MICRO FOCUS

Artix 5.6.3

Java Router, Defining Routes

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2015-03-06

Contents

Preface

Open Source Project Resources

Apache Incubator CXF

- **Website:** http://cxf.apache.org/
- **User's list:** <user@cxf.apache.org>

Apache Tomcat

- **Web site:** http://tomcat.apache.org/
- **User's list:** <users@tomcat.apache.org>

Apache ActiveMQ

- **Website:** http://activemq.apache.org/
- **User's list:** <users@activemq.apache.org>

Apache Camel

- **Web site:** http://camel.apache.org
- **User's list:** <users@camel.apache.org>

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`fixed width`	Fixed width (Courier New font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `javax.xml.ws.Endpoint` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `import java.util.logging.Logger;`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/YourUserName`
Italic	*Italic* words in normal text represent emphasis and introduce *new terms*.

Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the MS-DOS or Windows command prompt.
. . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in {} (braces).

The Artix ESB Documentation Library

For information on the organization of the Artix ESB library, the document conventions used, and where to find additional resources, see ***Using the Artix ESB Library***.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.

- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Defining Routes in Java DSL

You can define routing rules in Java, using a domain specific language (DSL). The routing rules represent the core of a router application and Java DSL is currently the most flexible way to define them.

Implementing a RouteBuilder Class

In Java Router, you define routes by implementing a `RouteBuilder` class. You must override a single method, `RouteBuilder.configure()`, and in this method define the routing rules you want to associate with the `RouteBuilder`. The rules themselves are defined using a *Domain Specific Language* (DSL), which is implemented as a Java API.

You can define as many `RouteBuilder` classes as you like in a router application. Ultimately, each `RouteBuilder` class must get instantiated once and registered with the `CamelContext` object. Normally, however, the lifecycle of the `RouteBuilder` objects is managed automatically by the container in which you deploy the router. The core task for a router developer is simply to implement one or more `RouteBuilder` classes.

RouteBuilder class

The `org.apache.camel.builder.RouteBuilder` class is the base class for implementing your own route builder types. It defines an abstract method, `configure()`, that you *must* override in your derived implementation class. In addition, `RouteBuilder` also defines methods that are used to initiate the routing rules (for example, `from()`, `intercept()`, and `onException()`).

Implementing a RouteBuilder

Example 1 shows an example of a simpler `RouteBuilder` implementation. You need only define a single method, `configure()`, which contains a list of routing rules (one Java statement for each rule).

Example 1. Implementation of a RouteBuilder Class

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

public class MyRouteBuilder extends RouteBuilder {

    public void configure() {
        // Define routing rules here:
        from("file:src/data?noop=true").to("file:target/mess
        ages");

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

Where the rule of the form `from(URL1).to(URL2)` instructs the router to read messages from the file system located in directory, `src/data`, and send them to files located in the directory, `target/messages`. The option, `?noop=true`, specifies that the source messages are *not* to be deleted from the `src/data` directory.

Basic Java DSL Syntax

What is a DSL?

A Domain Specific Language (DSL) is essentially a mini-language designed for a special purpose. The DSL is not required to be logically complete; it need only have enough expressive power to describe problems adequately in the chosen domain.

Typically, a DSL does *not* require a dedicated parser, interpreter, or compiler. You can piggyback a DSL on top of an existing object-oriented host language by observing that it is possible to map an API in a host language to a specialized language syntax: that is, a sequence of commands in the DSL maps to a chain of method invocations in the host language. For example, a sequence of commands in some hypothetical DSL that might look like this:

```
command0
1;
command0
```

Can be mapped to a chain of Java invocations, like this:

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

You could even define blocks, for example:

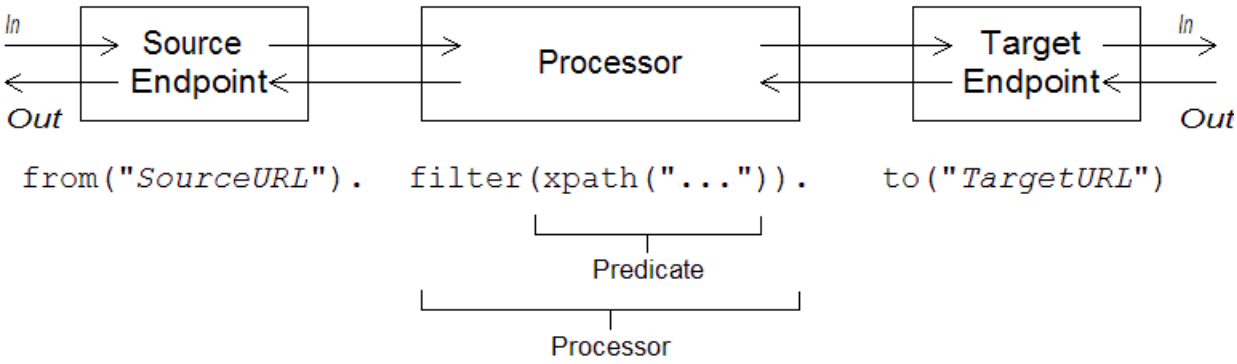
```
command01().startBlock().command02().command03().endBlock(
)
```

The syntax of the DSL is implicitly defined by the type system of the specialized API. For example, the return type of a method determines which methods can legally be invoked next (equivalent to the next command in the DSL).

Router rule syntax

The Java Router defines a *router DSL* for defining routing rules. You can use this DSL to define rules in the body of a `RouteBuilder.configure()` implementation. Figure 1 shows an overview of the basic syntax for defining local routing rules.

Figure 1. Local Routing Rules



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

NOTE: It is also possible to define a global routing rule, by starting the rule with a special processor type (such as `intercept()`, `onException()`, `errorHandler()`, and so on). This kind of rule lies outside the scope of the *Getting Started* guide..

Sources and targets

A local rule always starts by defining a source endpoint, using `from("`*`EndpointURL`*`")`, and typically (but not always) ends by defining a target 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`, a CXF endpoint, `cxf:MyServiceName`, or an ActiveMQ endpoint, `activemq:queue:MyQName`. For a complete list of component types, see http://camel.apache.org/components.html (`http://activemq.apache.org/camel/components.html`).

Processors

A *processor* is a method that can access and modify the stream of messages passing through a rule. If a message is part of a remote procedure call (*InOut* call), the processor can potentially act on the messages flowing in *both* directions: on

the request messages, flowing from source to target, and on the reply messages, flowing from target back to source (see Message exchanges). Processors can take *expression* or *predicate* arguments, that modify their behavior. For example, the rule shown in Figure 1 includes a `filter()` processor that takes an `xpath()` predicate as its argument.

Expressions and predicates

Expressions (evaluating to strings or other data types) and predicates (evaluating to true or false) occur frequently as arguments to the built-in processor types. You do not have to worry much about which type to pass to an expression argument, because they are usually automatically converted to the type you need. For example, you can usually just pass a string into an expression argument. Predicate expressions are useful for defining conditional behaviour in a route. For example, the following filter rule propagates *In* messages, only if the `foo` header is equal to the value `bar`:

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

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

Message exchanges

When a router rule is activated, it can process messages passing in either direction: that is, from source to target or from target back to source. For example, if a router rule is mediating a remote procedure call (RPC), the rule would process requests, replies, and faults. How do you manage message correlation in this case? One of the most effective and straightforward ways is to use a *message exchange* object as the basis for processing messages. Java Router uses message exchange objects (of `org.apache.camel.Exchange` type) in its API for processing router rules.

The basic idea of the message exchange is that, instead of accessing requests, replies, and faults separately, you encapsulate the correlated messages inside a single object (an `Exchange` object). Message correlation now becomes trivial from the perspective of a processor, because correlated messages are encapsulated in a single `Exchange` object and processors gain access to messages through the `Exchange` object.

Using an `Exchange` object makes it easy to generalize message processing to different kinds of *message exchange pattern*. For example, an asynchronous protocol might define a message exchange pattern that consists of a single message that flows

from the source to the target (an *In* message). An RPC protocol, on the other hand, might define a message exchange pattern that consists of a request message correlated with either a reply or fault message. Currently, Java Router supports the following message exchange patterns:

- `InOnly`
- `RobustInOnly`
- `InOut`
- `InOptionalOut`
- `OutOnly`
- `RobustOutOnly`
- `OutIn`
- `OutOptionalIn`

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

Processors

To enable the router to do something more interesting than simply connecting a source endpoint to a target endpoint, you can add *processors* to your route. A processor is a command you can insert into a routing rule in order to perform arbitrary processing of the messages that flow through the rule. Java Router provides a wide variety of different processors, as follows:

- Filter
- Choice
- Pipeline
- Recipient list
- Splitter
- Aggregator
- Resequencer
- Throttler
- Delayer
- Load balancer

- Custom processor

Filter

The `filter()` processor can be used to prevent uninteresting messages from reaching the target endpoint. It takes a single predicate argument: if the predicate is true, the message exchange is allowed through to the target; 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("Ta
```

Choice

The `choice()` processor is a conditional statement that is used to route incoming messages to alternative targets. The alternative targets are each 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("Target
1")
                        .when(Predicate2).to("Target2"
```

Pipeline

The `pipeline()` processor is used to link together a chain of targets, where the output of one target is fed into the input of the next target in the pipeline (analogous to the UNIX `pipe` command). The `pipeline()` method takes an arbitrary number of endpoint arguments, which specify the sequence of endpoints in the pipeline. For example, to pass messages from *`SourceURL`* to *`Target1`* to *`Target2`* to *`Target3`* in a pipeline, you could use the following rule:

```
from("SourceURL").pipeline("Target1","Target2","Target3");
```

Recipient list

If you want the messages from a source endpoint, SourceURL, to be sent to more than one target, there are two alternative approaches you can use. One approach is to invoke the to() method with multiple target endpoints (static recipient list), for example:

```
from("SourceURL").to("Target1","Target2","Target3");
```

The alternative approach is to invoke the `recipientList()` processor, which takes a list of recipients as its argument (dynamic recipient list). The advantage of the `recipientList`()

processor is that the list of recipients can be calculated at runtime.

For example, the following rule generates a recipient list by reading the contents of the `recipientListHeader` from the incoming message:

```
from("SourceURL").recipientList(header("recipientListHeader").
tokenize(","));
```

Splitter

The Splitter processor is used to split a message into parts, which are then processed as separate messages. The `split()` method takes a list argument, where each item in the list represents a message part that is to be re-sent as a separate message. For example, the following rule splits the body of an incoming message into separate lines and then sends each line to the target in a separate message:

```
from("SourceURL").split(bodyAs(String.class).tokenize("\n")).
to("TargetURL");
```

Aggregator

The Aggregateor processor is used to aggregate related incoming messages into a single message. In order to distinguish which messages are eligible to be aggregated together, you need to define a correlation key for the aggregator. The correlation key is normally derived from a field in the message (for example, a header field). Messages that have the same correlation key value are eligible to be aggregated together. You can also optionally specify an aggregation algorithm to the `aggregate()` processor (the default algorithm is to pick the latest message with a given value of the correlation key and to discard the older messages with that correlation key value).

For example, if you are monitoring a data stream that reports stock prices in real time, you might only be interested in the latest price of each stock symbol. In this case, you could configure an aggregator to transmit only the latest price for a given stock and discard the older (out-of-date) price notifications. The following rule implements this functionality, where the correlation key is read from the `stockSymbol` header and the default aggregator algorithm is used:

```
from("SourceURL").aggregate(header("stockSymbol")).to("TargetURL");
```

Resequencer

A Resequencer processor is used to re-arrange the order in which incoming messages are transmitted. The `resequence()` method takes a sequence number as its argument (where the sequence number is calculated from the contents of a field in the incoming message). Naturally, before you can start re-ordering messages, you need to wait until a certain number of messages have been received from the source. There are a

couple of different ways to specify how long the `resequence()` processor should wait before attempting to re-order the accumulated messages and forward them to the target, as follows:

Batch resequencing—(the default) wait until a specified number of messages have accumulated before starting to re-order and forward messages. This processing option is specified by invoking `resequence().batch()`. For example, the following resequencing rule would re-order messages based on the `timeOfDay` header, waiting until at least 300 messages have accumulated or 4000 ms have elapsed since the last message received.

```
from("SourceURL").resequence(header("timeOfDay").batch(new
 BatchResequencerConfig(300, 4000L)).to("TargetURL");
```

- *Stream resequencing*—transmit messages as soon as they arrive unless the resequencer detects a gap in the incoming message stream (missing sequence numbers), in which case the resequencer waits until the missing messages arrive and then forwards the messages in the correct order. To avoid the resequencer blocking forever, you can specify a timeout (default is 1000 ms), after which time the message sequence is transmitted with unresolved gaps. For example, the following resequencing rule detects gaps in the message stream by monitoring the value of the `sequenceNumber` header, where the maximum buffer size is limited to 5000 and the timeout is specified to be 4000 ms:

```
from("SourceURL").resequence(header("sequenceNumber")).stream
(new StreamResequencerConfig(5000, 4000L)).to("TargetURL");
```

Throttler

The Throttler processor is used to ensure that a target 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 target 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");
```

Delayer

The Delayer processor is used to hold up messages for a specified length of time. The delay can either be relative (wait a specified length of time after receipt of the incoming message) or absolute (wait until a specific time). For example, to add a delay of 2 seconds before transmitting received messages, you can use the following rule:

```
from("SourceURL").delay(2000).to("TargetURL");
```

To wait until the absolute time specified in the `processAfter` header, you can use the following rule:

```
from("SourceURL").delay(header("processAfter").to
("TargetURL");
```

The `delay()` method is overloaded, such that an integer is interpreted as a relative delay and an expression (for example, a string) is interpreted as an absolute delay.

Load Balancer

The Load Balancer processor is used to load balance message exchanges over a list of target endpoints. It is possible to customize the load balancing strategy. For example, to load balance incoming messages exchanges using a round robin algorithm (each endpoint in the target list is tried in sequence), you can use the following rule:

```
from("SourceURL").loadBalance().roundRobin().to
 ("TargetURL_01",  "TargetURL_02", "TargetURL_03");
```

Alternatively, you can customize the load balancing algorithm by implementing your own `LoadBalancer` class, as follows:

```
public class MyLoadBalancer implements
org.apache.camel.processor.loadbalancer.LoadBalancer {
    ...
};

from("SourceURL").loadBalance().setLoadBalancer(new MyLoadBalancer())
    .to("TargetURL_01", "TargetURL_02", "TargetURL_03");
```

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 override the `process()` method in this class. For example, the following custom processor, `MyProcessor`, removes the header named `foo` from incoming messages:

Example 2. Implementing a Custom Processor Class

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

To insert the custom processor into a router rule, invoke the `process()` method, which provides a generic mechanism for inserting processors into rules. For example, the following rule invokes the processor defined in Example 2:

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

Languages for Expressions and Predicates

To provide you with greater flexibility when parsing and processing messages, Java Router supports language plug-ins for various scripting languages. For example, if an incoming message is formatted as XML, it is relatively easy to extract the contents of particular XML elements or attributes from the message using a language such as XPath. The Java Router implements script builder classes, which encapsulate the imported languages. Each languages is accessed through a static method that takes a script expression as its argument, processes the current message using that script, and then returns an expression or a predicate. In order to be usable as an expression or a predicate, the script builder classes implement the following interfaces:

```
org.apache.camel.Expression<E>
org.apache.camel.Predicate<E>
```

In addition to this, the `ScriptBuilder` class (which wraps scripting languages such as JavaScript, and so on) inherits from the following interface:

```
org.apache.camel.Processor
```

Which implies that the languages associated with the `ScriptBuilder` class can also be used as message processors (see Custom processor).

Simple

The simple language is a very limited expression language that is built into the router core. This language can be useful, if you need to eliminate dependencies on third-party libraries whilst testing. Otherwise, you should use one of the other languages. To use the simple language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.language.simple.SimpleLanguage.simple;
```

The simple language provides various elementary expressions that return different parts of a message exchange. For example, the expression, `simple("header.timeOfDay")`, would return the contents of a header called `timeOfDay` from the incoming message. You can also construct predicates by testing expressions for equality. For example, the predicate, `simple("header.timeOfDay = '14:30'")`, tests whether the `timeOfDay` header in the incoming message is equal to `14:30`.

Table 1 shows the list of elementary expressions supported by the simple language.

Table 1. Properties for Simple Language

Elementary Expression	Description
`body`	Access the body of the incoming message.
`out.body`	Access the body of the outgoing message.
`Header.`*`HeaderName`*	Access the contents of the *HeaderName* header from the incoming message.
`out.header.`*`HeaderName`*	Access the contents of the *HeaderName* header from the outgoing message
`property.`*`PropertyName`*	Access the *PropertyName* property on the exchange.

Xpath

The `xpath()` static method parses message content using the XPath language (to learn about XPath, see the W3 Schools tutorial, http://www.w3schools.com/xpath/default.asp). To use the XPath language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.xml.XPathBuilder.xpath;
```

You can pass an XPath expression to `xpath()` as a string argument. The XPath expression implicitly acts on the message content and returns a node set as its result. Depending on the context, the return value is interpreted either as a predicate (where an empty node set is interpreted as false) or an expression. For example, if you are processing an XML message with the following content:

```
<person user="paddington">
  <firstName>Paddington</firstName>
  <lastName>Bear</lastName>
  <city>London</city>
</person>
```

You could choose which target endpoint to route the message to, based on the content of the `city` element, using the following rule:

```
from("file:src/data?noop=true").choice().
       when(xpath("/person/city =
'London'")).to("file:target/messages/uk").
       otherwise().to("file:target/messages/others");
```

Where the return value of `xpath()` is treated as a predicate in this example.

Xquery

The `xquery()` static method parses message content using the XQuery language (to learn about XQuery, see the W3 Schools tutorial, http://www.w3schools.com/xquery/default.asp). XQuery is a superset of the XPath language; hence, any valid XPath expression is also a valid XQuery expression. To use the XQuery language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.saxon.XQueryBuilder.xquery;
```

You can pass an XQuery expression to `xquery()` in several different ways. For simple expressions, you can pass the XQuery expressions as a string, `java.lang.String`. For longer XQuery expressions, on the other hand, you might prefer to store the expression in a file, which you can then reference by passing a `java.io.File` argument or a `java.net.URL` argument to the overloaded `xquery()` method. The XQuery expression implicitly acts on the message content and returns a node set as its result. Depending on the context, the return value is interpreted either as a predicate (where an empty node set is interpreted as false) or an expression.

JoSQL

The `sql()` static method enables you to call on the JoSQL (SQL for Java objects) language to evaluate predicates and expressions in Java Router. JoSQL employs a SQL-like query syntax to perform selection and ordering operations on data from in-memory Java objects—JoSQL is *not* a database, however. In the JoSQL syntax, each Java object instance is treated like a table row and each object method is treated like a column name. Using this syntax, it is possible to construct powerful statements for extracting and compiling data from collections of Java objects. For details, see http://josql.sourceforge.net/.

To use the JoSQL language in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.sql.SqlBuilder.sql;
```

OGNL

The `ognl()` static method enables you to call on OGNL (Object Graph Navigation Language) expressions, which can then be used as predicates and expressions in a router rule. For details, see http://www.ognl.org/.

To use the OGNL language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.language.ognl.OgnlExpression.ognl;
```

EL

The `el()` static method enables you to call on the Unified Expression Language (EL) to construct predicates and expressions in a router rule. The EL was originally specified as part of the JSP 2.1 standard (JSR-245), but is now available as a standalone language. Java Router integrates with JUEL (http://juel.sourceforge.net/), which is an open source implementation of the EL language.

To use the EL language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.language.juel.JuelExpression.el;
```

Groovy

The `groovy()` static method enables you to call on the Groovy scripting language to construct predicates and expressions in a route. To use the Groovy language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

JavaScript

The `javaScript()` static method enables you to call on the JavaScript scripting language to construct predicates and expressions in a route. To use the JavaScript language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

PHP

The `php()` static method enables you to call on the PHP scripting language to construct predicates and expressions in a route. To use the PHP language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

Python

The `python()` static method enables you to call on the Python scripting language to construct predicates and expressions in a route. To use the Python language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

Ruby

The `ruby()` static method enables you to call on the Ruby scripting language to construct predicates and expressions in a route. To use the Ruby language in your application code, include the following import statement in your Java source files:

```
import static
org.apache.camel.builder.camel.script.ScriptBuilder.*;
```

Bean

You can also use Java beans to evaluate predicates and expressions. For example, to evaluate the predicate on a filter using the `isGoldCustomer()` method on the bean instance, `myBean`, you can use a rule like the following:

```
from("SourceURL")
       .filter().method("myBean", "isGoldCustomer")
       .to("TargetURL");
```

A discussion of bean integration in Java Router is beyond the scope of this *Defining Routes* guide. For details, see http://activemq.apache.org/camel/bean-language.html.

Transforming Message Content

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

- Simple transformations
- Marshaling and unmarshaling

Simple transformations

The Java DSL has a built-in API that enables you to perform simple transformations on incoming and outgoing messages. For example, the rule shown in Example 3 would append the text, `World!`, to the end of the incoming message body.

Example 3. Simple Transformation of Incoming Messages

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

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

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

- `org.apache.camel.builder.ValueBuilder`

ProcessorDefinition class

The `org.apache.camel.model.ProcessorDefinition` class defines the DSL commands you can insert directly into a router rule—for example, the `setBody()` command in Example 3. Table 2 shows the `ProcessorDefinition` methods that are relevant to transforming message content:

Table 2. Transformation Methods from the ProcessorDefinition Class

Method	Description
`Type convertBodyTo(Class type)`	Converts the IN message body to the specified type.
`Type convertFaultBodyTo(Class type)`	Converts the FAULT message body to the specified type.
`Type convertOutBodyTo(Class type)`	Converts the OUT 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 removeOutHeader(String name)`	Adds a processor which removes the header on the OUT message.
`Type removeProperty(String name)`	Adds a processor which removes the exchange property.
`ExpressionClause<ProcessorType<Type>> setBody()`	Adds a processor which sets the body on the IN message.
`ExpressionClause<ProcessorType<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<ProcessorType<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<ProcessorType<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.

Method	Description
`ExpressionClause<ProcessorType<Type>> setProperty(String name)`	Adds a processor which sets the exchange property.
`Type setProperty(String name, Expression expression)`	Adds a processor which sets the exchange property.

Builder class

The `org.apache.camel.builder.Builder` class provides access to message content in contexts where expressions or predicates are expected. In other words, `Builder` methods are typically invoked in the *arguments* of DSL commands—for example, the `body()` command in Example 3. Table 3 summarizes the static methods available in the `Builder` class.

Table 3. Methods from the Builder Class

Method	Description
`static <E extends Exchange> ValueBuilder<E> body()`	Returns a predicate and value builder for the inbound body on an exchange.
`static <E extends Exchange,T> ValueBuilder<E> bodyAs(Class<T> type)`	Returns a predicate and value builder for the inbound message body as a specific type.
`static <E extends Exchange> ValueBuilder<E> constant(Object value)`	Returns a constant expression.
`static <E extends Exchange> ValueBuilder<E> faultBody()`	Returns a predicate and value builder for the fault body on an exchange.
`static <E extends Exchange,T> ValueBuilder<E> faultBodyAs(Class<T> type)`	Returns a predicate and value builder for the fault message body as a specific type.
`static <E extends Exchange> ValueBuilder<E> header(String name)`	Returns a predicate and value builder for headers on an exchange.
`static <E extends Exchange> ValueBuilder<E> outBody()`	Returns a predicate and value builder for the outbound body on an exchange.
`static <E extends Exchange> ValueBuilder<E> outBody()`	Returns a predicate and value builder for the outbound message body as a specific type.
`static <E extends Exchange> ValueBuilder<E> systemProperty(String name)`	Returns an expression for the given system property.

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

ValueBuilder class

The `org.apache.camel.builder.ValueBuilder` class enables you to modify values returned by the `Builder` methods. In other words, the methods in `ValueBuilder` provide a simple way of modifying message content.

Table 4 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 4. Modifier Methods from the ValueBuilder Class

Method	Description
`ValueBuilder<E> append(Object value)`	Appends the string evaluation of this expression with the given value.
`ValueBuilder<E> convertTo(Class type)`	Converts the current value to the given type using the registered type converters.
`ValueBuilder<E> convertToString()`	Converts the current value a String using the registered type converters.
`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<E> tokenize()`	
`ValueBuilder<E> tokenize(String token)`	Tokenizes the string conversion of this expression using the given token separator.

Marshaling and unmarshaling

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

- `marshal()`—convert a high-level data format to a low-level data format.
- `unmarshal()`—convert a low-level data format to a high-level data format.

Java Router supports marshaling and unmarshaling of the following data formats:

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

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

- *JAXB*—provides a mapping between XML schema types and Java types (see https://jaxb.dev.java.net/). For JAXB, unmarshaling converts an XML data type to a Java object and marshaling 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 in order to generate the Java classes that represent the XML data types in the schema. This is called *binding* the schema. After you have bound the schema, you can 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("Generated PackageName");

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.

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

```
from("SourceURL").unmarshal().xmlBeans()
```

- *XStream*—provides another mapping between XML types and Java types (see http://xstream.codehaus.org/). XStream is a serialization library (like Java serialization), enabling you to convert any Java object to XML. For XStream,

unmarshaling converts an XML data type to a Java object and marshaling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XStream, you can use code like the following:

```
from("SourceURL").unmarshal().xstream()
```

Defining Routes in XML

You can define routing rules in XML. This approach is not as flexible as Java DSL, but has the advantage that it is easy to reconfigure the routing rules at runtime.

Using the Router Schema in an XML File

The root element of the router schema is `camelContext`, which is defined in the XML namespace, `http://camel.apache.org/schema/spring`. Router configurations are typically embedded in other XML configuration files (for example, in a Spring configuration file). In general, whenever a router configuration is embedded in another configuration file, you need to specify the location of the router schema (so that the router configuration can be parsed). For example, Example 4 shows how to embed the router configuration, `camelContext`, in an arbitrary document, *`DocRootElement`*.

Example 4. Specifying the Router Schema Location

```
<DocRootElement ...
      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">

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

Where the schema location is specified to be `http://camel.apache.org/schema/spring/camel-spring.xsd`, which gives the location of the schema on the Apache Web site. This location always contains the latest, most up-to-date version of the XML schema.

Example 5 shows an example of embedding a router configuration, `camelContext`, in a Spring configuration file.

Example 5. Router Schema in a Spring Configuration File

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

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

Defining a Basic Route in XML

Basic concepts

In order to understand how to build a route using XML, you need to understand some of the basic concepts of the routing language—for example, sources and targets, processors, expressions and predicates, and message exchanges. For definitions and explanations of these concepts see Basic Java DSL Syntax.

Example of a basic route

Example 6 shows an example of a basic route in XML, which connects a source endpoint, *`SourceURL`*, directly to a destination endpoint, *`TargetURL`*.

Example 6. Basic Route in XML

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

Where *`CamelContextID`* is an arbitrary, unique identifier for the Camel context. The route is defined by a `route` element and there can be multiple `route` elements under the `camelContext` element.

Processors

To enable the router to something more interesting than simply connecting a source endpoint to a target endpoint, you can add *processors* to your route. A processor is a command you can insert into a routing rule in order to perform arbitrary processing of the messages that flow through the rule. Java Router provides a wide variety of different processors, as follows:

- Filter
- Choice
- Recipient list
- Splitter
- Aggregator
- Resequencer
- Throttler
- Delayer

Filter

The `filter` processor can be used to prevent uninteresting messages from reaching the target endpoint. It takes a single predicate argument: if the predicate is true, the message exchange is allowed through to the target; 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`:

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

Choice

The `choice` processor is a conditional statement that is used to route incoming messages to alternative targets. The alternative targets are each enclosed in a `when` element, which takes a predicate argument. If the predicate is true, the current target is selected, otherwise processing proceeds to the next `when` element in the rule. For example, the following `choice()` processor directs incoming messages to either *`Target1`*, *`Target2`*, or *`Target3`*, depending on the values of the predicates:

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

Recipient list

If you want the messages from a source endpoint, *SourceURL*, to be sent to more than one target, there are two alternative approaches you can use. One approach is to include multiple `to` elements in the route, for example:

```
<camelContext id="staticRecipientList"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <to uri="Target1"/>
    <to uri="Target2"/>
    <to uri="Target3"/>
  </route>
</camelContext>
```

The alternative approach is to add a `recipientList` element, which takes a list of recipients as its argument (dynamic recipient list). The advantage of using the `recipientList` element is that the list of recipients can be calculated at runtime. For example, the following rule generates a recipient list by reading the contents of the `recipientListHeader` from the incoming message:

```
<camelContext id="dynamicRecipientList"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <recipientList>
      <!-- Requires XPath 2.0 -->
      <xpath>tokenize(/headers/recipientListHeader,"\s+")</xpath>
    </recipientList>
  </route>
</camelContext>
```

Splitter

The `splitter` processor is used to split a message into parts, which are then processed as separate messages. The `splitter` element must contain an expression that returns a list, where each item in the list represents a message part that is to be re-sent as a separate message. For example, the following rule splits the body of an incoming message into separate sections (represented by a top-level `section` element) and then sends each section to the target in a separate message:

```
<camelContext id="splitterRoute"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:a"/>
    <splitter>
      <xpath>/section</xpath>
      <to uri="seda:b"/>
    </splitter>
  </route>
```

Aggregator

The `aggregator` processor is used to aggregate related incoming messages into a single message. In order to distinguish which messages are eligible to be aggregated together, you need to define a *correlation key* for the aggregator.

The correlation key is normally derived from a field in the message (for example, a header field). Messages that have the same correlation key value are eligible to be aggregated together. You can also optionally specify an aggregation algorithm to the `aggregator` processor (the default algorithm is to pick the latest message with a given value of the correlation key and to discard the older messages with that correlation key value).

For example, if you are monitoring a data stream that reports stock prices in real time, you might only be interested in the *latest* price of each stock symbol. In this case, you could configure an aggregator to transmit only the latest price for a given stock and discard the older (out-of-date) price notifications. The following rule implements this functionality, where the correlation key is read from the `stockSymbol` header and the default aggregator algorithm is used:

```
<camelContext id="aggregatorRoute"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <aggregator>
      <simple>header.stockSymbol</simple>
      <to uri="TargetURL"/>
    </aggregator>
  </route>
</camelContext>
```

Resequencer

A `resequencer` processor is used to re-arrange the order in which incoming messages are transmitted. The `resequencer` element needs to be provided with a sequence number (where the sequence number is calculated from the contents of a field in the incoming message). Naturally, before you can start re-ordering messages, you need to wait until a certain number of messages have been received from the source. There are a couple of different ways to specify how long the resequencer processor should wait before attempting to re-order the accumulated messages and forward them to the target, as follows:

- *Batch resequencing*—(the default) wait until a specified number of messages have accumulated before starting to re-order and forward messages. For example, the following resequencing rule would re-order messages based on the `timeOfDay` header, waiting until at least 300 messages have accumulated or 4000 ms have elapsed since the last message received.

```
<camelContext id="batchResequencer"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL" />
    <resequencer>
      <!-- Sequence ordering based on timeOfDay header -->
      <simple>header.timeOfDay</simple>
      <to uri="TargetURL" />
      <!--
        batch-config can be omitted for default (batch)
        resequencer settings
      -->
      <batch-config batchSize="300" batchTimeout="4000" />
    </resequencer>
  </route>
</camelContext>
```

- *Stream resequencing*—transmit messages as soon as they arrive *unless* the resequencer detects a gap in the incoming message stream (missing sequence numbers), in which case the resequencer waits until the missing messages arrive and then forwards the messages in the correct order. To avoid the resequencer blocking forever, you can specify a timeout (default is 1000 ms), after which time the message sequence is transmitted with unresolved gaps. For example, the following resequencing rule detects gaps in the message stream by monitoring the value of the `sequenceNumber` header, where the maximum buffer size is limited to 5000 and the timeout is specified to be 4000 ms:

```
<camelContext id="streamResequencer"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <resequencer>
      <simple>header.sequenceNumber</simple>
      <to uri="TargetURL" />
      <stream-config capacity="5000" timeout="4000"/>
    </resequencer>
  </route>
</camelContext>
```

Throttler

The `throttler` processor is used to ensure that a target 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 target endpoint. For example, to limit the rate of throughput to 100 messages per second, you can define the following rule:

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

Delayer

The `delayer` processor is used to hold up messages for a specified length of time. The delay can either be relative (wait a specified length of time after receipt of the incoming message) or absolute (wait until a specific time). For example, to add a delay of 2 seconds before transmitting received messages, you can use the following rule:

```
<camelContext id="delayerRelative"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <delayer>
           <delay>2000</delay>
           <to uri="TargetURL"/>
    </delayer>
  </route>
</camelContext>
```

To wait until the absolute time specified in the `processAfter` header, you can use the following rule:

```
<camelContext id="delayerRelative"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <delayer>
      <simple>header.processAfter</simple>
      <to uri="TargetURL"/>
    </delayer>
  </route>
</camelContext>
```

Load balancer

The `loadBalance` processor is used to load balance message exchanges over a list of target endpoints. For example, to load balance incoming messages exchanges using a round robin algorithm (each endpoint in the target list is tried in sequence), you can use the following rule:

```
<camelContext id="loadBalancer"
              xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <loadBalance>
      <to uri="TargetURL_01"/>
      <to uri="TargetURL_02"/>
      <roundRobin/>
    </loadBalance>
  </route>
</camelContext>
```

Currently, it is not possible to customize the load balancing algorithm in XML.

Languages for Expressions and Predicates

In the definition of a route, it is frequently necessary to evaluate expressions and predicates. For example, if a route includes a filter processor, you need to evaluate a predicate to determine whether or not a message is to be allowed through the filter. To facilitate the evaluation of expressions and predicates, Java Router supports multiple language plug-ins, which can be accessed through XML elements.

Elements for expressions and predicates

Table 5 lists the elements that you can insert whenever the context demands an expression or a predicate. The content of the element must be a script written in the relevant language. At runtime, the return value of the script is read by the parent element.

Table 5. Elements for Expression and Predicate Languages

Element	Language	Description
`simple`	N/A	A simple expression language, native to Java Router (see Simple).
`xpath`	XPath	The XPath language, which is used to select element, attribute, and text nodes from XML documents (see http://www.w3schools.com/xpath/default.asp). The XPath expression is applied to the current message.
`xquery`	XQuery	The XQuery language, which is an extension of XPath (see http://www.w3schools.com/xquery/default.asp). The XQuery expression is applied to the current message.
`sql`	JoSQL	The JoSQL language, which is a language for extracting and manipulating data from collections of Java objects, using a SQL-like syntax (see http://josql.sourceforge.net/).
`ognl`	OGNL	The OGNL (Object Graph Navigation Language) language (see http://www.ognl.org/).
`el`	EL	The Unified Expression Language (EL), originally developed as part of the JSP standard (see http://juel.sourceforge.net/).
`groovy`	Groovy	The Groovy scripting language (see http://groovy.codehaus.org/).
`javaScript`	JavaScript	The JavaScript scripting language (see http://developer.mozilla.org/en/docs/JavaScript), also known as ECMAScript (see http://www.ecmascript.org/).
`php`	PHP	The PHP scripting language (see http://www.php.net/).
`python`	Python	The Python scripting language (see http://www.python.org/).
`ruby`	Ruby	The Ruby scripting language (see http://www.ruby-lang.org/).
`bean`	Bean	Not really a language. The `bean` element is actually a mechanism for integrating with Java beans. You use the `bean` element to obtain an expression or predicate by invoking a method on a Java bean.

Transforming Message Content

This section describes how you can transform messages using the features provided in XML configuration.

Marshaling and unmarshaling

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

- `marshal`—convert a high-level data format to a low-level data format.
- `unmarshal`—convert a low-level data format to a high-level data format.

Java Router supports marshaling and unmarshaling of the following data formats:

- *Java serialization*—enables you to convert a Java object to a blob of binary data. For this data format, unmarshaling converts a binary blob to a Java object and marshaling converts a Java object to a binary blob. For example, to read a serialized Java object from an endpoint, *`SourceURL`*, and convert it to a Java object, you could use the following rule:

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

- *JAXB*—provides a mapping between XML schema types and Java types (see https://jaxb.dev.java.net/). For JAXB, unmarshaling converts an XML data type to a Java object and marshaling 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 in order to generate the Java classes that represent the XML data types in the schema. This is called *binding* the schema. After you have bound the schema, you can define a rule to unmarshal XML data to a Java object, as follows:

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

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

- *XMLBeans*—provides an alternative mapping between XML schema types and Java types (see http://xmlbeans.apache.org/). For XMLBeans, unmarshaling converts an XML data type to a Java object and marshaling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XMLBeans, define a rule like the following:

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

- *XStream*—You can set the XML encoding to XStream DataFormat either by setting the Exchange's property with the key Exchange.CHARSET_NAME, or by setting the encoding property on XStream from DSL or Spring configuration.

```
from("activemq:My.Queue").
  marshal().xstream("UTF-8").
  to("mqseries:Another.Queue");>
```

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

  <!- define the json xstream data formats to be used (xstream is default) ->
  <dataFormats>
        <xstream id="xstream-utf8" encoding="UTF-8"/>
        <xstream id="xstream-default"/>
  </dataFormats>

  <route>
```

```
        <from uri="direct:in"/>
        <marshal ref="xstream-default"/>
        <to uri="mock:result"/>
  </route>

  <route>
        <from uri="direct:in-UTF-8"/>
        <marshal ref="xstream-utf8"/>
        <to uri="mock:result"/>
  </route>

</camelContext>
```

See http://camel.apache.org/xstream.html for further details.

Basic Principles of Route Building

Java Router provides a great number of different processors and components, which you can link together to build a route. This chapter aims to give you some basic orientation by explaining the principles of building a route using the provided building blocks.

Pipeline Processing

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

Processor nodes

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

The first approach is where the processor simply modifies the exchange's *In* message, as shown in Figure 2. The exchange's *Out* message remains `null` in this case.

Figure 2. Processor Modifying an In Message

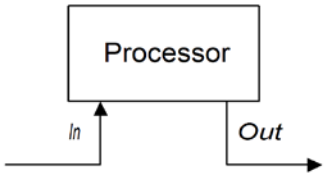
For example, the following route shows a `setHeader()` command that modifies the current *In* message by adding (or modifying) the `BillingSystem` heading:

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

    .to("activemq:billingQueue");
```

The second approach is where the processor creates an *Out* message to represent the result of the processing, as shown in Figure 3.

Figure 3. Processor Creating an Out Message



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

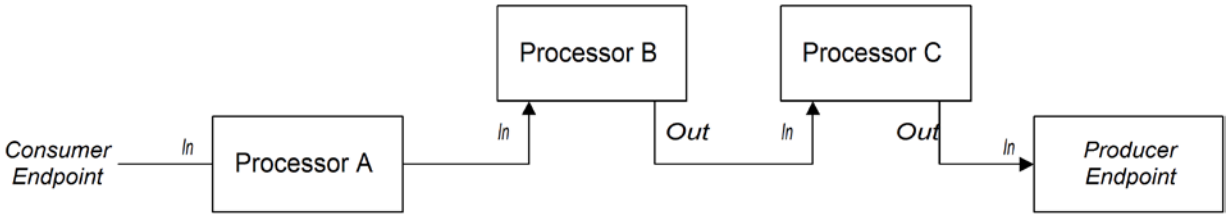
```
from("activemq:orderQueue")
    .transform(constant("DummyBody"))
    .to("activemq:billingQueue");
```

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

Pipeline for InOnly exchanges

Figure 4 shows an example of a processor pipeline for *InOnly* exchanges. Processor A acts by modifying the *In* message, while processors B and C create an *Out* message. The route builder links the processors together as shown. In particular, processors B and C are linked together in the form of a *pipeline*: that is, processor B's *Out* message is moved to the *In* message before feeding the exchange into processor C, and processor C's *Out* message is moved to the *In* message before feeding the exchange into the producer endpoint. Thus the processors' outputs and inputs are joined into a continuous pipeline, as shown in Figure 4.

Figure 4. Sample Pipeline for InOnly Exchanges



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

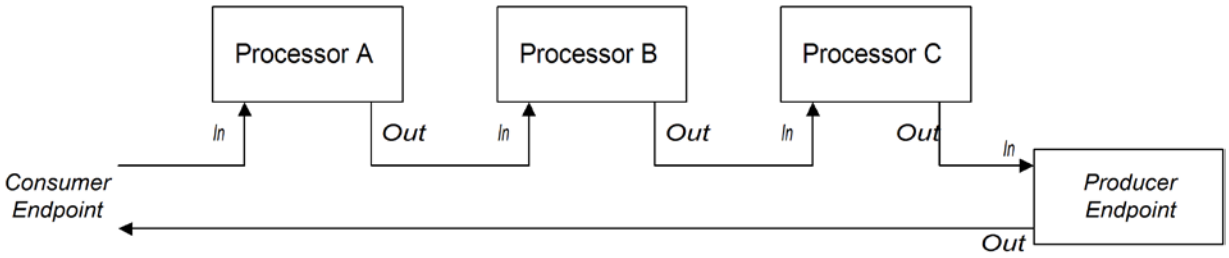
```
from("activemq:userdataQueue")
    .to("velocity:file:AdressTemplate.vm")
    .to("activemq:envelopeAddresses");
```

Where the Velocity endpoint, `velocity:file:AdressTemplate.vm`, specifies the location of a Velocity template file, `file:AdressTemplate.vm`, in the file system.

Pipeline for InOut exchanges

Figure 5 shows an example of a processor pipeline for *InOut* exchanges, which you typically use to support remote procedure call (RPC) semantics. Processors A, B, and C are linked together in the form of a pipeline, with the output of each processor being fed into the input of the next. The final *Out* message produced by the producer endpoint is sent all the way back to the consumer endpoint, where it provides the reply to the original request.

Figure 5. Sample Pipeline for InOut Exchanges



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

For example, consider the following route that processes payment requests, by processing incoming HTTP requests:

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

Where the incoming payment request is processed by passing it through a pipeline of Web services,

`cxf:bean:addAccountDetails`, `cxf:bean:getCreditRating`, and `cxf:bean:processTransaction`. The final Web service, `processTransaction`, generates a response (*Out* message) that is sent back through the JETTY endpoint.

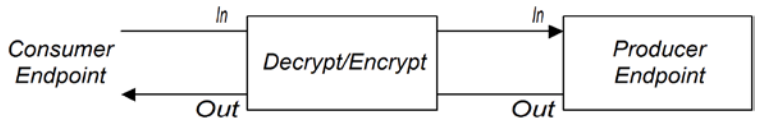
When the pipeline consists of just a sequence of endpoints, it is also possible to use the following alternative syntax:

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

Comparison of pipelining and interceptor chaining

An alternative paradigm for linking together the nodes of a route is *interceptor chaining*, where a processor in the route processes the exchange both before *and after* dispatching the exchange to the next processor in the chain. This style of processing is also supported by Java Router, but it is not the usual approach to use. Figure 6 shows an example of an interceptor processor that implements a custom encryption algorithm.

Figure 6. Example of Interceptor Chaining



In this example, incoming messages are encrypted in a custom format. The interceptor first decrypts the *In* message, then dispatches it to the Web services endpoint, `cxf:bean:processTxn`, and finally, the reply (*Out* message) is encrypted using the custom format, before being sent back through the consumer endpoint. Using the interceptor chaining approach, therefore, a single interceptor instance can modify both the request *and* the response.

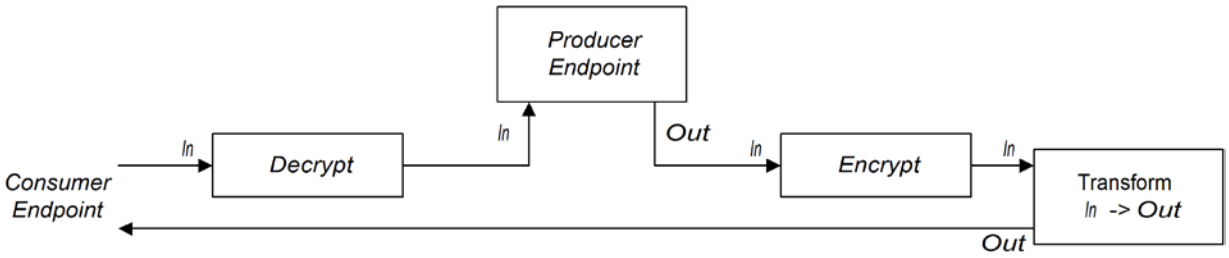
For example, if you want to define a route with a HTTP port that services incoming requests encoded using custom encryption, you could define a route like the following:

```
from("jetty:http://localhost:8080/foo")
    .intercept(new MyDecryptEncryptInterceptor())
    .to("cxf:bean:processTxn");
```

Where the class, MyDecryptEncryptInterceptor, is implemented by inheriting from the class, org.apache.camel.processor.DelegateProcessor.

Although it is possible to implement this kind of functionality using an interceptor processor, this is not a very idiomatic way of programming in Java Router. A more typical approach is shown in Figure 7.

Figure 7. Pipeline Alternative to Interceptor Chaining



In this example, the encrypt functionality is implemented in a separate processor from the decrypt functionality. The resulting processor pipeline is semantically equivalent to the original interceptor chain shown in Figure 6. One slight complication of this route, however, is that it turns out to be necessary to add a `transform` processor at the end of the route, in order to copy the *In* message to the *Out* message. This processor ensures that the reply message is available to the HTTP consumer endpoint (an alternative solution to this problem would be to implement the encrypt processor such that it creates an *Out* message directly).

For example, to implement the pipeline approach shown in Figure 7, you could define a route like the following:

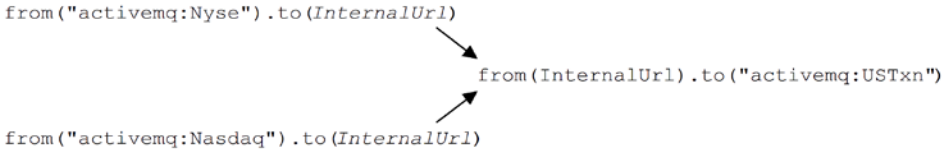
```
from("jetty:http://localhost:8080/foo")
    .process(new MyDecryptProcessor())
    .to("cxf:bean:processTxn")
    .process(new MyEncryptProcessor())
    .transform(body());
```

Where the final processor node, `transform(body())`, has the effect of copying the *In* message to the *Out* message (the *In* message body is copied explicitly and the *In* message headers are copied implicitly).

Multiple Inputs

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? For example, you might want to merge incoming messages from two different messaging systems and process them using the same route. In most cases, you can deal with multiple inputs by dividing your route into segments, as shown in Figure 8.

Figure 8. 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 a kind of endpoint that is intended only for use *within* a router application. The following types of endpoint are suitable for internal use:

- Direct endpoints
- SEDA endpoints
- VM endpoints

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

NOTE: The direct, SEDA, and VM components work only for the *InOnly* exchange pattern. If one of your inputs requires an *InOut* exchange pattern, you should take a look at the Content enricher pattern instead.

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, say 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 could 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 into the `activemq:USTxn` queue.

The implementation of the direct endpoint is very simple: whenever an exchange arrives at a producer endpoint (for example, `to("direct:mergeTxns")`), the direct endpoint passes

the exchange directly to all of the consumers endpoints that have the same endpoint ID (for example, `from("direct:mergeTxns")`). Direct endpoints can only be used to communicate between routes that belong to the same `CamelContext` in the same Java virtual machine (JVM) instance.

NOTE: If you connect multiple consumers to a direct endpoint, every exchange that passes through the endpoint will be processed by *all* of the attached consumers (multicast). Hence, each exchange would be processed *more than once*. If you don't want this to happen, consider using a SEDA endpoint, which behaves differently.

SEDA endpoints

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

- Processing of a SEDA endpoint is *not* synchronous. That is, when you send an exchange to a SEDA producer endpoint, control immediately returns to the preceding processor in the route.
- SEDA endpoints contain a queue buffer (of `java.util.concurrent.BlockingQueue` type), which stores all of the incoming exchanges prior to processing by the next route segment.
- Each SEDA consumer endpoint creates a thread pool (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. For example, 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 Java Router is simply to enable you to apply this design philosophy to your applications. For more details about SEDA, see http://www.eecs.harvard.edu/~mdw/ proj/seda/.

VM endpoints

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

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

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

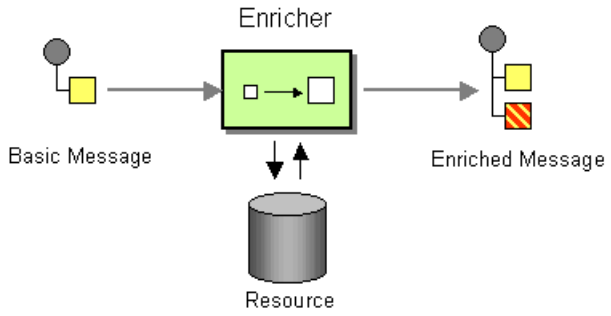
And in a separate router application (running in the same Java VM), you can define the second segment of the route:

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

Content enricher pattern

The content enricher pattern defines a fundamentally different way of dealing with multiple inputs to a route. A content enricher is a kind of processor that you can insert into a route, as shown in Figure 9. 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.

Figure 9. Processing Multiple Inputs with a Content Enricher



The key difference between the content enricher approach and the segmented route approach is that the content enricher is based on a radically different event model. In the segmented route

approach, each of the input sources independently generate new message events. Whereas, in the content enricher approach, only one input source generates new message events, while a second resource (accessed by the enricher) is effectively a *slave* of the first stream of events. That is, the enricher's resource is only polled for input whenever a message arrives on the main route.

Exception Handling

Two exception handling methods are provided for catching exceptions in Java DSL routes, as follows:

- `errorHandler()` clause—is a simple catch-all mechanism that re-routes the current exchange to a dead letter channel, if an error occurs. This mechanism is *not* able to discriminate between different exception types.
- `onException()` clause—enables you to discriminate between different exception types, performing different kinds of processing for different exception types.

Dead letter channel pattern

This section provides a brief introduction to exception handling in Java Router.

errorHandler clause

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 could define a `RouteBuilder` as follows:

```
public class MyRouteBuilder extends RouteBuilder {

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

    // The preceding error handler applies to all of the
    // following routes:
       from("activemq:orderQueue").to("pop3://fulfillment@acme.com");
       from("file:src/data?noop=true").to("file:target/messages");
       // ...
    }
}
```

Redirection to the dead letter channel does not occur, however, until attempts at redelivery have been exhausted (see Redelivery policy).

onException clause

The `onException(Class exceptionType)` 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 the specified type, `exceptionType`, occurs in one of the applicable routes. For example, you can define `onException` clauses for catching `NullPointerException`, `IOException`, and `Exception` exceptions as follows:

```
public class MyRouteBuilder extends RouteBuilder {

    public void configure() {
       onException(NullPointerException.class).to("activemq:ex.npex");
      onException(IOException.class).to("activemq:ex.ioex");
      onException(Exception.class).to("activemq:ex");

  // The preceding onException() clauses apply to all of the following routes:
       from("activemq:orderQueue").to("pop3://fulfillment@acme.com");
       from("file:src/data?noop=true").to("file:target/messages");
       // ...
    }
}
```

When an exception occurs, Java Router selects the `onException` clause that matches the given exception type most closely. If no other clause matches the raised exception, the `onException(Exception.class)` clause (if present) matches by default, because `java.lang.Exception` is the base class of all Java exceptions. The applicable `onException` clause does not initiate processing, however, until attempts at redelivery have been exhausted (see Redelivery policy).

Redelivery policy

Both the `errorHandler` clause and the `onException` clause support a *redelivery policy* that specifies how often Java Router attempts to redeliver the failed exchange before giving up and triggering the actions defined by the relevant error handler. The most important redelivery policy setting is the *maximum redeliveries* value, which specifies how many times redelivery is attempted. The default value is 6.

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 *bean binding*. Bean binding can use either or both of the following approaches in order to initialize a method's parameters:

- *Conventional method signatures*—if the method signature conforms to certain conventions, the bean binding can use Java reflection to determine what parameters to pass.

- *Annotations and dependency injection*—for a more flexible binding mechanism, employ Java annotations to specify what to inject into the method's arguments. This dependency injection mechanism relies on Spring component scanning. Normally, if you are deploying your Java Router application into a Spring container, the dependency injection mechanism will work automatically.

Accessing a bean created in Java

To process exchange objects using a Java bean (which is just 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. If you want to access the same `MyBeanProcessor` instance from multiple routes, however, use the variant of `bean()` that takes the `Object` type as its first argument, as follows:

```
MyBeanProcessor myBean = new MyBeanProcessor();

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

Basic method signatures

In order 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.

Method signature for processing message bodies

If you want to implement a bean method that access or modifies the incoming message body, define a method signature that takes a single `String` argument and returns a `String` value.

For example:

```
// Java
package com.acme;

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

Method signature for processing exchanges

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

```
// Java
package com.acme;

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

Accessing a bean created in Spring XML

Instead of creating a bean instance in Java, you can create an instance using Spring XML. This is the only feasible approach, in fact, if you are defining your routes in XML. To define a bean in XML, use the standard Spring `bean` element. For example, to create an instance of `MyBeanProcessor`:

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

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

For full details of how to use the Spring `bean` element, see The IoC Container from the Spring reference guide (`http://docs.spring.io/spring/docs/3.2.6.RELEASE/spring-framework-reference/html/beans.html`).

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

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

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

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

Bean binding annotations

The basic bean bindings described in Basic method signatures might not always be convenient to use. For example, if you have a legacy Java class that performs some data manipulation, you might want to extract data from an inbound exchange and map it to the arguments of an existing method signature. For this kind of bean binding, Java Router provides the following kinds of Java annotation:

- Basic annotations
- Expression language annotations

Basic annotations

Table 6 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 6. 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`	Binds to a `java.util.Map` of the inbound message headers.	
`@OutHeaders`	Binds to a `java.util.Map` of the outbound message headers.	
`@Property`	Binds to a named exchange property.	String name of the property.
`@Properties`	Binds to a `java.util.Map` of the exchange properties.	

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

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

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

    }
}
```

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

Expression language annotations

The expression language annotations provide a powerful mechanism for injecting message data into a bean method's arguments. Using these annotations, you can invoke an arbitrary script, written in the scripting language of your choice, in order to extract data from an inbound exchange and inject this data into a method argument. Table 7 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 7. Expression Language Annotations

Annotation	Description
`@Bean`	Inject a Bean expression.
`@BeanShell`	Inject a BeanShell expression.
`@Constant`	Inject a Constant expression
`@EL`	Inject an EL expression.
`@Groovy`	Inject a Groovy expression.
`@Header`	Inject a Header expression.
`@JavaScript`	Inject a JavaScript expression.
`@OGNL`	Inject an OGNL expression.
`@PHP`	Inject a PHP expression.
`@Python`	Inject a Python expression.

Annotation	Description
`@Ruby`	Inject a Ruby expression.
`@Simple`	Inject a Simple expression.
`@XPath`	Inject an XPath expression.
`@XQuery`	Inject an XQuery expression.

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

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

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

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

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

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

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

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

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

```
// Java
package com.acme;

public class MyIdGenerator {

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

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

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