XSLT

XSLT

The **xslt:** component allows you to process a message using an XSLT template. This can be ideal when using Templating to generate respopnses for requests.

URI format

```
xslt:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template. Refer to the Spring Documentation for more detail of the URI syntax

You can append query options to the URI in the following format, `?option=value&option=value&...`

Here are some example URIs

URI	Description
`xslt:com/acme/mytransform.xsl`	refers to the file com/acme/mytransform.xsl on the classpath
`xslt:file:///foo/bar.xsl`	refers to the file /foo/bar.xsl
`xslt:http://acme.com/cheese/foo.xsl`	refers to the remote http resource

Maven users will need to add the following dependency to their `pom.xml` for this component when using **Camel 2.8** or older:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

From Camel 2.9 onwards the XSLT component is provided directly in the camel-core.

Options

Name	Default Value	Description
`converter`	`null`	Option to override default XmlConverter. Will lookup for the converter in the Registry. The provided converted must be of type org.apache.camel.converter.jaxp.XmlConverter.
`transformerFactory`	`null`	Option to override default TransformerFactory. Will lookup for the transformerFactory in the Registry. The provided transformer factory must be of type javax.xml.transform.TransformerFactory.
`transformerFactoryClass`	`null`	Option to override default TransformerFactory. Will create a TransformerFactoryClass instance and set it to the converter.

uriResolverFactory	DefaultXsltUriResolverFactory	**Camel 2.17**: Reference to a `org.apache.camel.component.xslt.XsltUriResolverFactory` which creates an URI resolver per endpoint. The default implementation returns an instance of `org.apache.camel.component.xslt.DefaultXsltUriResolverFactory` which creates the default URI resolver `org.apache.camel.builder.xml`.XsltUriResolver per endpoint. The default URI resolver reads XSLT documents from the classpath and the file system. This option instead of the option `uriResolver` shall be used when the URI resolver depends on the resource URI of the root XSLT document specified in the endpoint; for example, if you want to extend the default URI resolver. This option is also available on the XSLT component, so that you can set the resource resolver factory only once for all endpoints.
uriResolver	null	**Camel 2.3**: Allows you to use a custom `javax.xml.transformation.URIResolver`. Camel will by default use its own implementation `org.apache.camel.builder.xml.XsltUriResolver` which is capable of loading from classpath.
resultHandlerFactory	null	**Camel 2.3:** Allows you to use a custom `org.apache.camel.builder.xml.ResultHandlerFactory` which is capable of using custom `org.apache.camel.builder.xml.ResultHandler` types.
failOnNullBody	true	**Camel 2.3:** Whether or not to throw an exception if the input body is null.
deleteOutputFile	false	**Camel 2.6:** If you have `output=file` then this option dictates whether or not the output file should be deleted when the Exchange is done processing. For example suppose the output file is a temporary file, then it can be a good idea to delete it after use.
output	string	**Camel 2.3:** Option to specify which output type to use. Possible values are: `string, bytes, DOM, file`. The first three options are all in memory based, where as `file` is streamed directly to a `java.io.File`. For `file` you **must** specify the filename in the IN header with the key `Exchange.XSLT_FILE_NAME` which is also `CamelXsltFileName`. Also any paths leading to the filename must be created beforehand, otherwise an exception is thrown at runtime.
contentCache	true	**Camel 2.6:** Cache for the resource content (the stylesheet file) when it is loaded. If set to `false` Camel will reload the stylesheet file on each message processing. This is good for development. Note: from **Camel 2.9** a cached stylesheet can be forced to reload at runtime via JMX using the `clearCachedStylesheet` operation.
allowStAX		**Camel 2.8.3/2.9:** Whether to allow using StAX as the `javax.xml.transform.Source`. The option is default `false` in Camel 2.11.3/2.12.2 or older. And default `true` in Camel 2.11.4/2.12.3 onwards.
transformerCacheSize	0	**Camel 2.9.3/2.10.1:** The number of `javax.xml.transform.Transformer` object that are cached for reuse to avoid calls to `Template.newTransformer()`.
saxon	false	**Camel 2.11:** Whether to use Saxon as the `transformerFactoryClass`. If enabled then the class `net.sf.saxon.TransformerFactoryImpl`. You would need to add Saxon to the classpath.
saxonExtensionFunctions	null	**Camel 2.17:** Allows to configure one or more custom net.sf.saxon.lib.ExtensionFunctionDefinition. You would need to add Saxon to the classpath. By setting this option, saxon option will be turned out automatically.
errorListener		**Camel 2.14:** Allows to configure to use a custom `javax.xml.transform.ErrorListener`. Beware when doing this then the default error listener which captures any errors or fatal errors and store information on the Exchange as properties is not in use. So only use this option for special use-cases.
entityResolver		**Camel 2.18:** To use a custom org.xml.sax.EntityResolver with javax.xml.transform.sax.SAXSource.

Using XSLT endpoints

For example you could use something like

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl");
```

To use an XSLT template to formulate a response for a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly and consume the message and send it to another destination you could use the following route:

```
from("activemq:My.Queue").
  to("xslt:com/acme/mytransform.xsl").
  to("activemq:Another.Queue");
```

Getting Parameters into the XSLT to work with

By default, all headers are added as parameters which are available in the XSLT.
To do this you will need to declare the parameter so it is then *useable*.

```
<setHeader headerName="myParam"><constant>42</constant></setHeader>
<to uri="xslt:MyTransform.xsl"/>
```

And the XSLT just needs to declare it at the top level for it to be available:

```
<xsl: ...... >

   <xsl:param name="myParam"/>

    <xsl:template ...>
```

Spring XML versions

To use the above examples in Spring XML you would use something like

```
  <camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
      <from uri="activemq:My.Queue"/>
      <to uri="xslt:org/apache/camel/spring/processor/example.xsl"/>
      <to uri="activemq:Another.Queue"/>
    </route>
  </camelContext>
```

There is a test case along with its Spring XML if you want a concrete example.

Using xsl:include

Camel 2.2 or older
If you use xsl:include in your XSL files then in Camel 2.2 or older it uses the default `javax.xml.transform.URIResolver` which means it can only lookup files from file system, and its does that relative from the JVM starting folder.

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

Will lookup the `staff_tempkalte.xsl` file from the starting folder where the application was started.

Camel 2.3 or newer
Now Camel provides its own implementation of `URIResolver` which allows Camel to load included files from the classpath and more intelligent than before.

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

Will now be located relative from the starting endpoint, which for example could be:

```
.to("xslt:org/apache/camel/component/xslt/staff_include_relative.xsl")
```

Which means Camel will locate the file in the **classpath** as `org/apache/camel/component/xslt/staff_template.xsl`.
This allows you to use xsl include and have xsl files located in the same folder such as we do in the example `org/apache/camel/component/xslt`.

You can use the following two prefixes `classpath:` or `file:` to instruct Camel to look either in classpath or file system. If you omit the prefix then Camel uses the prefix from the endpoint configuration. If that neither has one, then classpath is assumed.

You can also refer back in the paths such as

```
    <xsl:include href="../staff_other_template.xsl"/>
```

Which then will resolve the xsl file under `org/apache/camel/component`.

Using xsl:include and default prefix

When using xsl:include such as:

```
<xsl:include href="staff_template.xsl"/>
```

Then in Camel 2.10.3 and older, then Camel will use "classpath:" as the default prefix, and load the resource from the classpath. This works for most cases, but if you configure the starting resource to load from file,

```
.to("xslt:file:etc/xslt/staff_include_relative.xsl")
```

.. then you would have to prefix all your includes with "file:" as well.

```
<xsl:include href="file:staff_template.xsl"/>
```

From Camel 2.10.4 onwards we have made this easier as Camel will use the prefix from the endpoint configuration as the default prefix. So from Camel 2.10.4 onwards you can do:

```
<xsl:include href="staff_template.xsl"/>
```

Which will load the staff_template.xsl resource from the file system, as the endpoint was configured with "file:" as prefix.
You can still though explicit configure a prefix, and then mix and match. And have both file and classpath loading. But that would be unusual, as most people either use file or classpath based resources.

Using Saxon extension functions

Since Saxon 9.2, writing extension functions has been supplemented by a new mechanism, referred to as integrated extension functions you can now easily use camel:

- Java example:

```
SimpleRegistry registry = new SimpleRegistry();
registry.put("function1", new MyExtensionFunction1());
registry.put("function2", new MyExtensionFunction2());

CamelContext context = new DefaultCamelContext(registry);
context.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
            .to("xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?
saxonExtensionFunctions=#function1,#function2");
    }
});
```

Spring example:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:extensions"/>
    <to uri="xslt:org/apache/camel/component/xslt/extensions/extensions.xslt?saxonExtensionFunctions=#function1,
#function2"/>
  </route>
</camelContext>


<bean id="function1" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction1"/>
<bean id="function2" class="org.apache.camel.component.xslt.extensions.MyExtensionFunction2"/>
```

Dynamic stylesheets

To provide a dynamic stylesheet at runtime you can define a dynamic URI. See How to use a dynamic URI in to() for more information.

Available as of Camel 2.9 (removed in 2.11.4, 2.12.3 and 2.13.0)
Camel provides the `CamelXsltResourceUri` header which you can use to define a stylesheet to use instead of what is configured on the endpoint URI. This allows you to provide a dynamic stylesheet at runtime.

Accessing warnings, errors and fatalErrors from XSLT ErrorListener

Available as of Camel 2.14

From Camel 2.14 onwards, any warning/error or fatalError is stored on the current Exchange as a property with the keys `Exchange.XSLT_ERROR`, `Exchange.XSLT_FATAL_ERROR`, or `Exchange.XSLT_WARNING` which allows end users to get hold of any errors happening during transformation.

For example in the stylesheet below, we want to terminate if a staff has an empty dob field. And to include a custom error message using xsl:message.

```
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="staff/programmer">
          <p>Name: <xsl:value-of select="name"/><br />
            <xsl:if test="dob=''">
              <xsl:message terminate="yes">Error: DOB is an empty string!</xsl:message>
            </xsl:if>
          </p>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
```

This information is not available on the Exchange stored as an Exception that contains the message in the `getMessage()` method on the exception. The exception is stored on the Exchange as a warning with the key `Exchange.XSLT_WARNING.`

Notes on using XSLT and Java Versions

Here are some observations from Sameer, a Camel user, which he kindly shared with us:

> *In case anybody faces issues with the XSLT endpoint please review these points.*
>
> *I was trying to use an xslt endpoint for a simple transformation from one xml to another using a simple xsl. The output xml kept appearing (after the xslt processor in the route) with outermost xml tag with no content within.*
>
> *No explanations show up in the DEBUG logs. On the TRACE logs however I did find some error/warning indicating that the XMLConverter bean could no be initialized.*
>
> *After a few hours of cranking my mind, I had to do the following to get it to work (thanks to some posts on the users forum that gave some clue):*

> *1. Use the transformerFactory option in the route (`"xslt:my-transformer.xsl?transformerFactory=tFactory"`) with the `tFactory` bean having bean defined in the spring context for `class="org.apache.xalan.xsltc.trax.TransformerFactoryImpl"`.*
> *2. Added the Xalan jar into my maven pom.*
>
> *My guess is that the default xml parsing mechanism supplied within the JDK (I am using 1.6.0_03) does not work right in this context and does not throw up any error either. When I switched to Xalan this way it works. This is not a Camel issue, but might need a mention on the xslt component page.*
>
> *Another note, jdk 1.6.0_03 ships with JAXB 2.0 while Camel needs 2.1. One workaround is to add the 2.1 jar to the `jre/lib/endorsed` directory for the jvm or as specified by the container.*
>
> *Hope this post saves newbie Camel riders some time.*

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started