XPath

XPath

Camel supports XPath to allow an Expression or Predicate to be used in the DSL or Xml Configuration. For example you could use XPath to create an Predicate in a Message Filter or as an Expression for a Recipient List.

Streams
If the message body is stream based, which means the input is received by Camel as a stream, then you will only be able to read the content of the stream **once**. Oftentimes when using XPath as Message Filter or Content Based Router the data will be accessed multiple times. Therefore use Stream caching or convert the message body to a **`String`** beforehand. This makes it safe to be re-read multiple times.
from("queue:foo") .filter().xpath("//foo")) .to("queue:bar") from("queue:foo") .choice().xpath("//foo")).to("queue:bar") .otherwise().to("queue:others");

Namespaces

You can easily use namespaces with XPath expressions using the Namespaces helper class.{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/XPathWithNamespacesFilterTest.java}

Variables

Variables in XPath is defined in different namespaces. The default namespace is `http://camel.apache.org/schema/spring`.

Namespace URI	Local part	Type	Description
http://camel.apache.org/xml/in/	`in`	`Message`	The **`exchange.in`** message.
http://camel.apache.org/xml/out/	`out`	`Message`	The **`exchange.out`** message.
http://camel.apache.org/xml/function/	`functions`	`Object`	**Camel 2.5:** Additional functions.
http://camel.apache.org/xml/variables/environment-variables	`env`	`Object`	OS environment variables.
http://camel.apache.org/xml/variables/system-properties	`system`	`Object`	Java System properties.
http://camel.apache.org/xml/variables/exchange-property		`Object`	The exchange property.

Camel will resolve variables according to either:

- namespace given
- no namespace given

Namespace Given

If the namespace is given then Camel is instructed exactly what to return. However when resolving either **`IN`** or **`OUT`** Camel will try to resolve a header with the given local part first, and return it. If the local part has the value **`body`** then the body is returned instead.

No Namespace Given

If there is no namespace given then Camel resolves only based on the local part. Camel will try to resolve a variable in the following steps:

- From **`variables`** that has been set using the **`variable(name, value)`** fluent builder.
- From **`message.in.header`** if there is a header with the given key.
- From **`exchange.properties`** if there is a property with the given key.

Functions

Camel adds the following XPath functions that can be used to access the exchange:

Function	Argument	Type	Description
`in:body`	none	`Object`	Will return the **`IN`** message body.
`in:header`	the header name	`Object`	Will return the **`IN`** message header.
`out:body`	none	`Object`	Will return the **`OUT`** message body.
`out:header`	the header name	`Object`	Will return the **`OUT`** message header.
`function:properties`	key for property	`String`	**Camel 2.5:** To lookup a property using the Properties component (property placeholders).
`function:simple`	simple expression	`Object`	**Camel 2.5:** To evaluate a Simple expression.

Note: `function:properties` and **`function:simple`** is not supported when the return type is a **`NodeSet`**, such as when using with a Splitter EIP.

Here's an example showing some of these functions in use.{snippet:id=ex|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/language/XPathFunctionTest.java}And the new functions introduced in Camel 2.5:{snippet:id=ex|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/xml/XPathFunctionsTest.java}

Using XML Configuration

If you prefer to configure your routes in your Spring XML file then you can use XPath expressions as follows

xml<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd"> <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring" xmlns:foo="http://example.com/person"> <route> <from uri="activemq:MyQueue"/> <filter> <xpath>/foo:person[@name='James']</xpath> <to uri="mqseries:SomeOtherQueue"/> </filter> </route> </camelContext> </beans>

Notice how we can reuse the namespace prefixes, **`foo`** in this case, in the XPath expression for easier namespace based XPath expressions! See also this discussion on the mailinglist about using your own namespaces with XPath.

Setting the Result Type

The XPath expression will return a result type using native XML objects such as **`org.w3c.dom.NodeList`**. But many times you want a result type to be a **`String`**. To do this you have to instruct the XPath which result type to use.

In Java DSL:

javaxpath("/foo:person/@id", String.class)

In Spring DSL you use the **`resultType`** attribute to provide a fully qualified classname:

xml<xpath resultType="java.lang.String">/foo:person/@id</xpath>

In **`@XPath`**:
Available as of Camel 2.1

java@XPath(value = "concat('foo-',//order/name/)", resultType = String.class) String name)

Where we use the XPath function **`concat`** to prefix the order name with **`foo-`**. In this case we have to specify that we want a **`String`** as result type so the **`concat`** function works.

Using XPath on Headers

Available as of Camel 2.11

Some users may have XML stored in a header. To apply an XPath statement to a header's value you can do this by defining the **`headerName`** attribute.

In XML DSL:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/xpath/XPathHeaderNameTest.xml}And in Java DSL you specify the **`headerName`** as the 2nd parameter as shown:

javaxpath("/invoice/@orderType = 'premium'", "invoiceDetails")

Examples

Here is a simple example using an XPath expression as a predicate in a Message Filter{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/XPathFilterTest.java}If you have a standard set of namespaces you wish to work with and wish to share them across many different XPath expressions you can use the **`NamespaceBuilder`** as shown in this example{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/XPathWithNamespaceBuilderFilterTest.java}In this sample we have a **`choice`** construct. The first choice evaulates if the message has a header key **`type`** that has the value **`Camel`**. The 2nd **`choice`** evaluates if the message body has a name tag **`<name>`** which values is **`Kong`**.
If neither is true the message is routed in the otherwise block:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/builder/xml/XPathHeaderTest.java}And the spring XML equivalent of the route:{snippet:id=example|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringXPathHeaderTest-context.xml}

XPath Injection

You can use Bean Integration to invoke a method on a bean and use various languages such as XPath to extract a value from the message and bind it to a method parameter.

The default XPath annotation has SOAP and XML namespaces available. If you want to use your own namespace URIs in an XPath expression you can use your own copy of the XPath annotation to create whatever namespace prefixes you want to use.{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/component/xslt/MyXPath.java}e.g., cut and paste upper code to your own project in a different package and/or annotation name then add whatever namespace prefix/URIs you want in scope when you use your annotation on a method parameter. Then when you use your annotation on a method parameter all the namespaces you want will be available for use in your XPath expression.

Example:

javapublic class Foo { @MessageDriven(uri = "activemq:my.queue") public void doSomething(@MyXPath("/ns1:foo/ns2:bar/text()") String correlationID, @Body String body) { // process the inbound message here } }

Using XPathBuilder Without an Exchange

Available as of Camel 2.3

You can now use the **`org.apache.camel.builder.XPathBuilder`** without the need for an Exchange. This comes handy if you want to use it as a helper to do custom XPath evaluations. It requires that you pass in a CamelContext since a lot of the moving parts inside the **`XPathBuilder`** requires access to the Camel Type Converter and hence why CamelContext is needed.

For example you can do something like this:

javaboolean matches = XPathBuilder.xpath("/foo/bar/@xyz").matches(context, "<foo><bar xyz='cheese'/></foo>"));

This will match the given predicate.

You can also evaluate for example as shown in the following three examples:

javaString 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);

Evaluating with a String result is a common requirement and thus you can do it a bit simpler:

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

Using Saxon with XPathBuilder

Available as of Camel 2.3

You need to add **`camel-saxon`** as dependency to your project. It's now easier to use Saxon with the **`XPathBuilder`** which can be done in several ways as shown below. Where as the latter ones are the easiest ones.

Using a factory{snippet:id=e1|lang=java|url=camel/trunk/components/camel-saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}Using the object model
{snippet:id=e2|lang=java|url=camel/trunk/components/camel-saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}The easy one{snippet:id=e3|lang=java|url=camel/trunk/components/camel-saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}

Setting a Custom XPathFactory Using System Property

Available as of Camel 2.3

Camel now supports reading the JVM system property `javax.xml.xpath.XPathFactory` that can be used to set a custom **`XPathFactory`** to use.

This unit test shows how this can be done to use Saxon instead:{snippet:id=e4|lang=java|url=camel/trunk/components/camel-saxon/src/test/java/org/apache/camel/builder/saxon/XPathTest.java}Camel will log at **`INFO`** level if it uses a non default **`XPathFactory`** such as:

XPathBuilder INFO Using system property javax.xml.xpath.XPathFactory:http://saxon.sf.net/jaxp/xpath/om with value: net.sf.saxon.xpath.XPathFactoryImpl when creating XPathFactory

To use Apache Xerces you can configure the system property:

-Djavax.xml.xpath.XPathFactory=org.apache.xpath.jaxp.XPathFactoryImpl

Enabling Saxon from Spring DSL

Available as of Camel 2.10

Similarly to Java DSL, to enable Saxon from Spring DSL you have three options:

Specifying the factory

xml<xpath factoryRef="saxonFactory" resultType="java.lang.String">current-dateTime()</xpath>

Specifying the object model

xml<xpath objectModel="http://saxon.sf.net/jaxp/xpath/om" resultType="java.lang.String">current-dateTime()</xpath>

Shortcut

xml<xpath saxon="true" resultType="java.lang.String">current-dateTime()</xpath>

Namespace Auditing to Aid Debugging

Available as of Camel 2.10

A large number of XPath-related issues that users frequently face are linked to the usage of namespaces. You may have some misalignment between the namespaces present in your message and those that your XPath expression is aware of or referencing. XPath predicates or expressions that are unable to locate the XML elements and attributes due to namespaces issues may simply look like "they are not working", when in reality all there is to it is a lack of namespace definition.

Namespaces in XML are completely necessary, and while we would love to simplify their usage by implementing some magic or voodoo to wire namespaces automatically, truth is that any action down this path would disagree with the standards and would greatly hinder interoperability.

Therefore, the utmost we can do is assist you in debugging such issues by adding two new features to the XPath Expression Language and are thus accessible from both predicates and expressions.

Logging the Namespace Context of Your XPath Expression/Predicate

Every time a new XPath expression is created in the internal pool, Camel will log the namespace context of the expression under the **`org.apache.camel.builder.xml.XPathBuilder`** logger. Since Camel represents Namespace Contexts in a hierarchical fashion (parent-child relationships), the entire tree is output in a recursive manner with the following format:

[me: {prefix -> namespace}, {prefix -> namespace}], [parent: [me: {prefix -> namespace}, {prefix -> namespace}], [parent: [me: {prefix -> namespace}]]]

Any of these options can be used to activate this logging:

1. Enable **`TRACE`** logging on the **`org.apache.camel.builder.xml.XPathBuilder`** logger, or some parent logger such as **`org.apache.camel`** or the root logger.
2. Enable the **`logNamespaces`** option as indicated in Auditing Namespaces, in which case the logging will occur on the **`INFO`** level.

AuditingNamespaces

Auditing namespaces

Camel is able to discover and dump all namespaces present on every incoming message before evaluating an XPath expression, providing all the richness of information you need to help you analyse and pinpoint possible namespace issues. To achieve this, it in turn internally uses another specially tailored XPath expression to extract all namespace mappings that appear in the message, displaying the prefix and the full namespace URI(s) for each individual mapping.

Some points to take into account:

- The implicit XML namespace (xmlns:xml="http://www.w3.org/XML/1998/namespace") is suppressed from the output because it adds no value.
- Default namespaces are listed under the **`DEFAULT`** keyword in the output.
- Keep in mind that namespaces can be remapped under different scopes. Think of a top-level 'a' prefix which in inner elements can be assigned a different namespace, or the default namespace changing in inner scopes. For each discovered prefix, all associated URIs are listed.

You can enable this option in Java DSL and Spring DSL.

Java DSL:

javaXPathBuilder.xpath("/foo:person/@id", String.class).logNamespaces()

Spring DSL:

xml<xpath logNamespaces="true" resultType="String">/foo:person/@id</xpath>

The result of the auditing will be appear at the **`INFO`** level under the **`org.apache.camel.builder.xml.XPathBuilder`** logger and will look like the following:

2012-01-16 13:23:45,878 [stSaxonWithFlag] INFO XPathBuilder - Namespaces discovered in message: {xmlns:a=[http://apache.org/camel], DEFAULT=[http://apache.org/default], xmlns:b=[http://apache.org/camelA, http://apache.org/camelB]}

Loading Script from External Resource

Available as of Camel 2.11

You can externalize the script and have Camel load it from a resource such as: **`classpath:`**, **`file:`** or **`http:`**.
This is done using the following syntax: **`resource:scheme:location`**, e.g., to refer to a file on the classpath you can do:

.setHeader("myHeader").xpath("resource:classpath:myxpath.txt", String.class)

Dependencies

The XPath language is part of camel-core.