JBossWS Tutorials

jbossws-1.2.0.CR3

11-Jan-2007

Table of Contents

Preface

Tutorials covering the JAX-WS 2.0 web services stack for the JBoss Application Server.

If you have questions regarding JBossWS, please feel to ask the JBossWS team.

Web Service Lead - Thomas Diesler

WS-Security, WSTools - Jason Greene

WS-Eventing, Standards - Heiko Braun

1

Introduction

Welcome to the JBoss JAX-WS 2.0 series of tutorials. This set of tutorials is intended to help developers in getting started with the new JAX-WS 2.0 web services stack. The initial tutorial covers installation, a simple web service and client programs to use the simple web service from both Eclipse and NetBeans. Over the next few weeks, further tutorials will include adding security, MTOM example, wstools options and others.

The tutorials assume your working against JBoss Application Server 5 Beta 1 or after. If you haven't downloaded a version of AS 5 yet, you can get it here: http://labs.jboss.com/portal/jbossas/download/index.html

1.1. What is a Web Service?

Several approaches over the years (CORBA, DCOM, DSOM, RPC) aimed at providing distributed processing in a standardized manner have lead us to today's Web Services approach. It is built on XML which is multinational, human readable, platform independent and allows the expression and manipulation of very complicated data structures. Both the SOAP and RESTful web service approaches use XML to do the data transfer between the client and server distibuted processes.

While there are several different approaches to web services. The most common approach is embodied in the SOAP based web services specs and frameworks (now W3C specs) created jointly between IBM and Microsoft. They are a very rich set of specs, but along with that richness comes a lot of complexity aimed at dealing with many different kinds of scenerios. There is also the RESTful approach as well which tends to be more bare-bones and avoids dealing with many of the issues that the covered in the W3C specs.

The common reality of each of these approaches is being able to call some distributed (either remotely or on the same machine) process to get it to do some work for you and then return the results. So web services may best defined in this simple way: “ Web Services provide ability to call distributed behavior and receive results in a platform independent and well understood format. ”

2

Installation

JBossWS 2.0 (JAX-WS 2.0) is based on the JBoss Microcontainer [http://labs.jboss.com/portal/jbossmc], which supports direct POJO deployment and standalone use outside the JBoss application server. Currently, we have integration layers for the JBoss application server and Apache Tomcat.

For this series of tutorials we are going to focus on using the JBoss Web Services stack on the JBoss Application Server. If you need to know how to work with Tomcat, please see the JBoss Web Services User Guide for more details.

2.1. Web Services on JBoss Application Server

To take advantage of optimum integration it is best to install JBossWS on the appserver with embedded Tomcat. In this way you get access to the full set of advanced J2EE features (JMS, JAAS, EJB, etc.) advanced class loading, better integration with managment and security. Contrary to common believe JBossAS with embedded Tomcat is as lightweight and easy to administer as standalone Tomcat.

For additional information, see Tomcat Standalone vs Embedded [http://wiki.jboss.org/wiki/Wiki.jsp?page=TomcatStandalonevsEmbedded].

Additionally, please have a look at the release specific install instructions [http://labs.jboss.com/jbossws/user-guide/en/Install.txt].

2.1. Installation on JBoss Application Server 5

If you are using a beta of JBoss Application Server 5, you do not need to install anything as the JBoss Web Service JAX-WS 2.0 stack is included already.

3

Tutorial 1

In this tutorial we will see the basic structure of a JAX-WS 2.0 web service and the Ant file to build it. There are a few things to learn, so lets get to it.

3.1. Endpiont Interface Definition

We start out with a definition of our web services endpoint. Endpoints define the interface that we are going to export to the world thru WSDL for our web service. You can see that it is also much like a RMI object in that it is declared remote as well. The methods in our EndpointInterface definition must all be public as well.

```
package org.jboss.samples.helloworld;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface EndpointInterface extends Remote
{
    String sayHello(String toWhom) throws RemoteException;
}
```

Our interface is about as simple as it gets. We take a String instance input and return a String as an output.

3.2. Endpoint Class

Now we create the implementation of our Endpoint. Our example pays homage to the venerable "HelloWorld" known world over.

```
package org.jboss.samples.helloworld;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService(endpointinterface=")
public class HelloWorldWS
{
    @WebMethod
    public String sayHello(String toWhom)
    {
        return "Hello " + toWhom + "!" + " The date and time is: " + new java.util.Date();
    }
```

```
}
```

Here we see that we need our web service to import both the WebService and WebMethod classes from the javax.jws package for use in the annotations on our web service. We are using the simplest forms of both of those annotations. Doing so takes all the default settings that the annotation would define. We will go into that more in later tutorials.

Our class is flagged as a web service by the annotaion before the class declaration. The class must be public. We have our "sayHello" method implementation which takes in a string (someone's name) and then returns a appropriate greeting string. That's it.

3.3. ANT Build File

We'll see these main things in this build file:

- The libs needed to build
- How to get the WSDL file generated
- Packaging and deploying

In the program listing below you can see that the library we need is: jbossws-client.jar. This file is in the jboss application server "client" directory. This jar provides all the needed classes for us to build our web service.

You can also see the task definition for the "wstools" helper class that calls out to the wstools command line application. The "wstools" application is used to generate WSDL's from Java code or to create Java code from WSDL's. In our case we are going to create a WSDL from the Java code above. The "wstool" call in our example takes two arguments. One being the "wstools" configuration file and one being where to put the generated output files. We'll see the "wstools" configuration file more in the next section.

The packaging of a web service is done in this example by putting it in a standard WAR file. For EJB's you can also package it in a EAR file. In this example we use a web.xml file to specify the web services url mapping.

```
<?xml version="1.0"?>
<!-- Build file for Hello World Web Service Example using new JBoss JAX-WS 2.0 Stack -->
<project name="JAX-WS 2.0 Hello World Web Service Buildfile" default="main" basedir=".">
<!-- Standard Properties -->
        <property name="top.dir" value="${basedir}"/>
        <property name="server.config" value="default"/>
        <property name="src.dir" value="${top.dir}/src"/>
        <property name="build.classes" value="${top.dir}/classes"/>
        <property name="java.dir" value="${top.dir}/src/java"/>
        <property name="resources.dir" value="${top.dir}/resources"/>
        <property name="jboss.dir" value="/Users/sgriffith/jboss-5.0.0.Beta2"/>
        <property name="jboss.client" value="${jboss.dir}/client"/>
        <property name="jboss.lib" value="${jboss.dir}/lib"/>
        <property name="jboss.server" value="${jboss.dir}/server/${server.config}"/>
        <property name="jboss.server.lib" value="${jboss.server}/lib"/>
        <property name="jboss.server.deploy" value="${jboss.server}/deploy"/>
```

```
<!-- JDK Detection -->
        <available classname="java.lang.Enum" property="HAVE_JDK_1.5"/>

<!-- Check for new WS jar file -->
        <available file="${jboss.client}/jbossws-client.jar" value="jbossws-client.jar" property="jbossws

<!-- Path Definitions -->
        <path id="client.classpath">
                <fileset dir="${jboss.client}">
                        <include name="*.jar"/>
                </fileset>
        </path>
        <path id="compile.classpath">
                <path refid="client.classpath"/>
                <pathelement location="${jboss.server.lib}/jboss.jar"/>
        </path>

<!-- Targets -->

<!-- Helper target -->
        <target name="generate-sources" depends="" description="Generate the deployment resources and WSD
                <!-- Define a taskdef for the wstools ant task -->
                <echo message="Inside generate-sources"/>
                <taskdef name="wstools" classname="org.jboss.ws.tools.ant.wstools">
                        <classpath refid="client.classpath"/>
                        <classpath path="${top.dir}/classes"/>
                        <classpath path="${top.dir}"/>
                </taskdef>
                <echo message="wstools about to run"/>
                <wstools dest="${top.dir}/META-INF" config="${resources.dir}/wstools-config.xml"/>
        </target>

<!-- Reset everything for a build from scratch -->
        <target name="clean">
                <echo message="In clean"/>
                <delete file="${top.dir}/helloworldws.war"/>
                <delete dir="${build.classes}"/>
          <delete dir="${top.dir}/META-INF"/>
        </target>

<!--
        Compile the java sources
-->
<!--
  Compile with jdk-1.5 as JAX-WS 2.0 needs annotations so we must use Java 5 or above.
-->
        <target name="compile15" depends="" description="compile the sources" if="HAVE_JDK_1.5">
                <javac destdir="${build.classes}" classpathref="compile.classpath" debug="on">
                        <src path="${java.dir}"/>
                </javac>
        </target>

        <target name="main" depends="compile15, generate-sources" if="jbossws.client.jar">
                <echo message="In main"/>
                <mkdir dir="${build.classes}"/>
                <war warfile="helloworldws.war" webxml="${src.dir}/metadata/web.xml">
                        <classes dir="${top.dir}/classes">
                                <include name="**/*.class"/>
                        </classes>
                </war>
                <!-- <antcall target="deploy"/> -->
        </target>

<!--
        Deployment targets
-->
```

```
        <target name="deploy">
                <echo message="In deploy"/>
                <copy file="${top.dir}/helloworldws.war" todir="${jboss.server.deploy}"/>
                <delete file="${top.dir}/helloworldws.war"/>
        </target>

        <target name="undeploy">
                <echo message="In undeploy"/>
                <delete>
                        <fileset id="web_service_wars" dir="${jboss.server.deploy}" includes="*hello*.war
                </delete>
        </target>

</project>
```

Since we use a WAR file we can specify the web services url mapping in the "web.xml" file just like you can for any web application. For this sample we've got the web service mapped to url path /HelloWorldWS. You'll see when we deploy the web service that this url path and the service name are important to the final url to the web service.

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xs
        <display-name>HelloWorldWS</display-name>
        <servlet>
                <display-name>HelloWorldWS</display-name>
                <servlet-name>HelloWorldWS</servlet-name>
                <servlet-class>org.jboss.samples.helloworld.HelloWorldWS</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>HelloWorldWS</servlet-name>
                <url-pattern>/HelloWorldWS</url-pattern>
        </servlet-mapping>
        <session-config>
                <session-timeout>30</session-timeout>
        </session-config>
</web-app>
```

3.4. WSTools configuration file

The "wstools-config.xml" file is used by the "wstools" taskdef. The "wstools-config.xml" file contains the instructions about what we want "wstools" to do for us. In our example we are using "wstools" to generate the WSDL file for our sample web service. If we started with a WSDL, we could ask "wstools" to generate the Java source files for us as well. In this example though, we started with Java code so we only need to create the WSDL. The "java-wsdl" tag tells "wstools" that we want that WSDL generated for our service. In this example we tell "wstools" to use the file "HelloWorldWSEndpointInterface.java". The config file also tells "wstools" that we want our service to be know as "HellowWorldWS" and we want the SOAP invocation style to be "RPC". Finally, it also declares the target and type namespaces that the WSDL generated definitions should use.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!--
wstools -config wstools-config.xml
-->
<configuration xmlns="http://www.jboss.org/jbossws-tools" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
        <global>
                <package-namespace package="org.jboss.samples.helloworld" namespace="http://org.jboss.ws/
        </global>
        <java-wsdl>
                <service name="HelloWorldWS" endpoint="org.jboss.samples.helloworld.HelloWorldWSEndpointI
                <namespaces target-namespace="http://org.jboss.ws/samples/helloworld" type-namespace="htt
        </java-wsdl>
</configuration>
```

One thing you might notice is that we have a separate endpoint interface class defined in addition to our real web service class. This seperation is not required in JAX-WS 2.0. You could just have the endpoint property reference the actual implementation class "HelloWorldWS.java". If you do just use the real implementing class, then you do not need a separate endpoint interface class.

Having a separated endpoint is particularly relevant to people upgrading from the older JAX-RPC web services architecture under JavaEE 1.4. It allows for them to stay with there older definitions, but still use the new features of JAX-WS 2.0 and migrate to using the new annotaions and web service features gradually.

If you do not have the separate interface class then the "wstools" looks at the annotations and other metadata about the java class that you've specified as the endpoint and comes up with reasonable defaults for all the needed WSDL values. For all new code you more than likely just need the real annotated class that implements the web service. For this example thought, we are going to stay with having a separate interface class.

3.5. Building our web service

To build the web service example do the following:

- ant clean
- ant
- ant deploy

NOTE: In the current version of JBoss Application Server you have to manually deploy the "war" file created by the build, so do not do the "ant deploy" step above. We'll see how to manually deploy in the next section.

3.6. Manually Deploying the web service WAR file

To deploy the web service, open the JMX Console. On a system with the default install of JBoss you will normally find it at: http://localhost:8080/jmx-console/. Depending on your setup you may need to change the url host and port.

Once your at the console find the "Main Deployer" (jboss.system:service=MainDeployer). Select that link. That will take you to the JMX Bean View of the Main Deployer. Now find the "deploy" action that takes a "String" as

it's argument. Type in the full path to your WAR file and click the "invoke" button to deploy your WAR file. For example my full path is: /Users/sgriffith/svn_sandbox/jbossws_head/jbossws-docs/online/tutorial-src/jax-ws-hello-world/helloworldws.war. Your JMX Console will then display a new page titled "JMX MBean Operation Result". In the body of the page you should see "Operation completed successfully without a return value." if it succeeded in deploying.

NOTE: If you did the "ant deploy" step above you'll need to go remove the WAR file that that step deployed before doing this step or this won't work properly.

In your shell or console you should also see something like:

```
07:39:01,467 INFO  [MainDeployer] deploy, url=file:/Users/sgriffith/svn_sandbox/jbossws_head/jbossws-docs
07:39:08,755 INFO  [TomcatDeployment] deploy, ctxPath=/helloworldws, warUrl=.../tmp/deploy/helloworldws30
07:39:09,655 INFO  [StandardContext] Container org.apache.catalina.core.ContainerBase.[jboss.web].[localh
07:39:09,971 INFO  [WSDLFilePublisher] WSDL published to: file:/Users/sgriffith/jboss-5.0.0.Beta2/server/
07:39:10,446 INFO  [ServiceEndpointManager] WebService started: http://Sam-Griffiths.local:8080/hellowor
```

You can see in the console output that our web service was deployed and started. You can also see that it is called "HelloWorldWS" which is what we specified in our "wstools-config.xml" file for the service name. One other thing to notice is that because we have a web.xml file for our WAR, our web service is mapped according to the settings in there.

3.7. View web service in JBoss Web Console

We can also see the deployed web service in the JBoss Web Services Console. If you did a default server install you'll be able to find it at: http://localhost:8080/jbossws/. Once your there you should see a link entitled "View the list of deployed Web Services". Click the word "View" to go to the list of deployed web services. After you've clicked on "View" you'll be taken to a page entitled "Registered Service Endpoints". You should now see our web service Endpoint ID and Addresses displayed in the table. For example this Endpoint address: http://Sam-Griffiths.local:8080/helloworldws/HelloWorldWS?wsdl. If you click on the Endpoint address link, depending on your browser you may see the WSDL XML. On some browsers you can see the WSDL XML source by going to your "View Source" or equivalent menu. Below you can see the WSDL for our sample.

```
<definitions name='HelloWorldWSService' targetNamespace='http://helloworld.samples.jboss.org/' xmlns='htt
 <types>
  <xs:schema targetNamespace='http://helloworld.samples.jboss.org/' version='1.0' xmlns:tns='http://hello
   <xs:element name='sayHello' type='tns:sayHello'/>
   <xs:element name='sayHelloResponse' type='tns:sayHelloResponse'/>
   <xs:complexType name='sayHello'>
    <xs:sequence>
     <xs:element minOccurs='0' name='arg0' type='xs:string'/>
    </xs:sequence>
   </xs:complexType>
   <xs:complexType name='sayHelloResponse'>
    <xs:sequence>
     <xs:element minOccurs='0' name='return' type='xs:string'/>
    </xs:sequence>
   </xs:complexType>
  </xs:schema>
```

```
 </types>
 <message name='HelloWorldWS_sayHello'>
  <part element='tns:sayHello' name='sayHello'/>
 </message>
 <message name='HelloWorldWS_sayHelloResponse'>
  <part element='tns:sayHelloResponse' name='sayHelloResponse'/>
 </message>
 <portType name='HelloWorldWS'>
  <operation name='sayHello' parameterOrder='sayHello'>
   <input message='tns:HelloWorldWS_sayHello'/>
   <output message='tns:HelloWorldWS_sayHelloResponse'/>
  </operation>
 </portType>
 <binding name='HelloWorldWSBinding' type='tns:HelloWorldWS'>
  <soap:binding style='document' transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='sayHello'>
   <soap:operation soapAction=''/>
   <input>
    <soap:body use='literal'/>
   </input>
   <output>
    <soap:body use='literal'/>
   </output>
  </operation>
 </binding>
 <service name='HelloWorldWSService'>
  <port binding='tns:HelloWorldWSBinding' name='HelloWorldWSPort'>
   <soap:address location='http://Sam-Griffiths.local:8080/helloworldws/HelloWorldWS'/>
  </port>
 </service>
</definitions>
```

3.8. Test web service

To test the web service we are going to make use of a commercial XML editor that has a built in tool for testing web services. That tool is Oxygen, but you can use any tool that supports doing web service calls.

In Oxygen go to the "Tools" menu and choose the "WSDL SOAP Analyser" sub-menu. It will bring up a dialog box asking you for the WSDL URL. Put in the value from the JBoss Web Service Console for the ServiceEndpointAddress. For the tutorial web service it is: http://Sam-Griffiths.local:8080/helloworldws/HelloWorldWS?wsdl. Once you've done that, it will parse the WSDL and bring up a test dialog. Change the value for the "Arg0" tag to a string value of someone's name. Now click the "Send" button to call the web service. You should get back a response much like that in the figure below. The figure below shows the request and response values of the web service.

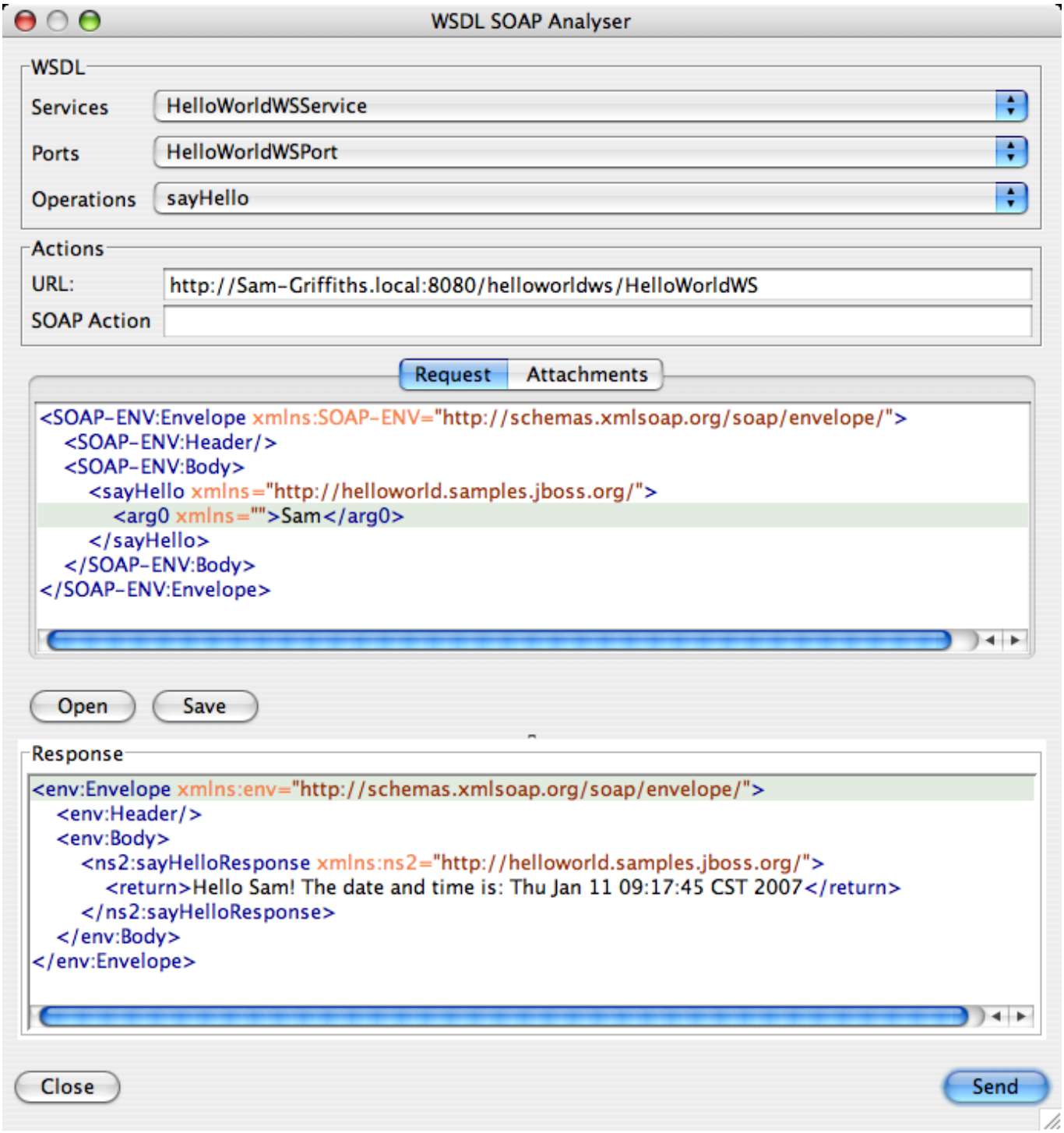

Figure 3.1. Test of Web Services Call in Oxygen

3.9. Where from here?

There is much more to the JAX-WS 2.0 web service framework and more can be found in the JBoss JAX-WS 2.0 Reference Manual. In there you will find more details about specific features such as adding security, mtom and others. Additonally, you'll also want to check out the JBoss IDE 2.0. It has GUI support for creating web services

from with the Eclipse IDE. If your not using the JBoss Eclipse IDE, you can also use NetBeans or the standard Eclipse release from Eclipse.org. Look for more tutorials on our site covering how to use some of these IDE's.

You may also be interested in the following list of links that provide much more information on JAX-WS 2.0 and web services in general.

- ????
- ????
- http://www.soapui.org/jbossws/gettingstarted.html