Progress FUSE

# FUSE™ Mediation Router

## Defining Routes

Version 1.6
April 2009

PROGRESS SOFTWARE

# Defining Routes

Version 1.6

Publication date 17 Jul 2009


***Legal Notices***

# Table of Contents

# List of Figures

# List of Tables

# List of Examples

# Chapter 1. Defining Routes in Java DSL

*The Java Domain Specific Language (DSL) leverages the code-completion feature of your IDE.*

# Implementing a RouteBuilder Class

**Overview**

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

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

**RouteBuilder class**

As a router developer, your core task is to implement one or more `RouteBuilder` classes. To do this, you extend the `org.apache.camel.builder.RouteBuilder` base class and override its abstract method, `configure()`.

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

**Implementing a RouteBuilder**

Example 1.1 on page 12 shows a minimal `RouteBuilder` implementation. The `configure()` method body contains a routing rule; each rule is a single Java statement.

***Example 1.1. Implementation of a RouteBuilder Class***

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

public class MyRouteBuilder extends RouteBuilder {

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


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

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

# Basic Java DSL Syntax

**What is a DSL?**

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

Consider the following sequence of commands in a hypothetical DSL:

```
command01;
command02;
command03;
```

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

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

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

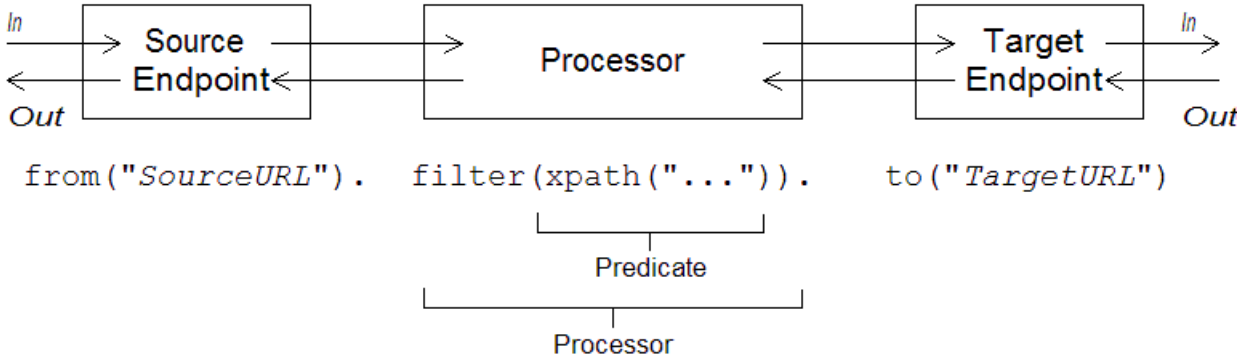
```
command01().startBlock().command02().command03().endBlock()
```

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

**Router rule syntax**

FUSE Mediation 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 1.1 on page 14 shows an overview of the basic syntax for defining local routing rules.

***Figure 1.1. Local Routing Rules***




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

## Note

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

**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 can use a file endpoint, `file:MyMessageDirectory`, a Apache CXF endpoint, `cxf:MyServiceName`, or an Apache ActiveMQ endpoint, `activemq:queue:MyQName`. For a complete list of component types, see http://camel.apache.org/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. It can act on request messages, flowing from source to target; and it can act on reply messages, flowing from target back to source (see "Message exchanges" on page 15). Processors can take *expression* or *predicate* arguments, that modify their behavior. For example, the rule shown in Figure 1.1 on page 14 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 be concerned about which type to pass to an expression argument, because arguments are usually automatically converted to the required type. For example, you can usually just pass a string into an expression argument. Predicate expressions are useful for defining conditional behavior 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(head
er("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" on page 22).

**Message exchanges**

When a router rule is activated, it can process messages passing in either direction: 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 processes 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. FUSE Mediation 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 *message exchange patterns*. 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 a fault message. Currently, FUSE Mediation 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

**Overview**

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 to perform arbitrary processing of messages that flow through the rule. FUSE Mediation Router provides a wide variety of different processors, as follows:



---

**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("TargetURL");
```

---

**Choice**

The `choice()` processor is a conditional statement that is used to route incoming messages to alternative targets. Each alternative target is preceded by a `when()` method, which takes a predicate argument. If the predicate is

true, the following target is selected, otherwise processing proceeds to the next `when()` method in the rule. For example, the following `choice()` processor directs incoming messages to either *`Target1`*, *`Target2`*, or *`Target3`*, depending on the values of *`Predicate1`* and *`Predicate2`*:

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

---

**Pipeline**

The `pipeline()` processor links a chain of targets together, 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 can 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, you can use two alternative approaches. The first is to invoke the `to()` method with multiple target endpoints (static recipient list), for example:

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

The second is to invoke the `recipientList()` processor, which takes as its argument a list of recipients (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("recipientListHead
er").tokenize(","));
```

---

**Splitter**

The `splitter()` processor splits a message into parts, which are then processed as separate messages. The `splitter()` method takes a list argument, where each item in the list represents a message part that is 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").splitter(bodyAs(String.class).token
ize("\n")).to("TargetURL");
```

### Aggregator

The `aggregator()` processor aggregates related incoming messages into a single message. To distinguish which messages are eligible to be aggregated together, you define a *correlation key* for the aggregator. This key is normally derived from a field in the messag, such as 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 can 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").aggregator(header("stockSymbol")).to("Tar
getURL");
```

### Resequencer

A `resequencer()` processor re-orders incoming messages and forwards them to the target. The `resequencer()` method takes a sequence number as its argument; this number is calculated from the contents of a field in the incoming message. The `resequencer()` processor waits for a specified amount of time to accumulate messages before reordering and forwarding them. You can specify the wait time in two ways:

- *Batch resequencing* — Wait until a specified number of messages have accumulated before reordering and forwarding them. This is the default processing option. You specify this option by invoking `resequencer().batch()`. For example, the following resequencing rule reorders messages based on the `timeOfDay` header, waiting until at least 300 messages have accumulated or until 4000 ms have elapsed since the last message was received:

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

- *Stream resequencing* — Transmits messages as soon as they arrive, unless the resequencer detects a gap in the incoming message stream (missing sequence numbers). If a gap occurs, the resequencer waits until the missing messages arrive and then forwards them in the correct order. To avoid the resequencer blocking forever, you can specify a timeout (the 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").resequencer(header("sequenceNum
ber")).stream(new StreamResequencerConfig(5000,
4000L)).to("TargetURL");
```

---

**Throttler**

The `throttler()` processor ensures 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").throttler(100).to("TargetURL");
```

---

**Delayer**

The `delayer()` processor delays 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").delayer(2000).to("TargetURL");
```

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

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

The `delayer()` 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.

### Custom processor

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

***Example 1.2. 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 1.2 on page 21:

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

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

# Languages for Expressions and Predicates

**Overview**

To provide greater flexibility when parsing and processing messages, FUSE Mediation 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 FUSE Mediation Router implements script builder classes, which encapsulate the imported languages. Each language 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. 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) inherits from the following interface:

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

This implies that the languages associated with the `ScriptBuilder` class can also be used as message processors (see "Custom processor" on page 21).

---

**Simple**

The simple language is a very limited expression language that is built into the router core. This language is useful when you need to eliminate dependencies on third-party libraries during 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.SimpleLan
guage.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 1.1 on page 23 shows the list of elementary expressions supported by the simple language.

*Table 1.1. Properties for Simple Language*

| Elementary Expression | Description |
|---|---|
| `body` | Accesses the body of the incoming message. |
| `out.body` | Accesses the body of the outgoing message. |
| `header.` *`HeaderName`* | Accesses the contents of the *`HeaderName`* header from the incoming message. |
| `out.header.` *`HeaderName`* | Accesses the contents of the *`HeaderName`* header from the outgoing message. |
| `property.` *`PropertyName`* | Accesses 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 at 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 as 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>
```

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

```
from("file:src/data?noop=true").
choice().
  when(xpath("/person/city = 'London'")).to("file:target/mes
sages/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.XQueryBuild
er.xquery;
```

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

**JoSQL**

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

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.OgnlExpres
sion.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 it is now available as a standalone language. FUSE Mediation 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.Script
Builder.*;
```

---

**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.Script
Builder.*;
```

---

**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.Script
Builder.*;
```

**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.Script
Builder.*;
```

**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.Script
Builder.*;
```

**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 FUSE Mediation Router is beyond the scope of this *Defining Routes* guide. For details, see http://activemq.apache.org/camel/bean-language.html.

# Transforming Message Content

**Overview**

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

- "Simple transformations" on page 27.
- "Marshalling and unmarshalling" on page 31.
- "Artix Data Services" on page 32.

---

**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 1.3 on page 27 appends the text, `World!`, to the end of the incoming message body.

***Example 1.3. 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.ProcessorType`
- `org.apache.camel.builder.Builder`
- `org.apache.camel.builder.ValueBuilder`

---

**ProcessorType class**

The `org.apache.camel.model.ProcessorType` class defines the DSL commands you can insert directly into a router rule—for example, the `setBody()` command in Example 1.3 on page 27. Table 1.2 on page 28 shows the `ProcessorType` methods that are relevant to transforming message content:

***Table 1.2. Transformation Methods from the ProcessorType 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. |
| `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>> setOutBody()` | Adds a processor which sets the body on the OUT message. |
| `Type setOutBody(Expression expression)` | Adds a processor which sets the body on the OUT 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 1.3 on page 27. Table 1.3 on page 29 summarizes the static methods available in the `Builder` class.

***Table 1.3. 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. |

| Method | Description |
|---|---|
| `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. |
| `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 1.4 on page 30 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 1.4. 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()` | |

| Method | Description |
| --- | --- |
| `ValueBuilder<E> tokenize(String token)` | Tokenizes the string conversion of this expression using the given token separator. |

### Marshalling and unmarshalling

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

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

FUSE Mediation Router supports marshalling and unmarshalling of the following data formats:

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

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

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

```
org.apache.camel.spi.DataFormat jaxb = new
org.apache.camel.model.dataformat.JaxbDataFormat("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, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XMLBeans, you use code like the following:

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

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

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

- *Artix Data Services* — FUSE Mediation Router also integrates with Artix Data Services, enabling you to integrate stub code generated by Artix Data Services. See "Artix Data Services" on page 32 for details.

---

**Artix Data Services**

Artix Data Services is a powerful tool for converting documents and messages between different data formats. In Artix Data Services, you can use a graphical tool to define complex mapping rules (including processing of data content) and then generate stub code to implement the mapping rules. See Progress Artix Data Services[1] home page and the Artix Data Services documentation[2] for more details. The `marshal()` and `unmarshal()` DSL commands are capable of consuming Artix Data Services stub code to perform transformations on message formats. Example 1.4 on page 32 shows a rule that unmarshals XML documents into a canonical format (Java objects) and then marshals the canonical format into the tag/value pair format.

***Example 1.4. Using Artix Data Services to Marshal and Unmarshal***

```
from("SourceURL").
unmarshal().artixDS(DocumentElement.class, ArtixDSContent
Type.Xml).
```

[1] http://www.iona.com/products/artix/data_services.htm?WT.mc_id=125795
[2] http://www.iona.com/support/docs/artix/data_services/3.6/index.xml

```
marshal().artixDS(ArtixDSContentType.TagValuePair).
    to("TargetURL");
```

## Note

Artix Data Services is licensed separately from FUSE Mediation Router.

# Chapter 2. Defining Routes in XML

*When you define routes in XML, you can reconfigure themm at runtime.*

# Using the Router Schema in an XML File

**Overview**

The root element of the router schema is `camelContext`, which is defined in the XML namespace, `http://activemq.apache.org/camel/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 must specify the location of the router schema (so that the router configuration can be parsed). Example 2.1 on page 36 shows how to embed the router configuration, `camelContext`, in an arbitrary document, *`DocRootElement`*.

***Example 2.1. Specifying the Router Schema Location***

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


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

where the schema location is specified as `http://activemq.apache.org/camel/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. If you prefer to tie your configuration to a specific version of the schema, change the schema file name to `camel-spring-`*`Version`*`.xsd`, where *`Version`* can be one of: `1.0.0`, `1.1.0`, `1.2.0`, `1.3.0`, or `1.4.0`. For example, the location of schema version 1.4.0 would be specified as `http://activemq.apache.org/camel/schema/spring/camel-spring-1.4.0.xsd`.

Example 2.2 on page 36 shows an example of embedding a router configuration, `camelContext`, in a Spring configuration file.

***Example 2.2. 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 ht
tp://www.springframework.org/schema/beans/spring-beans-2.0.xsd

       http://activemq.apache.org/camel/schema/spring ht
tp://activemq.apache.org/camel/schema/spring/camel-spring.xsd">


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

# Defining a Basic Route in XML

**Basic concepts**

To understand how to build a route using XML, you must understand some of the basic concepts of the routing language, such as sources and targets, processors, expressions and predicates, and message exchanges. For definitions and explanations of these concepts see "Basic Java DSL Syntax" on page 13.

**Example of a basic route**

Example 2.3 on page 38 shows an example of a basic route in XML, which connects a source endpoint, *SourceURL*, directly to a destination endpoint, *TargetURL*.

***Example 2.3. Basic Route in XML***

```
<camelContext id="CamelContextID" xmlns="http://act
ivemq.apache.org/camel/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

## Overview

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. FUSE Mediation Router provides a wide variety of different processors, as follows:



## 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. In the following example, the filter blocks a message exchange, unless the incoming message contains a header, `foo`, with value equal to `bar`:

```
<camelContext id="filterRoute" xmlns="http://act
ivemq.apache.org/camel/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 routes incoming messages to alternative targets. Each target is enclosed in a `when` element, which takes a predicate argument. If the predicate is true, the current target is selected; if false, processing proceeds to the next `when` element in the rule. In the following example, the `choice()` processor directs incoming messages to either *`Target1`*, *`Target2`*, or *`Target3`*, depending on the values of the predicates:

```
<camelContext id="buildSimpleRouteWithChoice" xmlns="http://act
ivemq.apache.org/camel/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. The first is to include multiple `to` elements in the route:

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

```
  </route>
</camelContext>
```

The second 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://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <recipientList>
      <!-- Requires XPath 2.0 -->
      <xpath>tokenize(/headers/recipientListHead
er,"\s+")</xpath>
    </recipientList>
  </route>
</camelContext>
```

---

**Splitter**

The `splitter` processor splits 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. The following example 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://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <splitter>
      <xpath>/section</xpath>
      <to uri="seda:b"/>
    </splitter>
  </route>
</camelContext>
```

---

**Aggregator**

The `aggregator` processor aggregates related incoming messages into a single message. In order to distinguish which messages are eligible to be aggregated together, you must 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 pck 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://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <aggregator>
      <simple>header.stockSymbol</simple>
      <to uri="TargetURL"/>
    </aggregator>
  </route>
</camelContext>
```

---

**Resequencer**

A `resequencer` processor reorders incoming messages and forwards them to the target. The `resequencer` element must be provided with a sequence number calculated from the contents of a field in the incoming message. Before you can start re-ordering messages, you must wait until a certain number of messages have been received from the source. There are two ways to specify how long the `resequencer` processor waits before attempting to re-order the accumulated messages and forward them to the target. They are:

- *Batch resequencing* — Waits until a specified number of messages have accumulated before starting to re-order and forward messages. This is the default. For example, the following resequencing rule re-orders messages based on the `timeOfDay` header, waiting until either 300 messages have accumulated or 4000 ms have elapsed since the last message was received:

```
<camelContext id="batchResequencer" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL" />
    <resequencer>
      <!-- Sequence ordering based on timeOfDay header -->
```

```
      <simple>header.timeOfDay</simple>
      <to uri="TargetURL" />
      <!--
        batch-config can be ommitted for default (batch)
resequencer settings
      -->
      <batch-config batchSize="300" batchTimeout="4000" />
    </resequencer>
  </route>
</camelContext>
```

- *Stream resequencing* — Transmits messages as soon as they arrive *unless* the resequencer detects a gap in the incoming message stream (missing sequence numbers). If there is a gap, the resequencer waits until the missing messages arrive and then forwards the messages in the correct order. To avoid the resequencer blocking forever, you specify a timeout (the 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://act
ivemq.apache.org/camel/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 ensures 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. To limit the rate of throughput to 100 messages per second, you can define the following rule:

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

```
    <from uri="SourceURL"/>
    <throttler maximumRequestsPerPeriod="100" timePeriodMil
lis="1000">
      <to uri="TargetURL"/>
    </throttler>
  </route>
</camelContext>
```

---

**Delayer**

The `delayer` processor delays messages for a specified length of time. The delay can either be relative (waits a specified length of time after receipt of the incoming message) or absolute (waits 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://act
ivemq.apache.org/camel/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://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <delayer>
      <simple>header.processAfter</simple>
      <to uri="TargetURL"/>
    </delayer>
  </route>
</camelContext>
```

# Languages for Expressions and Predicates

**Overview**

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, FUSE Mediation Router supports multiple language plug-ins, which can be accessed through XML elements.

**Elements for expressions and predicates**

Table 2.1 on page 45 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 2.1. Elements for Expression and Predicate Languages***

| Element | Language | Description |
|---|---|---|
| `simple` | N/A | A simple expression language, native to FUSE Mediation Router (see "Simple" on page 22). |
| `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/). |

| Element | Language | Description |
|---|---|---|
| `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

**Overview**

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

**Marshalling and unmarshalling**

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

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

FUSE Mediation Router supports marshalling and unmarshalling of the following data formats:

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

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

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

```
<camelContext id="jaxb" xmlns="http://act
ivemq.apache.org/camel/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, unmarshalling converts an XML data type to a Java object and marshalling converts a Java object to an XML data type. For example, to unmarshal XML data to a Java object using XMLBeans, define a rule like the following:

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

- *XStream* — Is currently *not* supported in XML configuration.
- *Artix Data Services* —FUSE Mediation Router also integrates with Artix Data Services, enabling you to integrate stub code generated by Artix Data Services. See "Artix Data Services" on page 48 for details.

---

**Artix Data Services**

Artix Data Services is a powerful tool for converting documents and messages between different data formats. In Artix Data Services, you can use a graphical tool to define complex mapping rules (including processing of data content) and then generate stub code to implement the mapping rules. See Progress

Artix Data Services[1] home page and the Artix Data Services documentation[2] for more details. The `marshal` and `unmarshal` elements are capable of consuming Artix Data Services stub code to perform transformations on message formats. Example 2.4 on page 49 shows a rule that unmarshals XML documents into a canonical format (Java objects) and then marshals the canonical format into the tag/value pair format.

***Example 2.4. Using Artix Data Services to Marshal and Unmarshal***

```
<camelContext id="artixDS" xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="SourceURL"/>
    <unmarshal>
      <artixDS contentType="Xml" elementType
Name="iso.std.iso.x20022.tech.xsd.pacs.x008.x001.x01.DocumentElement"/>
    </unmarshal>
    <marshal>
      <artixDS contentType="TagValuePair"/>
    </marshal>
    <to uri="TargetURL"/>
  </route>
</camelContext>
```

Where the `contentType` attribute can be set to one of the following values: `TagValuePair`, `Sax`, `Xml`, `Java`, `Text`, `Binary`, `Auto`, `Default`.

## Note

Artix Data Services is licensed separately from FUSE Mediation Router.

[1] http://www.iona.com/products/artix/data_services.htm?WT.mc_id=125795
[2] http://www.iona.com/support/docs/artix/data_services/3.6/index.xml

# Chapter 3. Basic Principles of Route Building

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

# Pipeline Processing

**Overview**

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

**Processor nodes**

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

The first approach is where the processor simply modifies the exchange's *In* message, as shown in Figure 3.1 on page 52. The exchange's *Out* message remains `null` in this case.

***Figure 3.1. Processor Modifying an In Message***

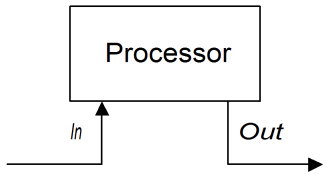

The following route shows a `setHeader()` command that modifies the current *In* message by adding (or modifying) the `BillingSystem` heading:

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

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

The second approach is where the processor creates an *Out* message to represent the result of the processing, as shown in Figure 3.2 on page 53.

***Figure 3.2. Processor Creating an Out Message***




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

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

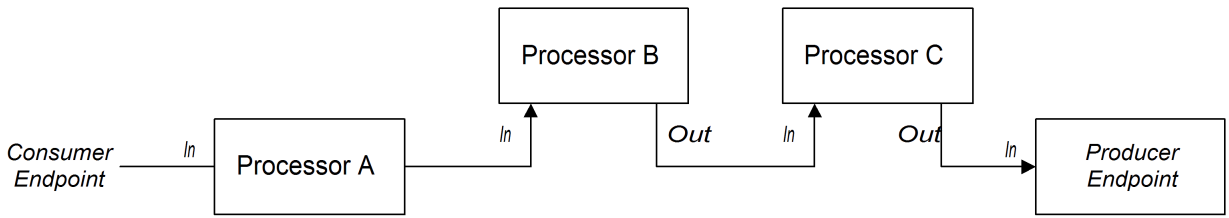
where `constant("DummyBody")` represents a constant expression. You cannot pass the string, `DummyBody`, directly, because the argument to `transform()` must be an expression type.

---

**Pipeline for InOnly exchanges**

Figure 3.3 on page 53 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 3.3 on page 53.

***Figure 3.3. Sample Pipeline for InOnly Exchanges***




FUSE Mediation Router employs the pipeline pattern by default, so you do not need to use any special syntax to create a pipeline in your routes. For example, the following route pulls messages from a `userdataQueue` queue,

pipes the message through a Velocity template (to produce a customer address in text format), and then sends the resulting text address to the queue, `envelopeAddressQueue`:

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

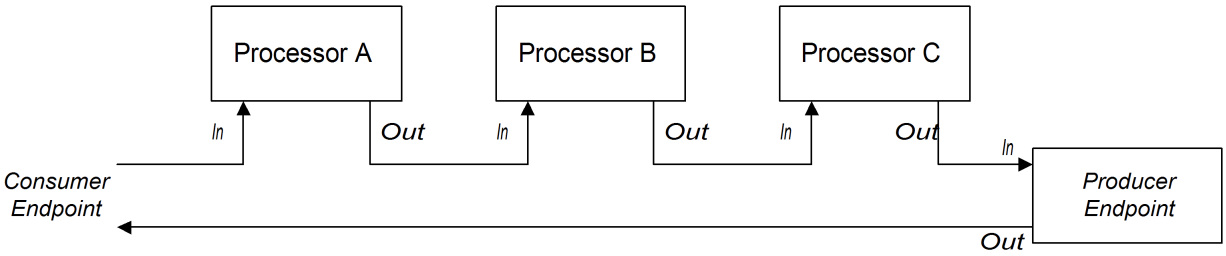
Where the Velocity endpoint, `velocity:file:AdressTemplate.vm`, specifies the location of a Velocity template file, `file:AdressTemplate.vm`, in the file system. For more details of the Velocity endpoint, see ????.

---

**Pipeline for InOut exchanges**

Figure 3.4 on page 54 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 3.4. Sample Pipeline for InOut Exchanges***




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

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

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

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

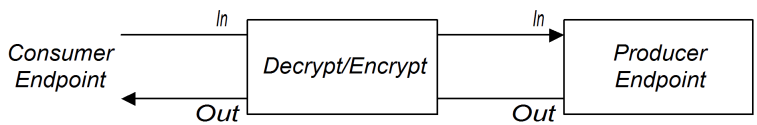
When the pipeline consists of just a sequence of endpoints, it is also possible to use the following alternative syntax:

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

**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 FUSE Mediation Router, but it is not the usual approach to use. Figure 3.5 on page 55 shows an example of an interceptor processor that implements a custom encryption algorithm.

***Figure 3.5. 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 an HTTP port that services incoming requests encoded using custom encryption, you can define a route like the following:

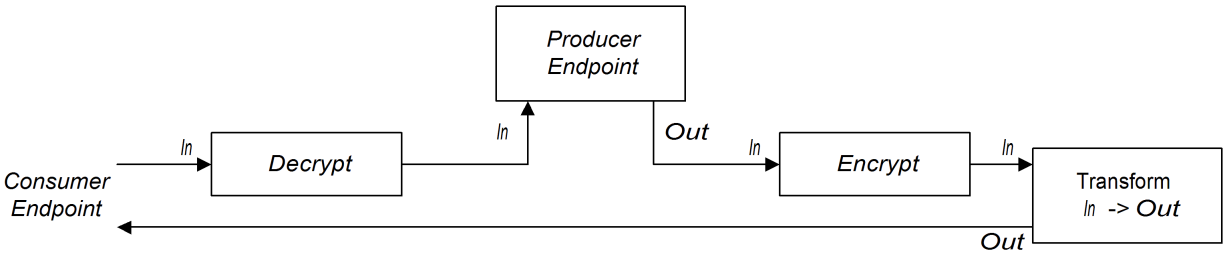
```
from("jetty:http://localhost:8080/foo")
    .intercept(new MyDecryptEncryptInterceptor())
    .to("cxf:bean:processTxn");
```

Where the class, `MyDecryptEncryptInterceptor`, is implemented by inheriting from the class,

`org.apache.camel.processor.DelegateProcessor`. For details of how to implement this kind of processor, see ????.

Although it is possible to implement this kind of functionality using an interceptor processor, this is not a very idiomatic way of programming in FUSE Mediation Router. A more typical approach is shown in Figure 3.6 on page 56.

***Figure 3.6. 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 3.5 on page 55. 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; the `transform` processor copies the *In* message to the *Out* message, ensuring that the reply message is available to the HTTP consumer endpoint. An alternative solution to this problem is to implement the encrypt processor so that it creates an *Out* message directly.

To implement the pipeline approach shown in Figure 3.6 on page 56, you can define a route like the following:

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

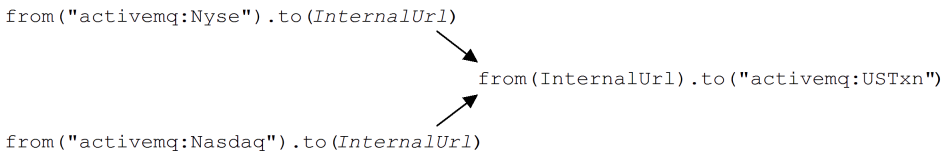
Where the final processor node, `transform(body())`, has the effect of copying the *In* message to the *Out* message (the *In* message body is copied explicitly and the *In* message headers are copied implicitly).

# Multiple Inputs

**Overview**

A standard route takes its input from just a single endpoint, using the `from(EndpointURL)` syntax in the Java DSL. But what if you need to define multiple inputs for your route? 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 3.7 on page 57.

***Figure 3.7. Processing Multiple Inputs with Segmented Routes***




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

- "Direct endpoints" on page 58.
- "SEDA endpoints" on page 59.
- "VM endpoints" on page 60.

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

## Note

The direct, SEDA, and VM components work only for the *InOnly* exchange pattern. If one of your inputs requires an *InOut* exchange pattern, refer to the "Content enricher pattern" on page 60 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, if you want to take the input from two message queues, `activemq:Nyse` and `activemq:Nasdaq`, and merge them into a single message queue, `activemq:USTxn`, you can do this by defining the following set of routes:

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

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

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

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

## 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 (the default size is 5) to process exchange objects from the blocking queue.
- The SEDA component supports the *competing consumers* pattern, which guarantees that each incoming exchange is processed only once, even if there are multiple consumers attached to a specific endpoint.

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

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

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

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

## Note

There is more to SEDA than simply pasting together route segments. The staged event-driven architecture (SEDA) encompasses a design philosophy for building more manageable multi-threaded applications. The purpose of the SEDA component in FUSE Mediation 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 FUSE Mediation Router applications, as long as they are running within the same Java virtual machine.

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

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

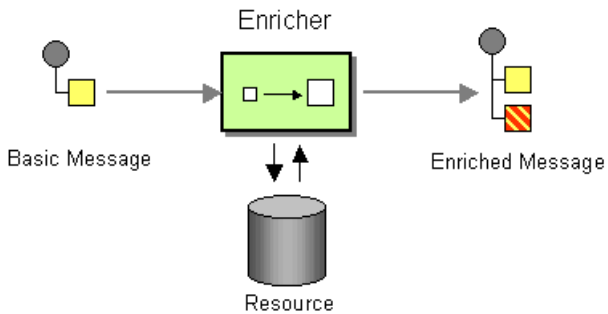
And in a separate router application (running in the same Java VM), you can define the second segment of the route as follows:

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

**Content enricher pattern**

The content enricher pattern defines a fundamentally different way of dealing with multiple inputs to a route. A content enricher is a processor that you can insert into a route, as shown in Figure 3.8 on page 60. 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 3.8. 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. 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.

For more details about the content enricher pattern, see "Content Enricher" in *Implementing Enterprise Integration Patterns*.

# Exception Handling

**Overview**

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

- `errorHandler()` clause — 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 FUSE Mediation Router. For full details, see "Dead Letter Channel" in *Implementing Enterprise Integration Patterns*.

---

**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 can define a `RouteBuilder` as follows:

```
public class MyRouteBuilder extends RouteBuilder {

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


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

Redirection to the dead letter channel does not occur, however, until all attempts at redelivery have been exhausted (see "Redelivery policy" on page 63).

---

**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("act
ivemq: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://fulfill
ment@acme.com");
        from("file:src/data?noop=true").to("file:target/mes
sages");
        // ...
    }
}
```

When an exception occurs, FUSE Mediation 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 all attempts at redelivery have been exhausted (see "Redelivery policy" on page 63).

---

**Redelivery policy**

Both the `errorHandler` clause and the `onException` clause support a *redelivery policy* that specifies how often FUSE Mediation 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.

For full details about the redelivery policy and its associated settings, see "Dead Letter Channel" in *Implementing Enterprise Integration Patterns*.

# Bean Integration

## Overview

Bean integration provides a general purpose mechanism for processing messages using arbitrary Java objects. By inserting a bean reference into a route, you can call an arbitrary method on a Java object, which can then access and modify the incoming exchange. The mechanism that maps an exchange's contents to the parameters and return values of a bean method is known as *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 2.5 component scanning. Normally, if you are deploying your FUSE Mediation 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 a plain old Java object or POJO), use the `bean()` processor, which binds the inbound exchange to a method on the Java object. For example, to process inbound exchanges using the class, `MyBeanProcessor`, define a route like the following:

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

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

```
MyBeanProcessor myBean = new MyBeanProcessor();

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

```
from("activemq:inboundData")
    .bean(myBean, "processBody")
    .to("activemq:outboundData");
```

### Basic method signatures

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

- "Method signature for processing message bodies" on page 66.
- "Method signature for processing exchanges" on page 66.

### Method signature for processing message bodies

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

```
// Java
package com.acme;

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

### Method signature for processing exchanges

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

```
// Java
package com.acme;

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

```
    }
}
```

### Accessing a bean created in Spring XML

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

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

It is also possible to pass data to the bean's constructor arguments using Spring syntax. For full details of how to use the Spring `bean` element, see The IoC Container[1] from the Spring reference guide.

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

```
from("file:data/inbound").beanRef("myBeanId", "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://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="file:data/inbound"/>
    <bean ref="myBeanId" method="processBody"/>
    <to uri="file:data/outbound"/>
```

[1] http://static.springframework.org/spring/docs/2.5.x/reference/beans.html

```
  </route>
</camelContext>
```

### Bean binding annotations

The basic bean bindings described in "Basic method signatures" on page 66 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, FUSE Mediation Router provides the following kinds of Java annotation:

- "Basic annotations" on page 68.
- "Expression language annotations" on page 69.

### Basic annotations

Table 3.1 on page 68 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 3.1. 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, to extract data from an inbound exchange and inject the data into a method argument. Table 3.2 on page 69 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 3.2. Expression Language Annotations***

| Annotation | Description |
|---|---|
| `@Bean` | Injects a Bean expression. |
| `@BeanShell` | Injects a BeanShell expression. |
| `@Constant` | Injects a Constant expression |
| `@EL` | Injects an EL expression. |
| `@Groovy` | Injects a Groovy expression. |
| `@Header` | Injects a Header expression. |
| `@JavaScript` | Injects a JavaScript expression. |
| `@OGNL` | Injects an OGNL expression. |
| `@PHP` | Injects a PHP expression. |

| Annotation | Description |
|---|---|
| `@Python` | Injects a Python expression. |
| `@Ruby` | Injects a Ruby expression. |
| `@Simple` | Injects a Simple expression. |
| `@XPath` | Injects an XPath expression. |
| `@XQuery` | Injects an XQuery expression. |

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

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

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

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

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

```
<beans ...>
    ...
    <bean id="myCorrIdGenerator" class="com.acme.MyIdGenerat
or"/>
</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(pay
load);
       return id;
   }
}
```

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

## Note

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