

1. JAX-WS

In this lecture we will learn what JAX-WS and some of its important annotations.

JAX-WS

stands for Java API for XML based web services like any other Java enterprise standard comprises of **specification** and **API**.

1.1. Specification

Specification

Is a set of rules or guidelines from Oracle when they write the JAX-WS standard. These guidelines in the specification help web services engines like Apache CXF and GlassFish to implement the JAX-WS standard.

1.2. API

API

Is for developers and it comprises of a set of Java annotations. We learn these annotations by marking simply our Java classes and methods. We can implement both web services providers and consumers using those annotations. Once we mark the classes and the methods, the web services engines like Apache CXF can read these annotations at runtime and take the appropriate action.

1.2.1. Core Annotations

Core Annotations

All the core annotations are inside the `javax.jws` package

@WebService	The web services engines know that this class or interface is a endpoint web services end point which can handle web service requests that are coming in.
@WebMethod	We use this against all of our web services methods each method in our web services end point.
@WebFault	Come up with our custom exceptions, which in turn will be converted, into SOAP faults.
@SOAPBinding	It allows developers to specify a particular type of binding. Binding controls how SOAP message is generated when it goes on the wire Apache CXF. By default is document/literal (recommended approach). We can specify a different type of binding by simply parametrizing this annotation.
@RequestWrapper	It allows us to wrap or map the incoming SOAP message to our Java objects in a customer manner and the response wrapper does the other way around.
@ResponseWrapper	It allows us to customize the way our response Java object is converted into a SOAP message. We very rarely do this because the JAX WS standard already does a great job in converting the SOAP messages into Java objects and Java objects into SOAP messages.

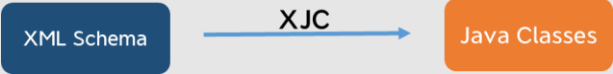

2. JAXB Introduction

Stands for Java Architecture for XML binding. It provides an easy way to map Java classes and XML schema hiding the complexity of XML programming. Instead of us as Java developers interacting with the abstract representation of XML up using API. So we can simply deal with Java objects directly and the JAXB allows us to convert this XML and XML schema into Java classes and objects.




2.1. Tools

Starting from Java 1.6 most of these tools are a part of your JDK. (for example if you go to JAVA folder on your machine and exactly to the bin folder).

BUILDTIME API	XJC	It stands for XML schema compiler. It generates Java classes from a given XML Schema.
		
	SCHEMAGEN	It generates XML Schema from a given set of java classes once you mark those java classes with JAXB annotations.
		

Note

We rarely use these tools directly. We use plugins on the top these tools (like maven JAXB plugin).

RUNTIME API	It allows us to do both marshalling as well as unmarshalling.	
		
	Composition	
	Marshall	To serialize XML into Java. Marshalling is the process of converting Java objects into XML.
	Unmarshall	To deserialize Java into XML. Unmarshalling is the other way around, so we convert XML back to java objects.
	Annotations	Used to mark our Java classes.

3. JAXB tools and plugins

Starting of Java 1.6, most of these tools are a part of your JDK. All the tools are available in bin folder inside JDK.

4. Steps to generate stubs from XML Schema

4.1. XML Schema to JAXB Classes

- Create the Project (with IntelliJ IDEA or Spring Tool Suite).
- Create the schemas (We will use the existing schema files, which you would have developed from the xml and xml schema section).
- Use the JAXB Plugin.
- Generate the stubs and use them (we will use them to serialize and deserialize Java objects into XML).

5. Generate the Stubs

In this lecture, we will generate the stubs (Java classes) using the JAXB maven plugin. To do that we need to:

- To copy the configuration of the plugin from the original project to the newly created project from the pom.xml file.
- We need to provide a configuration elements for the element to be as same as directory structure (where schema sources live).
- When we build or run our goal, the stubs are going to be generated dynamically.
- In this case, we are not going to generate the stubs for the employee schema. So later, we could try on that by binding the directory.
- The generated stubs will be found inside the generated folder in same structure as same is we declare it inside the configuration elements.

6. Customize Generated Code Using Binding File

In this lecture, we will learn how we can customize the stubs generation process (we can customize what happens internally and we do that using the .xjb format file included under the xsd file).

- The global.xjb file contains an XML configuration that will be used or read by our plugin.

- If we go to the pom.xml there are two optional parameters we didn't explain earlier.
- The first parameter is the binding directory, which tells the plugin where it can find a binding file.
- The second one is binding include (we are including the global.xjb). We can have multiple binding files on the fly.
- This binding files tells the xjc plugin or the JAXB plugin in how the Java code should look like when it gets generated.
- It will apply all the default rules but whenever we specify in here it will take them by consideration.

6.1. Describing the global.xjb file

Once we go to global.xjb (xjb stands for XML to Java binding). We can customize the JAXB code generation process. We don't need to just accept whatever the plugin generates by default. Using this binding file and by providing that binding to the maven project which will customize the way your Java code look like.

element	Description
jaxb:bindings	The root element is and here are all the namespaces from which we can use several elements in the file you can refer to JAXB documentation for each of these elements and there are more that you can use.
xjc:simple	Tells the xjc compiler that it should include the annotation simple type on every Java class that gets generated. If we go to the stubs, we will find that the annotation @XmlType . It will happened by default even when we take it out.
xjc:serializable	Affect the java serialization. In our case we are providing a default value of -1.
xjc:javaType	Allows us to bind the xsd types to a particular java type. In this case, when the tool find dateTime in the xml shema, I want you to convert that into java.util.Calendar in the Java code that gets generated.
	parseMethod
	printMethod
This are the converters that are available in the JAXB(documentation)	

7. Stubs Walk Through

In this lecture, we will walk through the Patient.java file that got generated from the patient.xsd

7.1. Anntation placed on a class

patient.xsd		
The root element hich is of type patient is a complexe type. It contains a sub types like name, age, gener ...		
Patient.java		
Annotations used		
@XmlRootElement	It associates the annotated class with a root node of an XML document and tells that the root element for an object of this class when it is serialized into xml should have a name equal to "patient".	
@XmlAccessorType	It indicates how JAXB should take in consideration the fields of a class (All the fields or properties of a class are taken into account by default in the process of generating an XML documents) except those that are annotated @XmlTransient . It is important to note that by default, if you annotate a private filed without annotating the class with @XmlAccessType.FIELD , things are likely to go wrong. Indeed JAXB will see the same field twice:	
	<ul style="list-style-type: none"> • because of the annotation. • because it takes in consideration the public getter. 	
	value	indication
	.FIELD	Indicates that all non-static fields of the class are taken into account.
	.PROPERTY	Indicates that all pairs of getters/setters are taken into account.
	.PUBLIC	Indicates that all pairs of getters/setters and all non-static public fields will be taken into account.
	.NONE	Indicates that no field or property is taken into account.

@XmlType	It specifies the order in which these fields should be serialized when they are converted into xml.
@XmlTransient	useful for resolving name collisions between a JavaBean property name and a field name or preventing the mapping of a field/property. A name collision can occur when the decapitalized JavaBean property name and a field name are the same. If the JavaBean property refers to the field, then the name collision can be resolved by preventing the mapping of either the field or the JavaBean property using the @XmlTransient annotation.

7.2. Annotation placed on a field or getter

Annotation	Definition
@XmlElement	Used to associate a field or a getter with a node of an XML document. It allows to specify the name of this element, its namespace, its default value ...By default, the name of XML elements will be the name of the fields but we can customize that by using this annotation within the specification of name inside with validation dynamically.
@XmlAttribute	Allows the annotated field to be written in an XML attribute rather than in sub-element of parent XML element. You can set the name of this attribute, the namespace to which it belongs, and require it to be present (required attribute).

8. Generating Java Classes from XML Schema

9. Marshalling and Unmarshalling

Marshalling	Is the process of converting Java objects into xml.
--------------------	---

To marshall the stubs that got generated in the previous steps, we have to create a class that contains a main method. The main entry point for JAXB API (runtime API) is the JAXB context class:

1. we start by creating an instance of it by calling the factory method on the JAXB context. The parameter to this context is the class that we want to **serialize** and **deserialize**, in our case is the Patient class.
2. Once we have the context, we create a marshaller.
3. In this step, we create a patient object and setting its properties.
4. Before the end, we create a method to marshall the created object. It takes two parameters,
 - It is the JAXB element which should be marshalled.
 - It is a handler which can be output stream or output writer (in our case we are going to create a simple String Writer).
5. Finally, we print the contents of that writer.

Note We need to handle a JAXB exception.

Unmarshalling	Is the process of converting XML into Java objects (the other way around).
----------------------	--

We do this kind of operations quite often in the web services world or using the frameworks like **Apache CXF** or the **SOAP** engines within the **Apache CXF**.

1. We start by getting the XML file to read.
2. We create the JAXB context.
3. We create the unmarshalling object.
4. We have to call the Unmarshall method.
5. Finally we print the information of object (patient) using toString() method.

Table des matières

1. JAX-WS.....	1
1.1. Specification.....	1
1.2. API.....	1
1.2.1. Core Annotations.....	1
2. JAXB Introduction	1
2.1. Tools	1
3. JAXB tools and plugins.....	2
4. Steps to generate stubs from XML Schema.....	2
4.1. XML Schema to JAXB Classes.....	2
5. Generate the Stubs.....	2
6. Customize Generated Code Using Binding File.....	2
6.1. Describing the global.xjb file.....	3
7. Stubs Walk Through	3
7.1. Anntation placed on a class.....	3
7.2. Annotation placed on a field or getter	4
8. Generating Java Classes from XML Schema.....	4
9. Marshalling and Unmarshalling	4