FuseSource

A Progress Software Company

Fuse Mediation Router

Expression and Predicate Languages

Version 2.7

March 2011

The experts in open source integration and messaging

Expression and Predicate Languages

Version 2.7
March 2011

Expression and Predicate Languages

Version 2.7

Publication date 22 Mar 2011

Legal Notices

Third Party Acknowledgments

Fuse Mediation Router v2.6.0 incorporates Apache Jakarta Commons DBCP v1.3 from the Apache Foundation. Such technology is subject to the following terms and conditions: Apache Software License Version 1.1 Copyright (c) 2000 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: “This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Apache"

and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache&apache.org[1]. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation. THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see http://www.apache.org/. Portions of this software are based upon public domain software originally written at the National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign.

[1] mailto:apache@apache.org

Table of Contents

List of Tables

Chapter 1. Introduction

This chapter provides an overview of all the expression languages supported by Fuse Mediation Router.

Overview of the Languages

Table of expression and predicate languages

Table 1.1 on page 12 gives an overview of the different syntaxes for invoking expression and predicate languages.

Table 1.1. Expression and Predicate Languages

Language	Static Method	Fluent DSL Method	XML Element	Annotation	Artifact
Bean	`bean()`	*EIP*`().method()`	`method`	`@Bean`	*Camel core*
Constant	`constant()`	*EIP*`().constant()`	`constant`	`@Constant`	*Camel core*
EL	`el()`	*EIP*`().el()`	`el`	`@EL`	`camel-juel`
Groovy	`groovy()`	*EIP*`().groovy()`	`groovy`	`@Groovy`	`camel-groovy`
Header	`header()`	*EIP*`().header()`	`header`	`@Header`	*Camel core*
JavaScript	`javaScript()`	*EIP*`().javaScript()`	`javaScript`	`@JavaScript`	`camel-script`
JoSQL	`sql()`	*EIP*`().sql()`	`sql`	`@SQL`	`camel-josql`
JXPath	*None*	*EIP*`().jxpath()`	`jxpath`	`@JXPath`	`camel-jxpath`
MVEL	`mvel()`	*None*	`mvel`	`@MVEL`	`camel-mvel`
OGNL	`ognl()`	*EIP*`().ognl()`	`ognl`	`@OGNL`	`camel-ognl`
PHP	`php()`	*EIP*`().php()`	`php`	`@PHP`	`camel-script`
Property	`property()`	*EIP*`().property()`	`property`	`@Property`	*Camel core*
Python	`python()`	*EIP*`().python()`	`python`	`@Python`	`camel-script`
Ruby	`ruby()`	*EIP*`().ruby()`	`ruby`	`@Ruby`	`camel-script`
Simple/File	`simple()`	*EIP*`().simple()`	`simple`	`@Simple`	*Camel core*
SpEL	`spel()`	*None*	`spel`	`@SpEL`	`camel-spring`
XPath	`xpath()`	*EIP*`().xpath()`	`xpath`	`@XPath`	*Camel core*
XQuery	`xquery()`	*EIP*`().xquery()`	`xquery`	`@XQuery`	`camel-saxon`

How to Invoke an Expression Language

Prerequisites

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

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

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-script</artifactId>
  <!-- Use the same version as your Camel core version -->
  <version>${camel.version}</version>
</dependency>
```

Approaches to invoking

As shown in Table 1.1 on page 12, there are several different syntaxes for invoking an expression language, depending on the context in which it is used. You can invoke an expression language:

- "As a static method" on page 13.
- "As a fluent DSL method" on page 14.
- "As an XML element" on page 14.
- "As an annotation" on page 15.

As a static method

Most of the languages define a static method that can be used in *any* context where an `org.apache.camel.Expression` type or an `org.apache.camel.Predicate` type is expected. The static method takes a string expression (or predicate) as its argument and returns an `Expression` object (which is usually also a `Predicate` object).

For example, to implement a content-based router that processes messages in XML format, you could route messages based on the value of the `/order/address/countryCode` element, as follows:

```
from("SourceURL")
  .choice
    .when(xpath("/order/address/countryCode = 'us'"))
      .to("file://countries/us/")
    .when(xpath("/order/address/countryCode = 'uk'"))
      .to("file://countries/uk/")
    .otherwise()
      .to("file://countries/other/")
  .to("TargetURL");
```

As a fluent DSL method

The Java fluent DSL supports another style of invoking expression languages. Instead of providing the expression as an argument to an Enterprise Integration Pattern (EIP), you can provide the expression as a sub-clause of the DSL command. For example, instead of invoking an XPath expression as `filter(xpath("Expression"))`, you can invoke the expression as, `filter().xpath("Expression")`.

For example, the preceding content-based router can be re-implemented in this style of invocation, as follows:

```
from("SourceURL")
  .choice
    .when().xpath("/order/address/countryCode = 'us'")
      .to("file://countries/us/")
    .when().xpath("/order/address/countryCode = 'uk'")
      .to("file://countries/uk/")
    .otherwise()
      .to("file://countries/other/")
  .to("TargetURL");
```

As an XML element

You can also invoke an expression language in Spring, by putting the expression string inside the relevant XML element.

For example, the XML element for invoking XPath in Spring is `xpath` (which belongs to the standard Apache Camel namespace). You can use XPath expressions in a Spring DSL content-based router, as follows:

```
<from uri="file://input/orders"/>
<choice>
  <when>
    <xpath>/order/address/countryCode = 'us'</xpath>
    <to uri="file://countries/us/"/>
  </when>
  <when>
```

```
    <xpath>/order/address/countryCode = 'uk'</xpath>
    <to uri="file://countries/uk/"/>
  </when>
  <otherwise>
    <to uri="file://countries/other/"/>
  </otherwise>
</choice>
```

As an annotation

Language annotations are used in the context of bean integration (see "Bean Integration" in *Implementing Enterprise Integration Patterns*). The annotations provide a convenient way of extracting information from a message or header and then injecting the extracted data into a bean's method parameters.

For example, consider the bean, `myBeanProc`, which is invoked as a predicate of the `filter()` EIP. If the bean's `checkCredentials` method returns `true`, the message is allowed to proceed; but if the method returns `false`, the message is blocked by the filter. The filter pattern is implemented as follows:

```
// Java
MyBeanProcessor myBeanProc = new MyBeanProcessor();

from("SourceURL")
  .filter().method(myBeanProc, "checkCredentials")
  .to("TargetURL");
```

The implementation of the `MyBeanProcessor` class exploits the `@XPath` annotation to extract the `username` and `password` from the underlying XML message, as follows:

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

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

The `@XPath` annotation is placed just before the parameter into which it gets injected. Notice how the XPath expression *explicitly* selects the text node, by appending `/text()` to the path, which ensures that just the content of the element is selected, not the enclosing tags.

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.

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");
```

For full details of bean integration, see "Bean Integration" in *Implementing Enterprise Integration Patterns*.

Constant

The constant language is a trivial built-in language that is used to specify a plain text string. This makes it possible to provide a plain text string in any context where an expression type is expected. For example, to set the `username` header to the value, `Jane Doe`:

```
from("SourceURL")
  .setHeader("username", constant("Jane Doe"))
  .to("TargetURL");
```

EL

The Unified Expression Language (EL) enables you 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 [1], which is an open source implementation of the EL language.

To use the `el()` static method in your application code, include the following import statement in your Java source files:

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

File

The File language is an extension to the Simple language that can only be used in conjunction with a File consumer endpoint or an FTP consumer endpoint. Because it is an extension to the simple language, it is invoked using the `simple()` method in the Java DSL and using the `simple` element in the Spring DSL.

For example, to resequence the exchanges read by a File consumer, so that the exchanges are alphabetically ordered by file name, you can define a route as follows:

```
from("file://target/filelanguage/")
  .resequence(simple("file:name"))
  .to("TargetURL");
```

A more elegant approach, however, is to use the File endpoint's built-in `sortBy` option, which takes a simple expression as its value. Using the `sortBy` option, you can ensure that files are processed in alphabetical order, as follows:

```
from("file://target/filelanguage/?sortBy=file:name")
  .to("TargetURL");
```

[1] http://juel.sourceforge.net/

For full details of the File language, see "The File Language" on page 39.

Groovy

The Groovy[2] scripting language enables you to construct predicates and expressions in a route. To use the `groovy()` static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.camel.script.Script
Builder.*;
```

Header

The header language provides a convenient way of accessing header values in the current message. When you supply a header name, the header language performs a case-insensitive lookup and returns the corresponding header value.

For example, to resequence incoming exchanges according to the value of a `TimeStamp` header, you can define a route as follows:

```
from("SourceURL")
  .resequence(header("TimeStamp"))
  .to("TargetURL");
```

JavaScript

The JavaScript[3] scripting language enables you to construct predicates and expressions in a route (see ECMAScript[4]). To use the `javaScript()` static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.camel.script.Script
Builder.*;
```

JoSQL

The JoSQL (SQL for Java objects) language enables you 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/.

[2] http://groovy.codehaus.org/
[3] http://developer.mozilla.org/en/docs/JavaScript
[4] http://www.ecmascript.org/

To use the `sql()` static method in your application code, include the following import statement in your Java source files:

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

JXPath

The JXPath language enables you to invoke Java beans using the Apache Commons JXPath[5] language. The JXPath language has a similar syntax to XPath, but instead of selecting element or attribute nodes from an XML document, it invokes methods on an object graph of Java beans. If one of the bean attributes returns an XML document (a DOM/JDOM instance), however, the remaining portion of the path is interpreted as an XPath expression and is used to extract an XML node from the document. In other words, the JXPath language provides a hybrid of object graph navigation and XML node selection.

When you invoke a JXPath expression in Fuse Mediation Router, the following bean instances are pre-defined:

`this`
: The current exchange is the root object.

`in`
: The current *In* message (equivalent to `this.in`).

`out`
: The current *Out* message (equivalent to `this.out`).

For example, if the body of the current *In* message contains the following XML fragment:

```
<person>
  <name surname="Bloggs" firstName="Joe"/>
</person>
```

You can test the value of the `surname` attribute using the following JXPath expression:

[5] http://commons.apache.org/jxpath/

```
from("SourceURL")
  .filter().jxpath("in/body/person/name/@surname='Bloggs'")
  .to("TargetURL");
```

MVEL

The MVEL[6] langauge is a dynamically-typed object graph navigation language (similar to OGNL and Groovy). You use the MVEL dot syntax to invoke Java methods, for example:

```
getRequest().getBody().getFamilyName()
```

Because MVEL is dynamically typed, it is unnecessary to cast the message body instance (of `Object` type) before invoking the `getFamilyName()` method. You can also use an abbreviated syntax for invoking bean attributes, for example:

```
request.body.familyName
```

When you invoke an MVEL expression in Fuse Mediation Router, the following variables and bean instances are pre-defined:

Variable	Type	Description
`this`	`Exchange`	The current exchange is the root object.
`exchange`	`Exchange`	The current exchange.
`exchangeId`	`String`	The current exchange's ID.
`exception`	`Throwable`	The exchange exception (if any).
`fault`	`Message`	The fault message (if any).
`request`	`Message`	The exchange's *In* message.
`response`	`Message`	The exchange's *Out* message (if any).
`properties`	`Map`	The exchange properties.
`property(`*`Name`*`)`	`Object`	The exchange property keyed by *`Name`*.
`property(`*`Name`*`,` *`Type`*`)`	`Type`	The exchange property keyed by *`Name`*, converted to the type, *`Type`*.

[6] http://mvel.codehaus.org/

For example, to select only those messages whose `Country` header has the value `USA`, you can use the following MVEL expression:

```
from("SourceURL")
  .filter().mvel("request.headers.Country == 'USA'")
  .to("TargetURL");
```

OGNL

The OGNL (Object Graph Navigation Language)[7] enables you to define predicates and expressions in a router rule.

To use the `ognl()` static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.language.ognl.OgnlExpres
sion.ognl;
```

PHP

The PHP[8] scripting language enables you to construct predicates and expressions in a route. To use the `php()` static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.camel.script.Script
Builder.*;
```

Property

The property language provides a convenient way of accessing exchange properties. When you supply a key that matches one of the property names, the property language returns the corresponding value.

For example, to implement the recipient list pattern when the `listOfEndpoints` exchange property contains the recipient list, you could define a route as follows:

```
from("direct:a").recipientList(property("listOfEndpoints"));
```

Python

The Python[9] scripting language enables you to construct predicates and expressions in a route. To use the `python()` static method in your application code, include the following import statement in your Java source files:

[7] http://www.ognl.org/
[8] http://www.php.net/
[9] http://www.python.org/

```
import static org.apache.camel.builder.camel.script.Script
Builder.*;
```

Ruby

The Ruby[10] scripting language enables you to construct predicates and expressions in a route. To use the `ruby()` static method in your application code, include the following import statement in your Java source files:

```
import static org.apache.camel.builder.camel.script.Script
Builder.*;
```

Simple

The simple language is a basic expression and predicate language built into the router core. This language is particularly useful, if you need to eliminate dependencies on third-party libraries during testing. To use the simple language in your Java application code, include the following import statement in your Java source files:

```
import static org.apache.camel.language.simple.SimpleLan
guage.simple;
```

For full details of the simple language, see "The Simple Language" on page 27.

SpEL

The Spring Expression Language (SpEL)[11] is an object graph navigation language provided with Spring 3, which can be used to construct predicates and expressions in a route. A notable feature of SpEL is the ease with which you can access beans from the registry.

When you invoke an SpEL expression in Fuse Mediation Router, the following variables and bean instances are pre-defined:

Variable	Type	Description
`this`	`Exchange`	The current exchange is the root object.
`exchange`	`Exchange`	The current exchange.
`exchangeId`	`String`	The current exchange's ID.
`exception`	`Throwable`	The exchange exception (if any).

[10] http://www.ruby-lang.org/
[11] http://static.springsource.org/spring/docs/current/spring-framework-reference/htmlsingle/spring-framework-reference.html#expressions

Variable	Type	Description
`fault`	`Message`	The fault message (if any).
`request`	`Message`	The exchange's *In* message.
`response`	`Message`	The exchange's *Out* message (if any).
`properties`	`Map`	The exchange properties.
`property(`*`Name`*`)`	`Object`	The exchange property keyed by *`Name`*.
`property(`*`Name`*`,` *`Type`*`)`	`Type`	The exchange property keyed by *`Name`*, converted to the type, *`Type`*.

The SpEL expressions must use the placeholder syntax, `#{`*`SpelExpression`*`}`, so that they can be embedded in a plain text string (in other words, SpEL has expression templating enabled).

For example, to select only those messages whose `Country` header has the value `USA`, you can use the following SpEL expression:

```
from("SourceURL")
  .filter().spel("#{request.headers['Country'] == 'USA'}")
  .to("TargetURL");
```

You can also use the SpEL expression in Spring DSL, as follows:

```
<route>
  <from uri="SourceURL"/>
  <filter>
    <spel>#{request.headers['Country'] == 'USA'}}</spel>
    <to uri="TargetURL"/>
  </filter>
</route>
```

The following example shows how to embed SpEL expressions within a plain text string:

```
from("SourceURL")
  .setBody(spel("Hello #{request.body}! What a beautiful
#{request.headers['dayOrNight']}"))
  .to("TargetURL");
```

SpEL can also look up beans in the registry (typically, the Spring registry), using the `@`*`BeanID`* syntax. For example, given a bean with the ID, `headerUtils`, and the method, `count()` (which counts the number of

headers on the current message), you could use the `headerUtils` bean in an SpEL predicate, as follows:

```
#{@headerUtils.count > 4}
```

XPath

The XPath language enables you to select parts of the current message, when the message is in XML format. To use the `xpath()` static method 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 endpoint to route the message to, based on the content of the `city` element, using the following rule:

```
from("file:src/data?noop=true").
choice().
  when(xpath("/person/city = 'London'")).to("file:target/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 language enables you to select parts of the current message, when the message is in XML format. XQuery is a superset of the XPath language; hence, any valid XPath expression is also a valid XQuery expression. To use the `xquery()` static method 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.

Chapter 2. The Simple Language

The simple language is a language that was developed in Apache Camel specifically for the purpose of accessing and manipulating the various parts of an exchange object. The language is not quite as simple as when it was originally created and it now features a comprehensive set of logical operators and conjunctions.

Java DSL

Simple expressions in Java DSL

In the Java DSL, there are two styles for using the `simple()` command in a route. You can either pass the `simple()` command as an argument to a processor, as follows:

```
from("seda:order")
  .filter(simple("in.header.foo"))
  .to("mock:fooOrders");
```

Or you can call the `simple()` command as a sub-clause on the processor, for example:

```
from("seda:order")
  .filter()
  .simple("in.header.foo")
  .to("mock:fooOrders");
```

Placeholder syntax

If you are embedding a simple expression inside a plain text string, you must use the placeholder syntax, `${`*`Expression`*`}`. For example, to embed the `in.header.name` expression in a string:

```
simple("Hello ${in.header.name}, how are you?")
```

Spring DSL

Simple expressions in Spring DSL

In the Spring DSL, you can use a simple expression by putting inside a `simple` element. For example, to define a route that performs filtering based on the contents of the `foo` header:

```
<route id="simpleExample">
  <from uri="seda:orders"/>
  <filter>
    <simple>in.header.foo</simple>
    <to uri="mock:fooOrders"/>
  </filter>
</route>
```

Placeholder syntax

If you are embedding a simple expression inside a plain text string, you must use the placeholder syntax, `${Expression}`. For example, to embed the `in.header.name` expression in a string:

```
<simple>Hello ${in.header.name}, how are you?</simple>
```

Sometimes—for example, if you have enabled Spring property placeholders or OSGi blueprint property placeholders—you might find that the `${Expression}` syntax clashes with another property placeholder syntax. In this case, you can disambiguate the placeholder using the alternative syntax, `$simple{Expression}`, for the simple expression. For example:

```
<simple>Hello $simple{in.header.name}, how are you?</simple>
```

Expressions

Overview

The simple language provides various elementary expressions that return different parts of a message exchange. For example, the expression, `simple("header.timeOfDay")`, would return the contents of a header called `timeOfDay` from the incoming message.

Contents of a single variable

You can use the simple language to define string expressions, based on the variables provided. For example, you can use a variable of the form, `in.header.`*`HeaderName`*, to obtain the value of the *`HeaderName`* header, as follows:

```
simple("in.header.foo")
```

Variables embedded in a string

You can embed simple variables in a string expression, but in this case you must enclose the variables in `${ }` (when you reference a variable on its own, the enclosing braces are optional)—for example:

```
simple("Received a message from ${in.header.user} on
$(date:in.header.date:yyyyMMdd}.")
```

date and bean variables

As well as providing variables that access all of the different parts of an exchange (see Table 2.1 on page 34), the simple language also provides special variables for formatting dates, `date:`*`command`*`:`*`pattern`*, and for calling bean methods, `bean:`*`beanRef`*. For example, you can use the date and the bean variables as follows:

```
simple("Todays date is ${date:now:yyyyMMdd}")
simple("The order type is ${bean:orderService?method=getOrder
Type}")
```

OGNL expressions

The Object Graph Navigation Language[1] (OGNL) is a notation for invoking bean methods in a chain-like fashion. If a message body contains a Java bean, you can easily access its bean properties using OGNL notation. For example, if the message body is a Java object with a `getAddress()` accessor,

[1] http://www.opensymphony.com/ognl/

you can access the `Address` object and the `Address` object's properties as follows:

```
simple("${body.address}")
simple("${body.address.street}")
simple("${body.address.zip}")
simple("${body.address.city}")
```

Where the notation, `${body.address.street}`, is shorthand for `${body.getAddress.getStreet}`.

OGNL null-safe operator

You can use the null-safe operator, `?.`, to avoid encountering null-pointer exceptions, in case the body does *not* have an address. For example:

```
simple("${body?.address?.street}")
```

If the body is a `java.util.Map` type, you can look up a value in the map with the key, `foo`, using the following notation:

```
simple("${body[foo]?.name}")
```

OGNL list element access

You can also use square brackets notation, `[k]`, to access the elements of a list. For example:

```
simple("${body.address.lines[0]}")
simple("${body.address.lines[1]}")
simple("${body.address.lines[2]}")
```

The `last` keyword returns the index of the last element of a list. For example, you can access the *second last* element of a list, as follows:

```
simple("${body.address.lines[last-1]}")
```

Predicates

Overview

You can construct predicates by testing expressions for equality. For example, the predicate, `simple("${header.timeOfDay} == '14:30'")`, tests whether the `timeOfDay` header in the incoming message is equal to `14:30`.

Syntax

You can also test various parts of an exchange (headers, message body, and so on) using simple predicates. Simple predicates have the following general syntax:

```
${LHSVariable} Op RHSValue
```

Where the variable on the left hand side, *`LHSVariable`*, is one of the variables shown in Table 2.1 on page 34 and the value on the right hand side, *`RHSValue`*, is one of the following:

- Another variable, `${RHSVariable}`.
- A string literal, enclosed in single quotes, `' '`.
- A string literal, *not* enclosed in quotes (no spaces allowed).
- A numeric constant.
- The null object, `null`.

The simple language always attempts to convert the RHS value to the type of the LHS value.

Examples

For example, you can perform simple string comparisons and numerical comparisons as follows:

```
simple("${in.header.user} == 'john'")
simple("${in.header.user} == john")  // Quotes are optional
here

simple("${in.header.number} > 100")
simple("${in.header.number} > '100'")  // String literal can
 be converted to integer
```

You can test whether the left hand side is a member of a comma-separated list, as follows:

```
simple("${in.header.type} not in 'gold,silver'")
```

You can test whether the left hand side matches a regular expression, as follows:

```
simple("${in.header.number} regex '\d{4}'")
```

You can test the type of the left hand side using the `is` operator, as follows:

```
simple("${in.header.type} is 'java.lang.String'")
simple("${in.header.type} is String") // You can abbreviate
java.lang. types
```

You can test whether the left hand side lies in a specified numerical range, as follows:

```
simple("${in.header.number} range 100..199")
```

Conjunctions

You can also combine predicates using the logical conjunctions, `and` and `or`.

For example, here is an expression using the `and` conjunction:

```
simple("${in.header.title} contains 'Camel' and ${in.head
er.type} == 'gold'")
```

And here is an expression using the `or` conjunction:

```
simple("${in.header.title} contains 'Camel' or ${in.head
er.type} == 'gold'")
```

Variable Reference

Table of variables

Table 2.1 on page 34 shows all of the variables supported by the simple language.

Table 2.1. Variables for the Simple Language

Variable	Type	Description
`exchangeId`	`String`	The exchange's ID value.
`id`	`String`	The *In* message ID value.
`body`	`Object`	The *In* message body. *Supports OGNL expressions.*
`in.body`	`Object`	The *In* message body. *Supports OGNL expressions.*
`out.body`	`Object`	The *Out* message body.
`bodyAs(`*`Type`*`)`	*`Type`*	The *In* message body, converted to the specified type. All types, *`Type`*, must be specified using their fully-qualified Java name, except for the types: `byte[]`, `String`, `Integer`, and `Long`. The converted body can be null.
`mandatoryBodyAs(`*`Type`*`)`	*`Type`*	The *In* message body, converted to the specified type. All types, *`Type`*, must be specified using their fully-qualified Java name, except for the types: `byte[]`, `String`, `Integer`, and `Long`. The converted body is expected to be non-null.
`header.`*`HeaderName`*	`Object`	The *In* message's *`HeaderName`* header. *Supports OGNL expressions.*
`headers.`*`HeaderName`*	`Object`	The *In* message's *`HeaderName`* header.
`in.header.`*`HeaderName`*	`Object`	The *In* message's *`HeaderName`* header. *Supports OGNL expressions.*
`in.headers.`*`HeaderName`*	`Object`	The *In* message's *`HeaderName`* header. *Supports OGNL expressions.*
`out.header.`*`HeaderName`*	`Object`	The *Out* message's *`HeaderName`* header.
`out.headers.`*`HeaderName`*	`Object`	The *Out* message's *`HeaderName`* header.
`headerAs(`*`Key`*`,`*`Type`*`)`	*`Type`*	The *`Key`* header, converted to the specified type. All types, *`Type`*, must be specified using their fully-qualified Java name, except for the types: `byte[]`, `String`, `Integer`, and `Long`. The converted value can be null.
`property.`*`PropertyName`*	`Object`	The *`PropertyName`* property on the exchange.

Variable	Type	Description
`sys.SysPropertyName`	`String`	The `SysPropertyName` Java system property.
`sysenv.SysEnvVar`	`String`	The `SysEnvVar` system environment variable.
`exception`	`String`	Either the exception object from `Exchange.getException()` or, if this value is null, the caught exception from the `Exchange.EXCEPTION_CAUGHT` property; otherwise null. *Supports OGNL expressions.*
`exception.message`	`String`	If an exception is set on the exchange, returns the value of `Exception.getMessage()`; otherwise, returns `null`.
`exception.stacktrace`	`String`	If an exception is set on the exchange, returns the value of `Exception.getStackTrace()`; otherwise, returns `null`. **Note:** The simple language first tries to retrieve an exception from `Exchange.getException()`. If that property is not set, it checks for a caught exception, by calling `Exchange.getProperty(Exchange.CAUGHT_EXCEPTION)`.
`date:command:pattern`	`String`	A date formatted using a java.text.SimpleDateFormat[2] pattern. The following commands are supported: `now`, for the current date and time; `header.HeaderName`, or `in.header.HeaderName` to use a java.util.Data[3] object in the `HeaderName` header from the *In* message; `out.header.HeaderName` to use a java.util.Data[4] object in the `HeaderName` header from the *Out* message;
`bean:beanRef`	`Object`	Invokes a method on the referenced bean. To specify a method name, you can either append a dot, `.`, followed by the method name; or you can use the `?method=methodName` syntax.
`properties:Key`	`String`	The value of the `Key` property placeholder (see ????).
`properties:Location:Key`	`String`	The value of the `Key` property placeholder, where the location of the properties file is given by `Location` (see ????).
`threadName`	`String`	The name of the current thread.

[2] http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html
[3] http://java.sun.com/j2se/1.5.0/docs/api/java/util/Date.html
[4] http://java.sun.com/j2se/1.5.0/docs/api/java/util/Date.html

Operator Reference

Table of operators

The complete set of operators for simple language predicates is shown in Table 2.2 on page 36.

Table 2.2. Operators for the Simple Language

Operator	Description
`==`	Equals.
`>`	Greater than.
`>=`	Greater than or equals.
`<`	Less than.
`<=`	Less than or equals.
`!=`	Not equal to.
`contains`	Test if LHS string contains RHS string.
`not contains`	Test if LHS string does *not* contain RHS string.
`regex`	Test if LHS string matches RHS regular expression.
`not regex`	Test if LHS string does *not* match RHS regular expression.
`in`	Test if LHS string appears in the RHS comma-separated list.
`not in`	Test if LHS string does *not* appear in the RHS comma-separated list.
`is`	Test if LHS is an instance of RHS Java type (using Java `instanceof` operator).
`not is`	Test if LHS is *not* an instance of RHS Java type (using Java `instanceof` operator).
`range`	Test if LHS number lies in the RHS range (where range has the format, *`min...max`*).

Operator	Description
`not range`	Test if LHS number does *not* lie in the RHS range (where range has the format, *`min...max`*).

Combining predicates

The conjunctions shown in Table 2.3 on page 37 can be used to combine two or more simple language predicates.

Table 2.3. Conjunctions for Simple Language Predicates

Operator	Description
`and`	Combine two predicates with logical *and*. Since Fuse Mediation Router 2.5, it is possible to combine more than two predicates with this operator.
`or`	Combine two predicates with logical *inclusive or*. Since Fuse Mediation Router 2.5, it is possible to combine more than two predicates with this operator.

For example, you could use the `and` conjunction to combine two predicate expressions as follows:

```
${header.foo} >= 0 and ${header.foo} < 100
```

Chapter 3. The File Language

The file language is an extension to the simple language, not an independent language in its own right. But the file language extension can only be used in conjunction with File or FTP endpoints.

When to Use the File Language

Overview

The file language is an extension to the simple language which is not always available. You can use it under the following circumstances:

- "In a File or FTP consumer endpoint" on page 40.
- "On exchanges created by a File or FTP consumer" on page 41.

In a File or FTP consumer endpoint

There are several URI options that you can set on a File or FTP consumer endpoint, which take a file language expression as their value. For example, in a File consumer endpoint URI you can set the `fileName`, `move`, `preMove`, `moveFailed`, and `sortBy` options using a file expression.

In a File consumer endpoint, the `fileName` option acts as a filter, determining which file will actually be read from the starting directory. If a plain text string is specified (for example, `fileName=report.txt`), the File consumer reads the same file each time it is updated. You can make this option more dynamic, however, by specifying a simple expression. For example, you could use a counter bean to select a different file each time the File consumer polls the starting directory, as follows:

```
file://target/filelanguage/bean/?file
Name=${bean:counter.next}.txt&delete=true
```

Where the `${bean:counter.next}` expression invokes the `next()` method on the bean registered under the ID, `counter`.

The `move` option is used to move files to a backup location after then have been read by a File consumer endpoint. For example, the following endpoint moves files to a backup directory, after they have been processed:

```
file://target/filelanguage/?move=backup/${date:now:yyyyMM
dd}/${file:name.noext}.bak&recursive=false
```

Where the `${file:name.noext}.bak` expression modifies the original file name, replacing the file extension with `.bak`.

You can use the `sortBy` option to specify the order in which file should be processed. For example, to process files according to the alphabetical order of their file name, you could use the following File consumer endpoint:

```
file://target/filelanguage/?sortBy=file:name
```

To process file according to the order in which they were last modified, you could use the following File consumer endpoint:

```
file://target/filelanguage/?sortBy=file:modified
```

You can reverse the order by adding the `reverse:` prefix—for example:

```
file://target/filelanguage/?sortBy=reverse:file:modified
```

On exchanges created by a File or FTP consumer

When an exchange originates from a File or FTP consumer endpoint, it is possible to apply file language expressions to the exchange throughout the route (as long as the original message headers are not erased). For example, you could define a content-based router, which routes messages according to their file extension, as follows:

```
<from uri="file://input/orders"/>
<choice>
  <when>
    <simple>${file:ext} == 'txt'</simple>
    <to uri="bean:orderService?method=handleTextFiles"/>
  </when>
  <when>
    <simple>${file:ext} == 'xml'</simple>
    <to uri="bean:orderService?method=handleXmlFiles"/>
  </when>
  <otherwise>
    <to uri="bean:orderService?method=handleOtherFiles"/>
  </otherwise>
</choice>
```

File Variables

Overview

File variables can be used whenever a route starts with a File or FTP consumer endpoint, which implies that the underlying message body is of `java.io.File` type. The file variables enable you to access various parts of the file pathname, almost as if you were invoking the methods of the `java.io.File` class (in fact, the file language extracts the information it needs from message headers that have been set by the File or FTP endpoint).

Starting directory

Some of file variables return paths that are defined relative to a *starting directory*, which is just the directory that is specified in the File or FTP endpoint. For example, the following File consumer endpoint has the starting directory, `./filetransfer` (a relative path):

```
file:filetransfer
```

The following FTP consumer endpoint has the starting directory, `./ftptransfer` (a relative path):

```
ftp://myhost:2100/ftptransfer
```

Naming convention of file variables

In general, the file variables are named after corresponding methods on the `java.io.File` class. For example, the `file:absolute` variable gives the value that would be returned by the `java.io.File.getAbsolute()` method.

Note

This naming convention is not strictly followed, however. For example, there is *no* such method as `java.io.File.getSize()`.

Table of variables

Table 3.1 on page 42 shows all of the variable supported by the file language.

Table 3.1. Variables for the File Language

Variable	Type	Description
`file:name`	`String`	The pathname relative to the starting directory.

Variable	Type	Description
`file:name.ext`	`String`	The file extension (characters following the last `.` character in the pathname).
`file:name.noext`	`String`	The pathname relative to the starting directory, omitting the file extension.
`file:onlyname`	`String`	The final segment of the pathname. That is, the file name without the parent directory path.
`file:onlyname.noext`	`String`	The final segment of the pathname, omitting the file extension.
`file:ext`	`String`	The file extension (same as `file:name.ext`).
`file:parent`	`String`	The pathname of the parent directory, including the starting directory in the path.
`file:path`	`String`	The file pathname, including the starting directory in the path.
`file:absolute`	`Boolean`	`true`, if the starting directory was specified as an absolute path; `false`, otherwise.
`file:absolute.path`	`String`	The absolute pathname of the file.
`file:length`	`Long`	The size of the referenced file.
`file:size`	`Long`	*Same as `file:length`.*
`file:modified`	`java.util.Date`	Date last modified.

Examples

Relative pathname

Consider a File consumer endpoint, where the starting directory is specified as a *relative pathname*. For example, the following File endpoint has the starting directory, `./filelanguage`:

```
file://filelanguage
```

Now, while scanning the `filelanguage` directory, suppose that the endpoint has just consumed the following file:

```
./filelanguage/test/hello.txt
```

And, finally, assume that the `filelanguage` directory itself has the following absolute location:

```
/workspace/camel/camel-core/target/filelanguage
```

Given the preceding scenario, the file language variables return the following values, when applied to the current exchange:

Expression	Result
`file:name`	`test/hello.txt`
`file:name.ext`	`txt`
`file:name.noext`	`test/hello`
`file:onlyname`	`hello.txt`
`file:onlyname.noext`	`hello`
`file:ext`	`txt`
`file:parent`	`filelanguage/test`
`file:path`	`filelanguage/test/hello.txt`
`file:absolute`	`false`

Expression	Result
file:absolute.path	/workspace/camel/camel-core/target/filelanguage/test/hello.txt

Absolute pathname

Consider a File consumer endpoint, where the starting directory is specified as an *absolute pathname*. For example, the following File endpoint has the starting directory, `/workspace/camel/camel-core/target/filelanguage`:

```
file:///workspace/camel/camel-core/target/filelanguage
```

Now, while scanning the `filelanguage` directory, suppose that the endpoint has just consumed the following file:

```
./filelanguage/test/hello.txt
```

Given the preceding scenario, the file language variables return the following values, when applied to the current exchange:

Expression	Result
file:name	test/hello.txt
file:name.ext	txt
file:name.noext	test/hello
file:onlyname	hello.txt
file:onlyname.noext	hello
file:ext	txt
file:parent	/workspace/camel/camel-core/target/filelanguage/test
file:path	/workspace/camel/camel-core/target/filelanguage/test/hello.txt
file:absolute	true
file:absolute.path	/workspace/camel/camel-core/target/filelanguage/test/hello.txt

Chapter 4. The XPath Language

When processing XML messages, the XPath language enables you to select part of a message, by specifying an XPath expression that acts on the message's Document Object Model (DOM). You can also define XPath predicates to test the contents of an element or an attribute.

Java DSL

Basic expressions

You can use `xpath("Expression")` to evaluate an XPath expression on the current exchange (where the XPath expression is applied to the body of the current *In* message). The result of the `xpath()` expression is an XML node (or node set, if more than one node matches).

For example, to extract the contents of the `/person/name` element from the current *In* message body and use it to set a header named `user`, you could define a route like the following:

```
from("queue:foo")
    .setHeader("user", xpath("/person/name/text()"))
    .to("direct:tie");
```

Instead of specifying `xpath()` as an argument to `setHeader()`, you can use the fluent builder `xpath()` command—for example:

```
from("queue:foo")
    .setHeader("user").xpath("/person/name/text()")
    .to("direct:tie");
```

If you want to convert the result to a specific type, specify the result type as the second argument of `xpath()`. For example, to specify explicitly that the result type is `String`:

```
xpath("/person/name/text()", String.class)
```

Namespaces

Typically, XML elements belong to a schema, which is identified by a namespace URI. When processing documents like this, it is necessary to associate namespace URIs with prefixes, so that you can identify element names unambiguously in your XPath expressions. Fuse Mediation Router provides the helper class, `org.apache.camel.builder.xml.Namespaces`, which enables you to define associations between namespaces and prefixes.

For example, to associate the prefix, `cust`, with the namespace, `http://acme.com/customer/record`, and then extract the contents of the element, `/cust:person/cust:name`, you could define a route like the following:

```
import org.apache.camel.builder.xml.Namespaces;
...
Namespaces ns = new Namespaces("cust", "http://acme.com/cus
```

```
tomer/record");

from("queue:foo")
   .setHeader("user", xpath("/cust:person/cust:name/text()",
 ns))
    .to("direct:tie");
```

Where you make the namespace definitions available to the `xpath()` expression builder by passing the `Namespaces` object, `ns`, as an additional argument. If you need to define multiple namespaces, use the `Namespace.add()` method, as follows:

```
import org.apache.camel.builder.xml.Namespaces;
...
Namespaces ns = new Namespaces("cust", "http://acme.com/cus
tomer/record");
ns.add("inv", "http://acme.com/invoice");
ns.add("xsi", "http://www.w3.org/2001/XMLSchema-instance");
```

If you need to specify the result type *and* define namespaces, you can use the three-argument form of `xpath()`, as follows:

```
xpath("/person/name/text()", String.class, ns)
```

Spring DSL

Basic expressions

To evaluate an XPath expression in the Spring DSL, put the XPath expression inside an `xpath` element. The XPath expression is applied to the body of the current *In* message and returns an XML node (or node set). Typically, the returned XML node is automatically converted to a string.

For example, to extract the contents of the `/person/name` element from the current *In* message body and use it to set a header named `user`, you could define a route like the following:

```
<beans ...>

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

    <route>
      <from uri="queue:foo"/>
      <setHeader headerName="user">
        <xpath>/person/name/text()</xpath>
      </setHeader>
      <to uri="direct:tie"/>
    </route>
  </camelContext>

</beans>
```

If you want to convert the result to a specific type, specify the result type by setting the `resultType` attribute to a Java type name (where you must specify the fully-qualified type name). For example, to specify explicitly that the result type is `String`:

```
<xpath resultType="java.lang.String">/per
son/name/text()</xpath>
```

Namespaces

When processing documents whose elements belong to one or more XML schemas, it is typically necessary to associate namespace URIs with prefixes, so that you can identify element names unambiguously in your XPath expressions. Because Spring DSL is itself written in XML, it is possible to use the standard XML mechanism for associating prefixes with namespace URIs. That is, you can set an attribute like this: `xmlns:`*`Prefix`*`="`*`NamespaceURI`*`"`.

For example, to associate the prefix, `cust`, with the namespace, `http://acme.com/customer/record`, and then extract the contents of the

element, `/cust:person/cust:name`, you could define a route like the following:

```
<beans ...>

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

                xmlns:cust="http://acme.com/customer/record"
 >
    <route>
      <from uri="queue:foo"/>
      <setHeader headerName="user">
        <xpath>/cust:person/cust:name/text()</xpath>
      </setHeader>
      <to uri="direct:tie"/>
    </route>
  </camelContext>

</beans>
```

XPath Injection

Parameter binding annotation

When using Fuse Mediation Router bean integration to invoke a method on a Java bean, you can use the `@XPath` annotation to extract a value from the exchange and bind it to a method parameter.

For example, consider the following route fragment, which invokes the `credit` method on an `AccountService` object:

```
from("queue:payments")
    .beanRef("accountService","credit")
    ...
```

The `credit` method uses parameter binding annotations to extract relevant data from the message body and inject it into its parameters, as follows:

```
public class AccountService {
    ...
    public void credit(
            @XPath("/transaction/transfer/receiver/text()")
String name,
            @XPath("/transaction/transfer/amount/text()")
String amount
            )
    {
        ...
    }
    ...
}
```

For more information about bean integration, see "Bean Integration" in *Implementing Enterprise Integration Patterns*.

Namespaces

Table 4.1 on page 52 shows the namespaces that are predefined for XPath. You can use these namespace prefixes in the `XPath` expression that appears in the `@XPath` annotation.

Table 4.1. Predefined Namespaces for @XPath

Namespace URI	Prefix
`http://www.w3.org/2001/XMLSchema`	`xsd`
`http://www.w3.org/2003/05/soap-envelope`	`soap`

It is not possible to add custom namespaces to use in the `@XPath` annotation. If you need to access your own custom namespaces, however, you could implement your own custom annotation, `@MyXPath` (you can look at the source code for `org.apache.camel.language.@XPath` to see how the annotation is implemented).

XPath Builder

Overview

The `org.apache.camel.builder.xml.XPathBuilder` class enables you to evaluate XPath expressions independently of an exchange. That is, if you have an XML fragment from any source, you can use `XPathBuilder` to evaluate an XPath expression on the XML fragment.

Matching expressions

Use the `matches()` method to check whether one or more XML nodes can be found that match the given XPath expression. The basic syntax for matching an XPath expression using `XPathBuilder` is as follows:

```
boolean matches = XPathBuilder
                    .xpath("Expression")
                    .matches(CamelContext, "XMLString");
```

Where the given expression, *`Expression`*, is evaluated against the XML fragment, *`XMLString`*, and the result is true, if at least one node is found that matches the expression. For example, the following example returns `true`, because the XPath expression finds a match in the `xyz` attribute.

```
boolean matches = XPathBuilder
                    .xpath("/foo/bar/@xyz")
                    .matches(getContext(), "<foo><bar
xyz='cheese'/></foo>"));
```

Evaluating expressions

Use the `evaluate()` method to return the contents of the first node that matches the given XPath expression. The basic syntax for evaluating an XPath expression using `XPathBuilder` is as follows:

```
String nodeValue = XPathBuilder
                    .xpath("Expression")
                    .evaluate(CamelContext, "XMLString");
```

You can also specify the result type by passing the required type as the second argument to `evaluate()`—for example:

```
String name = XPathBuilder
                   .xpath("foo/bar")
                   .evaluate(context,
"<foo><bar>cheese</bar></foo>", String.class);
Integer number = XPathBuilder
                   .xpath("foo/bar")
                   .evaluate(context,
```

```
"<foo><bar>123</bar></foo>", Integer.class);
Boolean bool = XPathBuilder
                   .xpath("foo/bar")
                   .evaluate(context,
"<foo><bar>true</bar></foo>", Boolean.class);
```

Using the Saxon parser

A prerequisite for using the Saxon parser is that you add a dependency on the `camel-saxon` artifact (either adding this dependency to your Maven POM, if you use Maven, or adding the `camel-saxon-2.7.0-fuse-00-00.jar` file to your classpath, otherwise).

The simplest way to enable the Saxon parser is to call the `saxon()` fluent builder method. For example, you could invoke the Saxon parser as shown in the following example:

```
// Java
// create a builder to evaluate the xpath using saxon
XPathBuilder builder = XPathBuilder.xpath("tokenize(/foo/bar,
 '_')[2]").saxon();

// evaluate as a String result
String result = builder.evaluate(context,
"<foo><bar>abc_def_ghi</bar></foo>");
```

Expressions

Result type

By default, an XPath expression returns a list of one or more XML nodes, of `org.w3c.dom.NodeList` type. You can use the type converter mechanism to convert the result to a different type, however. In the Java DSL, you can specify the result type in the second argument of the `xpath()` command. For example, to return the result of an XPath expression as a `String`:

```
xpath("/person/name/text()", String.class)
```

In the Spring DSL, you can specify the result type in the `resultType` attribute, as follows:

```
<xpath resultType="java.lang.String">/per
son/name/text()</xpath>
```

Patterns in location paths

You can use the following patterns in XPath location paths:

`/people/person`

The basic location path specifies the nested location of a particular element. That is, the preceding location path would match the person element in the following XML fragment:

```
<people>
  <person>...</person>
</people>
```

Note that this basic pattern can match *multiple* nodes—for example, if there is more than one `person` element inside the `people` element.

`/name/text()`

If you just want to access the *text* inside by the element, append `/text()` to the location path, otherwise the node includes the element's start and end tags (and these tags would be included when you convert the node to a string).

`/person/telephone/@isDayTime`

To select the value of an attribute, *`AttributeName`*, use the syntax `@`*`AttributeName`*. For example, the preceding location path returns `true` when applied to the following XML fragment:

```
<person>
  <telephone isDayTime="true">1234567890</telephone>
</person>
```

`*`

A wildcard that matches all elements in the specified scope. For example, `/people/person/*` matches all the child elements of `person`.

`@*`

A wildcard that matches all attributes of the matched elements. For example, `/person/name/@*` matches all attributes of every matched `name` element.

`//`

Match the location path at every nesting level. For example, the `//name` pattern matches every `name` element highlighted in the following XML fragment:

```
<invoice>
  <person>
    <name .../>
  </person>
</invoice>
<person>
  <name .../>
</person>
<name .../>
```

`..`

Selects the parent of the current context node. Not normally useful in the Fuse Mediation Router XPath language, because the current context node is the document root, which has no parent.

`node()`

Match any kind of node.

`text()`

Match a text node.

`comment()`

Match a comment node.

`processing-instruction()`

Match a processing-instruction node.

Predicate filters

You can filter the set of nodes matching a location path by appending a predicate in square brackets, `[Predicate]`. For example, you can select the N^{th} node from the list of matches by appending `[N]` to a location path. The following expression selects the first matching `person` element:

```
/people/person[1]
```

The following expression selects the second-last `person` element:

```
/people/person[last()-1]
```

You can test the value of attributes in order to select elements with particular attribute values. The following expression selects the `name` elements, whose `surname` attribute is either Strachan or Davies:

```
/person/name[@surname="Strachan" or @surname="Davies"]
```

You can combine predicate expressions using any of the conjunctions `and`, `or`, `not()`, and you can compare expressions using the comparators, `=`, `!=`, `>`, `>=`, `<`, `<=` (in practice, the less-than symbol must be replaced by the `<` entity). You can also use XPath functions in the predicate filter.

Axes

When you consider the structure of an XML document, the root element contains a sequence of children, and some of those child elements contain further children, and so on. Looked at in this way, where nested elements are linked together by the *child-of* relationship, the whole XML document has the structure of a *tree*. Now, if you choose a particular node in this element tree (call it the *context node*), you might want to refer to different parts of the tree relative to the chosen node. For example, you might want to refer to the children of the context node, to the parent of the context node, or to all of the nodes that share the same parent as the context node (*sibling nodes*).

An *XPath axis* is used to specify the scope of a node match, restricting the search to a particular part of the node tree, relative to the current context node. The axis is attached as a prefix to the node name that you want to match, using the syntax, `AxisType::MatchingNode`. For example, you can use the `child::` axis to search the children of the current context node, as follows:

```
/invoice/items/child::item
```

The context node of `child::item` is the `items` element that is selected by the path, `/invoice/items`. The `child::` axis restricts the search to the children of the context node, `items`, so that `child::item` matches the children of `items` that are named `item`. As a matter of fact, the `child::` axis is the default axis, so the preceding example can be written equivalently as:

```
/invoice/items/item
```

But there several other axes (13 in all), some of which you have already seen in abbreviated form: `@` is an abbreviation of `attribute::`, and `//` is an abbreviation of `descendant-or-self::`. The full list of axes is as follows (for details consult the reference below):

- `ancestor`
- `ancestor-or-self`
- `attribute`
- `child`
- `descendant`
- `descendant-or-self`
- `following`
- `following-sibling`
- `namespace`
- `parent`
- `preceding`
- `preceding-sibling`

- `self`

Functions

XPath provides a small set of standard functions, which can be useful when evaluating predicates. For example, to select the last matching node from a node set, you can use the last() function, which returns the index of the last node in a node set, as follows:

```
/people/person[last()]
```

Where the preceding example selects the last `person` element in a sequence (in document order).

For full details of all the functions that XPath provides, consult the reference below.

Reference

For full details of the XPath grammar, see the XML Path Language, Version 1.0[1] specification.

[1] http://www.w3.org/TR/xpath/

Predicates

Basic predicates

You can use `xpath` in the Java DSL or the Spring DSL in a context where a predicate is expected—for example, as the argument to a `filter()` processor or as the argument to a `when()` clause.

For example, the following route filters incoming messages, allowing a message to pass, only if the `/person/city` element contains the value, `London`:

```
from("direct:tie")
    .filter().xpath("/person/city = 'London'").to("file:tar
get/messages/uk");
```

The following route evaluates the XPath predicate in a `when()` clause:

```
from("direct:tie")
    .choice()
        .when(xpath("/person/city = 'London'")).to("file:tar
get/messages/uk")
        .otherwise().to("file:target/messages/others");
```

XPath predicate operators

The XPath language supports the standard XPath predicate operators, as shown in Table 4.2 on page 61.

Table 4.2. Operators for the XPath Language

Operator	Description
`=`	Equals.
`!=`	Not equal to.
`>`	Greater than.
`>=`	Greater than or equals.
`<`	Less than.
`<=`	Less than or equals.
`or`	Combine two predicates with logical *and*.
`and`	Combine two predicates with logical *inclusive or*.
`not()`	Negate predicate argument.

Using Variables and Functions

Evaluating variables in a route

When evaluating XPath expressions inside a route, you can use XPath variables to access the contents of the current exchange, as well as O/S environment variables and Java system properties. The syntax to access a variable value is `$VarName` or `$Prefix:VarName`, if the variable is accessed through an XML namespace.

For example, you can access the *In* message's body as `$in:body` and the *In* message's header value as `$in:HeaderName`. O/S environment variables can be accessed as `$env:EnvVar` and Java system properties can be accessed as `$system:SysVar`.

In the following example, the first route extracts the value of the `/person/city` element and inserts it into the `city` header. The second route filters exchanges using the XPath expression, `$in:city = 'London'`, where the `$in:city` variable is replaced by the value of the `city` header.

```
from("file:src/data?noop=true")
    .setHeader("city").xpath("/person/city/text()")
    .to("direct:tie");

from("direct:tie")
   .filter().xpath("$in:city = 'London'").to("file:target/mes
sages/uk");
```

Evaluating functions in a route

In addition to the standard XPath functions, the XPath language defines additional functions. These additional functions (which are listed in Table 4.4 on page 65) can be used to access the underlying exchange, to evaluate a simple expression or to look up a property in the Fuse Mediation Router property placeholder component.

For example, the following example uses the `in:header()` function and the `in:body()` function to access a head and the body from the underlying exchange:

```
from("direct:start").choice()
  .when().xpath("in:header('foo') = 'bar'").to("mock:x")
  .when().xpath("in:body() = '<two/>'").to("mock:y")
  .otherwise().to("mock:z");
```

Notice the similarity between theses functions and the corresponding `in:HeaderName` or `in:body` variables. The functions have a slightly different

syntax however: `in:header('HeaderName')` instead of `in:HeaderName`; and `in:body()` instead of `in:body`.

Evaluating variables in XPathBuilder

You can also use variables in expressions that are evaluated using the `XPathBuilder` class. In this case, you cannot use variables such as `$in:body` or `$in:HeaderName`, because there is no exchange object to evaluate against. But you *can* use variables that are defined inline using the `variable(Name, Value)` fluent builder method.

For example, the following XPathBuilder construction evaluates the `$test` variable, which is defined to have the value, `London`:

```
String var = XPathBuilder.xpath("$test")
               .variable("test", "London")
               .evaluate(getContext(), "<name>foo</name>");
```

Note that variables defined in this way are automatically entered into the global namespace (for example, the variable, `$test`, uses no prefix).

Variable Namespaces

Table of namespaces

Table 4.3 on page 64 shows the namespace URIs that are associated with the various namespace prefixes.

Table 4.3. XPath Variable Namespaces

Namespace URI	Prefix	Description
`http://camel.apache.org/schema/spring`	*None*	Default namespace (associated with variables that have no namespace prefix).
`http://camel.apache.org/xml/in/`	`in`	Used to reference header or body of the current exchange's *In* message.
`http://camel.apache.org/xml/out/`	`out`	Used to reference header or body of the current exchange's *Out* message.
`http://camel.apache.org/xml/functions/`	`functions`	Used to reference some custom functions.
`http://camel.apache.org/xml/variables/environment-variables`	`env`	Used to reference O/S environment variables.
`http://camel.apache.org/xml/variables/system-properties`	`system`	Used to reference Java system properties.
`http://camel.apache.org/xml/variables/exchange-property`	*Undefined*	Used to reference exchange properties. You must define your own prefix for this namespace.

Function Reference

Table of custom functions

Table 4.4 on page 65 shows the custom functions that you can use in Fuse Mediation Router XPath expressions. These functions can be used in addition to the standard XPath functions.

Table 4.4. XPath Custom Functions

Function	Description
`in:body()`	Returns the *In* message body.
`in:header(HeaderName)`	Returns the *In* message header with name, `HeaderName`.
`out:body()`	Returns the *Out* message body.
`out:header(HeaderName)`	Returns the *Out* message header with name, `HeaderName`.
`function:properties(PropKey)`	Looks up a property with the key, `PropKey` (see "Property Placeholders" in *Implementing Enterprise Integration Patterns*).
`function:simple(SimpleExp)`	Evaluates the specified simple expression, `SimpleExp`.