JAXB

JAXB

JAXB is a Data Format which uses the JAXB2 XML marshalling standard which is included in Java 6 to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.

Using the Java DSL

For example the following uses a named DataFormat of *jaxb* which is configured with a number of Java package names to initialize the JAXBContext.

DataFormat jaxb = new JaxbDataFormat("com.acme.model"); from("activemq:My.Queue"). unmarshal(jaxb). to("mqseries:Another.Queue");

You can if you prefer use a named reference to a data format which can then be defined in your Registry such as via your Spring XML file. e.g.

from("activemq:My.Queue"). unmarshal("myJaxbDataType"). to("mqseries:Another.Queue");

Using Spring XML

The following example shows how to use JAXB to unmarshal using Spring configuring the jaxb data type{snippet:id=example|lang=xml|url=camel/trunk /components/camel-jaxb/src/test/resources/org/apache/camel/example/springDataFormat.xml}This example shows how to configure the data type just once and reuse it on multiple routes.{snippet:id=example|lang=xml|url=camel/trunk/components/camel-jaxb/src/test/resources/org/apache/camel/example /marshalAndUnmarshalWithRef.xml}

Multiple context paths
It is possible to use this data format with more than one context path. You can specify context path using `:` as separator, for example `com.mycompany:com.mycompany2`. Note that this is handled by JAXB implementation and might change if you use different vendor than RI.

Partial marshalling/unmarshalling

This feature is new to Camel 2.2.0.
JAXB 2 supports marshalling and unmarshalling XML tree fragments. By default JAXB looks for `@XmlRootElement` annotation on given class to operate on whole XML tree. This is useful but not always - sometimes generated code does not have @XmlRootElement annotation, sometimes you need unmarshall only part of tree.
In that case you can use partial unmarshalling. To enable this behaviours you need set property `partClass`. Camel will pass this class to JAXB's unmarshaler.{snippet:id=example|lang=xml|url=camel/trunk/components/camel-jaxb/src/test/resources/org/apache/camel/example /springDataFormatPartial.xml}For marshalling you have to add `partNamespace` attribute with QName of destination namespace. Example of Spring DSL you can find above.

Fragment

This feature is new to Camel 2.8.0.
JaxbDataFormat has new property fragment which can set the the `Marshaller.JAXB_FRAGMENT` encoding property on the JAXB Marshaller. If you don't want the JAXB Marshaller to generate the XML declaration, you can set this option to be true. The default value of this property is false.

Ignoring the NonXML Character

This feature is new to Camel 2.2.0.
JaxbDataFromat supports to ignore the NonXML Character, you just need to set the filterNonXmlChars property to be true, JaxbDataFormat will replace the NonXML character with " " when it is marshaling or unmarshaling the message. You can also do it by setting the Exchange property `Exchange.FILTER_NON_XML_CHARS`.

	JDK 1.5	JDK 1.6+
Filtering in use	StAX API and implementation	No
Filtering not in use	StAX API only	No

This feature has been tested with Woodstox 3.2.9 and Sun JDK 1.6 StAX implementation.

New for Camel 2.12.1
JaxbDataFormat now allows you to customize the XMLStreamWriter used to marshal the stream to XML. Using this configuration, you can add your own stream writer to completely remove, escape, or replace non-xml characters.

java JaxbDataFormat customWriterFormat = new JaxbDataFormat("org.apache.camel.foo.bar"); customWriterFormat.setXmlStreamWriterWrapper(new TestXmlStreamWriter());

The following example shows using the Spring DSL and also enabling Camel's NonXML filtering:

xml<bean id="testXmlStreamWriterWrapper" class="org.apache.camel.jaxb.TestXmlStreamWriter"/> <jaxb filterNonXmlChars="true" contextPath="org.apache.camel.foo.bar" xmlStreamWriterWrapper="#testXmlStreamWriterWrapper" />

Working with the ObjectFactory

If you use XJC to create the java class from the schema, you will get an ObjectFactory for you JAXB context. Since the ObjectFactory uses JAXBElement to hold the reference of the schema and element instance value, jaxbDataformat will ignore the JAXBElement by default and you will get the element instance value instead of the JAXBElement object form the unmarshaled message body.
If you want to get the JAXBElement object form the unmarshaled message body, you need to set the JaxbDataFormat object's ignoreJAXBElement property to be false.

Setting encoding

You can set the **encoding** option to use when marshalling. Its the `Marshaller.JAXB_ENCODING` encoding property on the JAXB Marshaller.
You can setup which encoding to use when you declare the JAXB data format. You can also provide the encoding in the Exchange property `Exchange.CHARSET_NAME`. This property will overrule the encoding set on the JAXB data format.

In this Spring DSL we have defined to use `iso-8859-1` as the encoding:{snippet:id=example|lang=xml|url=camel/trunk/components/camel-jaxb/src/test/resources/org/apache/camel/example/springDataFormatWithEncoding.xml}

Controlling namespace prefix mapping

Available as of Camel 2.11

When marshalling using JAXB or SOAP then the JAXB implementation will automatic assign namespace prefixes, such as ns2, ns3, ns4 etc. To control this mapping, Camel allows you to refer to a map which contains the desired mapping.

Notice this requires having JAXB-RI 2.1 or better (from SUN) on the classpath, as the mapping functionality is dependent on the implementation of JAXB, whether its supported.

For example in Spring XML we can define a Map with the mapping. In the mapping file below, we map SOAP to use soap as prefix. While our custom namespace "http://www.mycompany.com/foo/2" is not using any prefix.

xml <util:map id="myMap"> <entry key="http://www.w3.org/2003/05/soap-envelope" value="soap"/> <!-- we dont want any prefix for our namespace --> <entry key="http://www.mycompany.com/foo/2" value=""/> </util:map>

To use this in JAXB or SOAP you refer to this map, using the `namespacePrefixRef` attribute as shown below. Then Camel will lookup in the Registry a `java.util.Map` with the id "myMap", which was what we defined above.

xml <marshal> <soapjaxb version="1.2" contextPath="com.mycompany.foo" namespacePrefixRef="myMap"/> </marshal>

Schema validation

Available as of Camel 2.11

The JAXB Data Format supports validation by marshalling and unmarshalling from/to XML. Your can use the prefix **classpath:**, **file:* or *http:** to specify how the resource should by resolved. You can separate multiple schema files by using the **','** character.

Known issue
Camel 2.11.0 and 2.11.1 has a known issue by validation multiple `Exchange`'s in parallel. See CAMEL-6630. This is fixed with Camel 2.11.2/2.12.0.

Using the Java DSL, you can configure it in the following way:

javaJaxbDataFormat jaxbDataFormat = new JaxbDataFormat(); jaxbDataFormat.setContextPath(Person.class.getPackage().getName()); jaxbDataFormat.setSchema("classpath:person.xsd,classpath:address.xsd");

You can do the same using the XML DSL:

xml<marshal> <jaxb id="jaxb" schema="classpath:person.xsd,classpath:address.xsd"/> </marshal>

Camel will create and pool the underling `SchemaFactory` instances on the fly, because the `SchemaFactory` shipped with the JDK is not thread safe. However, if you have a `SchemaFactory` implementation which is thread safe, you can configure the JAXB data format to use this one:

javaJaxbDataFormat jaxbDataFormat = new JaxbDataFormat(); jaxbDataFormat.setSchemaFactory(thradSafeSchemaFactory);

Schema Location

Available as of Camel 2.14

The JAXB Data Format supports to specify the SchemaLocation when marshaling the XML.

Using the Java DSL, you can configure it in the following way:

javaJaxbDataFormat jaxbDataFormat = new JaxbDataFormat(); jaxbDataFormat.setContextPath(Person.class.getPackage().getName()); jaxbDataFormat.setSchemaLocation("schema/person.xsd");

You can do the same using the XML DSL:

xml<marshal> <jaxb id="jaxb" schemaLocation="schema/person.xsd"/> </marshal>

Marshal data that is already XML

Available as of Camel 2.14.1

The JAXB marshaller requires that the message body is JAXB compatible, eg its a JAXBElement, eg a java instance that has JAXB annotations, or extend JAXBElement. There can be situations where the message body is already in XML, eg from a String type. There is a new option `mustBeJAXBElement` you can set to false, to relax this check, so the JAXB marshaller only attempts to marshal JAXBElements (javax.xml.bind.JAXBIntrospector#isElement returns true). And in those situations the marshaller fallbacks to marshal the message body as-is.

XmlRootElement objects

Available as of Camel 2.17.2

The JAXB Data Format option objectFactory has a default value equals to false. This is related to a performance degrading. For more information look at the issue CAMEL-10043

For the marshalling of non-XmlRootElement JaxB objects you'll need to call JaxbDataFormat#setObjectFactory(true)

Dependencies

To use JAXB in your camel routes you need to add the a dependency on **camel-jaxb** which implements this data format.

If you use maven you could just add the following to your pom.xml, substituting the version number for the latest & greatest release (see the download page for the latest versions).

<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-jaxb</artifactId> <version>x.x.x</version> </dependency>