mybinkey
```

it returns a path like **/tmp/camel1178754591615046747 4/mybinkey**

Before the Kubernetes property placeholder functions can be used they need to be configured with either (or both)

- path - A *mount path* that must be mounted to the running pod, to load the configmaps or secrets from local disk.
- kubernetes client - **Autowired** An **io.fabric8.kubernetes.client.KubernetesClient** instance to use for connecting to the Kubernetes API server.

Camel will first use *mount paths* (if configured) to lookup, and then fallback to use the **KubernetesClient**.

2.7.8.1.1. Configuring mount paths for ConfigMaps and Secrets

The configuration of the *mount path* are used by the given order:

1. Reading configuration property with keys **camel.kubernetes-config.mount-path-configmaps** and **camel.kubernetes-config.mount-path-secrets**.
2. Use JVM system property with key **camel.k.mount-path.configmaps** and **camel.k.mount-path.secrets** (Camel K compatible).
3. Use OS ENV variable with key **CAMEL_K_MOUNT_PATH_CONFIGMAPS** and **CAMEL_K_MOUNT_PATH_SECRETS** (Camel K compatible).

For example to use **/etc/camel/resources/** as mount path, you can configure this in the **application.properties**:

```
camel.kubernetes-config.mount-path-configmaps = /etc/camel/myconfig/
camel.kubernetes-config.mount-path-secrets = /etc/camel/mysecrets/
```

2.7.8.1.2. Configuring Kubernetes Client

Camel will autowire the **KubernetesClient** if a single instance of the client exists in the running application (lookup via the Registry). Otherwise, a new **KubernetesClient** is created. The client can be configured from either

- Using **camel.kubernetes-config.client.** properties (see below for example)
- Attempt to auto-configure itself by a combination of OS Environment variables, reading from **~./kube/config** configuration, and service account token file. For more details see the https://github.com/fabric8io/kubernetes-client documentation.

You most likely only need to explicit configure the **KubernetesClient** when you want to connect from a local computer to a remote Kubernetes cluster, where you can specify various options, such as the masterUrl and oauthToken as shown:

```
camel.kubernetes-config.client.masterUrl = https://127.0.0.1:50179/
camel.kubernetes-config.client.oauthToken = eyJhbGciOiJSUzI1NiIsImtpZCI...
```

The **KubernetesClient** has many options, see the https://github.com/fabric8io/kubernetes-client documentation.

If you only use *mount paths*, then it is good practice to disable **KubernetesClient** which can be done by setting enabled to false as show:

```
camel.kubernetes-config.client-enabled = false
```

When running your Camel applications inside an existing Kubernetes cluster, then you often would not need to explicit configure the **KubernetesClient** and can rely on default settings.

TIP

If you use Camel Quarkus, then it is recommended to use their https://quarkus.io/guides/kubernetes-config which automatic pre-configure the **KubernetesClient** which Camel then will reuse.

2.7.8.1.3. Using configmap with Kubernetes

Given a configmap named **myconfig** in Kubernetes that has two entries:

```
drink = beer
first = Carlsberg
```

Then these values can be used in your Camel routes such as:

```
<camelContext>
  <route>
    <from uri="direct:start"/>
    <log message="What {{configmap:myconfig/drink}} do you want?"/>
    <log message="I want {{configmap:myconfig/first}}"/>
  </route>
</camelContext>
```

You can also provide a default value in case a key does not exist:

```
    <log message="I want {{configmap:myconfig/second:Heineken}}"/>
```

2.7.8.1.4. Using secrets with Kubernetes

Camel reads ConfigMaps from the Kubernetes API Server. And when RBAC is enabled on the cluster, the ServiceAccount that is used to run the application needs to have the proper permissions for such access.

A secret named **mydb** could contain username and passwords to connect to a database such as:

```
myhost = killroy
myport = 5555
myuser = scott
mypass = tiger
```

This can be used in Camel with for example the Postrgres Sink Kamelet:

```
<camelContext>
  <route>
    <from uri="direct:rome"/>
    <setBody>
```

```
    <constant>{ "username":"oscerd", "city":"Rome"}</constant>
  </setBody>
  <to uri="kamelet:postgresql-sink?serverName={{secret:mydb/myhost}}
        &amp;serverPort={{secret:mydb/myport}}
        &amp;username={{secret:mydb/myuser}}
        &amp;password={{secret:mydb/mypass}}
        &amp;databaseName=cities
        &amp;query=INSERT INTO accounts (username,city) VALUES (:#username,:#city)"/>
 </route>
</camelContext>
```

The postgres-sink Kamelet can also be configured in **application.properties** which reduces the configuration in the route above:

```
camel.component.kamelet.postgresql-sink.databaseName={{secret:mydb/myhost}}
camel.component.kamelet.postgresql-sink.serverPort={{secret:mydb/myport}}
camel.component.kamelet.postgresql-sink.username={{secret:mydb/myuser}}
camel.component.kamelet.postgresql-sink.password={{secret:mydb/mypass}}
```

Which reduces the route to:

```
<camelContext>
 <route>
  <from uri="direct:rome"/>
  <setBody>
   <constant>{ "username":"oscerd", "city":"Rome"}</constant>
  </setBody>
  <to uri="kamelet:postgresql-sink?databaseName=cities
        &amp;query=INSERT INTO accounts (username,city) VALUES (:#username,:#city)"/>
 </route>
</camelContext>
```

2.7.8.1.5. Using configmap or secrets in local-mode

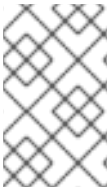
During development you may want to run in *local mode* where you do not need acces to a Kubernetes cluster, to lookup the configmap. In the local mode, then Camel will lookup the configmap *keys* from local properties, eg:

For example the example above with the postgresql kamelet, that was configured using a secret:

```
camel.component.kamelet.postgresql-sink.databaseName={{secret:mydb/myhost}}
camel.component.kamelet.postgresql-sink.serverPort={{secret:mydb/myport}}
camel.component.kamelet.postgresql-sink.username={{secret:mydb/myuser}}
camel.component.kamelet.postgresql-sink.password={{secret:mydb/mypass}}
```

Now suppose we have a local Postrgres database we want to use, then we can turn on *local mode* and specify the credentials in the same properties file:

```
camel.kubernetes-config.local-mode = true
mydb/myhost=localhost
mydb/myport=1234
mydb/myuser=scott
mydb/mypass=tiger
```



NOTE

Notice how the key is prefixed with the name of the secret and a slash, eg **name/key**. This makes it easy to copy/paste from the actual use of the configmap/secret and into the **application.properties** file.

2.7.8.2. Using custom property placeholder functions

The Properties component allow to plugin 3rd party functions which can be used during parsing of the property placeholders. These functions are then able to do custom logic to resolve the placeholders, such as looking up in databases, do custom computations, or whatnot. The name of the function becomes the prefix used in the placeholder.

This is best illustrated in the example route below, where we use **beer** as the prefix:

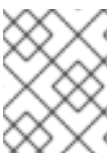
```
<route>
   <from uri="direct:start"/>
   <to uri="{{beer:FOO}}"/>
   <to uri="{{beer:BAR}}"/>
</route>
```

The implementation of the function is only two methods as shown below:

```
@org.apache.camel.spi.annotations.PropertiesFunction("beer")
public class MyBeerFunction implements PropertiesFunction {

   @Override
   public String getName() {
      return "beer";
   }

   @Override
   public String apply(String remainder) {
      return "mock:" + remainder.toLowerCase();
   }
}
```

The function must implement the **org.apache.camel.spi.PropertiesFunction** interface. The method **getName** is the name of the function (beer). And the **apply** method is where we implement the custom logic to do. As the sample code is from a unit test, it just returns a value to refer to a mock endpoint.

You also need to have **camel-component-maven-plugin** as part of building the component will then ensure that this custom properties function has necessary source code generated that makes Camel able to automatically discover the function.



NOTE

If the custom properties function need logic to startup and shutdown, then the function can extend **ServiceSupport** and have this logic in **doStart** and **doStop** methods.

TIP

For an example see the **camel-base64** component.

2.7.9. Using third party property sources

The properties component allows to plugin 3rd party sources to load and lookup properties via the **PropertySource** API from camel-api.

The regular **PropertySource** will lookup the property on-demand, for example to lookup values from a backend source such as a database or HashiCorp Vault etc.

A **PropertySource** can define that it supports loading all its properties (by implementing **LoadablePropertiesSource**) from the source at once, for example from file system. This allows Camel properties component to load these properties at once during startup.

For example the **camel-microprofile-config** component is implemented using this. The 3rd-party **PropertySource** can automatically be discovered from classpath when Camel is starting up. This is done by including the file **META-INF/services/org/apache/camel/property-source-factory** which refers to the fully qualified class name of the **PropertySource** implementation.

See MicroProfile Config component as an example.

You can also register 3rd-party property sources via Java API:

```
PropertiesComponent pc = context.getPropertiesComponent();
pc.addPropertiesSource(myPropertySource);
```

2.8. THREADING MODEL

2.8.1. 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.

2.8.2. 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 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.

2.8.3. Component threading model

Some Apache Camel components – such as SEDA, JMS, and Platform HTTP-- 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.

2.8.4. 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.

The Processor Threading Options table shows the various options for controlling and setting thread pools on the threading-aware processors built-in to Apache Camel.

Table 2.7. 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
threads	executorService() executorServiceRef() poolSize() maxPoolSize() keepAliveTime() timeUnit() maxQueueSize() rejectedPolicy()	@executorServiceRef @poolSize @maxPoolSize @keepAliveTime @timeUnit @maxQueueSize @rejectedPolicy

Processor	Java DSL	XML DSL
wireTap	wireTap(String uri, ExecutorService executorService) wireTap(String uri, String executorServiceRef)	@executorServiceRef

2.8.5. 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.

maxQueueSize()

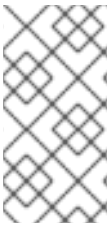
Maximum number of pending tasks that this thread pool can store in its incoming task queue.

allowCoreThreadTimeOut()

Sets default whether to allow core threads to timeout

rejectedPolicy()

Specifies what course of action to take, if the incoming task queue is full.



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.

2.8.6. 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>
```

2.8.7. 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 shownbelow (assuming that these settings have not been modified by the application code).

Table 2.8. Default Thread Pool Profile Settings

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

2.8.8. 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:

```
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:

```
<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 instead of replacing the default thread pool profile.

2.8.9. 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:

- **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.

2.8.10. 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 directly.

The following table shows the options supported by the **ThreadPoolBuilder** class, which you can set when defining a new custom thread pool.

Table 2.9. Thread Pool Builder Options

Builder Option	Description
maxQueueSize()	Sets the maximum number of pending tasks that this thread pool can store in its incoming task queue. A value of **-1** specifies an unbounded queue. Default value is taken from default thread pool profile.
poolSize()	Sets the minimum number of threads in the pool (this is also the initial pool size). Default value is taken from default thread pool profile.
maxPoolSize()	Sets the maximum number of threads that can be in the pool. Default value is taken from default thread pool profile.
keepAliveTime()	If any threads are idle for longer than this period of time (specified in seconds), they are terminated. This allows the thread pool to shrink when the load is light. Default value is taken from default thread pool profile.
rejectedPolicy()	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**:

```
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:

```
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:

```
<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>
```

2.8.11. 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:

```
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:

```
<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>
```

2.8.12. 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.

2.8.13. 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

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.

2.9.1. 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:

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

2.9.2. Disabling automatic start-up of routes

By default, all 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>
```

2.9.3. 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:

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

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

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

2.9.4. 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:

Startup Order in Java DSL

```
from("platform-http: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:

Startup Order in XML DSL

```
<route id="first" startupOrder="2">
    <from uri="platform-http: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).

2.9.5. 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.

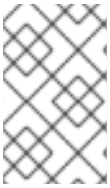
2. Normally, the shutdown strategy waits until the currently active exchanges have finshed processing. The treatment of running tasks is configurable, however.

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.

2.9.6. 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, 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).

2.9.7. 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 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:

```
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>
```

2.9.8. 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:

```
context = CamelContext instance
context.getShutdownStrategy().setTimeout(600);
```

2.9.9. Integration with custom components

If you are implementing a custom {CamelName} 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.

2.9.10. RouteIdFactory

Based on the consumer endpoints, you can add **RouteIdFactory** that can assign route ids with the logical names.

For example, when using the routes with SEDA or direct components as route inputs, then you may want to use their names as the route id, such as,

- **direct:foo** - foo
- **seda:bar** - bar
- **jms:orders** - orders

Instead of using auto-assigned names, you can use the **NodeIdFactory** that can assign logical names for routes. Also, you can use the context-path of route URL as the name. For example, execute the following to use the **RouteIDFactory**:

```
context.setNodeIdFactory(new RouteIdFactory());
```

NOTE

It is possible to get the custom route id from rest endpoints.

2.10. SCHEDULED ROUTE POLICY

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.

2.10.1. 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.

2.10.1.1. 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

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
```

2.10.2.1. 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.

2.10.2.2. Java DSL example

The Java DSL example below 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.

Java DSL Example of Simple Scheduled Route

```
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.

2.10.2.3. XML DSL example

The XML DSL example 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.

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.

2.10.2.4. 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:

```
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.

2.10.2.5. 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. 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 the route has finished processing the exchange.

2.10.2.6. 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, and so on, 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);
```

2.10.2.7. Scheduling tasks

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

- Starting a route
- Stopping a route
- Suspending a route
- Resuming a route

2.10.2.7.1. 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.

2.10.2.7.2. 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.

Parameter	Type	Default	Description
routeStopRepeatCount	**int**	**0**	When set to a non-zero value, specifies how many times the route should be stopped.
routeStopRepeatInterval	**long**	**0**	Specifies the time interval between stops, in units of milliseconds.
routeStopGracePeriod	**int**	**10000**	Specifies how long to wait for the current exchange to finish processing (grace period) before forcibly stopping the route. Set to 0 for an infinite grace period.
routeStopTimeUnit	**long**	**TimeUnit.MILLISECONDS**	Specifies the time unit of the grace period.

2.10.2.7.3. 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.

2.10.2.7.4. 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.
routeResumeRepeatCount	**int**	0	When set to a non-zero value, specifies how many times the route should be resumed.
routeResumeRepeatInterval	**long**	0	Specifies the time interval between resumes, in units of milliseconds.

2.10.3. Cron Scheduled Route Policy

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
```

2.10.3.1. 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.

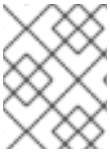
2.10.3.2. Java DSL example

Following example 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.

Java DSL Example of a Cron Scheduled Route

```
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.

2.10.3.3. XML DSL example

Following example 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.

XML DSL Example of a Cron Scheduled Route

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

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

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

NOTE

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

2.10.3.4. 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.

2.10.3.5. Scheduling tasks

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

- Starting a route
- Stopping a route
- Suspending a route
- Resuming a route

2.10.3.5.1. 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.

2.10.3.5.2. 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.

2.10.3.5.3. 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.

2.10.3.5.4. 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

2.10.4.1. Using Route Policy Factory

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. RELOADING CAMEL ROUTES

You can enable the live reload of your camel XML routes, which will trigger a reload, when you save the XML file from your editor. You can use this feature when using:

- Camel standalone with Camel Main class
- From Camel Spring Boot Maven plugin **mvn spring-boot:run**

You can also enable this manually, by setting a **ReloadStrategy** on the **CamelContext** and by providing your own custom strategies.

2.12. ONCOMPLETION

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. 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 to use a thread pool, though the default is no thread pool.

2.12.1. 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 invoked when the original route is complete.
       // This is similar to a 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 is executed when the exchange is done being routed. -->
   <!-- This callback is always triggered even if the exchange fails. -->
   <onCompletion>
      <!-- This is similar to 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 onCompletion is qualified to invoke only when the exchange fails (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>
```

2.12.2. 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");
```

2.12.3. 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");
```

2.12.4. Using onCompletion with or without a thread pool

The **onCompletion** option 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>
```

2.12.5. 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>
```

2.13. JMX NAMING

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

```
<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.

2.13.1. Customizing 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.

2.13.2. Specifying a name pattern in Java

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

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

2.13.3. Specifying a name pattern in XML

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

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

2.13.4. Name pattern tokens

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

Table 2.10. 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**).

2.13.5. 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>
```

2.13.6. 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>
```

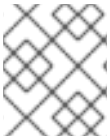
In this case, Apache Camel would fail on start-up and report an **MBean already exists** exception. You should, therefore, take extra care to ensure that you do not define ambiguous name patterns.

2.14. PERFORMANCE AND OPTIMIZATION

2.14.1. Message copying

The **allowUseOriginalMessage** option default setting is **false**, to cut down on copies being made of the original message when they are not needed. To enable the **allowUseOriginalMessage** option use the following commands:

- Set **useOriginalMessage=true** on any of the error handlers or on the **onException** element.
- In Java application code, set **AllowUseOriginalMessage=true**, then use the **getOriginalMessage** method.



NOTE

In Apache Camel, the default setting of `allowUseOriginalMessage` is true.

CHAPTER 3. SPRING BOOT MAVEN PLUGIN

The Spring Boot Maven Plugin provides Spring Boot support in Apache Maven. You must have Maven 3.6.3 or later to use it

The Spring Boot Maven Plugin allows you to package executable jar or war archives, run Spring Boot applications, generate build information and start your Spring Boot application prior to running integration tests.

3.1. GETTING STARTED

To use the Spring Boot Maven Plugin, include the appropriate XML in the **plugins** section of your **pom.xml**, as shown in the following example:

```
<project>
 <modelVersion>4.14.0</modelVersion>
 <artifactId>getting-started</artifactId>
 <!-- ... -->
 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
   </plugin>
  </plugins>
 </build>
</project>
```

3.2. USING THE PLUGIN

Maven users can inherit from the **spring-boot-starter-parent** project to obtain sensible defaults. The parent project provides the following features:

- Java 17 as the default compiler level.
- UTF-8 source encoding.
- Compilation with **-parameters**.
- A dependency management section, inherited from the **spring-boot-dependencies** POM, that manages the versions of common dependencies. This dependency management lets you omit **<version>** tags for those dependencies when used in your own POM.
- An execution of the **repackage** goal with a **repackage** execution id.
- A **native** profile that configures the build to be able to generate a Native image.
- Sensible resource filtering.
- Sensible plugin configuration ({url-git-commit-id-maven-plugin}[**Git Commit Id Plugin**], and shade).
- Sensible resource filtering for **application.properties** and **application.yml** including profile-specific files (for example, **application-dev.properties** and **application-dev.yml**)

NOTE

Since the **application.properties** and **application.yml** files accept Spring style placeholders (**${...}**), the Maven filtering is changed to use **@..@** placeholders.

(You can override that by setting a Maven property called **resource.delimiter**.)

NOTE

The **spring-boot-starter-parent** sets the **maven.compiler.release** property, which restricts the **--add-exports**, **--add-reads**, and **--patch-module** options if they modify system modules. In case you need to use those options, unset **maven.compiler.release**:

```
<maven.compiler.release></maven.compiler.release>
```

and then configure the source and the target options instead:

```
<maven.compiler.source>${java.version}</maven.compiler.source>
<maven.compiler.target>${java.version}</maven.compiler.target>
```

3.2.1. Inheriting the Starter Parent POM

To configure your project to inherit from the **spring-boot-starter-parent**, set the **parent** as follows:

```
<!-- Inherit defaults from Spring Boot -->
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>3.5.11</version>
</parent>
```

NOTE

You only need to specify the Spring Boot version number on this dependency.

If you import additional starters, you can safely omit the version number.

With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to use a different version of the SLF4J library and the Spring Data release train, you would add the following to your **pom.xml**:

```
<properties>
 <slf4j.version>2.0.17</slf4j.version>
 <spring-data-bom.version>2025.0.5</spring-data-bom.version>
</properties>
```

Browse the Dependency Versions Properties section in the Spring Boot reference for a complete list of dependency version properties.

WARNING

Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues and should be done with care.

3.2.2. Using Spring Boot without the Parent POM

There may be reasons for you not to inherit from the **spring-boot-starter-parent** POM. You may have your own corporate standard parent that you need to use or you may prefer to explicitly declare all your Maven configuration.

If you do not want to use the **spring-boot-starter-parent**, you can still keep the benefit of the dependency management (but not the plugin management) by using an **import** scoped dependency, as follows:

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <!-- Import dependency management from Spring Boot -->
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-dependencies</artifactId>
   <version>3.5.11</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

The preceding sample setup does not let you override individual dependencies by using properties, as explained above. To achieve the same result, you need to add entries in the **dependencyManagement** section of your project **before** the **spring-boot-dependencies** entry. For instance, to use a different version of the SLF4J library and the Spring Data release train, you could add the following elements to your **pom.xml**:

```
<dependencyManagement>
 <dependencies>
  <!-- Override SLF4J provided by Spring Boot -->
  <dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-api</artifactId>
   <version>1.7.30</version>
  </dependency>
  <!-- Override Spring Data release train provided by Spring Boot -->
  <dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-bom</artifactId>
   <version>2024.1.10</version>
   <type>pom</type>
   <scope>import</scope>
```

```
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-dependencies</artifactId>
   <version>3.5.11</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

3.2.3. Overriding Settings on the Command Line

The plugin offers a number of user properties, starting with **spring-boot**, to let you customize the configuration from the command line.

For instance, you could tune the profiles to enable when running the application as follows:

```
$ mvn spring-boot:run -Dspring-boot.run.profiles=dev,local
```

If you want to both have a default while allowing it to be overridden on the command line, you should use a combination of a user-provided project property and MOJO configuration.

```
<project>
 <properties>
  <app.profiles>local,dev</app.profiles>
 </properties>
 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
     <profiles>${app.profiles}</profiles>
    </configuration>
   </plugin>
  </plugins>
 </build>
</project>
```

The above makes sure that **local** and **dev** are enabled by default. Now a dedicated property has been exposed, this can be overridden on the command line as well:

```
$ mvn spring-boot:run -Dapp.profiles=test
```

3.3. GOALS

The Spring Boot Plugin has the following goals:

Goal	Description
spring-boot:build-image	Package an application into an OCI image using a buildpack, forking the lifecycle to make sure that package ran. This goal is suitable for command-line invocation. If you need to configure a goal execution in your build, use build-image-no-fork instead.
spring-boot:build-image-no-fork	Package an application into an OCI image using a buildpack, but without forking the lifecycle. This goal should be used when configuring a goal execution in your build. To invoke the goal on the command-line, use build-image instead.
spring-boot:build-info	Generate a build-info.properties file based on the content of the current MavenProject.
spring-boot:help	Display help information on spring-boot-maven-plugin. Call mvn spring-boot:help -Ddetail=true -Dgoal=<goal-name> to display parameter details.
spring-boot:process-aot	Invoke the AOT engine on the application.
spring-boot:process-test-aot	Invoke the AOT engine on tests.
spring-boot:repackage	Repackage existing JAR and WAR archives so that they can be executed from the command line using java -jar. With layout=NONE can also be used simply to package a JAR with nested dependencies (and no main class, so not executable).
spring-boot:run	Run an application in place.
spring-boot:start	Start a spring application. Contrary to the run goal, this does not block and allows other goals to operate on the application. This goal is typically used in integration test scenario where the application is started before a test suite and stopped after.
spring-boot:stop	Stop an application that has been started by the "start" goal. Typically invoked once a test suite has completed.
spring-boot:test-run	Run an application in place using the test runtime classpath. The main class that will be used to launch the application is determined as follows: The configured main class, if any. Then the main class found in the test classes directory, if any. Then the main class found in the classes directory, if any.

3.4. RUNNING YOUR APPLICATION WITH MAVEN

The plugin includes a run goal which can be used to launch your application from the command line, as shown in the following example:

```
$ mvn spring-boot:run
```

Application arguments can be specified using the **arguments** parameter.

The application is executed in a forked process and setting properties on the command-line will not affect the application. If you need to specify some JVM arguments (that is for debugging purposes), you can use the **jvmArguments** parameter, see Debug the application for more details. There is also explicit support for system properties and environment variables.

As enabling a profile is quite common, there is dedicated **profiles** property that offers a shortcut for **-Dspring-boot.run.jvmArguments="-Dspring.profiles.active=dev"**, see Specify active profiles.

Spring Boot **devtools** is a module to improve the development-time experience when working on Spring Boot applications. To enable it, just add the following dependency to your project:

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
 </dependency>
</dependencies>
```

When **devtools** is running, it detects changes when you recompile your application and automatically refreshes it. This works for not only resources but code as well. It also provides a LiveReload server so that it can automatically trigger a browser refresh whenever things change.

Devtools can also be configured to only refresh the browser whenever a static resource has changed (and ignore any change in the code). Just include the following property in your project:

```
spring.devtools.remote.restart.enabled=false
```

Prior to **devtools**, the plugin supported hot refreshing of resources by default which has now been disabled in favour of the solution described above. You can restore it at any time by configuring your project:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <configuration>
    <addResources>true</addResources>
   </configuration>
  </plugin>
 </plugins>
</build>
```

When **addResources** is enabled, any **src/main/resources** directory will be added to the application classpath when you run the application and any duplicate found in the classes output will be removed. This allows hot refreshing of resources which can be very useful when developing web applications. For example, you can work on HTML, CSS or JavaScript files and see your changes immediately without recompiling your application. It is also a helpful way of allowing your front end developers to work without needing to download and install a Java IDE.

NOTE

A side effect of using this feature is that filtering of resources at build time will not work.

In order to be consistent with the **repackage** goal, the **run** goal builds the classpath in such a way that any dependency that is excluded in the plugin's configuration gets excluded from the classpath as well.

Sometimes it is useful to run a test variant of your application.

Use the **test-run** goal with many of the same features and configuration options as **run** for this purpose.

3.4.1. spring-boot:run

org.springframework.boot:spring-boot-maven-plugin:3.5.6

Run an application in place.

3.4.1.1. Required parameters

Name	Type	Default
classesDirectory	**File**	**${project.build.outputDirectory}**

3.4.1.2. Optional parameters

Name	Type	Default
addResources	**boolean**	**false**
additionalClasspathElements	**String[]**	
arguments	**String[]**	
commandlineArguments	**String**	
environmentVariables	**Map**	
excludeGroupIds	**String**	
excludes	**List**	
includes	**List**	
jvmArguments	**String**	
mainClass	**String**	

Name	Type	Default
noverify	**boolean**	
optimizedLaunch	**boolean**	**true**
skip	**boolean**	**false**
systemPropertyVariables	**Map**	
useTestClasspath	**Boolean**	**false**
workingDirectory	**File**	

3.4.1.3. Parameter details

3.4.1.3.1. addResources

Add maven resources to the classpath directly, this allows live in-place editing of resources. Duplicate resources are removed from **target/classes** to prevent them from appearing twice if **ClassLoader.getResources()** is called. Please consider adding **spring-boot-devtools** to your project instead as it provides this feature and many more.

Name	**addResources**
Type	**boolean**
Default value	**false**
User property	**spring-boot.run.addResources**
Since	**1.0.0**

3.4.1.3.2. additionalClasspathElements

Additional classpath elements that should be added to the classpath. An element can be a directory with classes and resources or a jar file.

Name	**additionalClasspathElements**
Type	**java.lang.String[]**

Default value	
User property	**spring-boot.run.additional-classpath-elements**
Since	**3.2.0**

3.4.1.3.3. agents

Path to agent jars.

Name	**agents**
Type	**java.io.File[]**
Default value	
User property	**spring-boot.run.agents**
Since	**2.2.0**

3.4.1.3.4. arguments

Arguments that should be passed to the application.

Name	**arguments**
Type	**java.lang.String[]**
Default value	
User property	
Since	**1.0.0**

3.4.1.3.5. classesDirectory

Directory containing the classes and resource files that should be used to run the application.

Name	**classesDirectory**
Type	**java.io.File**
Default value	**${project.build.outputDirectory}**
User property	
Since	**1.0.0**

3.4.1.3.6. commandlineArguments

Arguments from the command line that should be passed to the application. Use spaces to separate multiple arguments and make sure to wrap multiple values between quotes. When specified, takes precedence over **#arguments**.

Name	**commandlineArguments**
Type	**java.lang.String**
Default value	
User property	**spring-boot.run.arguments**
Since	**2.2.3**

3.4.1.3.7. environmentVariables

List of Environment variables that should be associated with the forked process used to run the application.

Name	**environmentVariables**
Type	**java.util.Map**
Default value	
User property	

Since	**2.1.0**

3.4.1.3.8. **excludeGroupIds**

Comma separated list of groupId names to exclude (exact match).

Name	**excludeGroupIds**
Type	**java.lang.String**
Default value	
User property	**spring-boot.excludeGroupIds**
Since	**1.1.0**

3.4.1.3.9. **excludes**

Collection of artifact definitions to exclude. The **Exclude** element defines mandatory **groupId** and **artifactId** components and an optional **classifier** component. When configured as a property, values should be comma-separated with colon-separated components:
groupId:artifactId,groupId:artifactId:classifier

Name	**excludes**
Type	**java.util.List**
Default value	
User property	**spring-boot.excludes**
Since	**1.1.0**

3.4.1.3.10. **includes**

Collection of artifact definitions to include. The **Include** element defines mandatory **groupId** and **artifactId** components and an optional **classifier** component. When configured as a property, values should be comma-separated with colon-separated components:
groupId:artifactId,groupId:artifactId:classifier

Name	**includes**
Type	**java.util.List**
Default value	
User property	**spring-boot.includes**
Since	**1.2.0**

3.4.1.3.11. jvmArguments

JVM arguments that should be associated with the forked process used to run the application. On command line, make sure to wrap multiple values between quotes.

Name	**jvmArguments**
Type	**java.lang.String**
Default value	
User property	**spring-boot.run.jvmArguments**
Since	**1.1.0**

3.4.1.3.12. mainClass

The name of the main class. If not specified the first compiled class found that contains a 'main' method will be used.

Name	**mainClass**
Type	**java.lang.String**
Default value	
User property	**spring-boot.run.main-class**

Since	**1.0.0**

3.4.1.3.13. noverify

Flag to say that the agent requires -noverify.

Name	**noverify**
Type	**boolean**
Default value	
User property	**spring-boot.run.noverify**
Since	**1.0.0**

3.4.1.3.14. optimizedLaunch

Whether the JVM's launch should be optimized.

Name	**optimizedLaunch**
Type	**boolean**
Default value	**true**
User property	**spring-boot.run.optimizedLaunch**
Since	**2.2.0**

3.4.1.3.15. profiles

The spring profiles to activate. Convenience shortcut of specifying the 'spring.profiles.active' argument. On command line use commas to separate multiple profiles.

Name	**profiles**
Type	**java.lang.String[]**

Default value	
User property	**spring-boot.run.profiles**
Since	**1.3.0**

3.4.1.3.16. skip

Skip the execution.

Name	**skip**
Type	**boolean**
Default value	**false**
User property	**spring-boot.run.skip**
Since	**1.3.2**

3.4.1.3.17. systemPropertyVariables

List of JVM system properties to pass to the process.

Name	**systemPropertyVariables**
Type	**java.util.Map**
Default value	
User property	
Since	**2.1.0**

3.4.1.3.18. useTestClasspath

Flag to include the test classpath when running.

Name	**useTestClasspath**
Type	**java.lang.Boolean**
Default value	**false**
User property	**spring-boot.run.useTestClasspath**
Since	**1.3.0**

3.4.1.3.19. **workingDirectory**

Current working directory to use for the application. If not specified, basedir will be used.

Name	**workingDirectory**
Type	**java.io.File**
Default value	
User property	**spring-boot.run.workingDirectory**
Since	**1.5.0**

3.4.2. spring-boot:test-run

org.springframework.boot:spring-boot-maven-plugin:3.5.6

Run an application in place using the test runtime classpath. The main class that will be used to launch the application is determined as follows: The configured main class, if any. Then the main class found in the test classes directory, if any. Then the main class found in the classes directory, if any.

3.4.2.1. Required parameters

Name	Type	Default
classesDirectory	**File**	**${project.build.outputDirectory}**
testClassesDirectory	**File**	**${project.build.testOutputDirectory}**

3.4.2.2. Optional parameters

Name	Type	Default
addResources	**boolean**	**false**
additionalClasspathElements	**String[]**	
arguments	**String[]**	
commandlineArguments	**String**	
environmentVariables	**Map**	
excludeGroupIds	**String**	
excludes	**List**	
includes	**List**	
jvmArguments	**String**	
mainClass	**String**	
noverify	**boolean**	
optimizedLaunch	**boolean**	**true**
profiles	**String[]**	
skip	**boolean**	**false**
systemPropertyVariables	**Map**	
workingDirectory	**File**	

3.4.2.3. Parameter details

3.4.2.3.1. addResources

Add maven resources to the classpath directly, this allows live in-place editing of resources. Duplicate resources are removed from **target/classes** to prevent them from appearing twice if **ClassLoader.getResources()** is called. Please consider adding **spring-boot-devtools** to your project instead as it provides this feature and many more.

Name	**addResources**

Type	**boolean**
Default value	**false**
User property	**spring-boot.run.addResources**
Since	**1.0.0**

3.4.2.3.2. additionalClasspathElements

Additional classpath elements that should be added to the classpath. An element can be a directory with classes and resources or a jar file.

Name	**additionalClasspathElements**
Type	**java.lang.String[]**
Default value	
User property	**spring-boot.run.additional-classpath-elements**
Since	**3.2.0**

3.4.2.3.3. agents

Path to agent jars.

Name	**agents**
Type	**java.io.File[]**
Default value	
User property	**spring-boot.run.agents**
Since	**2.2.0**

3.4.2.3.4. **arguments**

Arguments that should be passed to the application.

Name	**arguments**
Type	**java.lang.String[]**
Default value	
User property	
Since	**1.0.0**

3.4.2.3.5. **classesDirectory**

Directory containing the classes and resource files that should be used to run the application.

Name	**classesDirectory**
Type	**java.io.File**
Default value	**${project.build.outputDirectory}**
User property	
Since	**1.0.0**

3.4.2.3.6. **commandlineArguments**

Arguments from the command line that should be passed to the application. Use spaces to separate multiple arguments and make sure to wrap multiple values between quotes. When specified, takes precedence over **#arguments**.

Name	**commandlineArguments**
Type	**java.lang.String**
Default value	

User property	**spring-boot.run.arguments**
Since	**2.2.3**

3.4.2.3.7. environmentVariables

List of Environment variables that should be associated with the forked process used to run the application.

Name	**environmentVariables**
Type	**java.util.Map**
Default value	
User property	
Since	**2.1.0**

3.4.2.3.8. excludeGroupIds

Comma separated list of groupId names to exclude (exact match).

Name	**excludeGroupIds**
Type	**java.lang.String**
Default value	
User property	**spring-boot.excludeGroupIds**
Since	**1.1.0**

3.4.2.3.9. excludes

Collection of artifact definitions to exclude. The **Exclude** element defines mandatory **groupId** and **artifactId** components and an optional **classifier** component. When configured as a property, values should be comma-separated with colon-separated components:

groupId:artifactId,groupId:artifactId:classifier

Name	**excludes**
Type	**java.util.List**
Default value	
User property	**spring-boot.excludes**
Since	**1.1.0**

3.4.2.3.10. includes

Collection of artifact definitions to include. The **Include** element defines mandatory **groupId** and **artifactId** components and an optional **classifier** component. When configured as a property, values should be comma-separated with colon-separated components:
groupId:artifactId,groupId:artifactId:classifier

Name	**includes**
Type	**java.util.List**
Default value	
User property	**spring-boot.includes**
Since	**1.2.0**

3.4.2.3.11. jvmArguments

JVM arguments that should be associated with the forked process used to run the application. On command line, make sure to wrap multiple values between quotes.

Name	**jvmArguments**
Type	**java.lang.String**
Default value	

User property	**spring-boot.run.jvmArguments**
Since	**1.1.0**

3.4.2.3.12. mainClass

The name of the main class. If not specified the first compiled class found that contains a 'main' method will be used.

Name	**mainClass**
Type	**java.lang.String**
Default value	
User property	**spring-boot.run.main-class**
Since	**1.0.0**

3.4.2.3.13. noverify

Flag to say that the agent requires -noverify.

Name	**noverify**
Type	**boolean**
Default value	
User property	**spring-boot.run.noverify**
Since	**1.0.0**

3.4.2.3.14. optimizedLaunch

Whether the JVM's launch should be optimized.

Name	**optimizedLaunch**
Type	**boolean**
Default value	**true**
User property	**spring-boot.test-run.optimizedLaunch**
Since	

3.4.2.3.15. profiles

The spring profiles to activate. Convenience shortcut of specifying the 'spring.profiles.active' argument. On command line use commas to separate multiple profiles.

Name	**profiles**
Type	**java.lang.String[]**
Default value	
User property	**spring-boot.run.profiles**
Since	**1.3.0**

3.4.2.3.16. skip

Skip the execution.

Name	**skip**
Type	**boolean**
Default value	**false**
User property	**spring-boot.run.skip**
Since	**1.3.2**

3.4.2.3.17. systemPropertyVariables

List of JVM system properties to pass to the process.

Name	**systemPropertyVariables**
Type	**java.util.Map**
Default value	
User property	
Since	**2.1.0**

3.4.2.3.18. testClassesDirectory

Directory containing the test classes and resource files that should be used to run the application.

Name	**testClassesDirectory**
Type	**java.io.File**
Default value	**${project.build.testOutputDirectory}**
User property	
Since	

3.4.2.3.19. workingDirectory

Current working directory to use for the application. If not specified, basedir will be used.

Name	**workingDirectory**
Type	**java.io.File**
Default value	

User property	**spring-boot.run.workingDirectory**
Since	**1.5.0**

3.4.3. Examples

3.4.3.1. Debug the Application

The **run** and **test-run** goals run your application in a forked process. If you need to debug it, you should add the necessary JVM arguments to enable remote debugging. The following configuration suspend the process until a debugger has joined on port 5005:

```
<project>
 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
     <jvmArguments>
      -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=*:5005
     </jvmArguments>
    </configuration>
   </plugin>
  </plugins>
 </build>
</project>
```

These arguments can be specified on the command line as well:

```
$ mvn spring-boot:run -Dspring-boot.run.jvmArguments=-
agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=*:5005
```

3.4.3.2. Using System Properties

System properties can be specified using the **systemPropertyVariables** attribute. The following example sets **property1** to **test** and **property2** to 42:

```
<project>
 <build>
  <properties>
   <my.value>42</my.value>
  </properties>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
     <systemPropertyVariables>
      <property1>test</property1>
```

```
      <property2>${my.value}</property2>
     </systemPropertyVariables>
    </configuration>
   </plugin>
  </plugins>
 </build>
</project>
```

If the value is empty or not defined (that is **<my-property**/>), the system property is set with an empty String as the value. Maven trims values specified in the pom, so it is not possible to specify a System property which needs to start or end with a space through this mechanism: consider using **jvmArguments** instead.

Any String typed Maven variable can be passed as system properties. Any attempt to pass any other Maven variable type (for example a **List** or a **URL** variable) will cause the variable expression to be passed literally (unevaluated).

The **jvmArguments** parameter takes precedence over system properties defined with the mechanism above. In the following example, the value for **property1** is **overridden**:

```
$ mvn spring-boot:run -Dspring-boot.run.jvmArguments="-Dproperty1=overridden"
```

3.4.3.3. Using Environment Variables

Environment variables can be specified using the **environmentVariables** attribute. The following example sets the 'ENV1', 'ENV2', 'ENV3', 'ENV4' env variables:

```
<project>
 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
     <environmentVariables>
      <ENV1>5000</ENV1>
      <ENV2>Some Text</ENV2>
      <ENV3/>
      <ENV4></ENV4>
     </environmentVariables>
    </configuration>
   </plugin>
  </plugins>
 </build>
</project>
```

If the value is empty or not defined (that is **<MY_ENV**/>), the env variable is set with an empty String as the value. Maven trims values specified in the pom so it is not possible to specify an env variable which needs to start or end with a space.

Any String typed Maven variable can be passed as system properties. Any attempt to pass any other Maven variable type (for example a **List** or a **URL** variable) will cause the variable expression to be passed literally (unevaluated).

Environment variables defined this way take precedence over existing values.

3.4.3.4. Using Application Arguments

Application arguments can be specified using the **arguments** attribute. The following example sets two arguments: **property1** and **property2=42**:

```
<project>
 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
     <arguments>
      <argument>property1</argument>
      <argument>property2=${my.value}</argument>
     </arguments>
    </configuration>
   </plugin>
  </plugins>
 </build>
</project>
```

On the command-line, arguments are separated by a space the same way **jvmArguments** are. If an argument contains a space, make sure to quote it. In the following example, two arguments are available: **property1** and **property2=Hello World**:

```
$ mvn spring-boot:run -Dspring-boot.run.arguments="property1 'property2=Hello World'"
```

3.4.3.5. Specify Active Profiles

The active profiles to use for a particular application can be specified using the **profiles** argument.

The following configuration enables the **local** and **dev** profiles:

```
<project>
 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
     <profiles>
      <profile>local</profile>
      <profile>dev</profile>
     </profiles>
    </configuration>
   </plugin>
  </plugins>
 </build>
</project>
```

The profiles to enable can be specified on the command line as well, make sure to separate them with a comma, as shown in the following example:

```
$ mvn spring-boot:run -Dspring-boot.run.profiles=local,dev
```

3.5. ADDITIONAL RESOURCES

For more information, see the following documentation:

- Spring Boot Maven Plugin
- Spring Boot Maven Plugin API.

CHAPTER 4. DEFINING REST SERVICES

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.

4.1. OVERVIEW OF REST IN CAMEL

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.

4.1.1. 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.

4.1.2. 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.

4.1.3. 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}");
```

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.
If you do not explicitly configure an HTTP transport component then the REST DSL automatically discovers which HTTP component to use by checking for available components on the classpath. The REST DSL looks for the default names of any HTTP components and uses the first one it finds. If there are no HTTP components on the classpath and you did not explicitly configure an HTTP transport then the default HTTP component is **camel-http**.

The following Java code shows how to define a simple **Hello World** service using the **camel-rest** component:

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

4.1.4. REST implementations

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

REST starter (Required)

The REST starter (camel-rest-starter) allows defining REST endpoints (consumer) using the Rest DSL and plugin to other Camel components as the REST transport.
The REST component can also be used as a client (producer) to call REST services.

URI format:

```
rest://method:path[:uriTemplate]?[options]
```

Platform HTTP component (Recommended)

The Platform HTTP component (in **camel-platform-http-starter**) is used to allow Camel to use the existing HTTP server from the runtime when running on Camel on Spring Boot, Quarkus, or other runtimes.
For example, the following Java code shows how to define a simple **Hello World** service:

```
from("platform-http:/hello").setBody(simple("Hello ${header.name}"));
```

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.

4.1.4.1. 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.

4.2. DEFINING SERVICES WITH REST DSL

4.2.1. 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).

4.2.2. 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.
- OpenAPI integration (through the **camel-openapi-java** component).

4.2.3. 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. We suggest you use the following Camel components with REST DSL:

- Servlet component (**camel-servlet**).
- Platform HTTP component (**camel-platform-http**).

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.

4.2.4. 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("platform-http").port(9091);
```

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

```
<restConfiguration component="platform-http" port="9091"/>
```

4.2.5. 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()**, **patch()** 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()
  ```

4.2.6. 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. A simple REST service can be defined as follows, where we use rest() to define the services as shown below:

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

```
    rest("/say")
        .get("/hello").to("direct:hello")
        .get("/bye").consumes("application/json").to("direct:bye")
        .post("/bye").to("mock:update");

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

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

The preceding example features three different kinds of builder:

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.

4.2.7. 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. The example above can be defined in XML as shown below:

```
<camel>


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

```
</camel>
```

4.2.8. 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 verb clauses. For example, given the following snippet of Java DSL:

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

Or given the following snippet of XML DSL:

```
<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");
```

4.2.9. 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>
```

4.2.10. 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**.

4.2.11. Embedded route syntax

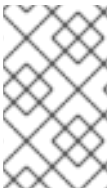
Instead of terminating a verb clause with the **to()** keyword (Java DSL) or the **to** element (XML DSL), you have the option of embedding an Apache Camel route directly into the REST DSL, using the **route()** keyword (Java DSL) or the **route** element (XML DSL). The **route()** keyword enables you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



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.

4.2.12. REST DSL and HTTP transport component

If you do not explicitly configure an HTTP transport component then the REST DSL automatically discovers which HTTP component to use by checking for available components on the classpath. The

REST DSL looks for the default names of any HTTP components and uses the first one it finds. If there are no HTTP components on the classpath and you did not explicitly configure an HTTP transport then the default HTTP component is **camel-http**.

4.2.13. 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:

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

4.2.14. 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()**, **patch()**. 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:

```
<camel>
    ...
    <rest path="/say">
        <get path="/hello" method="TRACE">
            <to uri="direct:hello"/>
        </get>
        ...
    </rest>
...
</camel>
```

4.2.15. 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.

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

```
    // Configure the REST DSL, with JSON binding mode
    restConfiguration().component("platform-
http").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:

```
public class UserErrorService {
    public void idTooLowError(Exchange exchange) {
```

```
        exchange.getMessage().setBody("id value is too low");
        exchange.getMessage().setHeader(Exchange.CONTENT_TYPE, "text/plain");
        exchange.getMessage().setHeader(Exchange.HTTP_RESPONSE_CODE, 400);
    }
}
```

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

4.2.16. 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");
```

4.2.17. 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:

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

4.2.18. 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()**):

```
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 subordinate verb clauses. Or the options can be applied to each individual verb clause:

```
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.

The REST DSL Options table 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.

Java DSL	XML DSL	Description
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.
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).

The folllwing table shows an overview of options supported by the REST DSL.

Table 4.2. 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**.

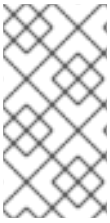
Java DSL	XML DSL	Description
consumes()	**@consumes**	Restricts the verb clause to accept only the specified Internet media type (MIME type) in an HTTP Request. Typical values are: **text/plain**, **text/http**, **text/xml**, **application/json**, **application/xml**.
customId()	**@customId**	Defines 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 an **HTTP Response** message.
produces()	**produces**	Restricts the verb clause to produce only the specified Internet media type (MIME type) in an HTTP Response. Typical values are: **text/plain**, **text/http**, **text/xml**, **application/json**, **application/xml**.

Java DSL	XML DSL	Description
type()	**@type**	When object binding is enabled (that is, when **bindingMode** option is enabled), this option specifies the Java type that represents an **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. REST DSL

Apache Camel offers a REST styled DSL.

The intention is to allow end users to define REST services (hosted by Camel) using a REST style with verbs such as get, post, delete, etc.



NOTE

From **Camel 4.6** onwards, the REST DSL has been improved with a *contract-first* approach using vanilla OpenAPI specification files. This is documented in the REST DSL with OpenAPI contract first page. This current page documents the *code-first* REST DSL that Camel provides for a long time.

4.3.1. How it works

The REST DSL is a facade that builds Rest endpoints as consumers for Camel routes. The actual REST transport is leveraged by using Camel REST components such as Netty HTTP, Servlet, and others that have native REST integration.

4.3.2. Components supporting REST DSL

We suggest the following Camel components for REST DSL:

- camel-rest **required** contains the base rest component needed by REST DSL
- camel-platform-http (recommended)
- camel-servlet

Also available is:

- camel-netty-http

4.3.3. REST DSL with Java DSL

To use the REST DSL in Java DSL, then just do as with regular Camel routes by extending the **RouteBuilder** and define the routes in the **configure** method.

A simple REST service can be defined as follows, where we use **rest()** to define the services as shown below:

```
@Override
public void configure() throws Exception {
    rest("/say")
        .get("/hello").to("direct:hello")
        .get("/bye").consumes("application/json").to("direct:bye")
        .post("/bye").to("mock:update");

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

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

This defines a REST service with the following url mappings:

Base Path	Uri template	Verb	Consumes
/say	***/hello***	**get**	*all*
/say	***/bye***	**get**	**application/json**
/say	***/bye***	**post**	*all*

Notice that in the REST service we route directly to a Camel endpoint using **to()**. This is because the REST DSL has a shorthand for routing directly to an endpoint using **to()**.

4.3.4. REST DSL with XML DSL

The example above can be defined in XML as shown below:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <rest path="/say">
    <get path="/hello">
      <to uri="direct:hello"/>
    </get>
    <get path="/bye" consumes="application/json">
      <to uri="direct:bye"/>
    </get>
    <post path="/bye">
      <to uri="mock:update"/>
    </post>
  </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>
```

4.3.5. Using a base path

The REST DSL allows defining a base path to help applying the *"don't repeat yourself"* (DRY) practice. For example, to define a customer path, we can set the base path in **rest("/customer")** and then provide the uri templates in the verbs, as shown below:

```
rest("/customers/")
    .get("/{id}").to("direct:customerDetail")
    .get("/{id}/orders").to("direct:customerOrders")
    .post("/neworder").to("direct:customerNewOrder");
```

And using XML DSL, it becomes:

```
<rest path="/customers/">
  <get path="/{id}">
    <to uri="direct:customerDetail"/>
  </get>
  <get path="/{id}/orders">
    <to uri="direct:customerOrders"/>
  </get>
  <post path="/neworder">
    <to uri="direct:customerNewOrder"/>
  </post>
</rest>
```

TIP

The REST DSL will take care of duplicate path separators when using base path and uri templates. In the example above the rest base path ends with a slash / and the verb starts with a slash /. Camel will take care of this and remove the duplicated slash.

It is not required to use both base path and uri templates. You can omit the base path and define the base path and uri template in the verbs only. The example above can be defined as:

```
<rest>
  <get path="/customers/{id}">
    <to uri="direct:customerDetail"/>
  </get>
  <get path="/customers/{id}/orders">
    <to uri="direct:customerOrders"/>
```

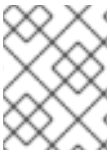
```
  </get>
  <post path="/customers/neworder">
    <to uri="direct:customerNewOrder"/>
  </post>
</rest>
```

You can combine path parameters to build complex expressions. For example:

```
rest("items/")
    .get("{id}/{filename}.{content-type}")
    .to("direct:item")
```

4.3.6. Managing Rest services

Each of the rest services becomes a Camel route, so in the first example, we have 2 x get and 1 x post REST service, which each becomes a Camel route. This makes it *the same* from Apache Camel to manage and run these services, as they are just Camel routes. This means any tooling and API today that deals with Camel routes, also work with the REST services.



NOTE

To use JMX with Camel then **camel-management** JAR must be included in the classpath.

This means you can use JMX to stop/start routes, and also get the JMX metrics about the routes, such as the number of messages processed, and their performance statistics.

There is also a Rest Registry JMX MBean that contains a registry of all REST services that has been defined.

4.3.7. Inline REST DSL as a single route

Since Camel 4.5 inline-routes are enabled by default.

Each of the rest services becomes a Camel route, and this means, that if the rest service is calling another Camel route via **direct**, which is a widespread practice. This means that each rest service then becomes two routes. This can become harder to manage if you have many rest services.

When you use **direct** endpoints then you can enable REST DSL to automatically *inline* the direct route in the rest route, meaning that there is only one route per rest service.

WARNING

When using inline-routes, then each REST endpoint should link 1:1 to a unique **direct** endpoint. The linked *direct* routes are inlined and therefore does not **exists** as independent routes, and they cannot be called from other regular Camel routes. In other words the inlined routes are essentially moved inside the rest-dsl and does not exist as a route.

To do this you **MUST** use **direct** endpoints, and each endpoint must be unique name per service. And the option **inlineRoutes** must be enabled.

For example, in the Java DSL below we have enabled inline routes and each rest service uses **direct** endpoints with unique names.

```
restConfiguration().inlineRoutes(true);

rest("/customers/")
    .get("/{id}").to("direct:customerDetail")
    .get("/{id}/orders").to("direct:customerOrders")
    .post("/neworder").to("direct:customerNewOrder");
```

And in XML:

```
<restConfiguration inlineRoutes="true"/>

<rest>
  <get path="/customers/{id}">
    <to uri="direct:customerDetail"/>
  </get>
  <get path="/customers/{id}/orders">
    <to uri="direct:customerOrders"/>
  </get>
  <post path="/customers/neworder">
    <to uri="direct:customerNewOrder"/>
  </post>
</rest>
```

If you use Camel Main, Camel Spring Boot, Camel Quarkus or Camel JBang, you can also enable this in **application.properties** such as:

```
camel.rest.inline-routes = true
```

The REST services above each use a unique 1:1 linked direct endpoint (direct:customerDetail, direct:customerOrders direct:customerNewOrder). This means that you cannot call these routes from another route such as the following would not function:

```
from("kafka:new-order")
    .to("direct:customerNewOrder");
```

So if you desire to call common routes from both REST DSL and other regular Camel routes then keep these in separate routes as shown:

```
restConfiguration().inlineRoutes(true);

rest("/customers/")
    .get("/{id}").to("direct:customerDetail")
    .get("/{id}/orders").to("direct:customerOrders")
    .post("/neworder").to("direct:customerNewOrder");

from("direct:customerNewOrder")
  // do some stuff here
  .to("direct:commonCustomerNewOrder"); // call common route
```

```
from("direct:commonCustomerNewOrder")
  // do stuff here
  .log("Created new order");

from("kafka:new-order")
   .to("direct:commonCustomerNewOrder"); // make sure to call the common route
```

The common shared route is separated into the route **direct:commonCustomerNewOrder**. This route can be called from both REST DSL and regular Camel routes.

4.3.8. Disabling REST services

While developing REST services using REST DSL, you may want to temporary disabled some REST endpoints, which you can do using **disabled** as shown in the following.

```
rest("/customers/")
    .get("/{id}").to("direct:customerDetail")
    .get("/{id}/orders").to("direct:customerOrders").disabled("{{ordersEnabled}}")
    .post("/neworder").to("direct:customerNewOrder").disabled();
```

And in XML:

```
<rest>
  <get path="/customers/{id}">
    <to uri="direct:customerDetail"/>
  </get>
  <get path="/customers/{id}/orders" disabled="{{ordersEnabled}}">
    <to uri="direct:customerOrders"/>
  </get>
  <post path="/customers/neworder" disabled="true">
    <to uri="direct:customerNewOrder"/>
  </post>
</rest>
```

In this example the last two REST endpoints are configured with **disabled**. You can use Property Placeholder to let an external configuration determine if the REST endpoint is disabled or not. In this example the **/customers/{id}/orders** endpoint is disabled via a placeholder. The last REST endpoint is hardcoded to be disabled.

4.3.9. Binding to POJOs using

The REST DSL supports automatic binding json/xml contents to/from POJOs using data formats. By default, the binding mode is off, meaning there is no automatic binding happening for incoming and outgoing messages.

You may want to use binding if you develop POJOs that maps to your REST services request and response types. This allows you as a developer to work with the POJOs in Java code.

The binding modes are:

Binding Mode	Description
off	Binding is turned off. This is the default option.
auto	Binding is enabled, and Camel is relaxed and supports JSON, XML or both if the necessary data formats are included in the classpath. Notice that if for example **camel-jaxb** is not on the classpath, then XML binding is not enabled.
json	Binding to/from JSON is enabled, and requires a JSON capable data format on the classpath. By default, Camel will use **jackson** as the data format.
xm	Binding to/from XML is enabled, and requires **camel-jaxb** on the classpath.
json_xml	Binding to/from JSON and XML is enabled and requires both data formats to be on the classpath.

When using camel-jaxb for XML bindings, then you can use the option **mustBeJAXBElement** to relax the output message body must be a class with JAXB annotations. You can use this in situations where the message body is already in XML format, and you want to use the message body as-is as the output type. If that is the case, then set the dataFormatProperty option **mustBeJAXBElement** to **false** value.

The binding from POJO to JSon/JAXB will only happen if the **content-type** header includes the word **json** or **xml** representatively. This allows you to specify a custom content-type if the message body should not attempt to be marshalled using the binding. For example, if the message body is a custom binary payload, etc.

When automatic binding from POJO to JSON/JAXB takes place the existing **content-type** header will by default be replaced with either **application/json** or **application/xml**. To disable the default behavior and be able to produce JSON/JAXB responses with custom **content-type** headers (e.g. **application/user.v2+json**) you configure this in Java DSL as shown below:

```
restConfiguration().dataFormatProperty("contentTypeHeader", "false");
```

To use binding you must include the necessary data formats on the classpath, such as **camel-jaxb** and/or **camel-jackson**. And then enable the binding mode. You can configure the binding mode globally on the rest configuration, and then override per rest service as well.

To enable binding, you configure this in Java DSL as shown below:

```
restConfiguration().component("platform-
http").host("localhost").port(portNum).bindingMode(RestBindingMode.auto);
```

And in XML DSL:

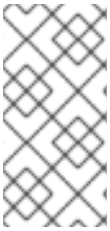
```
<restConfiguration bindingMode="auto" component="platform-http" port="8080"/>
```

When binding is enabled, Camel will bind the incoming and outgoing messages automatic, accordingly to the content type of the message. If the message is JSON, then JSON binding happens; and so if the message is XML, then XML binding happens. The binding happens for incoming and reply messages.

The table below summaries what binding occurs for incoming and reply messages.

Message Body	Direction	Binding Mode	Message Body
XML	Incoming	auto,xml,json_xml	POJO
POJO	Outgoing	auto,xml, json_xml	XML
JSON	Incoming	auto,json,json_xml	POJO
POJO	Outgoing	auto,json,json_xml	JSON

When using binding, you must also configure what POJO type to map to. This is mandatory for incoming messages, and optional for outgoing.



NOTE

When using binding mode **json**, **xml** or **json_xml** then Camel will automatically set **consumers** and **produces** on the rest endpoint (according to the mode), if not already explicit configured. For example, with binding mode **json** and setting the outType as **UserPojo** then Camel will define this rest endpoint as producing **application/json**.

For example, to map from xml/json to a pojo class **UserPojo** you do this in Java DSL as shown below:

```
// configure to use platformhttp on localhost with the given port
// and enable auto binding mode
restConfiguration().component("platform-
http").host("localhost").port(portNum).bindingMode(RestBindingMode.auto);

// use the rest DSL to define the rest services
rest("/users/")
    .post().type(UserPojo.class)
        .to("direct:newUser");
```

Notice we use **type** to define the incoming type. We can optionally define an outgoing type (which can be a good idea, to make it known from the DSL and also for tooling and JMX APIs to know both the incoming and outgoing types of the REST services). To define the outgoing type, we use **outType** as shown below:

```
// configure to use platformhttp on localhost with the given port
// and enable auto binding mode
restConfiguration().component("platform-
http").host("localhost").port(portNum).bindingMode(RestBindingMode.auto);

// use the rest DSL to define the rest services
rest("/users/")
    .post().type(UserPojo.class).outType(CountryPojo.class)
        .to("direct:newUser");
```

And in XML DSL:

```
<rest path="/users/">
  <post type="UserPojo" outType="CountryPojo">
    <to uri="direct:newUser"/>
  </post>
</rest>
```

To specify input and/or output using an array, append **[]** to the end of the canonical class name as shown in the following Java DSL:

```
// configure to use platformhttp on localhost with the given port
// and enable auto binding mode
restConfiguration().component("platform-
http").host("localhost").port(portNum).bindingMode(RestBindingMode.auto);

// use the rest DSL to define the rest services
rest("/users/")
    .post().type(UserPojo[].class).outType(CountryPojo[].class)
        .to("direct:newUser");
```

The **UserPojo** is just a plain pojo with getter/setter as shown:

```
public class UserPojo {
    private int id;
    private String 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 **UserPojo** only supports JSON, as XML requires using JAXB annotations, so we can add those annotations if we want to support XML also

```
@XmlRootElement(name = "user")
@XmlAccessorType(XmlAccessType.FIELD)
public class UserPojo {
    @XmlAttribute
    private int id;
    @XmlAttribute
    private String 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;
    }
}
```

By having the JAXB annotations, the POJO supports both JSON and XML bindings.

4.3.9.1. Camel Rest-DSL configurations

The REST DSL supports the following options:

Name	Description	Default	Type
apiComponent	Sets the name of the Camel component to use as the REST API (such as swagger or openapi)		String
apiContextPath	Sets a leading API context-path the REST API services will be using. This can be used when using components such as camel-servlet where the deployed web application is deployed using a context-path.		String
apiHost	To use a specific hostname for the API documentation (such as swagger or openapi) This can be used to override the generated host with this configured hostname		String
apiProperties	Sets additional options on api level		Map
apiVendorExtension	Whether a vendor extension is enabled in the Rest APIs. If enabled, then Camel will include additional information as a vendor extension (e.g., keys starting with ***x-***) such as route ids, class names etc. Not all third party API gateways and tools support vendor-extensions when importing your API docs.	false	boolean
bindingMode	Sets the binding mode to be used by the REST consumer	RestBindingMode.off	RestBindingMode

Name	Description	Default	Type
clientRequestValidation	Whether to enable validation of the client request to check: 1) Content-Type header matches what the REST DSL consumes; returns HTTP Status 415 if validation error. 2) Accept header matches what the REST DSL produces; returns HTTP Status 406 if validation error. 3) Missing required data (query parameters, HTTP headers, body); returns HTTP Status 400 if validation error. 4) Parsing error of the message body (JSON, XML or Auto binding mode must be enabled); returns HTTP Status 400 if validation error.	false	boolean
clientResponseValidation	Whether to check what Camel is returning as response to the client: 1) Status-code and Content-Type matches REST DSL response messages. 2) Check whether expected headers is included according to the REST DSL repose message headers. 3) If the response body is JSon then check whether its valid JSon. Returns 500 if validation error detected.	false	boolean
component	Sets the name of the Camel component to use as the REST consumer		String
componentProperties	Sets additional options on component level		Map
consumerProperties	Sets additional options on consumer level		Map
contextPath	Sets a leading context-path the REST services will be using. This can be used when using components such as camel-servlet where the deployed web application is deployed using a context-path. Or for components such that include an HTTP server.		String
corsHeaders	Sets the CORS headers to use if CORS has been enabled.		Map
dataFormatProperties	Sets additional options on data format level		Map
enableCORS	To specify whether to enable CORS, which means Camel will automatically include CORS in the HTTP headers in the response. This option is default false	false	boolean

Name	Description	Default	Type
enableNoContentResponse	To specify whether to return HTTP 204 with an empty body when a response contains an empty JSON object or XML root object.	false	boolean
endpointProperties	Sets additional options on endpoint level		Map
host	Sets the hostname to use by the REST consumer		String
hostNameResolver	Sets the resolver to use for resolving hostname	RestHostNameResolver.allLocalIp	RestHostNameResolver
inlineRoutes	Inline routes in rest-dsl which are linked using direct endpoints. By default, each service in REST DSL is an individual route, meaning that you would have at least two routes per service (rest-dsl, and the route linked from rest-dsl). Enabling this allows Camel to optimize and inline this as a single route. However, this requires using direct endpoints, which must be unique per service. This option is default false.	false	boolean
jsonDataFormat	Sets a custom JSON data format to be used Important: This option is only for setting a custom name of the data format, not to refer to an existing data format instance.		String
port	Sets the port to use by the REST consumer		int
producerApiDoc	Sets the location of the api document (swagger api) the REST producer will use to validate the REST uri and query parameters are valid accordingly to the api document. This requires adding camel-openapi-java to the classpath, and any miss configuration will let Camel fail on startup and report the error(s). The location of the api document is loaded from classpath by default, but you can use file: or http: to refer to resources to load from file or http url.		String
producerComponent	Sets the name of the Camel component to use as the REST producer		String
scheme	Sets the scheme to use by the REST consumer		String

Name	Description	Default	Type
skipBindingOnErrorCode	Whether to skip binding output if there is a custom HTTP error code, and instead use the response body as-is. This option is default true.	true	boolean
useXForwardHeaders	Whether to use X-Forward headers to set host etc. for Swagger. This option is default true.	true	boolean
xmlDataFormat	Sets a custom XML data format to be used. Important: This option is only for setting a custom name of the data format, not to refer to an existing data format instance.		String

For example, to configure to use the Platform HTTP component on port 9091, then we can do as follows:

```
restConfiguration().component("platform-http").port(9091).componentProperty("foo", "123");
```

And with XML DSL:

```
<restConfiguration component="platform-http" port="9091">
  <componentProperty key="foo" value="123"/>
</restConfiguration>
```

If no component has been explicitly configured, then Camel will look up if there is a Camel component that integrates with the REST DSL, or if a **org.apache.camel.spi.RestConsumerFactory** is registered in the registry. If either one is found, then that is being used.

You can configure properties on these levels.

- component - Is used to set any options on the Component class. You can also configure these directly on the component.
- endpoint - Is used set any option on the endpoint level. Many of the Camel components has many options you can set on endpoint level.
- consumer - Is used to set any option on the consumer level.
- data format - Is used to set any option on the data formats. For example, to enable pretty print in the JSON data format.
- cors headers - If cors is enabled, then custom CORS headers can be set. See below for the default values which are in used. If a custom header is set then that value takes precedence over the default value.

You can set multiple options of the same level, so you can, for example, configure two component options, and three endpoint options, etc.

4.3.10. Enabling or disabling Jackson JSON features

When using JSON binding, you may want to turn specific Jackson features on or off. For example, to disable failing on unknown properties (e.g., JSON input has a property which cannot be mapped to a POJO) then configure this using the **dataFormatProperty** as shown below:

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

You can disable more features by separating the values using comma, such as:

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

Likewise, you can enable features using the enableFeatures such as:

```
restConfiguration().component("platform-
http").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");
```

The values that can be used for enabling and disabling features on Jackson are the names of the enums from the following three Jackson classes

- **com.fasterxml.jackson.databind.SerializationFeature**
- **com.fasterxml.jackson.databind.DeserializationFeature**
- **com.fasterxml.jackson.databind.MapperFeature**

The rest configuration is, of course, also possible using XML DSL:

```
<restConfiguration component="platform-http" 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>
```

4.3.11. Default CORS headers

If CORS is enabled, then the *"follow headers"* is in use by default. You can configure custom CORS headers that take precedence over the default value.

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

Key	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

4.3.12. Defining a custom error message as-is

If you want to define custom error messages to be sent back to the client with a HTTP error code (e.g., such as 400, 404 etc.) then you set a header with the key **Exchange.HTTP_RESPONSE_CODE** to the error code (must be 300+) such as 404. And then the message body with any reply message, and optionally set the content-type header as well. There is a little example shown below:

```
restConfiguration().component("platform-
http").host("localhost").port(portNum).bindingMode(RestBindingMode.json);
// use the rest DSL to define the rest services
rest("/users/")
    .post("lives").type(UserPojo.class).outType(CountryPojo.class)
    .to("direct:users-lives");

from("direct:users-lives")
    .choice()
        .when().simple("${body.id} < 100")
            .bean(new UserErrorService(), "idToLowError")
        .otherwise()
            .bean(new UserService(), "livesWhere");
```

In this example, if the input id is a number that is below 100, we want to send back a custom error message, using the UserErrorService bean, which is implemented as shown:

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

In the ***UserErrorService*** bean, we build our custom error message, and set the HTTP error code to 400. This is important, as that tells rest-dsl that this is a custom error message, and the message should not use the output pojo binding (e.g., would otherwise bind to ***CountryPojo***).

4.3.12.1. Catching JsonParserException and returning a custom error message

You can return a custom message as-is (see previous section). So we can leverage this with Camel error handler to catch **JsonParserException**, handle that exception and build our custom response message. For example, to return a HTTP error code 400 with a hardcoded message, we can do as shown below:

```
onException(JsonParseException.class)
```

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

4.3.13. Query/Header Parameter default Values

You can specify default values for parameters in the rest-dsl, such as the verbose parameter below:

```
  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");
```

The default value is automatic set as header on the incoming Camel **Message**. So if the call to **/customers/id/orders** do not include a query parameter with key **verbose** then Camel will now include a header with key **verbose** and the value **false** because it was declared as the default value. This functionality is only applicable for query parameters. Request headers may also be defaulted in the same way.

```
  rest("/customers/")
      .get("/{id}").to("direct:customerDetail")
      .get("/{id}/orders")

.param().name("indicator").type(RestParamType.header).defaultValue("disabled").description("Featur
e Enabled Indicator").endParam()
          .to("direct:customerOrders")
      .post("/neworder").to("direct:customerNewOrder");
```

4.3.14. Client Request and Response Validation

It is possible to enable validation of the incoming client request. The validation checks for the following:

- Content-Type header matches what the REST DSL consumes. (Returns HTTP Status 415)
- Accept header matches what the REST DSL produces. (Returns HTTP Status 406)
- Missing required data (query parameters, HTTP headers, body). (Returns HTTP Status 400)
- Checking if query parameters or HTTP headers has not-allowed values. (Returns HTTP Status 400)
- Parsing error of the message body (JSON, XML or Auto binding mode must be enabled). (Returns HTTP Status 400)

If the validation fails, then REST DSL will return a response with an HTTP error code.

The validation is by default turned off (to be backwards compatible). It can be turned on via **clientRequestValidation** as shown below:

```
restConfiguration().component("platform-http").host("localhost")
    .clientRequestValidation(true);
```

The validator is pluggable and Camel provides a default implementation out of the box. However, the **camel-openapi-validator** uses the third party Atlassian Swagger Request Validator library instead for client request validator. This library is a more extensive validator than the default validator from **camel-core**, such as being able to validate the payload is structured according to the OpenAPI specification.

In **Camel 4.13** we added a *response validator* as well which is intended more as development assistance that you can enable while building your Camel integrations, and help ensure what Camel is sending back to the HTTP client is valid. The response validator checks for the following:

- Status-code and Content-Type matches REST DSL response messages.
- Check whether expected headers is included according to the REST DSL repose message headers.
- If the response body is JSon then check whether its valid JSon.

If any error is detected the HTTP Status 500 is returned.

Also, the **camel-openapi-validator** can be added to the classpath to have a more powerful response validator, that can be used to validate the response payload is structured according to the OpenAPI specification.

4.3.15. OpenAPI / Swagger API

The REST DSL supports OpenAPI and Swagger by the **camel-openapi-java** modules.

You can define each parameter fine-grained with details such as name, description, data type, parameter type and so on, using the **param**. For example, to define the id path parameter, you can do as shown below:

```
<!-- this is a rest GET to view an user by the given id -->
<get path="/{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" dataType="int"/>
  <to uri="bean:userService?method=getUser(${header.id})"/>
</get>
```

And 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").dataType("int").endParam()
    .to("bean:userService?method=getUser(${header.id})")
```

The body parameter type requires to use body as well for the name. For example, a REST PUT operation to create/update an user could be done as:

```
<!-- this is a rest PUT to create/update an user -->
<put type="org.apache.camel.example.rest.User">
  <description>Updates or create a user</description>
```

```
  <param name="body" type="body" description="The user to update or create"/>
  <to uri="bean:userService?method=updateUser"/>
</put>
```

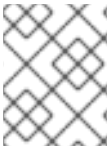
And in Java DSL:

```
.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")
```

4.3.15.1. Vendor Extensions

The generated API documentation can be configured to include vendor extensions (https://swagger.io/specification/#specificationExtensions) which document the operations and definitions with additional information, such as class name of model classes, camel context id and route id's. This information can be very helpful for developers, especially during troubleshooting. However, at production usage you may wish to not have this turned on to avoid leaking implementation details into your API docs.

The vendor extension information is stored in the API documentation with keys starting with **x-**.



NOTE

Not all third party API gateways and tools support vendor-extensions when importing your API docs.

The vendor extensions can be turned on **RestConfiguration** via the **apiVendorExtension** option:

```
restConfiguration()
    .component("servlet")
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("prettyPrint", "true")
    .apiContextPath("api-doc")
    .apiVendorExtension(true)
        .apiProperty("api.title", "User API").apiProperty("api.version", "1.0.0")
        .apiProperty("cors", "true");
```

And in XML DSL:

```
 <restConfiguration component="servlet" bindingMode="json"
                    apiContextPath="api-docs"
                    apiVendorExtension="true">

     <!-- we want json output in pretty mode -->
     <dataFormatProperty key="prettyPrint" value="true"/>

     <!-- setup swagger api descriptions -->
     <apiProperty key="api.version" value="1.0.0"/>
     <apiProperty key="api.title" value="User API"/>

 </restConfiguration>
```

4.3.15.2. Supported API properties

The following table lists supported API properties and explains their effect. To set them use **apiProperty(String, String)** in the Java DSL or **<apiProperty>** when defining the REST API via XML configuration. Properties in **bold** are required by the OpenAPI 2.0 specification. Most of the properties affect the OpenAPI Info object, License object or Contact object.

Property	Description
api.version	Version of the API
api.title	Title of the API
api.description	Description of the API
api.termsOfService	API Terms of Service of the API
api.license.name	License information of the API
api.license.url	URL for the License of the API
api.contact.name	The identifying name of the contact person/organization
api.contact.url	The URL pointing to the contact information
api.contact.email	The email address of the contact person/organization
api.specification.contentType.json	The Content-Type of the served OpenAPI JSON specification, **application/json** by default
api.specification.contentType.yaml	The Content-Type of the served OpenAPI YAML specification, **text/yaml** by default
externalDocs.url	The URI for the target documentation. This must be in the form of a URI
externalDocs.description	A description of the target documentation

4.4. MARSHALLING TO AND FROM JAVA OBJECTS

4.4.1. 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

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.

4.4.2. 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.

4.4.2.1. 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**)

4.4.3. 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.
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:

```
<project ...>
  ...
  <dependencies>
    ...
    *<!-- use for json binding -->
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-jackson</artifactId>
    </dependency>*
    ...
  </dependencies>
```

```
</project>
```

4.4.4. 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

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 4.3. 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**:

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:

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

4.4.5. 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 below.

User Class for JSON Response

```
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"
}
```

4.4.6. REST DSL route with JSON binding

The REST DSL configuration and the REST service definition for this example are shown below.

REST DSL Route with JSON Binding

```
<camel>

  <bean name="userService" type="org.apache.camel.example.rest.UserService">
  </bean>

  <rest description="User rest service" path="/user" produces="application/json"
consumes="application/json">
    <get description="Find user by id" outType="org.apache.camel.example.rest.User" path="/{id}">
      <param name="id"/>
      <to uri="bean:userService?method=getUser(${header.id})"/>
    </get>
    <put description="Updates or create a user" type="org.apache.camel.example.rest.User">
      <param dataType="org.apache.camel.example.rest.User" name="body" type="body"/>
      <to uri="bean:userService?method=updateUser"/>
    </put>
    <get description="Find all users" outType="org.apache.camel.example.rest.User[]"
path="/findAll">
      <to uri="bean:userService?method=listUsers"/>
    </get>
  </rest>


</camel>
```

4.4.7. REST operations

The REST service from the above example 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.

4.4.8. URLs to invoke the REST service

By inspecting the REST DSL definitions, 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/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
```

CHAPTER 5. OBSERVABILITY AND METRICS IN RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT

This section describes observability and metrics collection in Red Hat build of Apache Camel for Spring Boot applications, including

- Micrometer integration
- OpenTelemetry support
- JMX monitoring

5.1. OBSERVABILITY AND METRICS QUICK START

For the fastest setup of observability and metrics, use the **camel-observability-services-starter**:

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-observability-services-starter</artifactId>
</dependency>
```

That's it! No configuration needed. Your observability endpoints are available at:

- Health checks: **http://localhost:9876/observe/health**
- Metrics: **http://localhost:9876/observe/metrics**
- OpenTelemetry tracing: Automatically enabled

5.2. MICROMETER METRICS WITH SPRING BOOT ACTUATOR

Red Hat build of Apache Camel for Spring Boot provides extensive metrics integration through the **camel-micrometer-starter** component. When combined with Spring Boot's Actuator, it automatically exposes a rich set of metrics.

5.2.1. Dependencies

You have two options for setting up observability in your Camel Spring Boot application:

- Using the Observability Services Starter
- Individual Components

5.2.1.1. Option 1: All-in-One Observability Services Starter (Recommended)

For the simplest setup, use the **camel-observability-services-starter** which includes everything you need:

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-observability-services-starter</artifactId>
</dependency>
```

This starter automatically includes:

- **camel-micrometer-starter** - Micrometer metrics integration
- **micrometer-registry-prometheus** - Prometheus metrics registry (pre-configured)
- **camel-opentelemetry2-starter** - OpenTelemetry tracing support
- **camel-management-starter** - JMX management capabilities
- **spring-boot-starter-actuator** - Spring Boot actuator endpoints

5.2.1.1.1. Configuration using camel-observability-services-starter (Zero Configuration)

When using the **camel-observability-services-starter**, **no additional configuration is required**. The starter automatically configures:

- Management endpoints on port **9876** (separate from your application port **management.server.port=9876** property)
- Observability endpoints under /**observe** path:
- /**observe**/**health** - startup probe endpoint
- /**observe**/**health**/**live** - liveness probe endpoint
- /**observe**/**health**/**ready** - readiness probe endpoint
- /**observe**/**metrics** - Prometheus metrics endpoint
- OpenTelemetry tracing enabled
- Full health check details
- Pre-configured Prometheus registry bean

Access your metrics at:

- **http://localhost:9876/observe/metrics**

Known limitations

The Spring Boot Actuator endpoint configuration is overridden and no further customization is allowed. Only the above endpoints are available.

JKube/OpenShift maven pluging on openshift

If you want to deploy the application on OpenShift using JKube/OpenShift maven plugin, you need to specify the probe endpoints in the plugin configuration:

```
<configuration>
  <resources>
    <controller>
      <liveness>
        <getUrl>http://:9876/observe/health/live</getUrl>
      </liveness>
      <readiness>
        <getUrl>http://:9876/observe/health/ready</getUrl>
```

```
            </readiness>
        </controller>
    </resources>
</configuration>
```

5.2.1.2. Option 2: Individual components

If you prefer granular control, add individual dependencies:

```
<dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-micrometer-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

5.2.1.2.1. Manual Configuration (Individual Components)

If you use individual components, configure your **application.properties**:

```
# Enable actuator endpoints
management.endpoints.web.exposure.include=health,metrics,prometheus
management.endpoint.health.show-details=always

# Enable Camel metrics
camel.metrics.enable-exchange-event-notifier=true
camel.metrics.enable-message-history=true
camel.metrics.enable-route-event-notifier=true
camel.metrics.enable-route-policy=true

# Camel health checks
camel.health.routesEnabled=true
camel.health.consumersEnabled=true
camel.health.registryEnabled=true
camel.health.components-enabled=true
camel.health.exposure-level=full

# Spring Boot health indicators
management.health.livenessState.enabled=true
management.health.readinessState.enabled=true
```

5.2.2. Default Camel Metrics

Camel automatically provides these metrics out of the box:

Metric Name	Type	Description
camel.message.history	Timer	Performance of each node when message history is enabled
camel.routes.added	Gauge	Number of routes in total
camel.routes.running	Gauge	Number of routes currently running
camel.exchanges.inflight	Gauge	Route inflight messages
camel.exchanges.total	Counter	Total number of processed exchanges
camel.exchanges.succeeded	Counter	Number of successfully completed exchanges
camel.exchanges.failed	Counter	Number of failed exchanges
camel.exchanges.failures.handled	Counter	Number of failures handled
camel.route.policy	Gauge + Summary	Route performance metrics

5.2.3. Adding Custom Metrics with Micrometer Component in Routes

You can add custom metrics directly in your Camel routes:

```
@Component
public class MetricsRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        // Counter example
        from("timer:counter?period=5000")
            .setHeader("randomValue", simple("${random(1,100)}"))
            .to("micrometer:counter:custom.requests.total?increment=1&tags=source=timer")
            .log("Counter incremented");

        // Timer example
        from("direct:timedOperation")
```

```
            .to("micrometer:timer:custom.operation.timer?action=start")
            .delay(simple("${random(100,500)}"))
            .to("micrometer:timer:custom.operation.timer?action=stop")
            .log("Operation completed: ${body}");

        // Summary/Histogram example
        from("direct:histogramData")
            .to("micrometer:summary:custom.data.histogram?value=${header.dataSize}")
            .log("Data size recorded: ${header.dataSize}");
    }
}
```

5.2.4. Using Actuator to Visualize Metrics

Metrics

Access metrics at **http://localhost:8080/actuator/**:

```
{
  "names": [
    "camel.exchanges.total",
    "camel.exchanges.succeeded",
    "camel.exchanges.failed",
    "camel.routes.running",
    "custom.requests.total",
    "jvm.memory.used",
    "system.cpu.usage"
  ]
}
```

Details

Get specific metric details at **http://localhost:8080/actuator/metrics/camel.exchanges.total**:

```
{
  "name": "camel.exchanges.total",
  "description": "Total number of exchanges",
  "baseUnit": null,
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 1247.0
    }
  ],
  "availableTags": [
    {
      "tag": "camelContext",
      "values": ["camel-1"]
    }
  ]
}
```

5.3. OPENTELEMETRY INTEGRATION

5.3.1. Models

5.3.1.1. Pull Model (Micrometer only)

- **How it works**: Prometheus scrapes metrics from your application's **/actuator/prometheus** endpoint
- **Pros**: Simple setup, direct metric exposure
- **Cons**: Requires network access from Prometheus to application

5.3.1.2. Push Model (Micrometer + OpenTelemetry)

- **How it works**: OpenTelemetry agent pushes metrics to a collector, which then forwards to monitoring systems
- **Pros**: Better for dynamic environments, firewall-friendly, centralized collection
- **Cons**: Additional infrastructure complexity

5.3.2. Configuration and Setup

5.3.2.1. Dependencies

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-opentelemetry2-starter</artifactId>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-otlp</artifactId>
</dependency>
```

5.3.2.2. Application Configuration

```
# Enable OpenTelemetry
camel.opentelemetry2.enabled=true
camel.opentelemetry2.traceProcessors=true

# OpenTelemetry configuration (via system properties or environment variables)
otel.service.name=camel-spring-boot-app
otel.traces.exporter=otlp
otel.metrics.exporter=otlp
otel.exporter.otlp.endpoint=http://localhost:4318
```

5.3.2.3. Java Agent Setup

Download the OpenTelemetry Java agent and run your application:

```
java -javaagent:opentelemetry-javaagent.jar \
     -Dotel.service.name=camel-spring-boot-app \
     -Dotel.traces.exporter=otlp \
     -Dotel.metrics.exporter=otlp \
     -Dotel.exporter.otlp.endpoint=http://localhost:4318 \
     -jar your-application.jar
```

5.3.2.4. OpenTelemetry Collector Configuration

Example **otel-collector.yaml**:

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch:

exporters:
  prometheus:
    endpoint: "0.0.0.0:8889"
  jaeger:
    endpoint: jaeger:14250
    tls:
      insecure: true

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [jaeger]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [prometheus]
```

5.3.2.5. Example Route with Tracing

```
@Component
public class TracedRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:trace-demo?period=10000")
            .routeId("traced-route")
            .log("Starting traced operation")
            .to("direct:step1")
            .to("direct:step2")
            .log("Traced operation completed");
```

```
        from("direct:step1")
            .routeId("step1")
            .delay(simple("${random(100,300)}"))
            .setHeader("step1-completed", constant(true))
            .log("Step 1 completed");

        from("direct:step2")
            .routeId("step2")
            .delay(simple("${random(50,150)}"))
            .setHeader("step2-completed", constant(true))
            .log("Step 2 completed");
    }
}
```

5.4. JMX MONITORING

5.4.1. JConsole and Java Mission Control

5.4.1.1. Enabling JMX

Configure JMX in your **application.properties**:

```
# Enable Camel JMX
camel.springboot.jmx-enabled=true

# Spring Boot JMX configuration
spring.jmx.enabled=true
management.endpoints.jmx.exposure.include=*
```

5.4.1.2. JConsole Usage

1. Start your Camel Spring Boot application
2. Launch JConsole: **jconsole**
3. Connect to your local process
4. Navigate to MBeans tab
5. Explore Camel MBeans under **org.apache.camel**

Key Camel MBeans to monitor:

- **org.apache.camel:context=camel-1,type=context,name="camel-1"** - CamelContext information
- **org.apache.camel:context=camel-1,type=routes,name="route1"** - Individual route statistics
- **org.apache.camel:context=camel-1,type=endpoints,name="timer://foo"** - Endpoint statistics

5.4.1.3. Java Mission Control

1. Start JMC: **jmc**
2. Connect to your running JVM
3. Use Flight Recorder for detailed performance analysis
4. Monitor Camel-specific metrics in the MBean browser

5.4.2. Jolokia Integration

Jolokia provides HTTP/JSON access to JMX, making it cloud and web-friendly.

5.4.2.1. Dependencies

Use the Camel Jolokia starter for Spring Boot integration:

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-jolokia-starter</artifactId>
</dependency>
```

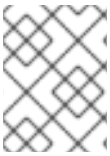
NOTE

If you're using the **camel-observability-services-starter**, Jolokia is not included by default. You'll need to add the **camel-jolokia-starter** dependency separately if you want HTTP/JSON access to JMX.

5.4.2.2. Configuration

When using **camel-jolokia-starter**, the Jolokia endpoint is automatically configured and exposed. For additional control, you can add to **application.properties**:

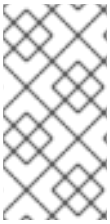
```
# Jolokia endpoint is automatically exposed when using camel-jolokia-starter
# Optionally configure additional endpoints
management.endpoints.web.exposure.include=health,metrics,prometheus,jolokia

# Optional Jolokia-specific configuration
management.endpoint.jolokia.enabled=true
```



NOTE

The **camel-jolokia-starter** automatically handles the Jolokia configuration, so minimal setup is required compared to using **jolokia-server-core** directly.

More information about the starter configuration are available [here] (https://github.com/apache/camel-spring-boot/blob/main/components-starter/camel-jolokia-starter/src/main/docs/jolokia.adoc)



NOTE

If you are using both **camel-observability-services-starter** and **camel-jolokia-starter** the actuator endpoint is not available, you can connect to the jolokia server directly using the exposed endpoint at http://0.0.0.0:8778/jolokia as defined in the above starter documentation

5.4.2.3. Accessing JMX via HTTP

Get CamelContext information:

```
curl http://localhost:8080/actuator/jolokia/read/org.apache.camel:context=*,type=context,name=*
```

Get route statistics:

```
curl http://localhost:8080/actuator/jolokia/read/org.apache.camel:context=*,type=routes,name=*
```

Execute JMX operations:

```
curl -X POST \
  http://localhost:8080/actuator/jolokia/exec/org.apache.camel:context=camel-
1,type=routes,name=\"route1\"/stop
```

5.4.2.4. Example Jolokia Response

```
{
  "request": {
    "mbean": "org.apache.camel:context=camel-1,type=routes,name=\"myRoute\"",
    "type": "read"
  },
  "value": {
    "ExchangesCompleted": 1250,
    "ExchangesFailed": 2,
    "ExchangesTotal": 1252,
    "MeanProcessingTime": 45,
    "MaxProcessingTime": 890,
    "MinProcessingTime": 12,
    "State": "Started"
  },
  "timestamp": 1634567890,
  "status": 200
}
```

CHAPTER 6. BUILDING ROUTES WITH XML IO DSL

6.1. XML IO DSL

The **xml-io-dsl** is the Camel optimized XML DSL with a very fast and low overhead XML parser. It is a source code generated parser that is Camel specific and can only parse Camel **.xml** route files (not classic Spring **<beans>** XML files).

We recommend that you use **xml-io-dsl** instead of **xml-jaxb-dsl** for Camel XML DSL. It works with all Camel runtimes.

NOTE

When you are using XML IO DSL, the application will by default look for xml files in **src/main/resources/camel/*.xml**.

You can configure this behavior by providing a different path in the **camel.springboot.routes-include-pattern** property:

camel.springboot.routes-include-pattern=/path/to/*.xml

6.1.1. Example

The following **my-route.xml** source file can be loaded and run with Camel CLI:

my-route.xml

```
<routes xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="timer:tick"/>
      <setBody>
         <constant>Hello Camel K!</constant>
       </setBody>
      <to uri="log:info"/>
   </route>
</routes>
```

TIP

You can omit the **xmlns** namespace.

If there is only a single route, you can use **<route>** as the root XML tag instead of **<routes>**.

Running with Camel K

```
kamel run my-route.xml
```

Running with Camel CLI

```
camel run my-route.xml
```

You can use **xml-io-dsl** to declare some beans to be bound to the Camel Registry.

You can declare and Beans define their properties (including **nested** properties) in XML. For example:

Bean declaration and definition

```
<camel>

 <bean name="beanFromProps" type="com.acme.MyBean">
  <properties>
   <property key="field1" value="f1_p" />
   <property key="field2" value="f2_p" />
   <property key="nested.field1" value="nf1_p" />
   <property key="nested.field2" value="nf2_p" />
  </properties>
 </bean>

</camel>
```

While keeping all benefits of fast XML parser used by **xml-io-dsl**, Camel can also process XML elements declared in other XML namespaces and process them separately. With this mechanism it is possible to include XML elements using Spring's **http://www.springframework.org/schema/beans** namespace.

This brings the flexibility of beans into Camel main without actually running any Spring Application Context (or Spring Boot).

When elements from Spring namespace are found, they are used to populate and configure an instance of **org.springframework.beans.factory.support.DefaultListableBeanFactory** and leverage Spring dependency injection to wire the beans together.

These beans are then exposed through normal Camel Registry and may be used by Camel routes.

Here's an example **camel.xml** file, which defines both the routes and beans used (referred to) by the route definition:

camel.xml

```
<camel>
    <bean name="greeterMessage" type="org.apache.camel.main.app.GreeterMessage">
        <properties>
            <property key="msg" value="Hello"/>
        </properties>
    </bean>

    <bean name="greeter" type="org.apache.camel.main.app.Greeter">
        <properties>
            <!-- use # sign for bean references -->
            <property key="message" value="#greeterMessage"/>
        </properties>
    </bean>

    <route id="greeting-route">
        <from uri="timer:start?period=1000"/>
        <setBody>
            <simple>Camel</simple>
```

```
        </setBody>
        <bean ref="greeter"/>
        <log message="${body}"/>
    </route>
</camel>
```

A **my-route** route is referring to **greeter** bean which is defined using Spring **<bean>** element.

More examples can be found on the ApacheCamel JBang page.

6.1.2. Using beans with constructors

When you want to create beans with constructor arguments, from Camel 4.1 onwards you can add them as XML tags. For example:

Camel 4.1+: Beans with constructor tags

```
<camel>

    <bean name="beanFromProps" type="com.acme.MyBean">
        <constructors>
          <constructor index="0" value="true"/>
          <constructor index="1" value="Hello World"/>
        </constructors>
        <!-- and you can still have properties -->
        <properties>
            <property key="field1" value="f1_p" />
            <property key="field2" value="f2_p" />
            <property key="nested.field1" value="nf1_p" />
            <property key="nested.field2" value="nf2_p" />
        </properties>
    </bean>

</camel>
```

If you use Camel 4.0, you must put then constructor arguments in the **type** attribute:

Camel 4.0: Beans with constructor arguments in the type attribute

```
<bean name="beanFromProps" type="com.acme.MyBean(true, 'Hello World')">
    <properties>
        <property key="field1" value="f1_p" />
        <property key="field2" value="f2_p" />
        <property key="nested.field1" value="nf1_p" />
        <property key="nested.field2" value="nf2_p" />
    </properties>
</bean>
```

6.1.3. Creating beans from factory method

A bean can also be created from a **public static** factory method:

Factory method XML

-

```
<bean name="myBean" type="com.acme.MyBean" factoryMethod="createMyBean">
      <constructors>
        <constructor index="0" value="true"/>
        <constructor index="1" value="Hello World"/>
      </constructors>
</bean>
```

When you use a **factoryMethod**, you must provide **constructor** tags for the arguments.

For example, this means that the class **com.acme.MyBean** should be as follows:

Factory method

```
public class MyBean {

   public static MyBean createMyBean(boolean important, String message) {
      MyBean answer = ...
      // create and configure the bean
      return answer;
   }
}
```

NOTE

You must make the factory method **public static** in the created class.

6.1.4. Creating beans from builder classes

You can create a bean created from another builder class as shown below:

Builder XML

```
<bean name="myBean" type="com.acme.MyBean"
      builderClass="com.acme.MyBeanBuilder" builderMethod="createMyBean">
      <properties>
        <property key="id" value="123"/>
        <property key="name" value="Acme"/>
      </constructors>
</bean>
```

NOTE

You must make the builder class **public** with a no-arg default constructor.

You can then use the builder class to create the actual bean by using fluent builder style configuration.

Set the properties on the builder class, and create the bean by invoking the **builderMethod** at the end.

You invocate this method via Java reflection.

6.1.5. Creating beans from factory bean

You can create a bean from a factory bean as shown below:

Factory XML

```
<bean name="myBean" type="com.acme.MyBean"
        factoryBean="com.acme.MyHelper" factoryMethod="createMyBean">
      <constructors>
        <constructor index="0" value="true"/>
        <constructor index="1" value="Hello World"/>
      </constructors>
</bean>
```

TIP

You can also use **factoryBean** to refer to an existing bean by bean id instead of the FQN classname.

When you use a **factoryBean** the, you must provide arguments as **constructor** tags.

For example, the class **com.acme.MyHelper** should be as follows:

Factory bean

```
public class MyHelper {

    public static MyBean createMyBean(boolean important, String message) {
        MyBean answer = ...
        // create and configure the bean
        return answer;
    }
}
```

NOTE

You must make the factory method **public static**.

6.1.6. Creating beans using script language

If you have advanced use-cases, you can inline a script language, such as groovy, java, javascript, and so on, to create the bean.

With scripting, you can be more flexible and use a bit of programming to create and configure the bean:

Scripting

```
<bean name="myBean" type="com.acme.MyBean" scriptLanguage="groovy">
      <script>
    // some groovy script here to create the bean
    bean = ...
    ...
    return bean
      </script>
</bean>
```

NOTE

When you use **script**, the constructors, factory bean, and factory method are not used.

6.1.7. Using init and destroy methods on beans

If you need to do initialization and cleanup work before you use a bean, you can use the **initMethod** and **destroyMethod** which are triggered as appropriate by Camel.

Those methods must be **public void** and have no arguments, as shown below:

Initialization and cleanup methods

```
public class MyBean {

    public void initMe() {
        // do init work here
    }

    public void destroyMe() {
        // do cleanup work here
    }

}
```

You also have to declare those methods in the XML DSL as follows:

Initialization and cleanup XML

```
<bean name="myBean" type="com.acme.MyBean"
        initMethod="initMe" destroyMethod="destroyMe">
      <constructors>
        <constructor index="0" value="true"/>
        <constructor index="1" value="Hello World"/>
      </constructors>
</bean>
```

Both **initMethod** and **destroyMethod** are optional, so a bean does not have to have both.

6.1.8. REST and routes in the same XML IO DSL file

You can have both REST and routes in the same DSL file:

REST and routes in the same XML IO DSL file

```
<camel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://camel.apache.org/schema/spring"
      xsi:schemaLocation="
          http://camel.apache.org/schema/spring
          https://camel.apache.org/schema/spring/camel-spring.xsd">
  <rest id="rest">
      <post id="post" path="start">
          <to uri="direct:start"/>
      </post>
```

```
    </rest>

    <route>
        <from uri="direct:start"/>
        <to uri="amqp:queue:Test.Broker.StreamMessage?
jmsMessageType=Stream&amp;amp;disableReplyTo=true"/>
    </route>
</camel>
```

CHAPTER 7. ROUTING EXPRESSION AND PREDICATE LANGUAGES

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

7.1. INTRODUCTION

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

7.1.1. Overview of the languages

This table gives an overview of the different syntaxes for invoking expression and predicate languages.

Table 7.1. Expression and Predicate Languages

	Language	Static Method	Fluent DSL Method	XML Element	Annotation	Artifact
1	Bean	**bean()**	**EIP().method()**	**method**	**@Bean**	Camel core
2	Constant	**constant()**	**EIP().constant()**	**constant**	**@Constant**	Camel core
3	CSimple	**csimple()**	**EIP().csimple()**	**csimple**	**@CSimple**	Camel core
4	ExchangeProperty	**exchangeProperty()**	**EIP().exchangeProperty()**	**exchangeProperty**	**@exchangeProperty**	Camel core
5	File	**file()**	**EIP().file()**	**file**	**@File**	Camel core
6	Groovy	**groovy()**	**EIP().groovy()**	**groovy**	**@Groovy**	**camel-groovy**
7	Header	**header()**	**EIP().header()**	**header**	**@Header**	Camel core
8	HL7 Terser	**terser()**	**EIP().terser()**	**terser**	**@terser**	**camel-hl7**
9	JQ	**jq()**	**EIP().jq()**	**jq**	**@jq**	**camel-jq**
10	Jsonpath	None	**EIP().jsonpath()**	**jsonpath**	**@JsonPath**	**camel-jsonpath**

	Language	Static Method	Fluent DSL Method	XML Element	Annotation	Artifact
11	Ref	**ref()**	**EIP().ref()**	**ref**	N/A	Camel core
12	Simple	**simple()**	**EIP().simple()**	**simple**	**@Simple**	Camel core
13	Xpath	**xpath()**	**EIP().xpath()**	**xpath**	**@XPath**	Camel core
14	Xquery	**xquery()**	**EIP().xquery()**	**xquery**	**@XQuery**	**camel-saxon**
15	Tokenize	**tokenize()**	**EIP().tokenize()**	**tokenize**	**@Tokenize**	Camel core

7.1.2. How to invoke an expression language

7.1.2.1. 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 {CamelName} 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>
  <version>4.14.0</version>
</dependency>
```

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**.

7.1.2.2. Approaches to invoking

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:

- As a static method
- As a fluent DSL method

- As an XML element
- As an annotation
- As a camel endpoint uri

7.1.2.3. Invoking an expression language 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:

```
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 ");
```

7.1.2.4. Invoking an expression language 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:

```
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 ");
```

7.1.2.5. Invoking an expression language 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 {CamelName} namespace). You can use XPath expressions in an XML DSL content-based router:

```
<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>
```

7.1.2.6. Invoking an expression language as an annotation

Language annotations are used in the context of 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:

```
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:

```
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.

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.

7.1.2.7. Invoking an expression language 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 the Apache Camel Component reference .

CHAPTER 8. TESTING CAMEL SPRING BOOT APPLICATIONS

8.1. TESTING WITH JUNIT 5

For testing, Maven users will need to add the following dependencies to their **pom.xml**:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <version>3.5.11</version> <!-- Use the same version as your Spring Boot version -->
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-spring-junit5</artifactId>
  <version>4.14.4.redhat-00008</version> <!-- use the same version as your Camel core version -->
  <scope>test</scope>
</dependency>
```

To test a Camel Spring Boot application, annotate your test class(es) with **@CamelSpringBootTest**. This brings Camel's Spring Test support to your application, so that you can write tests using Spring Boot test conventions.

To get the **CamelContext** or **ProducerTemplate**, you can inject them into the class in the normal Spring manner, using **@Autowired**.

You can also use camel-test-spring-junit5 to configure tests declaratively. This example uses the **@MockEndpoints** annotation to auto-mock an endpoint:

```
@CamelSpringBootTest
@SpringBootApplication
@MockEndpoints("direct:end")
public class MyApplicationTest {

  @Autowired
  private ProducerTemplate template;

  @EndpointInject("mock:direct:end")
  private MockEndpoint mock;

  @Test
  public void testReceive() throws Exception {
    mock.expectedBodiesReceived("Hello");
    template.sendBody("direct:start", "Hello");
    mock.assertIsSatisfied();
  }

}
```

TIP

For a sample application, see the Infinispan example in the camel-spring-boot-examples repository.

The **@CamelSpringBootTest** extends **@SpringBootTest** functionality, including features, plus: - Provides route testing utilities - Includes CamelContext, route advisors, and mock endpoints - Enables Camel test annotations - Works with @MockEndpoints, @UseAdviceWith, etc.

An example of **@UseAdviceWith** is when a route modification is applied at runtime:

```
@CamelSpringBootTest
@SpringBootApplication
public class MySpringBootApplicationTest {

 @Autowired
 private CamelContext camelContext;

 @Autowired
 private ProducerTemplate producerTemplate;

 public static void main(String[] args) {
  SpringApplication.run(MySpringBootApplicationTest.class, args);
 }

 // Spring context fixtures
 @Configuration
 static class TestConfig {

  @Bean
  RoutesBuilder route() {
   return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
     from("timer:hello1?period={{timer.period}}").routeId("hello1")
       .transform().method("myBean", "saySomething")
       .filter(simple("${body} contains 'foo'"))
       .to("log:foo")
       .end()
       .to("stream:out");
    }
   };
  }
 }

 @Test
 public void test() throws Exception {
  // Apply advice before getting the mock endpoint
  AdviceWith.adviceWith(camelContext, "hello1",
    // intercepting an exchange on route
    r -> {
     // replacing consumer with direct component
     r.replaceFromWith("direct:start");
     // mocking producer
     r.mockEndpoints("stream*");
    }
  );

  // Start the context after applying advice
  camelContext.start();
```

```
    // Get mock endpoint after advice is applied
    MockEndpoint mock = camelContext.getEndpoint("mock:stream:out", MockEndpoint.class);

    // setting expectations
    mock.expectedMessageCount(1);
    mock.expectedBodiesReceived("Hello World");

    // invoking consumer
    producerTemplate.sendBody("direct:start", null);

    // asserting mock is satisfied
    mock.assertIsSatisfied();
  }
}
```

In some case we need to declare both, a combination of **@CamelSpringBootTest** and **@SpringBootTest**, when an applicationContext is strictly required, just adding **@SpringBootTest(classes = DocTest.TestApplication.class)** to specify which configuration class to use:

```
@CamelSpringBootTest
@SpringBootTest(classes = MySpringBootApplicationTest.TestConfig.class)
@SpringBootApplication
@MockEndpoints("direct:end")
public class MySpringBootApplicationTest {

 @Autowired
 private ProducerTemplate template;

 @EndpointInject("mock:direct:end")
 private MockEndpoint mock;

 @Configuration
 static class TestConfig {

  @Bean
  RoutesBuilder route() {
   return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
     from("direct:start").routeId("hello2")
       .setBody(simple("Hello World"))
       .log("Received: ${body}")
       .to("direct:end");

     from("direct:end")
       .log("${body}");
    }
   };
  }
 }

 @Test
 public void testReceive() throws Exception {
  mock.expectedMessageCount(1);
  mock.expectedBodiesReceived("Hello World");
```

```
    template.sendBody("direct:start", null);
    mock.assertIsSatisfied();
  }

}
```

Another approach is to test classic Spring XML context setup. In this case, we need to use **@ContextConfiguration** to specify the XML file, retrieving the Main Camel runtime context through dependency injection:

```
@Autowired
protected CamelContext camelContext;
```

Then, to reloads the Spring ApplicationContext after each test method we can use:

```
@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
```

Alternative modes: **BEFORE_CLASS**, **AFTER_CLASS**, **BEFORE_EACH_TEST_METHOD**

A full example is described here:

```
package com.redhat.plain;
 .....
@CamelSpringBootTest
@ContextConfiguration
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_EACH_TEST_METHOD)
public class MySpringBootApplicationPlainTest {

 @Autowired
 protected CamelContext camelContext;

 @EndpointInject("mock:a")
 protected MockEndpoint mockA;

 @Produce("direct:start")
 protected ProducerTemplate start;

 @Test
 public void testPositive() throws Exception {
  assertEquals(ServiceStatus.Started, camelContext.getStatus());
  start.sendBody("Hello");
  MockEndpoint.assertIsSatisfied(camelContext);
  mockA.expectedBodiesReceived("Hello");
 }
}
```

Where Spring XML context requires to be placed under resources at declared class package **com.redhat.plain** using the naming pattern className-context.xml

Project Structure

```
src/
 ├─── main/
 |    ├─── java/
 |    |    └─── com/redhat/
```

```
|   |       ├── Application.java
|   |       └── route/
|   |           └── MyRoute.java
|   └── resources/
|       └── application.yml
└── test/
    ├── java/
    |   └── com/redhat/
    └── resources/
        └── MySpringBootApplicationPlainTest-context.yml
```

XML declaration for the example above:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
    ">

 <!-- Camel Context configuration -->
 <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">

  <route xmlns="http://camel.apache.org/schema/spring" id="helloX">
   <from id="from1" uri="direct:start"/>
   <filter id="filter1">
    <simple>${body} contains 'foo'</simple>
    <to id="to1" uri="log:foo"/>
   </filter>
   <to id="to2" uri="stream:out"/>
  </route>

 </camelContext>

</beans>
```

CHAPTER 9. ADVANCED CAMEL PROGRAMMING

This guide describes how to use the Apache Camel API.

9.1. UNDERSTANDING MESSAGE FORMATS

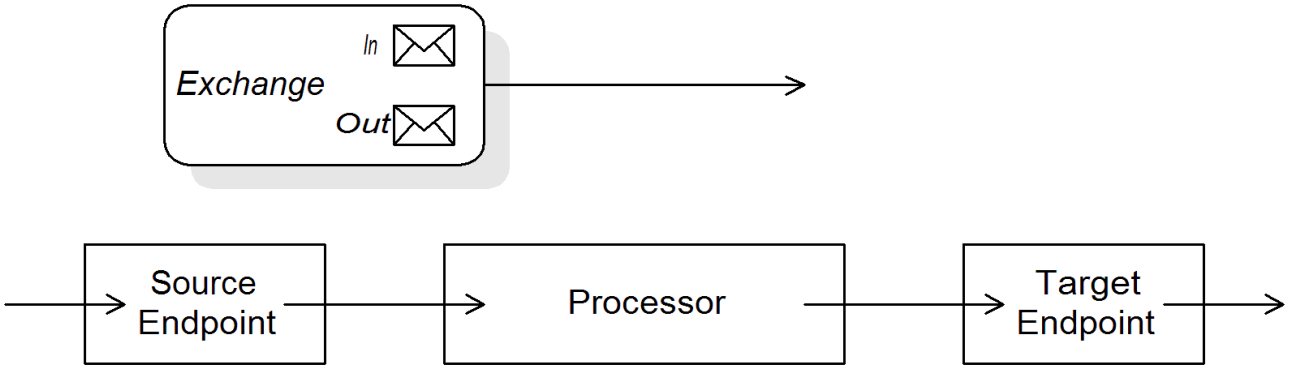
Before you can begin programming with Red Hat build of Apache Camel, you should have a clear understanding of how messages and message exchanges are modelled. Because Red Hat build of Apache Camel can process many message formats, the basic message type is designed to have an abstract format. Red Hat build of Apache Camel provides the APIs needed to access and transform the data formats that underly message bodies and message headers.

9.1.1. Exchanges

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 Red Hat build of 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 9.1. 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 processors in between.

9.1.1.1. The exchange interface

The `org.apache.camel.Exchange`interface defines methods to access **In** and **Out** messages, as shown in Example 9.1, “Exchange Methods”.

Example 9.1. Exchange Methods

```
// Access the In message
Message getMessage();
void    setIn(Message in);

// Access the Out message (if any)
Message getMessage();
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 9.5.1.1.5, "Exchange".

9.1.1.2. Lazy creation of messages

Red Hat build of 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 **getMessage()** or **getMessage()**). The lazy message creation semantics are implemented by the **org.apache.camel.impl.DefaultExchange** class.

If you call one of the no-argument accessors (**getMessage()** or **getMessage()**), 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.[1]

9.1.1.3. Lazy creation of exchange ids

Red Hat build of 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, about how to register UUID generators with the **CamelContext**, Section 9.1.4.2, "Custom uuid generator".

9.1.2. Messages

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 Red Hat build 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.

9.1.2.1. The message interface definition

The `org.apache.camel.Message`interface defines methods to access the message body, message headers and message attachments, as shown in Example 9.2, "Message Interface".

Example 9.2. 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 information about methods in the Message interface, see Section 9.5.1.1.6, "Message".

9.1.2.2. Lazy creation of bodies, headers, and attachments

Red Hat build of 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 9.5.1.1.6, “Message”.

9.1.2.3. Lazy creation of message ids

Red Hat build of 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, about how to register UUID generators with the **CamelContext**, Section 9.1.4.2, “Custom uuid generator”.

9.1.2.4. 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 Red Hat build of 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*.

9.1.2.5. 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.

There are various methods in the Red Hat build of 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:

```
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.

9.1.2.6. 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 information about supported conversion types, see Section 9.1.2.5, “Type converters”.

9.1.2.7. 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.

9.1.2.8. 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 9.3, “Unmarshalling a Java Object” .

Example 9.3. Unmarshalling a Java Object

```
from("file://tmp/appfiles/serialized")
    .unmarshal()
    .serialization()
    .<FurtherProcessing>
    .to("TargetURL ");
```

9.1.2.9. 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 Red Hat build of 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.

9.1.3. Built-in type converters

This section describes the conversions supported by the master type converter. These conversions are built into the Red Hat build of 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 9.4, “Converting a Value to a String”.

Example 9.4. Converting a Value to a String

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
String str_value = tc.convertTo(String.class, value);
```

9.1.3.1. Basic type converters

Red Hat build of 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 allse 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.

9.1.3.2. Collection type converters

Red Hat build of 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.

9.1.3.3. Map type converters

Red Hat build of 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.

9.1.3.4. 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.

9.1.3.5. 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**

9.1.3.6. 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"/>
```

9.1.3.7. Custom type converters

Red Hat build of Apache Camel also enables you to implement your own custom type converters.

9.1.4. Built-in uuid generators

Red Hat build of Apache Camel enables you to register a UUID generator in the **CamelContext**. This UUID generator is then used whenever Red Hat build of 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.

9.1.4.1. Provided uuid generators

You can configure Red Hat build of 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.

9.1.4.2. Custom uuid generator

To implement a custom UUID generator, implement the **org.apache.camel.spi.UuidGenerator** interface, which is shown in Example 9.5, "UuidGenerator Interface".

The **generateUuid()** must be implemented to return a unique ID string.

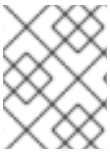
Example 9.5. UuidGenerator Interface

```
package org.apache.camel.spi;

/**
 * Generator to generate UUID strings.
 */
public interface UuidGenerator {
    String generateUuid();
}
```

9.1.4.3. 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**:

```
getContext().setUuidGenerator(new org.apache.camel.impl.SimpleUuidGenerator());
```



NOTE

The **setUuidGenerator()** method should be called during startup, **before** any routes are activated.

9.1.4.4. 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>
```

9.2. IMPLEMENTING A PROCESSOR

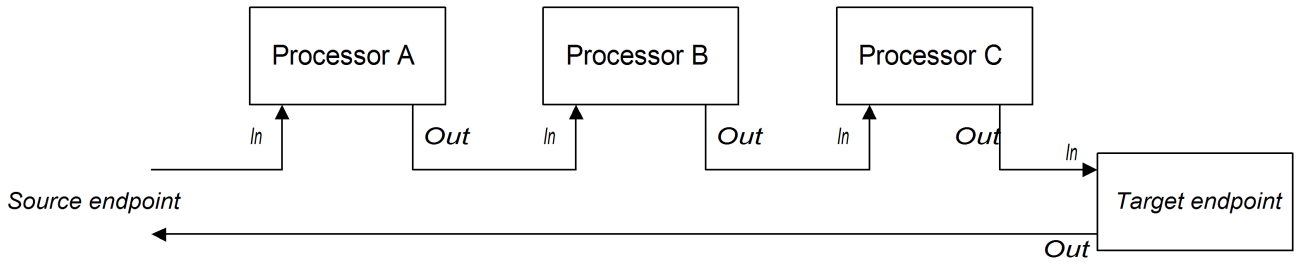
Red Hat build of Apache Camel allows you to implement a custom processor. You can then insert the custom processor into a route to perform operations on exchange objects as they pass through the route.

9.2.1. Processing model

9.2.1.1. Pipelining model

The *pipelining model* describes the way in which processors are arranged in Pipes. Pipelining is the most common way to process a sequence of endpoints (a producer endpoint is just a special type of processor).

Figure 9.2. 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 9.6, “Java DSL Pipeline” shows a Java DSL pipeline constructed from a sequence of two processors, **ProcessorA**, **ProcessorB**, and a producer endpoint, *TargetURI*.

Example 9.6. Java DSL Pipeline

```
from(_SourceURI_).pipeline(ProcessorA, ProcessorB, _TargetURI_);
```

9.2.2. Implementing a simple processor

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.

9.2.2.1. Processor interface

Simple processors are created by implementing the org.apache.camel.Processor interface. As shown in Example 9.7, "Processor Interface", the interface defines a single method, **process()**, which processes an exchange object.

Example 9.7. Processor Interface

```
package org.apache.camel;

public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

9.2.2.2. Implementing the processor interface

To create a simple processor you must implement the Processor interface and provide the logic for the **process()** method. Example 9.8, "Simple Processor Implementation" shows the outline of a simple processor implementation.

Example 9.8. 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 code in the **process()** method gets executed **before** the exchange object is delegated to the next processor in the chain.

9.2.2.3. 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:

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL ").process(myProc).to("TargetURL ");
```

9.2.3. Accessing message content

9.2.3.1. 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.getMessage()**), and then use the Message interface to retrieve the individual headers (for example, using **Message.getHeader()**).

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.

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 more information about the Message interface, see Section 9.1.2, “Messages”

9.2.3.2. 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 9.9, “Accessing the Message Body” .

Example 9.9. 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.getMessage();
        in.setBody(in.getBody(String.class) + " World!");
    }
}
```

9.2.3.3. Accessing message attachments

You can access a message’s attachments using either the **Message.getAttachment()** method or the **Message.getAttachments()** method.

9.2.4. The exchangehelper class

The **org.apache.camel.util.ExchangeHelper** class is a Red Hat build of Apache Camel utility class that provides methods that are useful when implementing a processor.

9.2.4.1. 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 9.10. 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 9.11, "Creating a File Endpoint" shows how to create a new file endpoint from an exchange instance **exchange**

Example 9.11. Creating a File Endpoint

```
Endpoint file_endp = ExchangeHelper.resolveEndpoint(exchange, "file://tmp/messages/in.xml");
```

9.2.4.2. Wrapping the exchange accessors

The **ExchangeHelper** class provides several static methods of the form **getMandatoryBeanProperty()**, which wrap the corresponding **getBeanProperty()** methods on the **Exchange** class. The difference between them is that the original **getBeanProperty()** accessors return **null**, if the corresponding property is unavailable, and the **getMandatoryBeanProperty()** 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 { ... }
    ...
}
```

9.2.4.3. 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) { ... }
    ...
}
```

9.2.4.4. 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).

9.3. TYPE CONVERTERS

Red Hat build of 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.

9.3.1. Type converter architecture

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 Section 9.1.2.5, "Type converters".

9.3.1.1. Type converter interface

All type converters must implement **org.apache.camel.TypeConverter** interface definition.

Example 9.12. TypeConverter Interface

```
package org.apache.camel;

public interface TypeConverter {
```

```
    <T> T convertTo(Class<T> type, Object value);
}
```

9.3.1.2. Controller type converter

The Red Hat build of Apache Camel type converter mechanism follows a controller/worker pattern. There are many **worker** type converters, which are each capable of performing a limited number of type conversions, and a single **controller** type converter, which aggregates the type conversions performed by the workers. The controller type converter acts as a front-end for the worker type converters. When you request the controller to perform a type conversion, it selects the appropriate worker and delegates the conversion task to that worker.

For users of the type conversion mechanism, the controller type converter is the most important because it provides the entry point for accessing the conversion mechanism. During start up, Red Hat build of Apache Camel automatically associates a controller type converter instance with the **CamelContext** object. To obtain a reference to the controller type converter, you call the **CamelContext.getTypeConverter()** method. For example, if you have an exchange object, **exchange**.

Example 9.13. Getting a Controller Type Converter

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
```

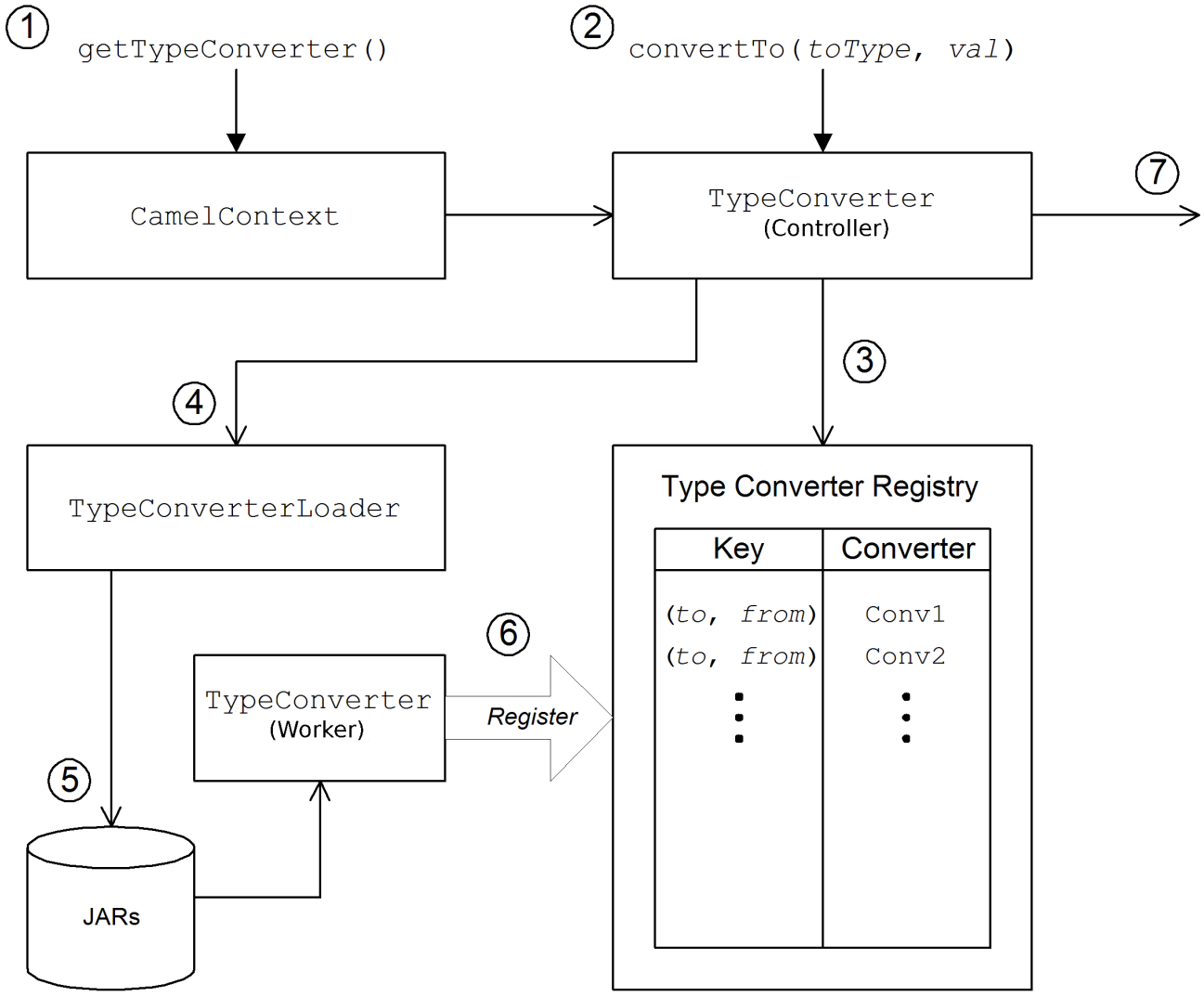
9.3.1.3. Type converter loader

The controller type converter uses a *type converter loader* to populate the registry of worker type converters. A type converter loader is any class that implements the TypeConverterLoader interface. Red Hat build of Apache Camel currently uses only one kind of type converter loader – the *annotation type converter loader* (of **AnnotationTypeConverterLoader** type).

9.3.1.4. Type conversion process

The following overview shows the type conversion process for converting a given data value, **value**, to a specified type, **toType**.

Figure 9.3. Type Conversion Process



The type conversion mechanism proceeds as follows:

1. The **CamelContext** object holds a reference to the controller TypeConverter instance. The first step in the conversion process is to retrieve the controller type converter by calling **CamelContext.getTypeConverter()**.

2. Type conversion is initiated by calling the **convertTo()** method on the controller 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 controller type converter is a front end for many different worker type converters, it looks up the appropriate worker 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 controller type converter calls the worker's **convertTo()** method and returns the result.

4. If a suitable type converter **cannot** be found in the registry, the controller 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.

6. If the type converter loader is successful, a new worker 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.

9.3.2. Handling duplicate type converters

You can configure what must happen if a duplicate type converter is added.

In the **TypeConverterRegistry** you can set the action to **Override**, **Ignore** or **Fail** using the following code:

```
typeconverterregistry = camelContext.getTypeConverter()
// Define the behavior if the TypeConverter already exists
typeconverterregistry.setTypeConverterExists(TypeConverterExists.Override);
```

Override in this code can be replaced by **Ignore** or **Fail**, depending on your requirements.

9.3.2.1. 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

}
```

9.3.3. Implementing type converter using annotations

The type conversion mechanism can easily be customized by adding a new worker type converter. This section describes how to implement a worker type converter and how to integrate it with Red Hat build of Apache Camel, so that it is automatically loaded by the annotation type converter loader.

9.3.3.1. How to implement a type converter

To implement a custom type converter, perform the following steps:

1. Section 9.3.3.2, “Implement an annotated converter class”

2. Section 9.3.3.3, “Create a typeconverter file”

3. Section 9.3.3.4, "Package the type converter"

9.3.3.2. Implement an annotated converter class

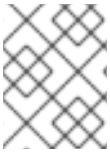
You can implement a custom type converter class using the **@Converter** annotation. You must annotate the class itself and each of the **static** methods intended to perform type conversion. Each converter method takes an argument that defines the **from** type, optionally takes a second **Exchange** 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.

The following example uses 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 9.14. Example of an Annotated Converter Class

```
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.

9.3.3.3. 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.

9.3.3.4. 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 Red Hat build of Apache Camel application.

9.3.3.5. 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 controller 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 controller 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;
    }
    ...
}
```

9.3.4. Implementing a type converter directly

Generally, the recommended way to implement a type converter is to use an annotated class.

But if you want to have complete control over the registration of your type converter, you can implement a custom worker type converter and add it directly to the type converter registry, as described here.

9.3.4.1. 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);
    }
}
```

9.3.4.2. 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.

9.4. PRODUCER AND CONSUMER TEMPLATES

The producer and consumer templates in {CamelName} 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 {CamelName}, the producer template and consumer template provide simplified interfaces for sending messages to and receiving messages from producer endpoints and consumer endpoints.

9.4.1. Using the producer template

9.4.1.1. Introduction to the producer template

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:

- Section 9.4.1.1.1, “Synchronous invocation”
- Section 9.4.1.1.2, “Synchronous invocation with a processor”
- Section 9.4.1.1.3, “Asynchronous invocation”
- Section 9.4.1.1.4, “Asynchronous invocation with a callback”

Alternatively, see Section 9.4.2, “Using fluent producer templates” .

9.4.1.1.1. Synchronous invocation

The methods for invoking endpoints synchronously have names of the form **sendSuffix()** and **requestSuffix()**. 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" );
```

9.4.1.1.2. 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 header 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.getMessage().setBody("<hello>world!</hello>");
   }
}
```

9.4.1.1.3. Asynchronous invocation

The methods for invoking endpoints **asynchronously** have names of the form **asyncSendSuffix()** and **asyncRequestSuffix()**. 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.getMessage().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);
```

9.4.1.1.4. 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.getMessage().setBody("Hello");

Future<Exchange> future = template.asyncCallback("direct:start", exchange, new
SynchronizationAdapter() {
    @Override
    public void onComplete(Exchange exchange) {
        assertEquals("Hello World", exchange.getMessage().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);
```

9.4.1.2. Synchronous send

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.

9.4.1.2.1. 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);
```

9.4.1.2.2. 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.

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
);
```

9.4.1.2.3. 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)]] ===== 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
);
```

9.4.1.2.4. 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
);
```

9.4.1.3. Synchronous request with inout pattern

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.

9.4.1.3.1. 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);
```

9.4.1.3.2. 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.

```
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
);
```

9.4.1.3.3. 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.

```
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
);
```

9.4.1.4. Asynchronous send

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.

9.4.1.4.1. 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.

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);
```

9.4.1.4.2. 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);
```

9.4.1.4.3. 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);
```

9.4.1.5. Asynchronous request with inout pattern

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.

9.4.1.5.1. 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.

```
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
);
```

9.4.1.5.2. 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.

```
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
);
```

9.4.1.6. Asynchronous send with callback

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 associated processing – invocation of the producer endpoint, waiting for a reply and processing the reply – occur asynchronously in a sub-thread.

9.4.1.6.1. 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()**.

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
);
```

9.4.1.6.2. 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
);
```

9.4.1.6.3. 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
);
```

9.4.1.6.4. 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
);
```

9.4.2. Using fluent producer templates

The **FluentProducerTemplate** interface provides a fluent syntax for building a producer. The **DefaultFluentProducerTemplate** class implements **FluentProducerTemplate**.

The following example uses a **DefaultFluentProducerTemplate** object to set headers and a body:

```
Integer result = DefaultFluentProducerTemplate.on(context)
    .withHeader("key-1", "value-1")
    .withHeader("key-2", "value-2")
    .withBody("Hello")
    .to("direct:inout")
    .request(Integer.class);
```

The following example shows how to specify a processor in a **DefaultFluentProducerTemplate** object:

```
Integer result = DefaultFluentProducerTemplate.on(context)
    .withProcessor(exchange -> exchange.getMessage().setBody("Hello World"))
    .to("direct:exception")
    .request(Integer.class);
```

The next example shows how to customize the default fluent producer template:

```
Object result = DefaultFluentProducerTemplate.on(context)
    .withTemplateCustomizer(
        template -> {
            template.setExecutorService(myExecutor);
            template.setMaximumCacheSize(10);
        }
    )
    .withBody("the body")
    .to("direct:start")
    .request();
```

To create a **FluentProducerTemplate** instance, call the **createFluentProducerTemplate()** method on the Camel context. For example:

```
FluentProducerTemplate fluentProducerTemplate = context.createFluentProducerTemplate();
```

9.4.3. Using the consumer template

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).

9.4.3.1. 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()**.

9.4.3.2. 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*();
```

9.4.3.3. 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);
```

9.4.3.4. 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);
```

9.5. IMPLEMENTING A COMPONENT

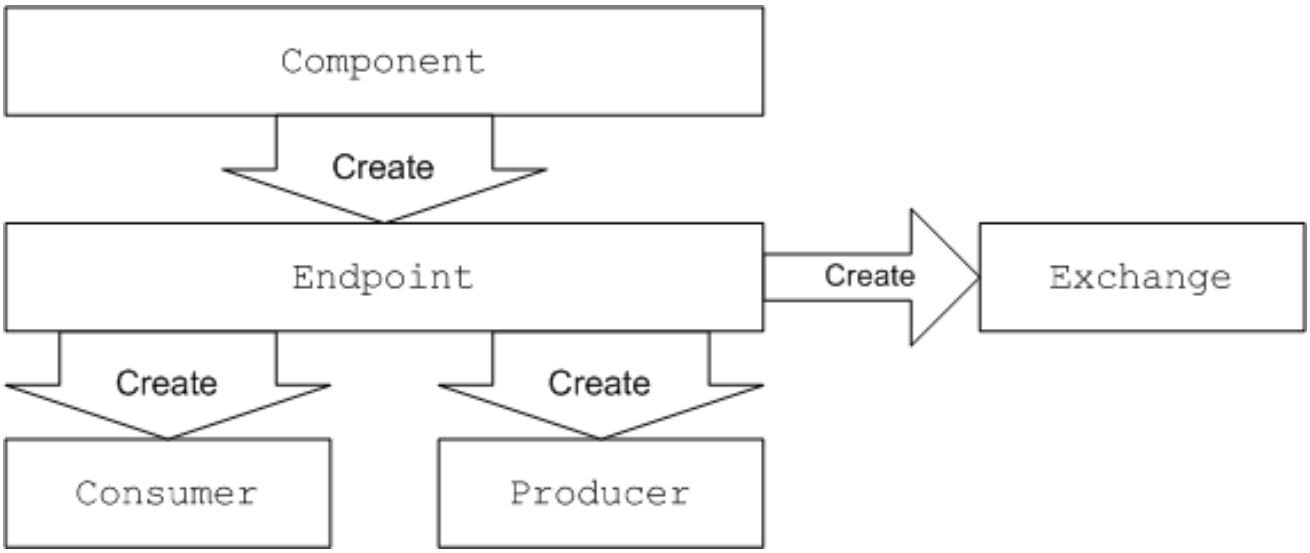
This chapter provides a general overview of the approaches can be used to implement a Red Hat build of Apache Camel component.

9.5.1. Component architecture

9.5.1.1. Factory patterns for a component

An Red Hat build of 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.

Figure 9.4. Component Factory Patterns



9.5.1.1.1. 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 Red Hat build of Apache Camel, you must define the association between a particular component prefix and the name of the class that implements the component.

9.5.1.1.2. Endpoint

Each endpoint instance encapsulates a particular endpoint URI. Every time Red Hat build of 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.

9.5.1.1.3. 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 perspective 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.

9.5.1.1.4. 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 perspective 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.

9.5.1.1.5. 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.

9.5.1.1.6. Message

There are two different message slots in an **Exchange** object:

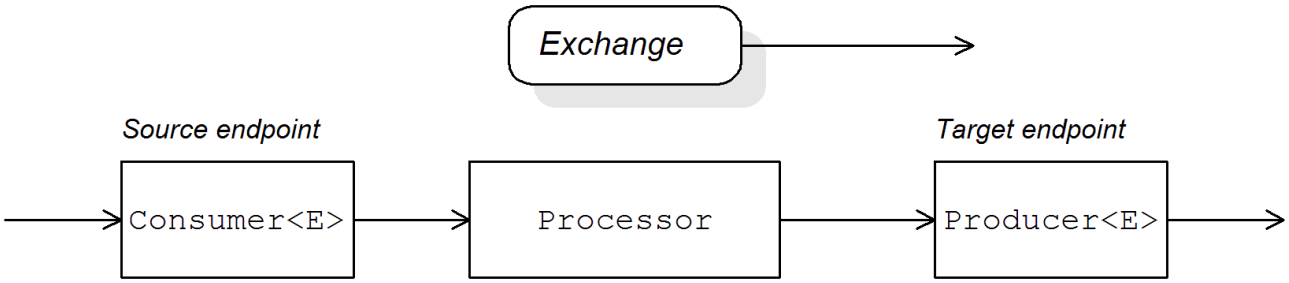
- **In** message – holds the current message.
- **Out** message – temporarily holds a reply message.

all 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.

9.5.1.2. Using a component in a route

A Red Hat build of 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.

Figure 9.5. Consumer and Producer Instances in a Route



9.5.1.2.1. 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, Red Hat build of Apache Camel creates the appropriate **Consumer** type based on the component prefix from the endpoint URI.

9.5.1.2.2. 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.

9.5.1.2.3. 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, Red Hat build of Apache Camel creates the appropriate **Producer** type based on the component prefix from the endpoint URI.

9.5.1.3. Consumer patterns and threading

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:

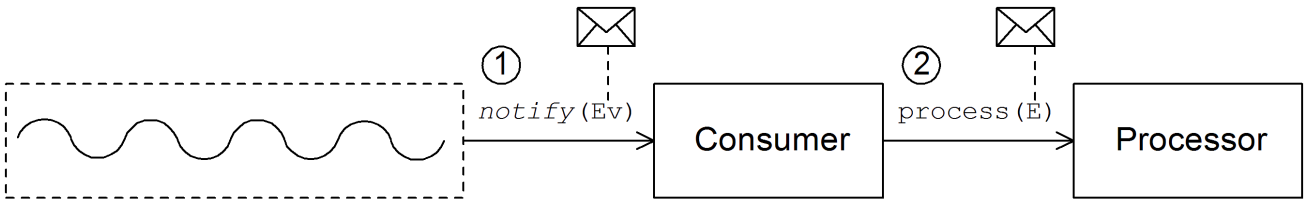
- Section 9.5.1.3.1, "Event-driven pattern" – The consumer is driven by an external thread.
- Section 9.5.1.3.2, "Scheduled poll pattern" – The consumer is driven by a dedicated thread pool.
- Section 9.5.1.3.3, "Polling pattern" – The threading model is left undefined.

9.5.1.3.1. 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 Red Hat build of Apache Camel JMX component, where events are initiated by the JMX library. The JMX library calls the **handleNotification()** method to initiate request processing.

In the following 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 9.6. Event-Driven Consumer



The event-driven consumer processes incoming requests as follows:

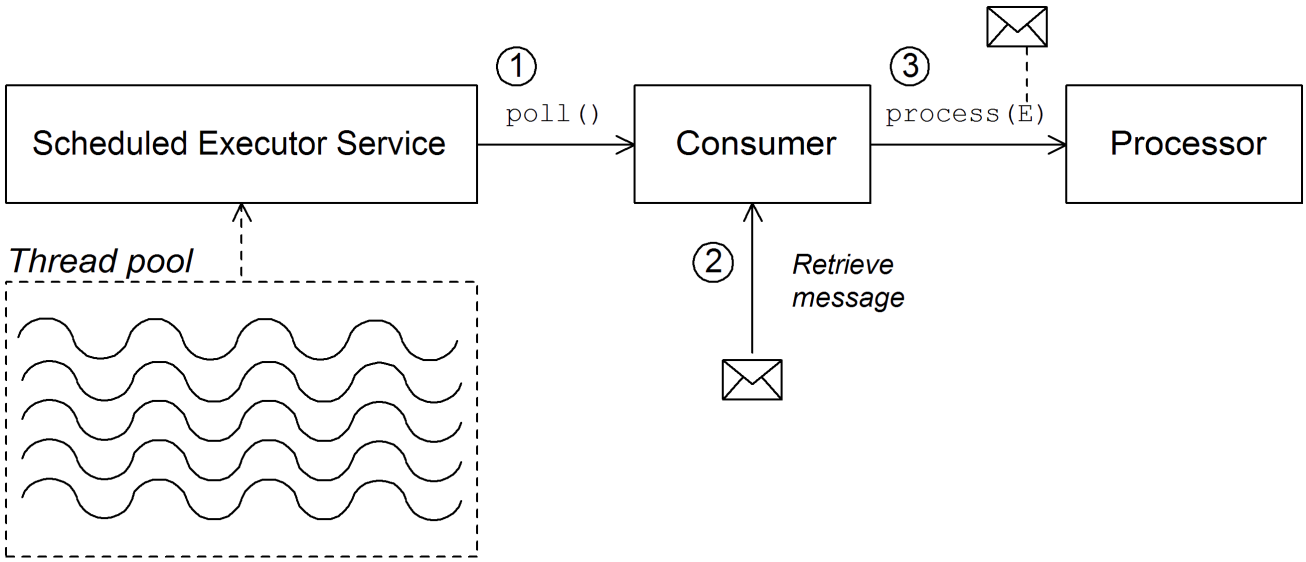
1. The consumer must implement a method to receive the incoming event. This is represented by the **notify()** method). The thread that calls **notify()** is normally a separate part of the application, so the consumer's threading policy is externally driven.
   For example, in the case of the JMX consumer 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.

9.5.1.3.2. Scheduled poll pattern

In the scheduled poll pattern, the consumer retrieves incoming requests by checking at regular time intervals whether 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 9.7. 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.

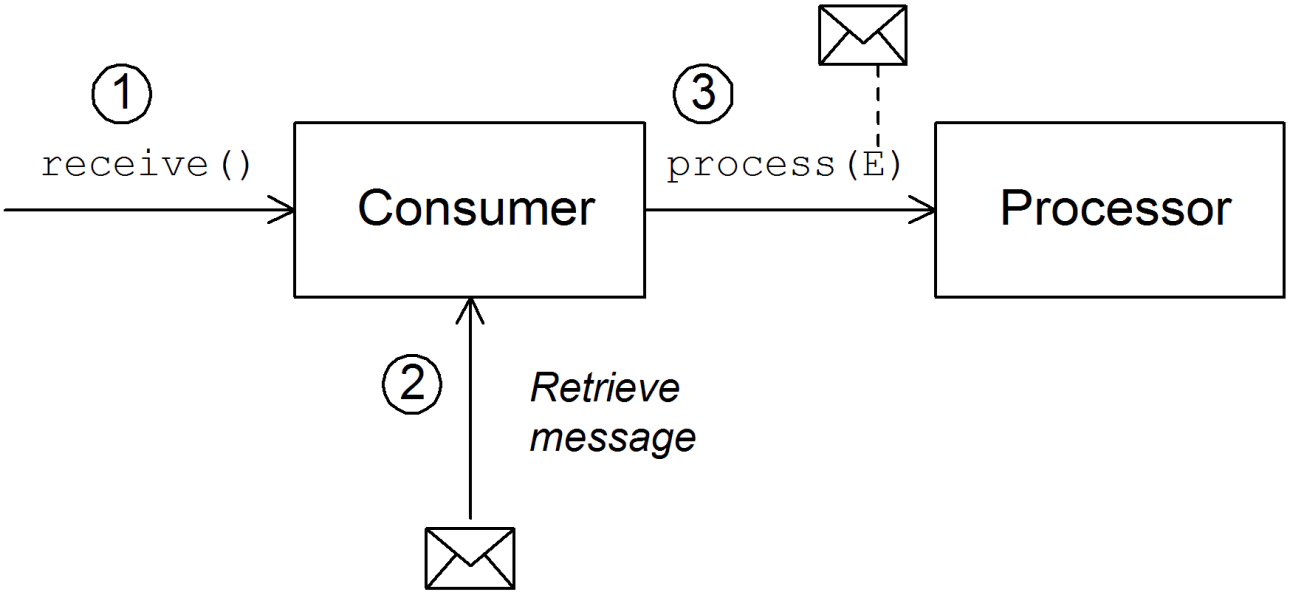
9.5.1.3.3. Polling pattern

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 in the polling pattern.

Figure 9.8. 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 defined by the component implementation.

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 [2] 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.

9.5.1.4. Asynchronous processing

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.

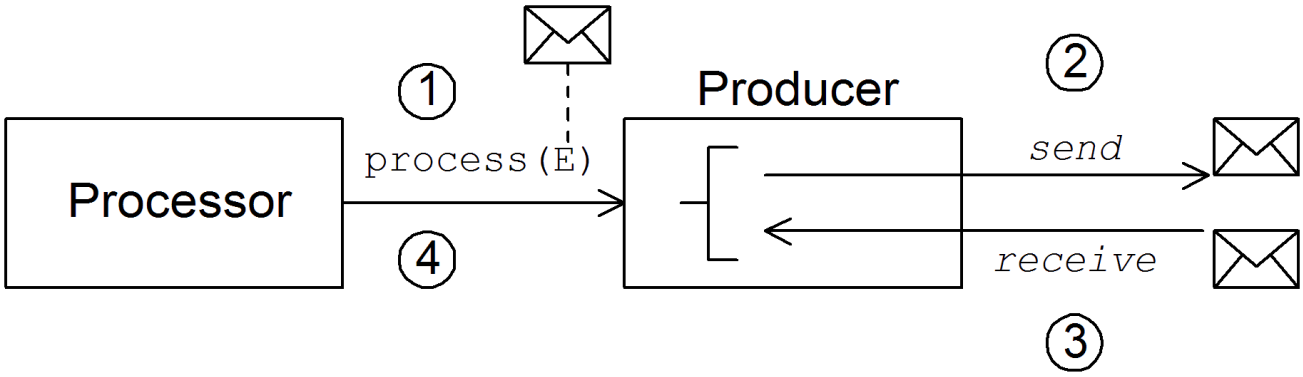
Sometimes, You might prefer to decouple the preceding processor from the producer, so that the processor's thread is released immediately and the **process()** call does **not** block. In this case, you should implement the producer using an *asynchronous* pattern, which gives the preceding processor 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.

9.5.1.4.1. Synchronous producer

The following overview shows an outline of a synchronous producer, where the preceding processor blocks until the producer has finished processing the exchange.

Figure 9.9. 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.

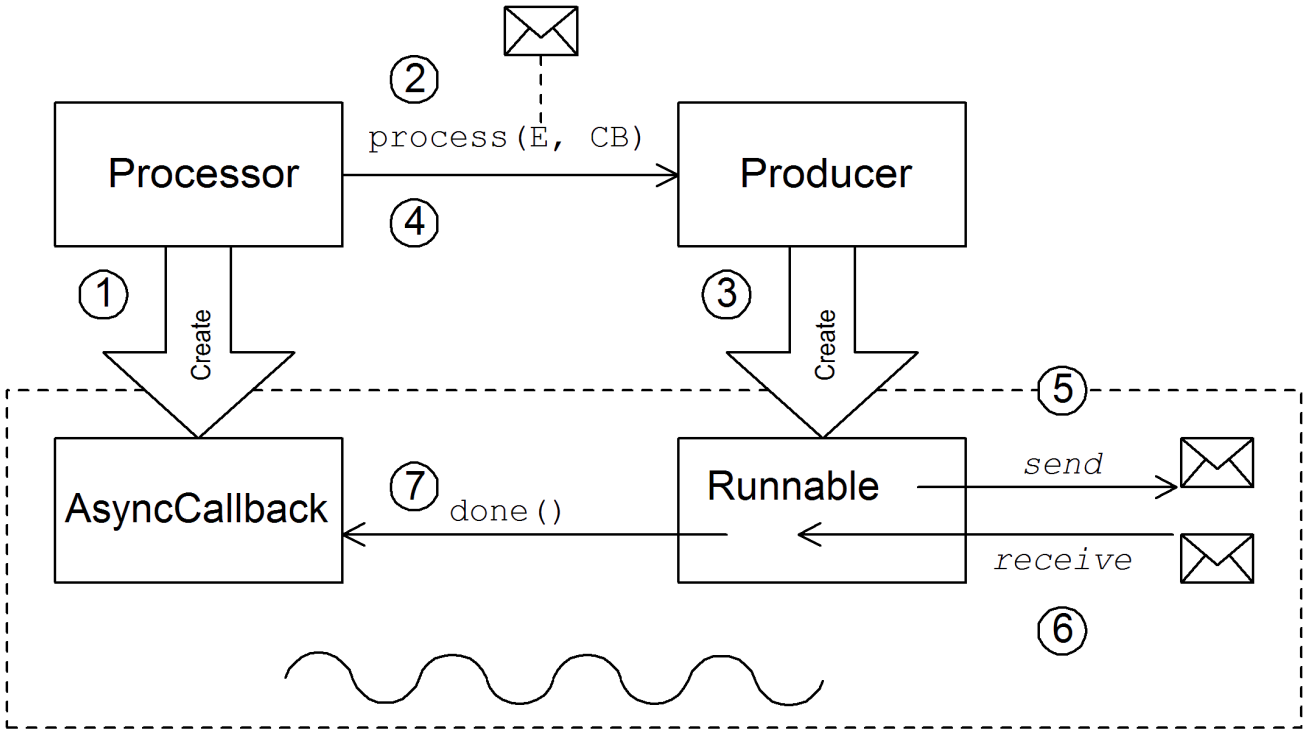
3. If required by the exchange pattern, the producer waits for the reply (**Out** message) to arrive from the endpoint. This step can cause the **process()** method to block indefinitely. However, if the exchange pattern does not mandate a reply, the **process()** method can return immediately after sending the request.

4. When the **process()** method returns, the exchange object contains the reply from the synchronous call (an **Out** message message).

9.5.1.4.2. Asynchronous producer

The following overview 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 9.10. 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).

9.5.2. How to implement a component

This section gives a brief overview of the steps required to implement a custom Red Hat build of Apache Camel component.

9.5.2.1. 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

9.5.2.2. 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.
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()**

1. **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.

2. **Implement the Producer interface**– To implement a producer, you extend the **DefaultProducer** class and implement the **process()** method.

3. **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.

9.5.2.3. 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.

- **Configure Red Hat build of Apache Camel to auto-discover the component**– Auto-discovery, ensures that Red Hat build of Apache Camel automatically loads the component on demand.

9.5.3. Auto-discovery and configuration

9.5.3.1. Setting up auto-discovery

Auto-discovery is a mechanism that enables you to dynamically add components to your Red Hat build of Apache Camel application. The component URI prefix is used as a key to load components on demand. For example, if Red Hat build of 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.

9.5.3.1.1. 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.

9.5.3.1.2. 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.

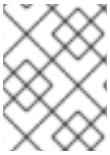
9.5.3.1.3. Example

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**.

9.5.3.2. Configuring a component

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.

9.5.3.2.1. 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 **getProperty()** method and the **setProperty()** method access the value of *property*.

9.5.3.2.2. Configure the component in spring

To configure a component in Spring, edit the configuration file, **META-INF/spring/camel-context.xml**.

Example 9.15. Configuring a Component in Spring

```
<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 **setProperty()** on the component.

9.5.3.2.3. Examples

The following example configures the Apache Camel's JMS component by defining a bean element with the ID **jms**.

These settings are added to the Spring configuration file, **camel-context.xml**.

Example 9.16. JMS Component Spring Configuration

```
<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> (1)
  </camelContext>

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent"> (2)
    <property name="connectionFactory" (3)
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL"
                value="vm://localhost?broker.persistent=false&amp;broker.useJmx=false"/> (4)
      </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).

9.6. COMPONENT INTERFACE

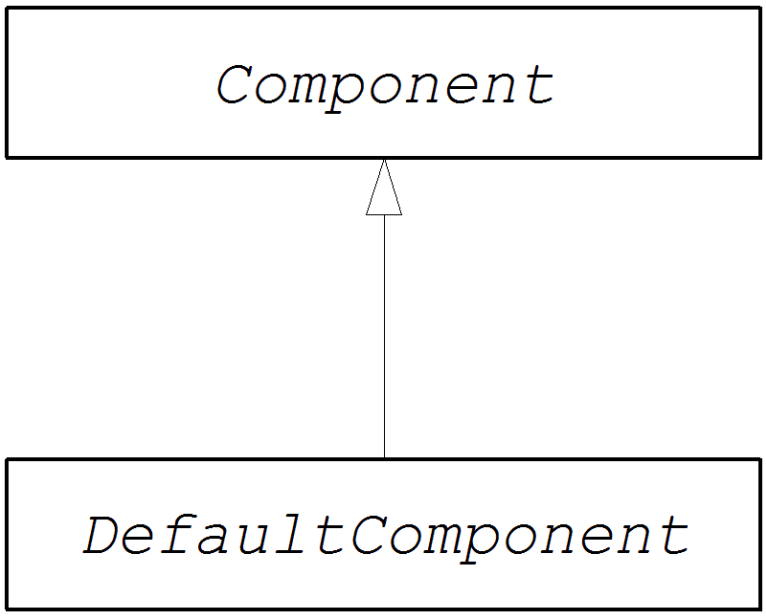
This chapter describes how to implement the Component interface.

9.6.1. The component interface

To implement a Red Hat build of 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.

The following overview shows the relevant Java interfaces and classes that make up the **Component** inheritance hierarchy.

Figure 9.11. Component Inheritance Hierarchy



9.6.1.1. The component interface definition

Example 9.17, "Component Interface" shows the definition of the org.apache.camel.Component interface.

Example 9.17. Component Interface

```
package org.apache.camel;

public interface Component {
    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    Endpoint createEndpoint(String uri) throws Exception;
}
```

9.6.1.2. 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.

9.6.2. Implementing the component interface

9.6.2.1. 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).

9.6.2.2. 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**
parameters	Two entries are set in **java.util.Map**: • parameter **delete** is boolean **true** • parameter **moveNamePostfix** has the string value, **.old**.

9.6.2.3. 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:

```
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.

9.6.2.4. 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**:

```
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.

9.6.2.5. 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. .

NOTE

Prior to Red Hat build of 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.

9.6.2.6. 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.

9.6.2.7. Creating an endpoint

The following example implements the **DefaultComponent.createEndpoint()** method, which is responsible for creating endpoint instances on demand.

Example 9.18. Implementation of createEndpoint()

```
public class CustomComponent extends DefaultComponent { (1)
    ...
    protected Endpoint createEndpoint(String uri, String remaining, Map parameters) throws
Exception { (2)
        _CustomEndpoint_ result = new _CustomEndpoint_(uri, this); (3)
        // ...
        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.

(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**.

9.6.2.8. Example

Example 9.19, "FileComponent Implementation" shows a sample implementation of a **FileComponent** class.

Example 9.19. 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() { (1)
    }

    public FileComponent(CamelContext context) { (2)
        super(context);
    }

    protected Endpoint createEndpoint(String uri, String remaining, Map parameters) throws
Exception { (3)
        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()** creates a **FileEndpoint** object.

9.6.2.9. 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:
    a. The consumer of the route writes any response back to the caller (if in **InOut** mode)
    b. The **UnitOfWork** is done by calling either **Synchronization.onComplete(org.apache.camel.Exchange)** or **Synchronization.onFailure(org.apache.camel.Exchange)**

9.7. ENDPOINT INTERFACE

This chapter describes how to implement the Endpoint interface, which is an essential step in the implementation of a Red Hat build of Apache Camel component.

9.7.1. The endpoint interface definition

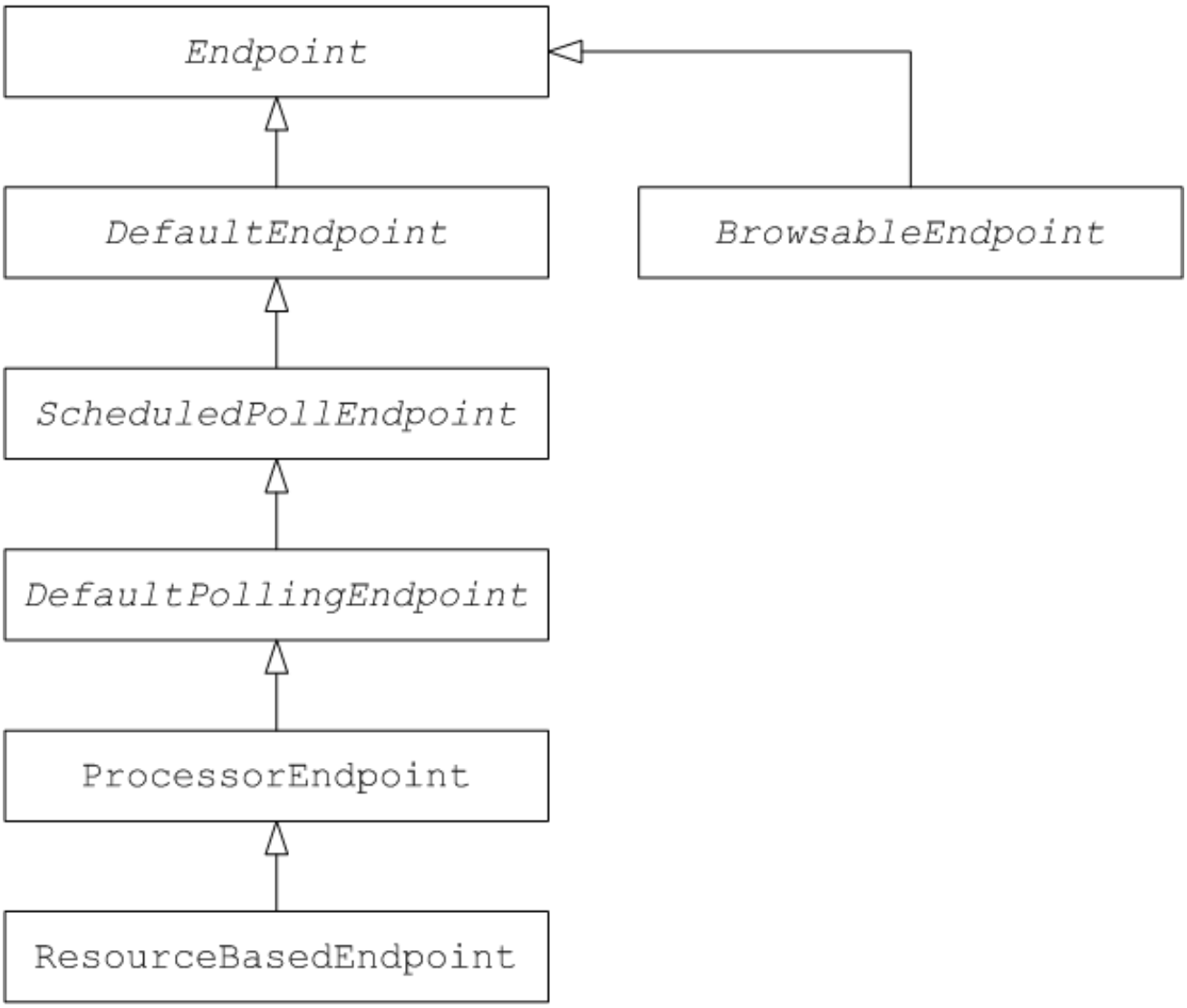
An instance of **org.apache.camel.Endpoint`type encapsulates an endpoint URI, and it also serves as a factory for `Consumer**, **Producer**, and **Exchange** objects. There are three different approaches to implementing an endpoint:

- Event-driven
- scheduled poll
- polling

These endpoint implementation patterns complement the corresponding patterns for implementing a consumer.

The following overview shows the relevant Java interfaces and classes that make up the **Endpoint** inheritance hierarchy.

Figure 9.12. Endpoint Inheritance Hierarchy



9.7.1.1. The endpoint interface

Example 9.20. 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;
}
```

9.7.1.2. 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.

9.7.1.3. 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.

9.7.2. Implementing the endpoint interface

9.7.2.1. Alternative ways of implementing an endpoint

The following alternative endpoint implementation patterns are supported:

- Section 9.7.2.2, "Event-driven endpoint implementation"
- Section 9.7.2.3, "Scheduled poll endpoint implementation"
- Section 9.7.2.4, "Polling endpoint implementation"

9.7.2.2. Event-driven endpoint implementation

If your custom endpoint conforms to the event-driven pattern, it is implemented by extending the abstract class, **org.apache.camel.impl.DefaultEndpoint**.

Example 9.21. 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 { 1

    public _CustomEndpoint_(String endpointUri, Component component) { 2
        super(endpointUri, component);
        // Do any other initialization...
    }
```

```
    public Producer createProducer() throws Exception { 3
        return new _CustomProducer_(this);
    }

    public Consumer createConsumer(Processor processor) throws Exception { 4
        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() { 5
        return this.createExchange(getExchangePattern());
    }

    public Exchange createExchange(ExchangePattern pattern) {
        Exchange result = new DefaultExchange(getCamelContext(), pattern);
        // Set exchange properties
        ...
        return result;
    }
}
```

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.
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 {CamelName} 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.

IMPORTANT

Do **not** override the **createPollingConsumer()** method.

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.
- **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.

9.7.2.3. Scheduled poll endpoint implementation

If your custom endpoint conforms to the scheduled poll pattern it is implemented by inheriting from the abstract class, **org.apache.camel.impl.ScheduledPollEndpoint**:

Example 9.22. ScheduledPollEndpoint 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.ScheduledPollEndpoint;

public class _CustomEndpoint_ extends ScheduledPollEndpoint { (1)

    protected _CustomEndpoint_(String endpointUri, _CustomComponent_ component) { (2)
        super(endpointUri, component);
        // Do any other initialization...
    }

    public Producer createProducer() throws Exception { (3)
        Producer result = new _CustomProducer_(this);
        return result;
    }

    public Consumer createConsumer(Processor processor) throws Exception { (4)
        Consumer result = new _CustomConsumer_(this, processor);
        configureConsumer(result); (5)
        return result;
    }

    public boolean isSingleton() {
        return true;
    }

    // Implement the following methods, only if you need to set exchange properties.
    //
    public Exchange createExchange() { (6)
        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 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.
5. The **configureConsumer()** method, defined in the **ScheduledPollEndpoint** base class, is responsible for injecting consumer query options into the consumer.
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 {CamelName} 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.

IMPORTANT

Do **not** override the **createPollingConsumer()** method.

9.7.2.4. Polling endpoint implementation

If your custom endpoint conforms to the polling consumer pattern it is implemented by inheriting from the abstract class, **org.apache.camel.impl.DefaultPollingEndpoint**:

Example 9.23. 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.

Apart from the implementation of the **createPollingConsumer()** method, the steps for implementing a **DefaultPollingEndpoint** are similar to the steps for implementing a **ScheduledPollEndpoint**.

9.7.2.5. 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 9.24, "BrowsableEndpoint Interface". It makes sense to implement this interface if the endpoint performs some sort of buffering of incoming events. For example, the Red Hat build of Apache Camel SEDA endpoint implements the BrowsableEndpoint interface – see Example 9.25, "SedaEndpoint Implementation".

Example 9.24. BrowsableEndpoint Interface

```
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();
}
```

9.7.2.6. Example

Example 9.25, "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 9.25. SedaEndpoint Implementation

```
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;

public class SedaEndpoint extends DefaultEndpoint implements BrowsableEndpoint { (1)
    private BlockingQueue<Exchange> queue;

    public SedaEndpoint(String endpointUri, Component component, BlockingQueue<Exchange>
queue) { (2)
        super(endpointUri, component);
        this.queue = queue;
    }

    public SedaEndpoint(String uri, SedaComponent component, Map parameters) { (3)
        this(uri, component, component.createQueue(uri, parameters));
    }

    public Producer createProducer() throws Exception { (4)
        return new CollectionProducer(this, getQueue());
    }

    public Consumer createConsumer(Processor processor) throws Exception { (5)
        return new SedaConsumer(this, processor);
    }

    public BlockingQueue<Exchange> getQueue() { (6)
        return queue;
    }

    public boolean isSingleton() { (7)
        return true;
    }

    public List<Exchange> getExchanges() { (8)
        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.

9.8. CONSUMER INTERFACE

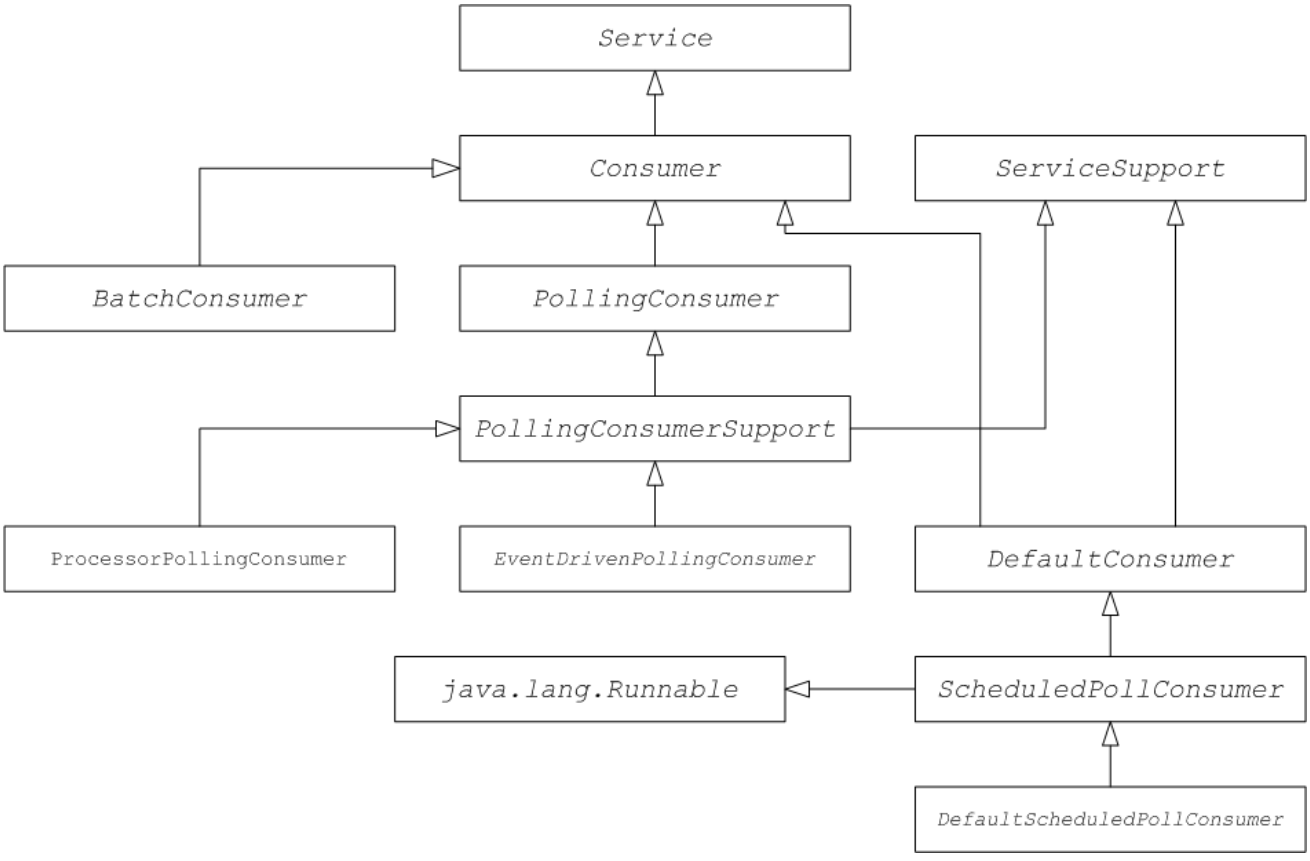
This chapter describes how to implement the Consumer interface, which is an essential step in the implementation of a Apache Camel component.

9.8.1. The Consumer Interface

9.8.1.1. 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, and this degree of flexibility is reflected in the inheritance hierarchy.

Figure 9.13. Consumer Inheritance Hierarchy



9.8.1.2. Consumer parameter injection

For consumers that follow the 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()**.

The following example shows a **createConsumer()** method implementation, taken from the **FileEndpoint** class in the file component:

Example 9.26. 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.

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.

9.8.1.3. Scheduled poll parameters

The following table lists consumer parameters supported by the scheduled poll pattern.

Table 9.1. 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.

9.8.1.4. 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 to 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).

9.8.1.5. 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 9.27, "ShutdownPrepared Interface" shows the definition of the **ShutdownPrepared** interface.

Example 9.27. 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:

a. **Graceful shutdown** – where the **forced** argument has the value **false**. Attempt to clean up

resources gracefully. For example, by stopping threads gracefully.

b. **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.

9.8.1.6. 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 exchanges in the queue before shutting down the SEDA consumer.

Example 9.28. ShutdownAware Interface definition

```
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 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 9.32, "Custom Threading Implementation".

9.8.2. Implementing the Consumer Interface

9.8.2.1. Alternative ways of implementing a consumer

You can implement a consumer in one of the following ways:

- Section 9.8.2.2, "Event-driven consumer implementation"
- Section 9.8.2.3, "Scheduled poll consumer implementation"
- Section 9.8.2.4, "Polling consumer implementation"
- Section 9.8.2.5, "Custom threading implementation"

9.8.2.2. 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 9.29, "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 9.29, "JMXConsumer Implementation".

Example 9.29. 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 { (1)

    JMXEndpoint jmxEndpoint;

    public JMXConsumer(JMXEndpoint endpoint, Processor processor) { (2)
        super(endpoint, processor);
        this.jmxEndpoint = endpoint;
    }

    public void handleNotification(Notification notification, Object handback) { (3)
        try {
            getProcessor().process(jmxEndpoint.createExchange(notification)); (4)
        } catch (Throwable e) {
            handleException(e); (5)
        }
    }
}
```

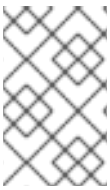
1. The **JMXConsumer** pattern follows the usual pattern for event-driven consumers by extending the **DefaultConsumer** class. Additionally, because this consumer is designed to receive events from JMX (which are represented by JMX notifications), it is necessary to implement the **NotificationListener** 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.

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.



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.

9.8.2.3. 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.

Example 9.30, "ScheduledPollConsumer Implementation" shows how to implement a consumer that follows the scheduled poll pattern, which is implemented by extending the **ScheduledPollConsumer** class.

Example 9.30. 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 { 1
    private final CustomEndpoint endpoint;
```

```
    public CustomConsumer(CustomEndpoint endpoint, Processor processor) { 2
        super(endpoint, processor);
        this.endpoint = endpoint;
    }

    protected void poll() throws Exception { 3
        Exchange exchange = /* Receive exchange object ... */;

        // Example of a synchronous processor.
        getProcessor().process(exchange); 4
    }

    @Override
    protected void doStart() throws Exception { 5
        // 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 { 6
        // Pre-Stop:
        // Place code here to execute just before processing stops.
        super.doStop();
        // Post-Stop:
        // Place code here to execute just after processing stops.
    }
}
```

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()**.

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.

9.8.2.4. Polling consumer implementation

Example 9.31, "PollingConsumerSupport Implementation" outlines how to implement a consumer that follows the polling pattern, which is implemented by extending the **PollingConsumerSupport** class.

Example 9.31. PollingConsumerSupport Implementation

```
import org.apache.camel.Exchange;
import org.apache.camel.RuntimeCamelException;
import org.apache.camel.impl.PollingConsumerSupport;

public class CustomConsumer extends PollingConsumerSupport { (1)
    private final CustomEndpoint endpoint;

    public CustomConsumer(CustomEndpoint endpoint) { (2)
        super(endpoint);
        this.endpoint = endpoint;
    }

    public Exchange receiveNoWait() { (3)
        Exchange exchange = /* Obtain an exchange object. */;
        // Further processing ...
        return exchange;
    }

    public Exchange receive() { (4)
        // Blocking poll ...
    }

    public Exchange receive(long timeout) { (5)
        // Poll with timeout ...
    }

    protected void doStart() throws Exception { (6)
        // Code to execute whilst starting up.
    }

    protected void doStop() throws Exception {
        // Code to execute whilst shutting down.
    }
}
```

(1) Implement your polling consumer class, *CustomConsumer*, by extending the **org.apache.camel.impl.PollingConsumerSupport** class.

(2) 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.

(3) The **receiveNoWait()** method should implement a non-blocking algorithm for retrieving an event (exchange object). If no event is available, it should return **null**.

(4) The **receive()** method should implement a blocking algorithm for retrieving an event. This method can block indefinitely, if events remain unavailable.

(5) The **receive(long timeout)** method implements an algorithm that can block for as long as the specified timeout (typically specified in units of milliseconds).

specified timeout (typically specified in units of milliseconds).

6 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.

9.8.2.5. 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.

For example, the SEDA component from **camel-core** implements its own consumer threading, which is consistent with the Apache Camel threading model. Example 9.32, "Custom Threading Implementation" shows an outline of how the **SedaConsumer** class implements its threading.

Example 9.32. Custom Threading Implementation

```
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 { 1
    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() { (2)
        BlockingQueue<Exchange> queue = endpoint.getQueue();
        // Poll the queue and process exchanges
        ...
    }

    ...
    protected void doStart() throws Exception { (3)
        int poolSize = endpoint.getConcurrentConsumers();
        executor = endpoint.getCamelContext().getExecutorServiceStrategy()
            .newFixedThreadPool(this, endpoint.getEndpointUri(), poolSize); (4)
        for (int i = 0; i < poolSize; i++) { (5)
            executor.execute(this);
        }
        endpoint.onStarted(this);
    }

    protected void doStop() throws Exception { (6)
        endpoint.onStopped(this);
        // must shutdown executor on stop to avoid overhead of having them running
        endpoint.getCamelContext().getExecutorServiceStrategy().shutdownNow(executor); (7)

        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.

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.

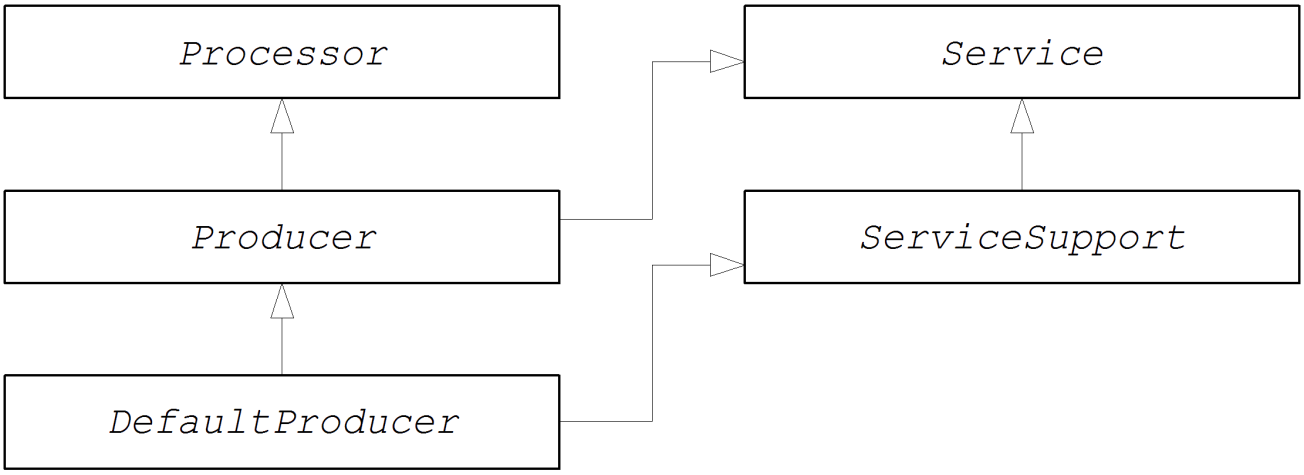
9.9. PRODUCER INTERFACE

This chapter describes how to implement the Producer interface, which is an essential step in the implementation of a Red Hat build of Apache Camel component.

9.9.1. The producer interface

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 9.14. Producer Inheritance Hierarchy



9.9.1.1. The producer interface

Example 9.33, "Producer Interface" shows the definition of the **org.apache.camel.Producer** interface.

Example 9.33. 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);
}
```

9.9.1.2. 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.

9.9.1.3. 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.

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 9.34, “AsyncProcessor Interface” .

Example 9.34. 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 9.35, “AsyncCallback Interface”.

Example 9.35. 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**.

9.9.1.4. 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 9.2.4, "The exchangehelper class".

9.9.2. Implementing the producer interface

9.9.2.1. Alternative ways of implementing a producer

You can implement a producer in one of the following ways:

- As a synchronous producer
- As an asynchronous producer

9.9.2.2. How to implement a synchronous producer

Example 9.36, "DefaultProducer Implementation" outlines how to implement a synchronous producer. In this case, call to **Producer.process()** blocks until a reply is received.

Example 9.36. DefaultProducer Implementation

```
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultProducer;

public class _CustomProducer_ extends DefaultProducer { 1

    public _CustomProducer_(Endpoint endpoint) { 2
        super(endpoint);
        // Perform other initialization tasks...
    }
```

```
    public void process(Exchange exchange) throws Exception { 3
        // 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)**.

9.9.2.3. How to implement an asynchronous producer

Example 9.37, "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 9.37. 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;

public class _CustomProducer_ extends DefaultProducer implements AsyncProcessor { 1

    public _CustomProducer_(Endpoint endpoint) { 2
        super(endpoint);
        // ...
    }

    public void process(Exchange exchange) throws Exception { 3
        // Process exchange synchronously.
```

```
        // ...
    }

    public boolean process(Exchange exchange, AsyncCallback callback) { 4
        // Process exchange asynchronously.
        CustomProducerTask task = new CustomProducerTask(exchange, callback);
        // Process 'task' in a separate thread...
        // ...
        return false; 5
    }
}

public class CustomProducerTask implements Runnable { 6
    private Exchange exchange;
    private AsyncCallback callback;

    public CustomProducerTask(Exchange exchange, AsyncCallback callback) {
        this.exchange = exchange;
        this.callback = callback;
    }

    public void run() { 7
        // Process exchange.
        // ...
        callback.done(false);
    }
}
```

1. Implement a custom asynchronous producer class, *CustomProducer*, by extending the **org.apache.camel.impl.DefaultProducer** class, and implementing the AsyncProcessor interface.

2. Implement a constructor that takes a reference to the parent endpoint.

3. Implement the synchronous **process()** method.

4. 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).

5. Normally, you return **false** from the asynchronous **process()** method, to indicate that the exchange was processed asynchronously.

6. The **CustomProducerTask** 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.

7. 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.

9.10. EXCHANGE INTERFACE

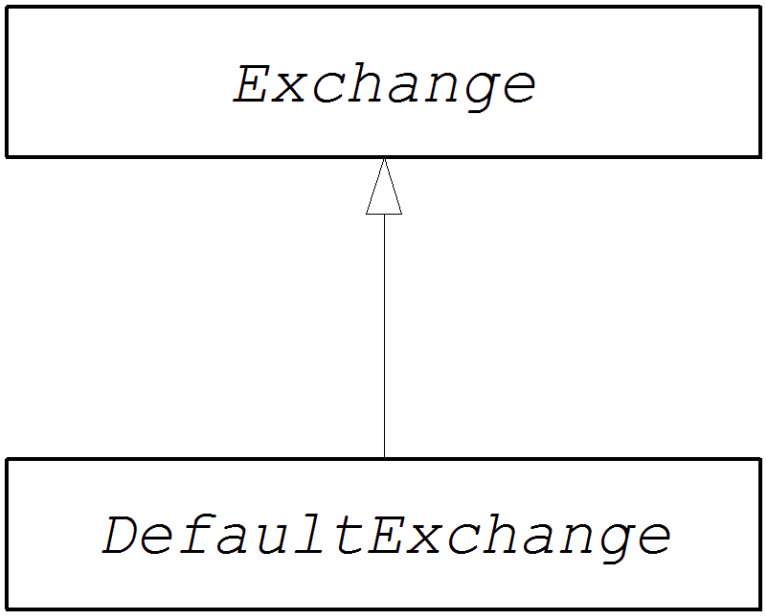
This chapter describes the Exchange interface. Since the refactoring of the camel-core module performed in {CamelName} 2.0, there is no longer any necessity to define custom exchange types. The **DefaultExchange** implementation can now be used in all cases.

9.10.1. The exchange interface

An instance of `org.apache.camel.Exchange` type encapsulates the current message passing through a route, with additional metadata encoded as exchange properties.

The following overview shoes the inheritance hierarchy for the exchange type. The default implementation, **DefaultExchange**, is always used.

Figure 9.15. Exchange Inheritance Hierarchy



9.10.1.1. The exchange interface definition

Example 9.38, "Exchange Interface" shows the definition of the org.apache.camel.Exchange interface.

Example 9.38. 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 getMessage();
    <T> T   getIn(Class<T> type);
    void    setIn(Message in);

    Message getMessage();
    <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);
}
```

9.10.1.2. 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**
- **InOut**

- **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()**, **getMessage()** – Setter and getter methods for the **In** message.
  The **getMessage()** implementation provided by the **DefaultExchange** class implements lazy creation semantics: if the *In* message is null when **getMessage()** is called, the **DefaultExchange** class creates a default **In** message.
- **setOut()**, **getMessage()**, **hasOut()** – Setter and getter methods for the **Out** message.
  The **getMessage()** 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 an 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 **OnCompletion** callback objects to the specified exchange object.

9.11. MESSAGE INTERFACE

This chapter describes how to implement the Message interface, which is an optional step in the implementation of a Red Hat build of Apache Camel component.

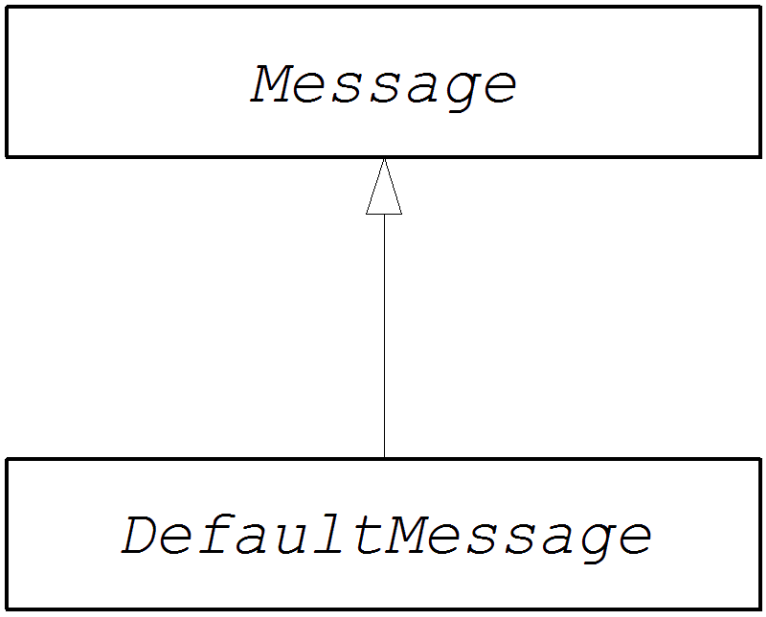
9.11.1. The message interface

An instance of **org.apache.camel.Message** type can represent any kind of message (**In** or **Out**).

You do not always need to implement a custom message type for a component. In many cases, the default implementation, **DefaultMessage**, is adequate.

The following overview shows the inheritance hierarchy for the message type.

Figure 9.16. Message Inheritance Hierarchy



9.11.1.1. The message interface definition

Example 9.39, "Message Interface" shows the definition of the **org.apache.camel.Message** interface.

Example 9.39. 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();
}
```

9.11.1.2. 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 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**.

9.11.2. Implementing the message interface

9.11.2.1. How to implement a custom message

The following exaxmple implements a message by extending the **DefaultMessage** class.

Example 9.40. Custom Message Implementation

```
import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultMessage;

public class _CustomMessage_ extends DefaultMessage { (1)

    public _CustomMessage_() { (2)
        // Create message with default properties...
    }

    @Override
    public String toString() { (3)
        // Return a stringified message...
    }

    @Override
    public _CustomMessage_ newInstance() { (4)
        return new _CustomMessage_( ... );
    }

    @Override
    protected Object createBody() { (5)
        // Return message body (lazy creation).
    }

    @Override
    protected void populateInitialHeaders(Map<String, Object> map) { (6)
        // Initialize headers from underlying message (lazy creation).
    }

    @Override
    protected void populateInitialAttachments(Map<String, DataHandler> map) { (7)
```

```
        // 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 **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()**.

---

[1] If there is no active method the returned value will be **null**.

[2] The timeout interval is typically specified in milliseconds.