



FUSE Mediation Router

FUSE Mediation Router Defining Routes

Version 1.3.6.0
May 2008

Making Software Work Together™

FUSE Mediation Router Defining Routes

IONA Technologies

Version 1.3.6.0

Published 04 Jun 2008

Copyright © 1999-2007 IONA Technologies PLC

Trademark and Disclaimer Notice

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix, FUSE, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright Notice

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice. Portions of this document may include Apache Foundation documentation, all rights reserved.

Table of Contents

Preface	11
Open Source Project Resources	12
Document Conventions	13
Defining Routes in Java DSL	15
Implementing a RouteBuilder Class	16
Basic Java DSL Syntax	18
Processors	22
Languages for Expressions and Predicates	27
Transforming Message Content	32
Defining Routes in XML	39
Using the Router Schema in an XML File	40
Defining a Basic Route in XML	42
Processors	43
Languages for Expressions and Predicates	50
Transforming Message Content	52

List of Figures

1. Local Routing Rules	19
------------------------------	----

List of Tables

1. Properties for Simple Language	28
2. Transformation Methods from the ProcessorType Class	33
3. Methods from the Builder Class	34
4. Modifier Methods from the ValueBuilder Class	35
5. Elements for Expression and Predicate Languages	50

List of Examples

1. Implementation of a RouteBuilder Class	16
2. Implementing a Custom Processor Class	26
3. Simple Transformation of Incoming Messages	32
4. Using Artix Data Services to Marshal and Unmarshal	38
5. Specifying the Router Schema Location	40
6. Router Schema in a Spring Configuration File	41
7. Basic Route in XML	42
8. Using Artix Data Services to Marshal and Unmarshal	54

Preface

Table of Contents

Open Source Project Resources	12
Document Conventions	13

Open Source Project Resources

Apache Incubator CXF

Web site: <http://incubator.apache.org/cxf/>

User's list: <cxf-user@incubator.apache.org>

Apache Tomcat

Web site: <http://tomcat.apache.org/>

User's list: <users@tomcat.apache.org>

Apache ActiveMQ

Web site: <http://activemq.apache.org/>

User's list: <users@@activemq.apache.org>

Apache Camel

Web site:
<http://activemq.apache.org/camel/enterprise-integration-patterns.html>

User's list: <camel-user@activemq.apache.org>

Apache ServiceMix

Web site: <http://servicemix.org/site/home.html>

User's list: <servicemix-users@geronimo.apache.org>

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

<code>fixed width</code>	<p>Fixed width (Courier 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 <code>javax.xml.ws.Endpoint</code> class.</p> <p>Constant width paragraphs represent code examples or information a system displays on the screen. For example:</p> <pre>import java.util.logging.Logger;</pre>
<i>Fixed width italic</i>	<p>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:</p> <pre>% cd /users/YourUserName</pre>
<i>Italic</i>	Italic words in normal text represent <i>emphasis</i> and introduce <i>new terms</i> .
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).
--	---

Admonition conventions

This book uses the following conventions for admonitions:

	Notes display information that may be useful, but not critical.
	Tips provide hints about completing a task or using a tool. They may also provide information about workarounds to possible problems.
	Important notes display information that is critical to the task at hand.
	Cautions display information about likely errors that can be encountered. These errors are unlikely to cause damage to your data or your systems.
	Warnings display information about errors that may cause damage to your systems. Possible damage from these errors include system failures and loss of data.

Defining Routes in Java DSL

Summary

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.

Table of Contents

Implementing a RouteBuilder Class	16
Basic Java DSL Syntax	18
Processors	22
Languages for Expressions and Predicates	27
Transforming Message Content	32

Implementing a RouteBuilder Class

Overview

In Mediation 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 `exception()`).

Implementing a RouteBuilder

Example 1, “Implementation of a RouteBuilder Class” 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/messages");

        // 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:

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

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

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

You could even define blocks, for example:

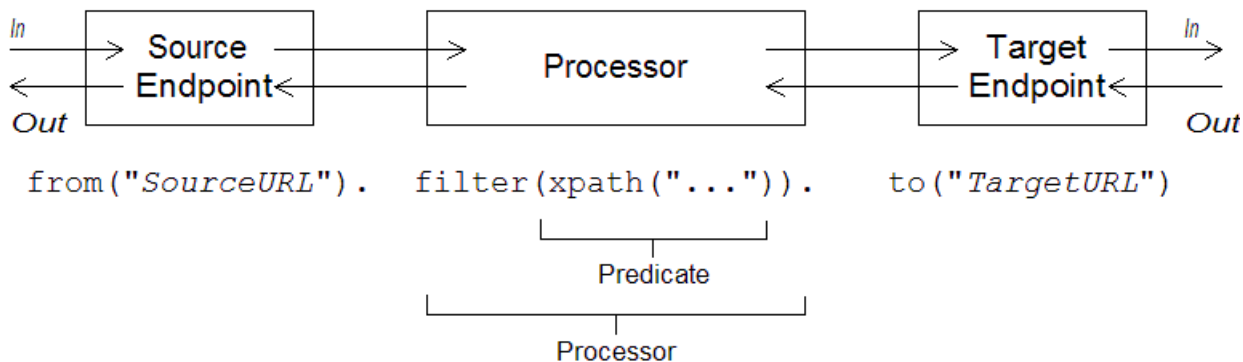
```
command01().startBlock().command02().command03().endBlock()
```

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 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, “Local Routing Rules” 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()`, `exception()`, `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://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, “Local Routing Rules” includes a `filter()` processor that takes an `xpath()` predicate as its argument.

Expressions and predicates

Expressions (evaluating to strings or other data types) and predicates (evaluating to true or false) occur frequently as arguments to the built-in processor types. 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. 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 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, 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 in order to perform arbitrary processing of the messages that flow through the rule. Mediation Router provides a wide variety of different processors, as follows:

- Filter.
- Choice.
- Pipeline
- Recipient list.
- Splitter.
- Aggregator.
- Resequencer.
- Throttler.
- Delayer.
- 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("TargetURL");
```

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

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 `splitter()` 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").splitter(bodyAs(String.class).tokenize("\n")).to("TargetURL");
```

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:

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

Resequencer

A `resequencer()` processor is used to re-arrange the order in which incoming messages are transmitted. The `resequencer()` 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 `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. This processing option is specified by invoking `resequencer().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").resequencer(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").resequencer(header("sequenceNum
ber")).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").throttler(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").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 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, “Implementing a Custom Processor Class”:

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

Languages for Expressions and Predicates

Overview

To provide you with greater flexibility when parsing and processing messages, 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 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. 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 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, “Properties for Simple Language” shows the list of elementary expressions supported by the simple language.

Table 1. Properties for Simple Language

Elementary Expression	Description
<code>body</code>	Access the body of the incoming message.
<code>out.body</code>	Access the body of the outgoing message.
<code>header.HeaderName</code>	Access the contents of the <i>HeaderName</i> header from the incoming message.
<code>out.header.HeaderName</code>	Access the contents of the <i>HeaderName</i> header from the outgoing message.
<code>property.PropertyName</code>	Access the <i>PropertyName</i> 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:tar
```

```
get/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 Mediation 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. 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.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.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 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

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

- Simple transformations.
- Marshalling and unmarshalling.
- Artix Data Services.

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, “Simple Transformation of Incoming Messages” 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.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 3, “Simple Transformation of Incoming Messages”. Table 2, “Transformation Methods from the ProcessorType Class”

shows the `ProcessorType` methods that are relevant to transforming message content:

Table 2. Transformation Methods from the ProcessorType Class

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

Method	Description
<code>ExpressionClause<ProcessorType<Type>> setOutHeader(String name)</code>	Adds a processor which sets the header on the OUT message.
<code>Type setOutHeader(String name, Expression expression)</code>	Adds a processor which sets the header on the OUT message.
<code>ExpressionClause<ProcessorType<Type>> setProperty(String name)</code>	Adds a processor which sets the exchange property.
<code>Type setProperty(String name, Expression expression)</code>	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, “Simple Transformation of Incoming Messages”. Table 3, “Methods from the Builder Class” summarizes the static methods available in the `Builder` class.

Table 3. Methods from the Builder Class

Method	Description
<code>static <E extends Exchange> ValueBuilder<E> body()</code>	Returns a predicate and value builder for the inbound body on an exchange.
<code>static <E extends Exchange, T> ValueBuilder<E> bodyAs(Class<T> type)</code>	Returns a predicate and value builder for the inbound message body as a specific type.
<code>static <E extends Exchange> ValueBuilder<E> constant(Object value)</code>	Returns a constant expression.
<code>static <E extends Exchange> ValueBuilder<E> faultBody()</code>	Returns a predicate and value builder for the fault body on an exchange.
<code>static <E extends Exchange, T> ValueBuilder<E> faultBodyAs(Class<T> type)</code>	Returns a predicate and value builder for the fault message body as a specific type.

Method	Description
<code>static <E extends Exchange> ValueBuilder<E> header(String name)</code>	Returns a predicate and value builder for headers on an exchange.
<code>static <E extends Exchange> ValueBuilder<E> outBody()</code>	Returns a predicate and value builder for the outbound body on an exchange.
<code>static <E extends Exchange> ValueBuilder<E> outBody()</code>	Returns a predicate and value builder for the outbound message body as a specific type.
<code>static <E extends Exchange> ValueBuilder<E> systemProperty(String name)</code>	Returns an expression for the given system property.
<code>static <E extends Exchange> ValueBuilder<E> systemProperty(String name, String defaultValue)</code>	Returns an expression for the given system property.

ValueBuilder class

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

Table 4, “Modifier Methods from the ValueBuilder Class” summarizes the methods available in the `ValueBuilder` class. That is, the table shows only the methods that are used to modify the value they are invoked on (for full details, see the *API Reference* documentation).

Table 4. Modifier Methods from the ValueBuilder Class

Method	Description
<code>ValueBuilder<E> append(Object value)</code>	Appends the string evaluation of this expression with the given value.
<code>ValueBuilder<E> convertTo(Class type)</code>	Converts the current value to the given type using the registered type converters.
<code>ValueBuilder<E> convertToString()</code>	Converts the current value a <code>String</code> using the registered type converters.
<code>ValueBuilder<E> regexReplaceAll(String regex, Expression<E> replacement)</code>	Replaces all occurrences of the regular expression with the given replacement.

Method	Description
<code>ValueBuilder<E> regexReplaceAll(String regex, String replacement)</code>	Replaces all occurrences of the regular expression with the given replacement.
<code>ValueBuilder<E> regexTokenize(String regex)</code>	Tokenizes the string conversion of this expression using the given regular expression.
<code>ValueBuilder<E> tokenize()</code>	
<code>ValueBuilder<E> tokenize(String token)</code>	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()`—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.

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 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, 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 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, 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 can 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 can use code like the following:

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

- *Artix Data Services*—Mediation Router also integrates with Artix Data Services, enabling you to integrate stub code generated by Artix Data Services. See Artix Data Services 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 IONA Artix Data Services

[http://www.iona.com/products/artix/data_services.htm?WT.mc_id=125795]

home page and the Artix Data Services documentation

[http://www.iona.com/support/docs/artix/data_services/3.6/index.xml] for

more details). The `marshal()` and `unmarshal()` DSL commands are capable

of consuming Artix Data Services stub code in order to perform transformations on message formats. For example, Example 4, “Using Artix Data Services to Marshal and Unmarshal” 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 4. Using Artix Data Services to Marshal and Unmarshal

```
from("SourceURL").
    unmarshal().artixDS(DocumentElement.class, ArtixDSContent
Type.Xml).
    marshal().artixDS(ArtixDSContentType.TagValuePair).
        to("TargetURL");
```



Note

Artix Data Services is licensed separately from Mediation Router.

Defining Routes in XML

Summary

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.

Table of Contents

Using the Router Schema in an XML File	40
Defining a Basic Route in XML	42
Processors	43
Languages for Expressions and Predicates	50
Transforming Message Content	52

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 need to specify the location of the router schema (so that the router configuration can be parsed). For example, Example 5, “Specifying the Router Schema Location” shows how to embed the router configuration, `camelContext`, in an arbitrary document, `DocRootElement`.

Example 5. 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 to be `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, or 1.2.0. For example, the location of schema version 1.2.0 would be specified as `http://activemq.apache.org/camel/schema/spring/camel-spring-1.2.0.xsd`.

Example 6, “Router Schema in a Spring Configuration File” shows an example of embedding a router configuration, `camelContext`, in a Spring configuration file.

Example 6. 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

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 7, “Basic Route in XML” shows an example of a basic route in XML, which connects a source endpoint, *SourceURL*, directly to a destination endpoint, *TargetURL*.

Example 7. Basic Route in XML

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

```
</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://activemq.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 you can use. One approach is to include multiple `to` elements in the route, for example:

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

```
<to uri="Target3"/>
</route>
</camelContext>
```

The alternative approach is to add `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://activemq.apache.org/camel/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://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <splitter>
      <xpath>/section</xpath>
      <to uri="seda:b"/>
    </splitter>
  </route>
</camelContext>
```

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

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://activemq.apache.org/camel/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://activemq.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 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://activemq.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>
```

```
</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://activemq.apache.org/camel/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://activemq.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://activemq.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, Mediation Router supports multiple language plug-ins, which can be accessed through XML elements.

Elements for expressions and predicates

Table 5, “Elements for Expression and Predicate Languages” 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 Mediation 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/).

Languages for Expressions and Predicates

Element	Language	Description
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 <code>bean</code> element is actually a mechanism for integrating with Java beans. You use the <code>bean</code> element to obtain an expression or predicate by invoking a method on a Java bean.

Transforming Message Content

Overview

This section describes how you can 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`—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.

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 could use the following rule:

```
<camelContext id="serialization" xmlns="http://activemq.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 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://activemq.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://activemq.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*—Mediation Router also integrates with Artix Data Services, enabling you to integrate stub code generated by Artix Data Services. See Artix Data Services 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 IONA Artix Data Services

[http://www.iona.com/products/artix/data_services.htm?WT.mc_id=125795] home page and the Artix Data Services documentation

[http://www.ionac.com/support/docs/artix/data_services/3.6/index.xml] for more details). The `marshal` and `unmarshal` elements are capable of consuming Artix Data Services stub code in order to perform transformations on message formats. For example, Example 8, “Using Artix Data Services to Marshal and Unmarshal” 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 8. 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 Mediation Router.