XmlJson

XML JSON Data Format (camel-xmljson)

Available as of Camel 2.10

Camel already supports a number of data formats to perform XML and JSON-related conversions, but all of them require a POJO either as an input (for marshalling) or produce a POJO as output (for un-marshaling). This data format provides the capability to convert from XML to JSON and viceversa directly, without stepping through intermediate POJOs.

This data format leverages the Json-lib library to achieve direct conversion. In this context, XML is considered the high-level format, while JSON is the low-level format. Hence, the marshal/unmarshal semantics are assigned as follows:

- marshalling => converting from XML to JSON
- un-marshaling => converting from JSON to XML.

Options

This data format supports the following options. You can set them via all DSLs. The defaults marked with **(*)** are determined by **json-lib**, rather than the code of the data format itself. They are reproduced here for convenience to avoid having to consult the **json-lib** documentation directly.

Option	Type	Default	Description
encoding	String	UTF-8 (*)	**Used when un-marshaling (JSON to XML conversion).** Sets the encoding for the call to `XMLSerializer.write()` method, hence it is only used when producing XML. However, when producing JSON, the encoding is determined by the input String being processed. If the conversion is performed on an **InputStream**, **json-lib** uses the platform's default encoding, e.g., determined by the **file.encoding** system property.
elementName	String	'e' (*)	**Used when un-marshaling (JSON to XML conversion).** Specifies the name of the XML elements representing each array element. See json-lib doc.
arrayName	String	'a' (*)	**Used when un-marshaling (JSON to XML conversion).** Specifies the name of the top-level XML element. For example, when converting: **[1, 2, 3]** it is, by default, translated as: **<a><e>1</e><e>2</e><e>3</e>** By setting this option or **rootName**, you can alter the name of the element **a**.
rootName	String	none (*)	**Used when un-marshaling (JSON to XML conversion).** When converting any JSON construct (**object**, **array**, **null**) to XML (un-marshaling), this option specifies the name of the top-level XML element. If not set, **json-lib** will use **arrayName** or **objectName** (default value: **o**, at the current time it is not configurable in this data format). If set to **root**, the JSON string: **{ "x": "value1", "y" : "value2" }** is translated as: **<root><x>value1</x><y>value2</y></root>** otherwise the **root** element would be named **o**.
namespaceLenient	Boolean	false (*)	**Used when un-marshaling (JSON to XML conversion).** According to the **json-lib** docs: "*Flag to be tolerant to incomplete namespace prefixes.*" In most cases, **json-lib** automatically changes this flag at runtime to match the processing.

`namespaceMappings`	`List<NamespacesPerElementMapping>`	`none`	**Used when un-marshaling (JSON to XML conversion).** Binds namespace prefixes and URIs to specific JSON elements. **`NamespacesPerElementMapping`** is a wrapper around an element name + a map of prefixes against URIs.
`expandableProperties`	`List<String>`	`none`	**Used when un-marshaling (JSON to XML conversion).** With expandable properties, JSON array elements are converted to XML as a sequence of repetitive XML elements with the local name equal to the JSON key. For example, the following JSON: **`{ "number": 1,2,3 }`** is normally translated as: **`<number><e>1</e><e>2</e><e>3</e></number>`** where **`e`** can be modified by setting **`elementName`**. However, if **`number`** is set as an expandable property, it's translated as: **`<number>1</number><number>2</number><number>3</number>`**
`typeHints`	`TypeHintsEnum`	`YES`	**Used when un-marshaling (JSON to XML conversion).** Adds type hints to the resulting XML to aid conversion back to JSON. See documentation here for an explanation. **`TypeHintsEnum`** comprises the following values, which lead to different combinations of the underlying XMLSerializer's **`typeHintsEnabled`** and **`typeHintsCompatibility`** flags: • `TypeHintsEnum.NO` **`=>`** `typeHintsEnabled` = `false` • `TypeHintsEnum.YES` **`=>`** `typeHintsEnabled` = `true`, `typeHintsCompatibility` = `true` • `TypeHintsEnum.WITH_PREFIX` **`=>`** `typeHintsEnabled` = `true`, `typeHintsCompatibility` = `false`
`forceTopLevelObject`	`Boolean`	`false` (*)	**Used when marshaling (XML to JSON conversion).** Determines whether the resulting JSON will start off with a top-most element whose name matches the XML root element. If this option is **`false`**, the XML string: **`<a><x>1</x><y>2</y></a>`** is translated as: **`{"x": "1", "y": "2"}`** If **`true`**, it's translated as: **`{ "a":  { "x": "1", "y": "2" }}`**
`skipWhitespace`	`Boolean`	`false` (*)	**Used when marshaling (XML to JSON conversion).** Determines whether white spaces between XML elements will be regarded as text values or disregarded.
`trimSpaces`	`Boolean`	`false` (*)	**Used when marshaling (XML to JSON conversion).** Determines whether leading and trailing white spaces will be omitted from String values.
`skipNamespaces`	`Boolean`	`false` (*)	**Used when marshaling (XML to JSON conversion).** Signals whether namespaces should be ignored. By default they will be added to the JSON output using **`@xmlns`** elements.
`removeNamespacePrefixes`	`Boolean`	`false` (*)	**Used when marshaling (XML to JSON conversion).** Removes the namespace prefixes from XML qualified elements, so that the resulting JSON string does not contain them.

Basic Usage With the Java DSL

Explicitly Instantiating the DataFormat

Just instantiate the **`XmlJsonDataFormat`** from package **`org.apache.camel.dataformat.xmljson`**. Make sure you have installed the **`camel-xmljson`** feature (if running on OSGi) or that you've included **`camel-xmljson-{version}.jar`** and its transitive dependencies in your classpath.

Example, initialization with a default configuration:

```
XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();
```

To tune the behavior of the data format as per the options above, use the appropriate setters:

```
XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();
xmlJsonFormat.setEncoding("UTF-8");
xmlJsonFormat.setForceTopLevelObject(true);
xmlJsonFormat.setTrimSpaces(true);
xmlJsonFormat.setRootName("newRoot");
xmlJsonFormat.setSkipNamespaces(true);
xmlJsonFormat.setRemoveNamespacePrefixes(true);
xmlJsonFormat.setExpandableProperties(Arrays.asList("d", "e"));
```

Once the **`DataFormat`** is instantiated, the next step is to use it as a parameter to either of the **`marshal()/unmarshal()`** DSL elements:

```
// From XML to JSON
from("direct:marshal")
  .marshal(xmlJsonFormat)
  .to("mock:json");

// From JSON to XML
from("direct:unmarshal")
  .unmarshal(xmlJsonFormat)
  .to("mock:xml");
```

Defining the DataFormat in-line

Alternatively, you can define the data format inline by using the **`xmljson()`** DSL element.

```
// From XML to JSON - inline dataformat
from("direct:marshalInline")
  .marshal()
  .xmljson()
  .to("mock:jsonInline");

// From JSON to XML - inline dataformat
from("direct:unmarshalInline")
  .unmarshal()
  .xmljson()
  .to("mock:xmlInline");
```

If you wish, you can even pass in a Map<String, String> to the inline methods to provide custom options:

```
Map<String, String> xmlJsonOptions = new HashMap<String, String>();

xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.ENCODING, "UTF-8");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.ROOT_NAME, "newRoot");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.SKIP_NAMESPACES, "true");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.REMOVE_NAMESPACE_PREFIXES, "true");
xmlJsonOptions.put(org.apache.camel.model.dataformat.XmlJsonDataFormat.EXPANDABLE_PROPERTIES, "d e");

// From XML to JSON - inline dataformat w/options
from("direct:marshalInlineOptions")
  .marshal()
  .xmljson(xmlJsonOptions)
  .to("mock:jsonInlineOptions");

// From JSON to XML - inline dataformat w/options
from("direct:unmarshalInlineOptions")
  .unmarshal()
  .xmljson(xmlJsonOptions)
  .to("mock:xmlInlineOptions");
```

Basic Usage with Spring or Blueprint DSL

Within the **<dataFormats>** block, simply configure an **xmljson** element with unique IDs:

```
<dataFormats>
    <xmljson id="xmljson"/>
    <xmljson id="xmljsonWithOptions"
             forceTopLevelObject="true"
             trimSpaces="true"
             rootName="newRoot"
             skipNamespaces="true"
             removeNamespacePrefixes="true"
             expandableProperties="d e"/>
</dataFormats>
```

Then you simply refer to the data format object within your **<marshal/>** and **<unmarshal/>** DSLs:

```
<route>
    <from uri="direct:marshal"/>
    <marshal ref="xmljson"/>
    <to uri="mock:json"/>
</route>

<route>
    <from uri="direct:unmarshalWithOptions"/>
    <unmarshal ref="xmljsonWithOptions"/>
    <to uri="mock:xmlWithOptions"/>
</route>
```

Enabling XML DSL autocompletion for this component is easy: just refer to the appropriate Schema locations, depending on whether you're using Spring or Blueprint DSL. Remember that this data format is available from Camel 2.10. Therefore only schemas from that version or later will include these new XML elements and attributes.

The syntax with Blueprint is identical to that of the Spring DSL. Just ensure the correct namespaces and **schemaLocations** are in use.

Namespace Mappings

XML has namespaces to fully qualify elements and attributes; JSON doesn't. You need to take this into account when performing XML-JSON conversions.

To bridge the gap, Json-lib has an option to bind namespace declarations in the form of prefixes and namespace URIs to XML output elements while unmarshaling, e.g., converting from JSON to XML.

For example, provided the following JSON string:

```
{ "pref1:a": "value1", "pref2:b": "value2" }
```

you can ask **json-lib** to output namespace declarations on elements **pref1:a** and **pref2:b** to bind the prefixes **pref1** and **pref2** to specific namespace URIs.

To use this feature, simply create **XmlJsonDataFormat.NamespacesPerElementMapping** objects and add them to the **namespaceMappings** option (which is a **List**).

The **XmlJsonDataFormat.NamespacesPerElementMapping** holds an element name and a Map of **[prefix => namespace URI]**. To facilitate mapping multiple prefixes and namespace URIs, the **NamespacesPerElementMapping(String element, String pipeSeparatedMappings)** constructor takes a String-based pipe-separated sequence of **[prefix, namespaceURI]** pairs in the following way: **|ns2|http://camel.apache.org/personalData|ns3|http://camel.apache.org/personalData2|**.

In order to define a default namespace, just leave the corresponding key field empty: **|ns1|http://camel.apache.org/test1||http://camel.apache.org/default|.**

Binding namespace declarations to an element name = empty string will attach those namespaces to the root element.

The code for this is:

```
XmlJsonDataFormat namespacesFormat = new XmlJsonDataFormat();
List<XmlJsonDataFormat.NamespacesPerElementMapping> namespaces = new ArrayList<XmlJsonDataFormat.
NamespacesPerElementMapping>();

namespaces.add(new XmlJsonDataFormat.NamespacesPerElementMapping("", "|ns1|http://camel.apache.
org/test1||http://camel.apache.org/default|"));
namespaces.add(new XmlJsonDataFormat.NamespacesPerElementMapping("surname", "|ns2|http://camel.apache.
org/personalData|ns3|http://camel.apache.org/personalData2|"));
namespacesFormat.setNamespaceMappings(namespaces);
namespacesFormat.setRootElement("person");
```

And you can achieve the same in Spring DSL.

Example

Using the namespace bindings in the Java snippet above on the following JSON string:

```
{ "name": "Raul", "surname": "Kripalani", "f": true, "g": null}
```

Would yield the following XML:

```
<person xmlns="http://camel.apache.org/default" xmlns:ns1="http://camel.apache.org/test1">
    <f>true</f>
    <g null="true"/>
    <name>Raul</name>
    <surname xmlns:ns2="http://camel.apache.org/personalData" xmlns:ns3="http://camel.apache.org/personalData2"
>Kripalani</surname>
</person>
```

Remember that the JSON spec defines a JSON object as follows:

> *An object is an unordered set of name/value pairs. [...].*

That's why the elements are in a different order in the output XML.

Dependencies

To use the XmlJson dataformat in your camel routes you need to add the following dependency to your pom.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xmljson</artifactId>
  <version>x.x.x</version>
  <!-- Use the same version as camel-core, but remember that this component is only available from Camel 2.10 --
>
</dependency>

<!-- And also XOM must be included. XOM cannot be included by default due to an incompatible
license with ASF; so add this manually -->
<dependency>
  <groupId>xom</groupId>
  <artifactId>xom</artifactId>
  <version>1.2.5</version>
</dependency>
```

See Also

- Data Format
- json-lib