JAX-RS Advanced XML

JAX-RS : Advanced XML

XPath support

XPath is supported on the server and client sides with the help of XMLSource utility and XPathProvider provider classes. The difference between the two is that XPathProvider allows for specifying the XPath expressions in the external configuration.

Introduction

XMLSource allows for converting matched XML DOM element, attribute and text nodes into typed complex and primitive classes. DOM Node and Element classes as well as JAXP Source and DOMSource can also be used. A single XMLSource instance can be used to query the same input source multiple times if the buffering mode is enabled.

Here are some examples:

```
InputStream is = new ByteArrayInputStream("<foo><bar attr=\"3\">barValue</bar></foo>".getBytes());
XMLSource xp = new XMLSource(is);
xp.setBuffering(true);
// query 1
String value = xp.getValue("/foo/bar/@attr");
assertEquals("3", value);

// query 2
Integer intValue = xp.getNode("/foo/bar/@attr", Integer.class);
assertEquals(3, intValue);

// query 3
Node node = xp.getNode("/foo/bar/@attr", Node.class);
assertEquals("3", node.getTextValue());
```

In the above example a primitive attribute node is accessed in a number of ways, using the same XMLSource instance.
Matched XML complex (element) nodes can be converted in a similar way:

```
public class Bar {
    @XmlAttribute
    private String attr;
    public String getAttribute() {
        return attr;
    }
}

InputStream is = new ByteArrayInputStream("<foo><bar attr=\"3\">barValue</bar></foo>".getBytes());
XMLSource xp = new XMLSource(is);
xp.setBuffering(true);

// query 1
Bar bean = xp.getNode("/foo/bar", Bar.class);
assertEquals("3", bean.getAttribute());

// query 2
String value = xp.getValue("/foo/bar");
assertEquals("<bar attr=\"3\">barValue</bar>", value);
```

Note that JAXB is used internally to convert the matched XML element into a class like Bar which does not have to have an @XmlRootElement annotation. The 2nd query in the above example how a String representation of the matched complex node can be captured.

XMLSource also provides methods for capturing multiple nodes, example:

```
InputStream is = new ByteArrayInputStream("<foo><bar attr=\"3\">value1</bar><bar attr=\"4\">value2</bar></foo>".
getBytes());
XMLSource xp = new XMLSource(is);
xp.setBuffering(true);
// query 1
String[] values = xp.getValue("/foo/bar/text()");
assertEquals("value1", values[0]);
assertEquals("value2", values[1]);

// query 2
Integer[] intValues = xp.getNodes("/foo/bar/@attr", Integer.class);
assertEquals(3, intValues[0]);
assertEquals(4, intValues[1]);

// query 3
Bar[] nodes = xp.getNodes("/foo/bar", Bar.class);
```

All the above examples have been simplified in that no namespaces have been used. Most real XML instances will have plenty of them and XMLSource has methods accepting optional maps containing prefixes as keys and namespaces as values:

```
InputStream is = new ByteArrayInputStream("<foo xmlns=\"http://foo\"><ns1:bar xmlns:ns1=\"http://bar\" attr=\"
3\">barValue</bar></foo>".getBytes());
XMLSource xp = new XMLSource(is);
xp.setBuffering(true);

Foo foo = xp.getNode("/ps1:foo", Collections.singletonMap("ps1", "http://foo"), Foo.class);
assertNotNull(foo);

Bar foo = xp.getNode("/ps2:bar", Collections.singletonMap("ps2", "http://bar"), Bar.class);
assertNotNull(foo);
```

In the above example, a default "http://foo" namespace qualifies the root "foo" element while its 'bar' children are qualified with the "http://bar" namespace. Note that XMLSource is configured to make sure these namespaces are checked but the prefixes used do not have to match the ones in the actual XML instance which may not even have them at all, for example, when default namespaces are used.

XMLSource also provides few methods for capturing attribute or text values representing the HTTP links:

```
String xmlString = "<customers xmlns=\"http://customers\">
                   + "<customer id="1" homepage=\"http://customers/1\"/>"
                   + "</customers>";
InputStream is = new ByteArrayInputStream(xmlString.getBytes());
XMLSource xp = new XMLSource(is);

URI homePage = xp.getLink("/ps1:customer[@id='1']/@homePage",
                          Collections.singletonMap("ps1", "http://customers"));
WebClient client = WebClient.create(homePage);
// access the home page
```

In the above example the link to the home page of the customer with id equal to '1' is retrieved and is used to create a WebClient instance.

In some cases, the links are relative. In such cases the base URI is already either known to the application code or it may be specified as the value of the xml:base attribute. In the latter case XMLSource makes it easy to get this base URI:

```
String xmlString = "<customers xmlns=\"http://customers\" xml:base="http://customers">
                   + "<customer id="1"/><customer id="2"/>"
                   + "</customers>";
InputStream is = new ByteArrayInputStream(xmlString.getBytes());
XMLSource xp = new XMLSource(is);
xp.setBuffering(true);

URI baseURI = xp.getBaseURI();
URI[] relativeURIs = xp.getLinks("/ps1:customer/@id", Collections.singletonMap("ps1", "http://customers"));

WebClient client = WebClient.create(baseURI);
for (URI uri: relativeURIs) {
  client.path(uri);
  // access the home page

  // and get back to the base URI
  client.back(true);
}
```

In the last example, a customer id also represents a relative URI, with xml:base attribute pointing to a base URI.
XMLSource is used to get all the relative URIs, creates a WebClient with the base URI and then iterates using individual relative URIs.

Using XMLSource and XPathProvider in the application code

Please see this section on how http-centric WebClients can use XPath, and here is an example for the server side:

```
@Path("/root")
public class Root {
   @POST
   public void post(XMLSource source) {
       String value = source.getProperty("/books/book/@name");
   }
}
```

Users have an option to hide XPath expressions, by registering an XPathProvider which is a JAX-RS MessageBodyReader, either on the client or server sides. For example:

```
XPathProvider provider = new XPathProvider();
provider.setGlobalExpression("/books/book[position() = 1]");
WebClient wc = WebClient.create("http://aggregated/data", Collections.singletonList(provider));
Book b = wc.get(Book.class);
```

XSLT support

XSLT is currently supported by XSLTJaxbProvider. This provider works relies on JAXB to initiate the transformation process and can be used to produce pretty much any format, including non-XML ones. Likewise, it can be used to extract XML data out of incoming XML fragments. This provider can be used on the server and client sides.

Please see this blog entry for an overview of how this provider can help with separating the presentation logic from the main application code.

XSLTJaxbProvider can be configured to handle input or output data, scoped by media types if needed. For example, one may configure it such that one template handles "application/xml" formats only while the other one handles "application/json" writes only.

XSLTJaxbProvider uses an injected JAX-RS UriInfo to inject all the usual JAX-RS information like template or query parameters into a given XSLT template. These parameters will be injected with the names equal to corresponding @PathParam, QueryParam, etc values. Additionally, an absolute, base and relative path URIs will be injected as "absolute.path", "base.path" and "relative.path" parameters. Finally, custom inParameters and outParameters map properties can also be injected and they will be added to all the XSLT template instances.

For example, given this resource method definition:

```
@Path("/root")
public class Root {
   @GET
   @Path("{id}")
   @Produces({"application/xml", "application/json", "text/html"} )
   public Book get(@PathParam("id") String id, @QueryParam("name") String name) {
       return getBook(id, name);
   }
}
```

the XSLT templates will have parameters with name 'id' and 'name' injected. In this particular case it may make sense to have two templates, one for processing a Book XML stream (generated by JAXB) only, and the other one for adding HTML specific elements. The HTML-specific template will most likely import the template dealing with the Book stream.

Here are some examples of how XSLTJaxbTemplate can be configured:

```
   <map id="outTemplates">
      <entry key="application/xml" value="classpath:/WEB-INF/templates/book-xml.xsl"/>
      <entry key="text/html" value="classpath:/WEB-INF/templates/book-html.xsl"/>
      <entry key="application/json" value="classpath:/WEB-INF/templates/book-json.xsl"/>
  </map>

  <bean id="uriResolver" class="org.apache.cxf.systest.jaxrs.URIResolverImpl"/>

  <bean id="xsltProvider" class="org.apache.cxf.jaxrs.provider.XSLTJaxbProvider">
      <property name="outMediaTemplates" ref="outTemplates"/>
      <property name="resolver" ref="uriResolver"/>
  </bean>
```

In this example, the provider is injected with three out templates, one for modifying the Book XML stream, the other one - for creating an HTML Book representation and the last one for writing a JSON sequence which the existing JSON providers may not be able to generate.

A 'uriResolver' property can be used to point to a custom javax.xml.transform.URIResolver implementation which can be used to resolve relative links in few XSLT instructions such as 'import'/etc. A 'systemId' property can also be set. Additionally, inProperties and outProperties map properties can be injected and they will be used when XSLT template instances will be created.

Here is a simpler configuration example:

```
   <bean id="xsltProvider" class="org.apache.cxf.jaxrs.provider.XSLTJaxbProvider">
      <property name="outTemplate" value="classpath:/WEB-INF/templates/book-xml.xsl"/>
      <property name="inTemplate" class="classpath:/WEB-INF/templates/fromNewBookToOldBook.xsl"/>
  </bean>
```

The provider is configured with one output template and one input template which will apply to all request and response data.

When XSLTJaxbProvider is used for processing the incoming requests, its main goal is to ensure the incoming XML can be adapted as needed for the JAXB unmarshalling to succeed. The in template can modify the incoming XML in a number of ways or it can extract the XML fragment out of the bigger XML instance which is similar to what XMLSource and XPathProvider can do as well. Please also check the new Transformation feature.

Note that XSLTJaxbProvider may be used with other XML-aware providers on the same endpoint if needed. In such cases one needs to restrict the set of classes it can handle, for it to not interfere with the other XML provider. This can be done using inClassNames and outClassNames list properties which enumerate supported class names. Alternatively, a new "supportJaxbOnly" property can be set, in which case XSLTJaxbProvider will simply delegate to JAXBElementProvider if no template is available for a given class.

Note that when XSLTJaxbProvider is used on the client side, it may not always be possible for template parameters be injected in cases when http-centric clients are used (as opposed to proxies). For example :

```
WebClient client = WebClient.create("http://books");
client.path("/store/1").get();
```

it is not possible to deduce that '1' represents a template parameter in the "/store/1" expression. However, one can use the following code instead if '1' needs to be available to XSLT templates :

```
WebClient client = WebClient.create("http://books");
client.path("/store/{id}", 1).get();
```

XML Processing Instructions

One way to get outbound XML transformed to HTML or get an XHTML payload further decorated with CSS tags is to associate an xml-stylesheet processing instruction with the XML payload, for example:

```
<?xml-stylesheet type="text/xsl" href="http://localhost/myapp/stylesheets/toHTML.xsl"?>
<products xmlns="http://products">
   <product id="1"/>
</products>
```

The browser will fetch a stylesheet from the server and will transform this XML on the client's machine.
This option is alternative to using XSLTJaxbProvider to create HTML on the server side. It may become less easy to use if XSL stylesheets importing other documents or rely on additional parameters to produce the HTML but it can definitely be used to offload the cost of doing the transformations from the server in some cases.

You can also combine XSLTJaxbProvider to create complex HTML on the server and use xml-stylesheet instructions to get the browser to download and cache the CSS stylesheets, for example:

```
<?xml-stylesheet type="text/css" href="http://localhost/myapp/stylesheets/HTMLDecorator.css"?>
<!
   XSLTJaxbProvider transformed the XML products payload into well-formed XHTML.
   The browser will get HTMLDecorator.css at the next step and apply it to this HTML
-->
<html>
   <title>The products</title>
   <table>
     <!-- description of products -->
   </table>
</html>
```

When working with JAXB, the way to add such xml processing instructions is to set a "com.sun.xml.bind.xmlHeaders" or "com.sun.xml.internal.bind.xmlHeaders" on the JAXB Marshaller.

Up until CXF 2.5.1 the way to do is to use a JAXBElementProvider 'marshallerProperties' map property, for example:

```
<beans xmlns:util="http://www.springframework.org/schema/util">
<bean id="jaxbProvider" class="org.apache.cxf.jaxrs.provider.JAXBElementProvider">
<map>
<entry key="com.sun.xml.bind.xmlHeaders"
       value="<?xml-stylesheet type='text/xsl' href='/stylesheets/toHTML.xsl'?>"/>
</map>
</bean>
</beans>
```

The problem here is how to correctly point to an absolute URI identifying the toHTML.xsl resource.
Assuming that the browser will correctly resolve a relative URI such as "/stylesheets/toHTML.xsl", then all is set.
Otherwise the best option is to extend JAXBElementProvider and set marshaller properties there by using JAX-RS UriInfo to get to the base request URI.

Starting from CXF 2.5.1 and 2.4.5 one can use an XMLInstruction annotation:

```
@Path("products")
public class Resource {
   @GET
   @Produces("application/xml")
   @XMLInstruction("<?xml-stylesheet type='text/xsl' href='/stylesheets/toHTML.xsl'?>")
   public Products getProducts() {}
}
```

Lets assume one runs a 'myapp' web application with CXFServlet listening on "/services/*", which can accept requests such as "GET http://localhost/myapp/services/products".

The above relative href value will be converted to "http://localhost/myapp/stylesheets/toHTML.xsl" thus making it easy to resources not 'covered' by CXFServlet. What if you prefer not to even list 'stylesheets' in href='/stylesheets/toHTML.xsl' given that the name of the resource folder may change ? Use an 'xmlResourceOffset' property of JAXBElementProvider:

```
<bean id="jaxbProvider" class="org.apache.cxf.jaxrs.provider.JAXBElementProvider">
<property name="xmlResourceOffset" value="stylesheets"/>
</bean>
</beans>
```

and only have href='toHTML.xsl'. You can also use xmlResourceOffset to make sure the absolute URI will be covered by CXFServlet if preferred.

XSLTTransform

XSLTTransform is a new annotation introduced in CXF 3.0.0. It can support either server or client driven transformations.

Single XSLTTransform annotation can also support both client and server transformations.

Its 'mediaTypes' property, if enabled, is used to restrict the server-based transformations to the listed media types only and expected to be a subset of JAX-RS Produces value.

XSLTTransform can be used in SERVER, CLIENT or BOTH modes.

For it to be effective for the server side transformations (SERVER, BOTH), it has to be used in conjunction with XSLTJaxbProvider.

Default JAXBElementProvider will support XSLTransform CLIENT mode independently.

When it is used either in CLIENT or BOTH modes to let the clients run the transformations it is processed by JAXBElementProvider and is converted into XML XSL instruction.

Using it in BOTH mode effectively supports a transformation 'negotiation': the clients which can do the transformation themselves will have an XML XSL instruction added to the response XML payloads,

and in other case the server will run the transformations.

Examples:

```
// server based Products to XML transformation
@Path("products")
public class Resource {
   @GET
   @Produces("application/xml")
   @XSLTTransform("template.xsl")
   public Products getProducts() {}
}

// server based Products to HTML transformation,
// Product to XML is not affected
@Path("products")
public class Resource {
   @GET
   @Produces("application/xml", "text/html")
   @XSLTTransform(value = "template.xsl", mediaTypes = {"text/html"})
   public Products getProducts() {}
}

// client based Products XML to XML transformation
@Path("products")
public class Resource {
   @GET
   @Produces("application/xml")
   @XSLTTransform(value = "template.xsl", type = CLIENT)
   public Products getProducts() {}
}

// client based Products XML to HTML transformation,
// Product to XML is not affected
@Path("products")
public class Resource {
   @GET
   @Produces("application/xml", "text/html")
   @XSLTTransform(value = "template.xsl", type = CLIENT, mediaTypes = {"text/html"})
   public Products getProducts() {}
}

// clients with "Accept: application/xml" will do Products XML to HTML transformation
// clients with "Accept: text/html" will have server-based Products to HTML transformation
@Path("products")
public class Resource {
   @GET
   @Produces("application/xml", "text/html")
   @XSLTTransform(value = "template.xsl", type = BOTH, mediaTypes = {"text/html"})
   public Products getProducts() {}
}
```

Typically you do not need to configure either XSLT or JAXB providers with the additional properties for XSLTTransform be effective unless BOTH mode is used, in this case simply set XSLTJaxbProvider "supportJaxbOnly":

```
<bean id="xsltProvider" class="org.apache.cxf.jaxrs.provider.XSLTJaxbProvider">
<property name="supportJaxbOnly" value="true"/>
</bean>
```

The above configuration is sufficient to have XSLTTransform BOTH mode supported.

XSI Schema Location

Some tools such as Microsoft Excel can do WEB queries and import the XML payload but this payload is expected to use an xsi:schemaLocation attribute pointing to the XML schema document describing this XML, for example:

```
<products xmlns="http://products"
   xmlns:xsi=""http://www.w3.org/2000/10/XMLSchema-instance""
   xsi:schemaLocation="http://localhost/myapp/schemas/product.xsd">
   <product id="1"/>
</products>
```

In order to get this attribute set with JAXB, one needs to use a "javax.xml.bind.Marshaller.JAXB_SCHEMA_LOCATION" property which can be set on JAXBElementProvider directly (similarly to the way XML Processing Instructions set, see the previous section) or, starting with CXF 2.5.1, with the help of a new XSISchemaLocation annotation:

```
@Path("products")
public class Resource {
   @GET
   @Produces("application/xml")
   @XSISchemaLocation("schemas/products.xsd")
   public Products getProducts() {}
}
```

Please see the previous section on how to affect the absolute URI to a given schema with the help of JAXBElementProvider.
Note that XSISchemaLocation may have a noNamespace attribute set to 'true' if a given schema has no target namespace.