# More about Webservices

1. **What is web service and what are the advantages of exposing web services ?**

**What are Web Services?**

Web services are application components
Web services communicate using open protocols
Web services are self-contained and self-describing
Web services can be discovered using UDDI
Web services can be used by other applications
XML is the basis for Web services

**How Does it Work?**

The basic Web services platform is XML + HTTP.
XML provides a language which can be used between different platforms and programming languages and still express complex messages and functions.
The HTTP protocol is the most used Internet protocol.

**Web services platform elements:**

SOAP (Simple Object Access Protocol)
UDDI (Universal Description, Discovery and Integration)
WSDL (Web Services Description Language)

- **Interoperability**
- **Implementation Neutrality**
- **Platform Neutrality**

**Why Web Services?**

Interoperability has Highest Priority or System implementation nutrality and platform nutrality.
By using Web services, your application can publish its function or message to the rest of the world.

**What is SOAP?**

SOAP is an XML-based protocol to let applications exchange information over HTTP.
Or more simple: SOAP is a protocol for accessing a Web Service.

SOAP stands for Simple Object Access Protocol
SOAP is a communication protocol
SOAP is a format for sending messages
SOAP is designed to communicate via Internet
SOAP is platform independent
SOAP is language independent
SOAP is based on XML
SOAP is simple and extensible
SOAP allows you to get around firewalls
SOAP is a W3C standard

## SOAP is a vehicle to convey the data

2. **What is WSDL ?**

## What is WSDL?

WSDL is an XML-based language for locating and describing Web services.

WSDL stands for Web Services Description Language
WSDL is based on XML
WSDL is used to describe Web services
WSDL is used to locate Web services
WSDL is a W3C standard

## What is UDDI?

UDDI is a directory service where companies can register and search for Web services.
UDDI stands for Universal Description, Discovery and Integration
UDDI is a directory for storing information about web services
UDDI is a directory of web service interfaces described by WSDL
UDDI communicates via SOAP
UDDI is built into the Microsoft .NET platform

## What are the components of WSDL ?

Three major elements of WSDL that can be defined separately and they are:

**Definition**
**Types**
**Message**
**PortType**
**Binding**
**Service**

## The WSDL Document Structure

The main structure of a WSDL document looks like this:

**<definitions>**
**<types>**

```
  definition of types.....
</types>

<message>
  definition of a message....
</message>

<portType>
    <operation>
        definition of a operation.....
    </operation>
</portType>

<binding>
  definition of a binding....
</binding>

<service>
  definition of a service....
</service>

</definitions>
```

**As per WSDL1.1 there are total 5 components of WSDl.**
**They are Imports,Types,Services,Bindings,Port Type and Messages (ITSBPM -- BITSPM)**

**Imports**

<wsdl:import> references a separate WSDL 1.1 document, with descriptions to be incorporated into this document.
Import: element is used to import other WSDL documents or XML Schemas.

**Messages**

Message: an abstract definition of the data, in the form of a message presented either as an entire document or as arguments to be mapped to a method invocation.
The <message> element describes the data being exchanged between the Web service providers and consumers.
Each Web Service has two messages: input and output.
The input describes the parameters for the Web Service and the output describes the return data from the Web Service.
Each message contains zero or more <part> parameters, one for each parameter of the Web Service's function.
Each <part> parameter associates with a concrete type defined in the <types> container element.
Message is just like the method with method signature and return type.

Lets take a piece of code from the Example Session:

```
  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>
```

**Types**

A Web service needs to define its inputs and outputs and how they are mapped into and out of services.
WSDL <types> element take care of defining the data types that are used by the web service. Types are XML documents, or document parts.
The <types> element defines the data types that are used by the web service.
For maximum platform neutrality, WSDL uses XML Schema syntax to define data types.

**Port Type**

Port type : an abstract set of operations mapped to one or more end points.

The <portType> element is the most important WSDL element.
It describes a web service, the operations that can be performed, and the messages that are involved.
It defines the input and output message of a function .
Port Type–an abstract set of operations supported by one or more endpoints.

a <portType> specifies a
set of named <operation>s which
refer to the <message>s used by each <operation> for <input> and <output>

The <portType> element combines multiple message elements to form a complete oneway or round-trip operation. For example, a <portType> can combine one request and one response message into a single request/response operation. This is most commonly used in SOAP services. A portType can define multiple operations.

```
<portType name="Hello_PortType">
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>
```

The portType element defines a single operation, called sayHello.

The operation itself consists of a single input message SayHelloRequest

The operation itself consists of a single output message SayHelloResponse

**Services**

Service: a collection of related end points encompassing the service definitions in the file; the services map the binding to the port and include any extensibility definitions.

The <service> element defines the ports supported by the Web service.
For each of the supported protocols, there is one port element.
The service element is a collection of ports.
a <service>, specifies a <port> (URI) at which it is available and refers to a binding for the port.

Here is a pice of code from Example Session:

```
 <service name="Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address
        location="http://www.examples.com/SayHello/">
    </port>
  </service>
```

**What do you mean by port in WSDL ?**

A <port> element defines an individual endpoint by specifying a single address for a binding.

The port element has two attributes - the name attribute and the binding attribute.
The name attribute provides a unique name among all ports defined within in the enclosing WSDL document.
The binding attribute refers to the binding using the linking rules defined by WSDL.
Binding extensibility elements (1) are used to specify the address information for the port.
A port MUST NOT specify more than one address.
A port MUST NOT specify any binding information other than address information.

**Bindings**

Binding: the concrete protocol and data formats for the operations and messages defined for a particular port type.

The <binding> element provides specific details on how a portType operation will actually be transmitted over the wire.
The bindings can be made available via multiple transports, including HTTP GET, HTTP POST, or SOAP.
The bindings provide concrete information on what protocol is being used to transfer portType operations.
The bindings provide information where the service is located.
For SOAP protocol, the binding is <soap:binding>, and the transport is SOAP messages on top of HTTP protocol.
You can specify multiple bindings for a single portType.
The binding element has two attributes - the name attribute and the type attribute.
A binding defines message format and protocol details for operations and messages defined by a particular portType.
There may be any number of bindings for a given portType.

a <binding> specifies the
style of interaction (e.g. RPC)
the transport protocol used (e.g. HTTP)
the <operation>s defined along with encodingStyle for their <input> and <output>

The binding element has two attributes - the name attribute and the type attribute.

<binding name="Hello_Binding" type="tns:Hello_PortType">

**What is WSDL style and Which style of WSDL should I use?**
A WSDL binding describes how the service is bound to a messaging protocol, particularly the SOAP messaging protocol.
A WSDL SOAP binding can be either a Remote Procedure Call (RPC) style binding or a document style binding.
A SOAP binding can also have an encoded use or a literal use. This gives you four style/use models:
RPC/encoded
RPC/literal
Document/encoded
Document/literal

**What can be done with the Web services API ?**

There are three things that can be performed with the webservice API.

1. Web service creation using java2wsdl
2. Web service client creation using wsdl2java
3. Skeleton creation from an existing WSDL using wsdl2java

# Webservice Technology Apache Axis2

## How to create a web service using Apache Axis2

### Capsule

**1. Write a java class with the appropriate java methods with mandatory relevent method signature or parameters and write the business logic for the web service method.**
**2. Create or modify the services.xml file with the entry of the above defined class name with the package name.**
**3. Write the web.xml file for the Axis servlet.**
**4. Copy all the relevent *.mar files inside the directory called modules. It is better to copy the complete module directory.**
**5. Copy all the relevent *.jar files inside the lib.**
**6. Run the ant script which will do "from java to wsdl".**
**7. Create the war file for deployment with the following structure.**

```
<webservice.war>
        /WEB-INF
                /classes
                /lib/*.jar
                /modules/*.mar
                /services/*.aar
                /web.xml
```

The following is the step by step process to create a web service in axis2.

Step 1. Create a java class which will be your web service class and define all the methods with appropriate parameters and return type.

**SimpleWebService.java**

```
package com.ddlab.webservice.axis2;

public class SimpleWebService
{
        public String getName( String name )
        {
                System.out.println("Name from webservice client--->"+name);
                return "Name from server "+name;
        }
}
```

Step 2. Define the "**services.xml**" file in the folder **resource/META-INF** as follows.

```
<service name="simplewebservice" scope="application" targetNamespace="http://service.simplewebservice/">
   <description>Simple Webservice using axis2 , developed by - Debadatta Mishra</description>
   <messageReceivers>
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
   </messageReceivers>
   <schema schemaNamespace="http://service.simplewebservice/xsd"/>
   <parameter name="ServiceClass">com.ddlab.webservice.axis2.SimpleWebService</parameter>
</service>
```

Step 3. Configure Axis servlet in your **web.xml** file as follows.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>Apache-Axis2</display-name>
   <servlet>
      <servlet-name>AxisServlet</servlet-name>
      <display-name>Apache-Axis Servlet</display-name>
      <servlet-class>org.apache.axis2.transport.http.AxisServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>AxisServlet</servlet-name>
      <url-pattern>/services/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Step 4. Configure the structure of WEB-INF as follows.

```
WEB-INF/
        classes
        lib/*.jar
        modules/*.mar
        services/*.aar
        web.xml
```

The structure of **web.xml** file for Apache Axis2 is given below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>Apache-Axis2</display-name>
   <servlet>
      <servlet-name>AxisServlet</servlet-name>
      <display-name>Apache-Axis Servlet</display-name>
      <servlet-class>org.apache.axis2.transport.http.AxisServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>AxisServlet</servlet-name>
      <url-pattern>/services/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Step 5. Run the following ant script to create and deploy your web service in web/application server

```
<!-- ANT Script to generate the *.AAR file. -->

<project basedir="." default="create.war">

        <property name="server.src.dir"                 value="${basedir}/serversrc" />
        <property name="lib.dir"                        value="${basedir}/lib" />
        <property name="build.dir"                      value="${basedir}/build"/>
        <property name="dist.dir"                       value="${basedir}/dist"/>
```

```
<property name="web.dir"                    value="${basedir}/web"/>
<property name="webinf.dir"                 value="${web.dir}/WEB-INF"/>
<property name="services.dir"               value="${webinf.dir}/services"/>
<property name="class.name"
value="com.ddlab.webservice.axis2.SimpleWebService"/>
<property name="aar.file.name"              value="simplewebservice.aar"/>
<property name="war.file.name"              value="simplewebservice.war"/>

<!-- Create classpath setting -->
<path id="axis2.classpath">
        <fileset dir="${lib.dir}">
                <include name="*.jar"/>
        </fileset>
</path>

<!-- Delete and Create the required directories -->
<target name="clean">
        <delete dir="${build.dir}"/>
        <delete dir="${dist.dir}"/>
        <delete dir="${services.dir}"/>
        <mkdir dir="${services.dir}"/>
        <mkdir dir="${build.dir}"/>
        <mkdir dir="${dist.dir}"/>
        <mkdir dir="${build.dir}/classes"/>
</target>

<!-- Compile the services -->
<target name="compile.service" depends="clean">
        <javac debug="on"
  fork="true"
  destdir="${build.dir}/classes"
  srcdir="${server.src.dir}"
  classpathref="axis2.classpath">
        </javac>
</target>

<!-- Generate the WSDL -->
<target name="generate.wsdl" depends="compile.service">
        <taskdef name="java2wsdl"
    classname="org.apache.ws.java2wsdl.Java2WSDLTask"
    classpathref="axis2.classpath"/>
        <java2wsdl className="${class.name}"
     outputLocation="${build.dir}"
     targetNamespace="http://service.simplewebservice/"
     schemaTargetNamespace="http://service.simplewebservice/xsd">
                <classpath>
                        <pathelement path="${axis2.classpath}"/>
                        <pathelement location="${build.dir}/classes"/>
                </classpath>
        </java2wsdl>
</target>

<!-- Generate the Services -->
<target name="generate.service" depends="generate.wsdl">
        <copy toDir="${build.dir}/classes" failonerror="false">
                <fileset dir="${basedir}/resources">
                        <include name="**/*.xml"/>
```

```
                </fileset>
            </copy>
            <jar destfile="${build.dir}/${aar.file.name}">
                <fileset dir="${build.dir}/classes"/>
            </jar>
        </target>

        <!-- create the .war file to deploy into server -->
        <target name="create.war" depends="generate.service">
            <delete dir="${webinf.dir}/lib"/>
            <mkdir dir="${webinf.dir}/lib"/>
            <copy toDir="${webinf.dir}/lib" >
                <fileset dir="${lib.dir}">
                    <include name="**/*.jar"/>
                </fileset>
            </copy>
            <copy toDir="${services.dir}" >
                <fileset dir="${build.dir}">
                    <include name="**/*.aar"/>
                </fileset>
            </copy>
            <jar destfile="${dist.dir}/${war.file.name}">
                <fileset dir="${web.dir}"/>
            </jar>
        </target>
</project>
```

**How to create a web service from an existing wsdl file using Apache Axis2**

**Capsule**

**1. In a java project, copy the directory having the .wsdl file.**
**2. Copy all the related *.jar file inside lib from Apache Axis2.**
**3. Run the ant script "service-build.xml" to generate the skeleton java source code and services.xml.**
**4. Modify the java file having the name *Skeleton and write your business logic.**
**5. Run the ant script "modify-deploy.xml" to generate the *.aar file.**
**6. Write the web.xml file for the Axis servlet.**
**7. Copy all the relevent *.mar files inside the directory called modules. It is better to copy the complete module directory.**
**8. Copy all the relevent *.jar files inside the lib.**
**9. Run the ant script which will do "from java to wsdl".**

**10. Create the war file for deployment with the following structure.**

**<webservice.war>**
**/WEB-INF**
**/classes**
**/lib/*.jar**
**/modules/*.mar**
**/services/*.aar**
**/web.xml**

The initial ant script is given below.

File name "**service-build.xml**"

```
<project name="simplewebservice" basedir="." default="gen.service">

        <property name="wsdl.file.name"          value="simplewebservice" />
        <property name="wsdl.file"             value="${basedir}/wsdls/${wsdl.file.name}" />
        <property name="wsdl.uri"              value="${wsdl.file}.wsdl" />
        <property name="gen.dir"              value="${basedir}/gensrc" />
        <property name="build.dir"             value="build" />
        <property name="lib.dir"              value="lib" />
        <property name="dist.dir"             value="${basedir}/dist"/>
        <property name="web.dir"              value="${basedir}/web"/>
        <property name="webinf.dir"             value="${web.dir}/WEB-INF"/>
        <property name="services.dir"           value="${webinf.dir}/services"/>
        <property name="war.file.name"            value="${ant.project.name}.war"/>

        <path id="classpath">
                <fileset dir="${lib.dir}">
                        <include name="**/*.jar" />
                </fileset>
        </path>

   <taskdef name="codegen" classname="org.apache.axis2.tool.ant.AntCodegenTask"
classpathref="classpath"/>

        <target name="clean">
                <echo message="deleting the old directories and creation of new directories" />
                <delete dir="${build.dir}" />
                <delete dir="${gen.dir}" />
                <delete dir="${webinf.dir}/lib"/>
                <delete dir="${dist.dir}"/>
        </target>

        <target name="init" depends="clean">
      <echo message="deleting the old directories and creation of new directories" />
      <mkdir dir="${build.dir}" />
      <mkdir dir="${build.dir}/${war.file.name}"/>
      <mkdir dir="${gen.dir}" />
      <mkdir dir="${webinf.dir}/lib"/>
      <mkdir dir="${dist.dir}"/>
   </target>

        <target name="gen.service" depends="init">
           <codegen
            wsdlfilename="${wsdl.uri}"
            output="${gen.dir}"
            serverside="true"
            generateservicexml="true" />
        </target>
</project>
```

The ant script after modification and deployment is given below.

File name "**modify-deploy.xml**"

```
<project name="simplewebservice" basedir="." default="create.war">

   <property name="build.dir"              value="build" />
```

```
  <property name="lib.dir"                value="lib" />
  <property name="dist.dir"               value="${basedir}/dist"/>
  <property name="gen.dir"                value="${basedir}/gensrc" />
        <property name="web.dir"                value="${basedir}/web"/>
  <property name="webinf.dir"             value="${web.dir}/WEB-INF"/>
  <property name="services.dir"           value="${webinf.dir}/services"/>
  <property name="war.file.name"          value="${ant.project.name}.war"/>

 <target name="create.war">
    <ant antfile="${gen.dir}/build.xml" dir="${gen.dir}" inheritrefs="false" />
    <delete dir="${webinf.dir}/lib"/>
    <mkdir dir="${webinf.dir}/lib"/>
    <copy toDir="${webinf.dir}/lib" flatten="true">
       <fileset dir="${lib.dir}">
          <include name="**/*.jar"/>
       </fileset>
    </copy>
    <copy toDir="${services.dir}" flatten="true">
       <fileset dir="${gen.dir}/${build.dir}">
          <include name="**/*.aar"/>
       </fileset>
    </copy>
    <echo file="${services.dir}/services.list" message="${aar.file.name}"/>
    <copy toDir="${build.dir}/${war.file.name}">
       <fileset dir="${web.dir}"/>
    </copy>
    <jar destfile="${dist.dir}/${war.file.name}">
       <fileset dir="${web.dir}"/>
    </jar>
 </target>
</project>
```

**How to consume a web service using Generated stub in Apache Axis2**

**Capsule**

**1. Access the already hosted WSDL url in any browser. If you are able to see the xml structure, save it as <anyname>.wsdl in a directory.**
**2. In a java project, copy the directory having the .wsdl file.**
**3. Copy all the related *.jar file inside lib from Apache Axis2.**
**4. Run the ant script to execute "From wsdl to java".**
**5. Finally you will get the client stub jar and put it in the classpath.**
**6. Write the code to invoke the web service by passing the relevent data.**

Let us see the sample java code.

**TestUsingAxis2GeneratedStub.java**

```
import com.ddlab.webservice.client.SimpleWebServiceStub;
import com.ddlab.webservice.client.SimpleWebServiceStub.GetName;
import com.ddlab.webservice.client.SimpleWebServiceStub.GetNameResponse;
```

```
public class TestUsingAxis2GeneratedStub
{
	public static void main(String[] args) throws Exception
	{
		String targetEndPoint = "http://localhost:8080/simplewebservice/services/simplewebservice?wsdl";
		try
		{
			SimpleWebServiceStub stub = new SimpleWebServiceStub(targetEndPoint);
			GetName name = new GetName();
			name.setName("Deba11");
			GetNameResponse response = stub.getName(name);
			System.out.println("Response Value :::"+response.get_return());
		}
		catch (Exception e)
		{
			e.printStackTrace();
		}
	}

}
```

Let us see the ant script which will generate the stub.

```
<project name="testucwebservice" basedir="." default="generate.client">

	<property name="wsdl.uri"		value="${basedir}/wsdlsrc/SimpleWebService.wsdl"/>
  <property name="clientbuild.dir"	value="${basedir}/clientbuild"/>
	<property name="client.target"			value="${clientbuild.dir}/client"/>
	<property name="lib.dir"		value="lib/axis2-1.5"/>
	<property name="package.name"		value="com.ddlab.webservice.client"/>

	<path id="axis2.classpath">
    <fileset dir="${lib.dir}">
      <include name="*.jar"/>
    </fileset>
  </path>

	<target name="clean">
			<echo message="deleting the old directories "/>
			<delete dir="${clientbuild.dir}"/>
	 </target>

	<target name="init">
		<echo message="deleting the old directories and creation of new directories"/>
		<delete dir="${clientbuild.dir}"/>
    <mkdir dir="${clientbuild.dir}"/>
    <mkdir dir="${clientbuild.dir}/client"/>
  </target>

	<target name="generate.client">
		<delete dir="${client.target}"/>
		<mkdir dir="${client.target}"/>
     <java classname="org.apache.axis2.wsdl.WSDL2Java" fork="true" classpathref="axis2.classpath">
	<arg line="-uri ${wsdl.uri}"/>
			<arg line="-s"/>
			<arg line="-l java"/>
	<arg line="-p ${package.name}"/>
```

```
        <arg line="-d adb"/>
        <arg line="-o ${clientbuild.dir}/client"/>
      </java>
      <ant antfile="${clientbuild.dir}/client/build.xml" inheritall="false" />
   </target>

</project>
```

**How to invoke a web service dynamically using Apache Axis2**

The sample code is given below.

**AxiomWebserviceConsumer1.java**

```
import java.io.StringReader;
import java.util.Iterator;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;

import org.apache.axiom.om.OMAbstractFactory;
import org.apache.axiom.om.OMElement;
import org.apache.axiom.om.OMFactory;
import org.apache.axiom.om.OMNamespace;
import org.apache.axiom.soap.SOAPBody;
import org.apache.axiom.soap.SOAPEnvelope;
import org.apache.axiom.soap.SOAPFactory;
import org.apache.axiom.soap.impl.builder.StAXSOAPModelBuilder;
import org.apache.axis2.Constants;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;

public class AxiomWebserviceConsumer1
{
        private static EndpointReference targetEPR =
                new EndpointReference(
                "http://localhost:8080/simplewebservice/services/simplewebservice?wsdl");
        private static String methodName = "getName";

        public static OMElement getPayload( ) throws Exception
        {
                OMFactory fac = OMAbstractFactory.getOMFactory();
                OMNamespace omNs = fac.createOMNamespace("http://service.simplewebservice/xsd", "tns");
                OMElement method = fac.createOMElement(methodName, omNs);

                OMElement empValue = fac.createOMElement("name", omNs);
                empValue.setText("Debadatta Mishra");//Data

                method.addChild(empValue);

                return method;
        }

        public static SOAPEnvelope getSoapEvelope( OMElement omElement )
        {
```

```
        SOAPFactory fac = OMAbstractFactory.getSOAP11Factory();
SOAPEnvelope envelope = fac.getDefaultEnvelope();
OMNamespace omNs =   fac.createOMNamespace(
   "http://ws.apache.org/axis2/xsd", "ns1");
envelope.getBody().addChild(omElement);
return envelope;
   }

   public static void showResponseResult( Iterator itr ) throws Exception
   {
        System.out.println("---------- Response ----------");
        while( itr.hasNext() )
        {
            OMElement result11 = (OMElement) itr.next();
            System.out.println(  getSoapEvelope(result11).toStringWithConsume());

            StringReader sr = new StringReader(getSoapEvelope(result11).toStringWithConsume());
            XMLStreamReader parser = XMLInputFactory.newInstance().createXMLStreamReader(sr);

            StAXSOAPModelBuilder builder = new StAXSOAPModelBuilder(parser, null);
            System.out.println(builder.getDocumentElement().getLocalName());
            SOAPEnvelope envelope = (SOAPEnvelope) builder.getDocumentElement();
            SOAPBody soapBody = envelope.getBody();
            Iterator itr1 = soapBody.getAllAttributes();
            while( itr1.hasNext() )
            {
                System.out.println("--->"+itr1.next());
            }
        }
   }

   public static void main(String[] args) throws Exception
   {
        OMElement getPricePayload = getPayload( );

        Options options = new Options();
        options.setTo(targetEPR);
        options.setTransportInProtocol(Constants.TRANSPORT_HTTP);

        ServiceClient sender = new ServiceClient();
        sender.setOptions(options);

        OMElement result = sender.sendReceive(getPricePayload);
        System.out.println(result.getQName());

        System.out.println("--------------------XML Message----------------------");
        System.out.println(getPricePayload.toStringWithConsume());
        System.out.println("--------------------XML Message----------------------");

        SOAPFactory fac = OMAbstractFactory.getSOAP11Factory();
SOAPEnvelope envelope = fac.getDefaultEnvelope();
OMNamespace omNs =   fac.createOMNamespace(
   "http://ws.apache.org/axis2/xsd", "ns1");
envelope.getBody().addChild(getPricePayload);
System.out.println(envelope);
```

```
                Iterator itr = result.getChildElements();
                showResponseResult(itr);
        }

}
```

# Web service Technology - JAX-RPC

**Capsule**

To create a webservice using JAX-RPC, follow the following steps.

**1. Create an interface which extends java.rmi.Remote interface and provide a method with the appropriate method/s and parameter/s.**
**2. Write an implementation class for the above defined interface.**
**3. Write an XML file with the exact name "jaxrpc-ri-runtime.xml" and provide the relevent information for the web service name,interface name,**
**implementation class name,name of the wsdl, name of the service, name of the port and the url pattern. This file should be available inside the**
**war and the WEB-INF folder.**
**4. Write an XML file with the exact name "serverconfig.xml" and provide the information for name of the web service, target namespace,**
**java package name and the interface name. This file is used to generate the wsdl file.**
**5. Write "web.xml" file for the web application by providing the name of the JAX-RPC servlet.**
**The name of the Servlet is "com.sun.xml.rpc.server.http.JAXRPCServlet".**
**6. Run the build script and it will generate the required .wsdl file.**
**7. In the above generated wsdl file, you will find a key word "REPLACE_WITH_ACTUAL_URL". In this string you have to replace with the actual**
**web url. You will find the key word "REPLACE_WITH_ACTUAL_URL" in the following string line.**
**<soap:address location="REPLACE_WITH_ACTUAL_URL"/> and the modified line should be as**
**<soap:address location="http://localhost:8080/SimpleWebService_jaxrpc/SimpleWebService?WSDL"/>**
**8. Now run another build script which will package your war file for deployment.**
**9. The jar files required for development of web service using JAX-RPC is given below.**

activation.jar,ant.jar,FastInfoset.jar,jaxb-api.jar,jaxb-impl.jar,jaxb-xjc.jar,jaxb1-impl.jar,jaxr-api.jar,jaxr-impl.jar,jaxrpc-api.jar,jaxrpc-impl.jar,jaxrpc-spi.jar,mail.jar,relaxngDatatype.jar,saaj-api.jar,saaj-impl.jar,xalan.jar,xercesImpl.jar,xsdlib.jar

See the code below.

**SimpleWebService.java**

```
package com.ddlab.webservice.jaxrpc;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface SimpleWebService extends Remote
{
	public String getName( String name ) throws RemoteException;
}
```

**SimpleWebServiceImpl.java**

```
package com.ddlab.webservice.jaxrpc;

public class SimpleWebServiceImpl implements SimpleWebService
{
	public String getName( String name )
	{
		System.out.println("Name from webservice client--->"+name);
		return "Name from server "+name;
	}
}
```

**jaxrpc-ri-runtime.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-rpc/ri/runtime' version='1.0'>
  <endpoint
    name='SimpleWebService'
    interface='com.ddlab.webservice.jaxrpc.SimpleWebService'
    implementation='com.ddlab.webservice.jaxrpc.SimpleWebServiceImpl'
    tie='com.ddlab.webservice.jaxrpc.SimpleWebService_Tie'
    model='/WEB-INF/model-wsdl-rpcenc.xml.gz'
    wsdl='/WEB-INF/SimpleWebService.wsdl'
    service='{http://deba.org/wsdl}SimpleWebService'
    port='{http://deba.org/wsdl}SimpleWebServicePort'
    urlpattern='/SimpleWebService'/>
</endpoints>
```

**serverconfig.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service
     name="SimpleWebService"
     targetNamespace="http://deba.org/wsdl"
     typeNamespace="http://deba.org/types"
     packageName="com.ddlab.webservice.jaxrpc">
     <interface name="com.ddlab.webservice.jaxrpc.SimpleWebService"/>
  </service>
</configuration>
```

**web.xml**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<web-app>
        <display-name>JAX-RPC SimpleWebService Web Application Sample</display-name>
        <description>Sample Application for JAX-RPC using static stubs.</description>
        <listener>
                <listener-class>com.sun.xml.rpc.server.http.JAXRPCContextListener</listener-class>
        </listener>
        <servlet>
                <servlet-name>SimpleWebService</servlet-name>
                <display-name>SimpleWebService</display-name>
                <description>JAX-RPC endpoint - Hello</description>
                <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-class>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>SimpleWebService</servlet-name>
                <url-pattern>/SimpleWebService</url-pattern>
        </servlet-mapping>
        <session-config>
                <session-timeout>60</session-timeout>
        </session-config>
</web-app>
```

**server-build.xml**

```
<!--
        This is a generic ant build script for building the server side code.
        Upon running the script, it will create a war file and you can
        deploy in any application server.
        Author : Debadatta Mishra
```

```
        Project Name : SimpleWebService_jaxrpc
-->
<project name="SimpleWebService_jaxrpc" default="createwar" basedir=".">

        <property name="server.config.dir"        value="${basedir}/server-config"/>
        <property name="config.rpcenc.file"       value="${server.config.dir}/serverconfig.xml"/>
        <property name="src.dir"                  value="${basedir}/serversrc"/>
        <property name="bin.dir"                  value="${basedir}/bin"/>
        <property name="lib.dir"                  value="${basedir}/lib"/>
        <property name="dist.dir"                 value="${basedir}/dist"/>
        <property name="build.dir"                value="${basedir}/build"/>
        <property name="war.dir"                  value="${basedir}/${ant.project.name}.war"/>
        <property name="webinf.dir"               value="${war.dir}/WEB-INF"/>
        <property name="webinf.classes.dir"       value="${war.dir}/WEB-INF/classes"/>
        <property name="webinf.lib.dir"           value="${war.dir}/WEB-INF/lib"/>
        <property name="service.jar"              value="${dist.dir}/service.jar"/>
        <property name="webservice.jar"           value="${dist.dir}/${ant.project.name}.jar"/>
        <property name="model.rpcenc.file"        value="model-wsdl-rpcenc.xml.gz"/>
        <property name="wsdl.file.name"           value="SimpleWebService?WSDL"/>

        <!-- Setting the ClassPath -->
        <path id="classpath">
                <fileset dir="${lib.dir}">
                        <include name="**/*.jar" />
                </fileset>
        </path>

        <!-- Setting the wscompile -->
        <taskdef name="wscompile" classname="com.sun.xml.rpc.tools.ant.Wscompile" >
                <classpath refid="classpath" />
        </taskdef>

        <target name="clean">
                <delete dir="${dist.dir}" />
                <delete dir="${bin.dir}" />
                <delete dir="${war.dir}" />
                <delete dir="${build.dir}" />
        </target>

        <!-- Delete and create the required directories -->
        <target name="init" depends="clean" >
                <mkdir dir="${bin.dir}" />
                <mkdir dir="${dist.dir}" />
                <mkdir dir="${war.dir}" />
                <mkdir dir="${build.dir}" />
                <mkdir dir="${webinf.dir}/classes" />
        </target>

        <!-- Compile the source and create the jar file -->
        <target name="compile.src" depends="init">
                <javac srcdir="${src.dir}" destdir="${bin.dir}" debug="on" fork="yes">
                        <classpath refid="classpath" />
                </javac>
                <jar destfile="${service.jar}" basedir="${bin.dir}" />
        </target>

        <!-- Generate the webservice -->
```

```
        <target name="generate-service" depends="compile.src">
                <wscompile keep="true" server="true" base="${build.dir}" xPrintStackTrace="true"
verbose="false"
                        model="${build.dir}/${model.rpcenc.file}" classpath="${bin.dir}"
config="${config.rpcenc.file}" >
                        <classpath>
                                <path refid="classpath" />
                        </classpath>
                </wscompile>
                <copy todir="${build.dir}/classes">
                        <fileset dir="${build.dir}" includes="**/*.class"/>
                </copy>
                <jar destfile="${webservice.jar}" basedir="${build.dir}/classes" />

                <replace casesensitive="true" file="${build.dir}/SimpleWebService.wsdl"
token="REPLACE_WITH_ACTUAL_URL"
                        value="http://localhost:8080/${ant.project.name}/SimpleWebService?WSDL" />
        </target>

        <target name="createwar" depends="generate-service">
      <copy todir="${webinf.dir}/lib" flatten="true">
        <fileset dir="${lib.dir}" includes="**/*.jar" />
      </copy>
      <copy todir="${webinf.dir}/lib" flatten="true">
        <fileset dir="${dist.dir}" includes="**/*.jar" />
      </copy>
      <copy todir="${webinf.dir}" flatten="true">
        <fileset dir="${build.dir}" includes="**/*.wsdl,*.gz" />
      </copy>
      <copy todir="${webinf.dir}" flatten="true">
        <fileset dir="${server.config.dir}" excludes="serverconfig.xml" />
      </copy>
      <jar destfile="${dist.dir}/${ant.project.name}.war" basedir="${war.dir}" />
    </target>

</project>
```

In the above case, the URL to access is
http://localhost:8080/SimpleWebService_jaxrpc/SimpleWebService?WSDL

**Creation of Web service from the existing WSDL file**

There will be some situation where you will be able to generate a web service with your custom logic from a wsdl. Think about a situation where a company is using a web service and they want to expand and enhance it with more complex logic but they will sustain the same wsld format. In this situation, we have to generate the web service from the existing wsdl file.

**Capsule**

Follow the following steps.

**1. Create a directory called "server-config" and put the wsdl file. you may get the wsdl file from an external**

**location.**
**2. In the above directory create an xml file called "config-wsdl.xml" and provide the location of the wsdl file and the**
**java package name to generate the java skeleton classes. It is always advisable to keep the both "server-config" and**
**"config-wsdl.xml" file in one directory.**
**3. Write "web.xml" file for the web application by providing the name of the JAX-RPC servlet.**
**The name of the Servlet is "com.sun.xml.rpc.server.http.JAXRPCServlet".**
**4. Run the build script and generated the required skeleton java classes. In this build script we have to point to the**
**location of JAX-RPC directory, for example the location may be "F:/dev/jwsdp-2.0/jaxrpc/bin/wscompile.bat". We can generate the**
**web service from wsdl by executing "wscompile" provided by JAX-RPC.**
**5. After generating the java skeleton classes, modify the java file ending with the name "_PortTypeImpl" and provide your business**
**logic. If the file is not available, write a class called "<webservice name>_PortTypeImpl.java" and put it in the same package.**
**6. Write an XML file with the exact name "jaxrpc-ri-runtime.xml" and provide the relevent information for the web service name,interface name,**
**implementation class name,name of the wsdl, name of the service, name of the port and the url pattern. This file should be available inside the**
**war and the WEB-INF folder.**
**7. Run another build script to package your web application in the form of war for deployment.**
**8. The following jar files are required to develop this service.**

fastinfoset/FastInfoset.jar,
jaxb/jaxb-api.jar,jaxb/jaxb-impl.jar,jaxb/jaxb-xjc.jar,jaxb/jaxb1-impl.jar,
jaxp/dom.jar,jaxp/jaxp-api.jar,jaxp/sax.jar,jaxp/xalan.jar,jaxp/xercesImpl.jar,
jaxr/jaxr-api.jar,jaxr/jaxr-impl.jar,
jaxrpc/jaxrpc-api.jar,jaxrpc/jaxrpc-impl.jar,jaxrpc/jaxrpc-spi.jar,
jaxws/jaxws-api.jar,jaxws/jaxws-rt.jar,jaxws/jaxws-tools.jar,jaxws/jsr181-api.jar,jaxws/jsr250-api.jar,
jwsdp-shared/activation.jar,jwsdp-shared/jaas.jar,jwsdp-shared/jta-spec1_0_1.jar,jwsdp-shared/mail.jar,
jwsdp-shared/relaxngDatatype.jar,jwsdp-shared/resolver.jar,jwsdp-shared/xmlsec.jar,jwsdp-shared/xsdlib.jar,
saaj/saaj-api.jar,saaj/saaj-impl.jar, sjsxp/jsr173_api.jar,sjsxp/sjsxp.jar, xmldsig/xmldsig.jar

All the above jar files are located in the directory of JWSDP2.0. ie Java web service developers pack.

Let us see the all the relevent code below.

**config-wsdl.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
   xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
   <wsdl location="./server-config/SimpleWebService.wsdl"  packageName="com.ddlab.rnd.ws.jaxrpc">
   </wsdl>
</configuration>
```

**jaxrpc-ri-runtime.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-rpc/ri/runtime' version='1.0'>
  <endpoint
    name='SimpleWebService'
```

```
   interface='com.ddlab.rnd.ws.jaxrpc.SimpleWebService_PortType'
   implementation='com.ddlab.rnd.ws.jaxrpc.SimpleWebService_PortTypeImpl'
   tie='com.ddlab.rnd.ws.jaxrpc.SimpleWebService_PortType_Tie'
   model='/WEB-INF/model.gz'
   wsdl='/WEB-INF/SimpleWebService.wsdl'
   service='{http://deba.org/wsdl}SimpleWebService'
   port='{http://deba.org/wsdl}SimpleWebServicePort'
   urlpattern='/SimpleWebService'/>
</endpoints>
```

**web.xml**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<web-app>
        <display-name>JAX-RPC SimpleWebService Web Application Sample</display-name>
        <description>Sample Application for JAX-RPC using static stubs.</description>
        <listener>
                <listener-class>com.sun.xml.rpc.server.http.JAXRPCContextListener</listener-class>
        </listener>
        <servlet>
                <servlet-name>SimpleWebService</servlet-name>
                <display-name>SimpleWebService</display-name>
                <description>JAX-RPC endpoint - Hello</description>
                <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-class>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>SimpleWebService</servlet-name>
                <url-pattern>/SimpleWebService</url-pattern>
        </servlet-mapping>
        <session-config>
                <session-timeout>60</session-timeout>
        </session-config>
</web-app>
```

**SimpleWebService_PortTypeImpl.java**

```
package com.ddlab.rnd.ws.jaxrpc;
import java.rmi.RemoteException;

public class SimpleWebService_PortTypeImpl implements SimpleWebService_PortType
{
   @Override
   public String getName(String string_1) throws RemoteException {
      System.out.println("****************** JAX-RPC Webservice Method Invocation
********************");
      System.out.println("****************** JAX-RPC Webservice Method Invocation
********************");
      System.out.println("****************** JAX-RPC Webservice Method Invocation
********************");

      System.out.println("string_1---------->"+string_1);
```

```
        return string_1;
    }
}
```

**service-build.xml**

```
<project name="SimpleWebService_jaxrpc" basedir="." default="copy.src">

    <property name="wsdl.uri"              value="${basedir}/wsdls/SimpleWebService.wsdl" />
    <property name="gen.dir"               value="${basedir}/gensrc" />
    <property name="build.dir"             value="build" />
    <property name="lib.dir"               value="lib" />
    <property name="gen.src.dir"           value="${gen.dir}/src" />
    <property name="config.wsdl"           value="${basedir}/server-config/config-wsdl.xml"/>
    <property name="dist.dir"              value="${basedir}/dist"/>
    <property name="war.dir"               value="${basedir}/${ant.project.name}.war"/>
    <property name="webinf.dir"            value="${war.dir}/WEB-INF"/>
    <property name="webinf.classes.dir"    value="${war.dir}/WEB-INF/classes"/>
    <property name="webinf.lib.dir"        value="${war.dir}/WEB-INF/lib"/>
    <property name="bin.dir"               value="${basedir}/bin" />
    <property name="wscompile"             value="F:/dev/jwsdp-2.0/jaxrpc/bin/wscompile.bat"/>

    <path id="classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar" />
        </fileset>
    </path>

    <target name="clean">
        <echo message="deleting the old directories and creation of new directories" />
        <delete dir="${build.dir}" />
        <delete dir="${gen.dir}" />
        <delete dir="${dist.dir}" />
        <delete dir="${bin.dir}" />
        <delete dir="${war.dir}" />
    </target>

    <target name="init" depends="clean">
        <echo message="deleting the old directories and creation of new directories" />
        <mkdir dir="${build.dir}" />
        <mkdir dir="${gen.dir}" />
        <mkdir dir="${bin.dir}" />
        <mkdir dir="${dist.dir}" />
        <mkdir dir="${war.dir}" />
        <mkdir dir="${webinf.dir}/classes" />
    </target>

    <target name="generate.service" depends="init" description="Generates ties">
        <exec executable="${wscompile}">
            <arg line="-model ${build.dir}/model.gz"/>
            <arg line="-verbose -gen:server -d ${build.dir} ${config.wsdl}"/>
            <arg line="-keep"/>
        </exec>
    </target>

    <target name="copy.src" depends="generate.service">
        <copy todir="${gen.dir}">
            <fileset dir="${build.dir}" includes="**/*.java" />
```

```
		</copy>
	</target>

</project>
```

**server-deployer.xml**

```
<!--
	This is a generic ant build script for deploying the server side code.
	Upon running the script, it will create a war file and you can
	deploy in any application server.
	Author : Debadatta Mishra
	Project Name : SimpleWebService
-->
<project name="SimpleWebService_jaxrpc" default="createwar" basedir=".">

	<property name="dist.dir"		value="${basedir}/dist"/>
	<property name="gen.dir"		value="${basedir}/gensrc" />
	<property name="bin.dir"		value="${basedir}/bin" />
	<property name="build.dir"		value="build" />
	<property name="lib.dir"		value="lib" />
	<property name="gen.src.dir"		value="${gen.dir}/src" />
	<property name="service.jar"		value="${dist.dir}/service.jar"/>
	<property name="war.dir"		value="${basedir}/${ant.project.name}.war"/>
	<property name="webinf.dir"		value="${war.dir}/WEB-INF"/>
	<property name="service.jar"		value="${dist.dir}/service.jar"/>
	<property name="server.config.dir"	value="${basedir}/server-config"/>
	<property name="wscompile"		value="F:/dev/jwsdp-2.0/jaxrpc/bin/wscompile.bat"/>

	<path id="classpath">
		<fileset dir="${lib.dir}">
			<include name="**/*.jar" />
		</fileset>
	</path>

	<target name="compile.src">
		<javac srcdir="${gen.dir}" destdir="${bin.dir}" debug="on" fork="yes">
			<classpath refid="classpath" />
		</javac>
		<jar destfile="${service.jar}" basedir="${bin.dir}" />
	</target>

	<target name="createwar" depends="compile.src">
		<copy todir="${webinf.dir}/lib" flatten="true">
			<fileset dir="${lib.dir}" includes="**/*.jar" />
		</copy>
		<copy todir="${webinf.dir}/lib" flatten="true">
			<fileset dir="${dist.dir}" includes="**/*.jar" />
		</copy>
		<copy todir="${webinf.dir}" flatten="true">
			<fileset dir="${build.dir}" includes="**/*.wsdl,*.gz" />
		</copy>
		<copy todir="${webinf.dir}" flatten="true">
			<fileset dir="${server.config.dir}" excludes="serverconfig.xml" />
		</copy>
		<jar destfile="${dist.dir}/${ant.project.name}.war" basedir="${war.dir}" />
	</target>
```

</project>

**How to consume a web service using JAX-RPC stub**
**Capsule**

To consume a web service using JAX-RPC, follow the following steps.

**1. Create an XML file called "client-config.xml" and provide the path of the wsdl file location.**
**2. Run the build script to generate the java stub classes and finally create a jar file and put that jar file in the classpath.**
**3. Write a test program to test the web service.**

Let us see the code below.

**client-config.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
   xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
   <wsdl
      location="./config/SimpleWebService.wsdl"
      packageName="com.ddlab.client.jaxrpc">
   </wsdl>
</configuration>
```

**application-build-config.properties**

```
lib.home=${basedir}/lib
config.rpcenc.file=${basedir}/config/client-config.xml
client.home.src=${basedir}/generated_client_src
client.home.bin=${basedir}/generated_client_bin
dist.dir=${basedir}/dist
client.jar=${dist.dir}/${ant.project.name}.jar
```

**client-build.xml**

```
<project name="SimpleWebService_jaxrpc" default="create.client" basedir=".">

        <property file="application-build-config.properties"/>

        <path id="classpath">
                <fileset dir="${lib.home}">
                        <include name="**/*.jar" />
                </fileset>
        </path>

        <taskdef name="wscompile" classname="com.sun.xml.rpc.tools.ant.Wscompile">
                <classpath refid="classpath" />
        </taskdef>

        <target name="clean">
                <delete dir="${client.home.src}" />
                <delete dir="${client.home.bin}" />
                <delete dir="${dist.dir}" />
        </target>

        <target name="init">
                <delete dir="${client.home.src}" />
                <delete dir="${client.home.bin}" />
```

```
            <delete dir="${dist.dir}" />
            <mkdir dir="${client.home.src}" />
            <mkdir dir="${client.home.bin}" />
            <mkdir dir="${dist.dir}" />
        </target>

        <target name="generate.client" depends="init">
            <wscompile keep="true" client="true" base="${client.home.src}" xPrintStackTrace="true"
                verbose="false" classpath="${classpath}" config="${config.rpcenc.file}">
                <classpath>
                    <path refid="classpath" />
                </classpath>
            </wscompile>
        </target>

        <target name="create.client" depends="generate.client">
            <copy todir="${client.home.bin}">
               <fileset dir="${client.home.src}" includes="**/*.class"/>
            </copy>
            <jar destfile="${client.jar}" basedir="${client.home.bin}" />
        </target>

</project>
```

**Test.java**

```
import com.ddlab.client.jaxrpc.SimpleWebService_Service;
import com.ddlab.client.jaxrpc.SimpleWebService_Service_Impl;

public class Test
{
        public static void main(String[] args) throws Exception
        {
            SimpleWebService_Service service = new SimpleWebService_Service_Impl();
            String response = service.getSimpleWebServicePort().getName("Deba...");
            System.out.println("Response :::"+response);
        }

}
```

For the above the following jar files are required.

apache-ant/ant.jar
fastinfoset/FastInfoset.jar,
jaxb/jaxb-api.jar,jaxb/jaxb-impl.jar,jaxb/jaxb-xjc.jar,jaxb/jaxb1-impl.jar,
jaxp/dom.jar,jaxp/jaxp-api.jar,jaxp/sax.jar,jaxp/xalan.jar,jaxp/xercesImpl.jar,
jaxr/jaxr-api.jar,jaxr/jaxr-impl.jar,
jaxrpc/jaxrpc-api.jar,jaxrpc/jaxrpc-impl.jar,jaxrpc/jaxrpc-spi.jar,
jaxws/jaxws-api.jar,jaxws/jaxws-rt.jar,jaxws/jaxws-tools.jar,jaxws/jsr181-api.jar,jaxws/jsr250-api.jar,
jwsdp-shared/activation.jar,jwsdp-shared/jaas.jar,jwsdp-shared/jta-spec1_0_1.jar,jwsdp-shared/mail.jar,
jwsdp-shared/relaxngDatatype.jar,jwsdp-shared/resolver.jar,jwsdp-shared/xmlsec.jar,jwsdp-shared/xsdlib.jar,
saaj/saaj-api.jar,saaj/saaj-impl.jar,

# Web service Technology - Metro from java.net Glassfish community

**How to create a web service using Metro**

This is based on annotation provided by Metro.

**Capsule**

**1. Create a java class with the method and the appropriate parameters and provide the proper annotations.**
**2. Write an XML file called "sun-jaxws.xml" by providing the web service name, implementation class and the url pattern to access the web service.**
**This file should be available inside <war file>/WEB-INF/**
**3. Write the deployment descriptor "web.xml" by providing the metro WSServlet.The name of the servlet is "com.sun.xml.ws.transport.http.servlet.WSServlet".**
**4. Run the build script to generate and package the final war file for deployment.**

Let us see the code below.

**SimpleWebServiceImpl.java**

```
package com.ddlab.rnd.webservice.metro;

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

@WebService
public class SimpleWebServiceImpl
{
	@WebMethod
	public String getName( String name ) throws Exception
	{
		System.out.println("Name from webservice client--->"+name);
		return "Name from server "+name;
	}
}
```

**sun-jaxws.xml**

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  <endpoint
    name="SimpleWebService"
    implementation="com.ddlab.rnd.webservice.metro.SimpleWebServiceImpl"
    url-pattern="/SimpleWebService"/>
</endpoints>
```

**web.xml**

```
<web-app>
   <description>SimpleWebService</description>
   <display-name>SimpleWebService</display-name>
   <listener>
      <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>
   </listener>
   <servlet>
      <description>JAX-WS endpoint - SimpleObjectSebservice</description>
      <display-name>SimpleWebService</display-name>
      <servlet-name>SimpleWebService</servlet-name>
      <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>SimpleWebService</servlet-name>
      <url-pattern>/SimpleWebService</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>
</web-app>
```

**server-build.xml**

```
<!--
        This is a generic ant build script for building the server side code.
        Upon running the script, it will create a war file and you can
        deploy in any application server.
        Author : Debadatta Mishra
        Project Name : simpleobjectwebservice_metro
-->
<project name="SimpleWebService_metro" default="createwar" basedir=".">

            <property environment="env"/>
            <property name="server.config.dir"                value="${basedir}/server-config" />
            <property name="src.dir"                        value="${basedir}/serversrc" />
            <property name="bin.dir"                        value="${basedir}/bin" />
            <property name="lib.dir"                        value="${basedir}/lib" />
            <property name="dist.dir"                       value="${basedir}/dist" />
            <property name="build.dir"                      value="${basedir}/build" />
            <property name="war.dir"                        value="${basedir}/${ant.project.name}.war" />
            <property name="webinf.dir"                     value="${war.dir}/WEB-INF" />
            <property name="webinf.classes.dir"             value="${war.dir}/WEB-INF/classes" />
            <property name="webinf.lib.dir"                 value="${war.dir}/WEB-INF/lib" />
            <property name="webservice.jar"                 value="${dist.dir}/${ant.project.name}.jar" />
            <property name="java.home"                      value="${env.JAVA_HOME}"/>


        <!-- Setting the ClassPath -->
        <path id="classpath">
                <pathelement location="${java.home}/../lib/tools.jar"/>
                <fileset dir="${lib.dir}">
                        <include name="**/*.jar" />
                </fileset>
        </path>
```

```
<taskdef name="apt" classname="com.sun.tools.ws.ant.Apt">
        <classpath refid="classpath"/>
</taskdef>

<taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
        <classpath refid="classpath"/>
</taskdef>

<target name="clean">
        <delete dir="${dist.dir}" />
        <delete dir="${bin.dir}" />
        <delete dir="${war.dir}" />
        <delete dir="${build.dir}" />
</target>

<!-- Delete and create the required directories -->
<target name="init" depends="clean" >
        <mkdir dir="${bin.dir}" />
        <mkdir dir="${dist.dir}" />
        <mkdir dir="${war.dir}" />
        <mkdir dir="${build.dir}" />
        <mkdir dir="${webinf.dir}/classes" />
</target>

<!-- Compile the source and create the jar file -->
<target name="compile.src" depends="init">
        <javac srcdir="${src.dir}" destdir="${bin.dir}" debug="on" fork="yes">
                <classpath refid="classpath" />
        </javac>
        <jar destfile="${webservice.jar}" basedir="${bin.dir}" />
</target>

<!-- Generate the webservice -->
<target name="generate-service" depends="compile.src">
        <apt
   fork="true"
   debug="true"
   verbose="${verbose}"
   destdir="${build.dir}"
   sourcedestdir="${build.dir}"
   sourcepath="${src.dir}">
                <classpath>
                        <path refid="classpath"/>
                        <pathelement location="${src.dir}"/>
                </classpath>
                <option key="r" value="${build.dir}"/>
                <source dir="${src.dir}">
                        <include name="**/*.java"/>
                </source>
        </apt>
        <copy todir="${build.dir}/classes">
                <fileset dir="${build.dir}" includes="**/*.class"/>
        </copy>
        <jar destfile="${webservice.jar}" basedir="${build.dir}/classes" />
</target>
```

```
    <!-- Creation of war file -->
    <target name="createwar" depends="generate-service">
        <copy todir="${webinf.dir}/lib" flatten="true">
            <fileset dir="${lib.dir}" includes="**/*.jar" />
        </copy>
        <copy todir="${webinf.dir}/lib" flatten="true">
            <fileset dir="${dist.dir}" includes="*.jar" />
        </copy>
        <copy todir="${webinf.dir}" flatten="true">
            <fileset dir="${server.config.dir}" includes="*.xml" />
        </copy>
        <jar destfile="${dist.dir}/${ant.project.name}.war" basedir="${war.dir}" />
    </target>

</project>
```

**NOTE**

If you face issue related to building the application using Metro, follow the following steps.
Before building the application , follow the following steps for JDK 6, otherwise you will not be able to build the application.
1. Create a directory called "endorsed" inside <JAVA_HOME>/jre/lib. So the path looks like the following.
**F:\dev\jdk6\jre\lib\endorsed**

2. Copy the jar files "**jaxb-api.jar**" and "**jaxws-api.jar**" from JAX-WS and paste it in the above "**endorsed**" directory. If required copy "**webservices-api.jar**"

Finally the your **<JAVA_HOME>/jre/lib/endorsed** directory looks like the following.

**F:/dev/jdk6/jre/lib\endorsed/**
**| jaxb-api.jar**
**| jaxws-api.jar**
**| webservices-api.jar**

3. Finally build the application using Apache Ant from command propmt, do not run Ant script from Eclipse.

4. Deploy the application in Jboss5.1.0GA. It works.

The jar files required for the application is given below.
**metro/**
**jsr173_api.jar**
**webservices-api.jar**
**webservices-extra-api.jar**
**webservices-extra.jar**
**webservices-rt.jar**
**webservices-tools.jar**

**How to create a web service from the existing wsdl file**

1. Get the wsdl file from the external system and store it in a directory called "wsdl".
2. Run the build script to generate the skeleton classes using the command of wsimport.
3. Modify the java class ending with Impl by providing the business logic. In this case the java class name is "SimpleWebServiceImpl.java"
4. Write an XML file called "sun-jaxws.xml" by providing the web service name, implementation class and the url pattern to access the web service.
This file should be available inside <war file>/WEB-INF/

5. Write the deployment descriptor "web.xml" by providing the metro WSServlet.The name of the servlet is "com.sun.xml.ws.transport.http.servlet.WSServlet".
6. Run the build script to generate and package the final war file for deployment.

See the code below.

**sun-jaxws.xml**

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
   <endpoint
      name="SimpleWebService"
      implementation="com.ddlab.rnd.webservice.metro.SimpleWebServiceImpl"
      url-pattern="/SimpleWebService"/>
</endpoints>
```

**web.xml**

```
<web-app>
   <description>SimpleWebService</description>
   <display-name>SimpleWebService</display-name>
   <listener>
      <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>
   </listener>
   <servlet>
      <description>JAX-WS endpoint - SimpleObjectSebservice</description>
      <display-name>SimpleWebService</display-name>
      <servlet-name>SimpleWebService</servlet-name>
      <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>SimpleWebService</servlet-name>
      <url-pattern>/SimpleWebService</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>
</web-app>
```

**server-build.xml**

```
<!--
        This is a generic ant build script for building the server side code.
        Upon running the script, it will create a war file and you can
        deploy in any application server.
        Author : Debadatta Mishra
        Project Name : simpleobjectwebservice_metro
-->
<project name="SimpleWebService_fromwsdl_metro" default="createwar" basedir=".">

            <property environment="env"/>
            <property name="wsdl.path"
value="${basedir}/wsdl/SimpleWebService.wsdl"/>
            <property name="server.config.dir"                    value="${basedir}/server-config" />
            <property name="src.dir"                            value="${basedir}/serversrc" />
            <property name="bin.dir"                             value="${basedir}/bin" />
            <property name="lib.dir"                            value="${basedir}/lib" />
            <property name="dist.dir"                            value="${basedir}/dist" />
      <property name="gensrc.dir"                          value="${basedir}/gensrc" />
      <property name="package.name"                          value="com.ddlab.rnd.ws.fromwsdl"/>
```

```
    <property name="build.dir"                value="${basedir}/build" />
<property name="build.dir.classes"            value="${build.dir}/classes" />
    <property name="war.dir"                  value="${basedir}/${ant.project.name}.war" />
    <property name="webinf.dir"               value="${war.dir}/WEB-INF" />
    <property name="webinf.classes.dir"       value="${war.dir}/WEB-INF/classes" />
    <property name="webinf.lib.dir"           value="${war.dir}/WEB-INF/lib" />
    <property name="webservice.jar"           value="${dist.dir}/${ant.project.name}.jar" />
    <property name="java.home"                value="${env.JAVA_HOME}"/>


  <!-- Setting the ClassPath -->
  <path id="classpath">
        <pathelement location="${java.home}/../lib/tools.jar"/>
        <fileset dir="${lib.dir}">
            <include name="**/*.jar" />
        </fileset>
  </path>

  <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
        <classpath refid="classpath"/>
  </taskdef>

  <target name="clean">
        <delete dir="${dist.dir}" />
        <delete dir="${bin.dir}" />
        <delete dir="${war.dir}" />
        <delete dir="${build.dir}" />
        <delete dir="${gensrc.dir}" />
        <delete dir="${build.dir.classes}" />
  </target>

  <!-- Delete and create the required directories -->
  <target name="init" depends="clean" >
        <mkdir dir="${bin.dir}" />
        <mkdir dir="${dist.dir}" />
        <mkdir dir="${war.dir}" />
        <mkdir dir="${build.dir}" />
        <mkdir dir="${webinf.dir}/classes" />
        <mkdir dir="${gensrc.dir}" />
        <mkdir dir="${build.dir.classes}" />
  </target>

  <!-- Compile the source and create the jar file -->

  <target name="compile.src" depends="init">
        <javac srcdir="${src.dir}" destdir="${bin.dir}" debug="on" fork="yes" includeantruntime="true">
            <classpath refid="classpath" />
        </javac>
        <jar destfile="${webservice.jar}" basedir="${bin.dir}" />
  </target>

  <target name="generate-service" depends="compile.src">
      <wsimport
          debug="true"
          verbose="${verbose}"
          keep="true"
          destdir="${gensrc.dir}"
```

```
            package="${package.name}"
            wsdl="${wsdl.path}">
        </wsimport>
        <javac
            fork="true"
            srcdir="${gensrc.dir}"
            destdir="${build.dir.classes}"
            includes="**/*.java">
          <classpath refid="classpath"/>
        </javac>
          <jar destfile="${dist.dir}/genservice.jar" basedir="${build.dir.classes}" />
      </target>

    <!-- Creation of war file -->
    <target name="createwar" depends="generate-service">
        <copy todir="${webinf.dir}/lib" flatten="true">
            <fileset dir="${lib.dir}" includes="**/*.jar" />
        </copy>
        <copy todir="${webinf.dir}/lib" flatten="true">
            <fileset dir="${dist.dir}" includes="*.jar" />
        </copy>
        <copy todir="${webinf.dir}" flatten="true">
            <fileset dir="${server.config.dir}" includes="*.xml" />
        </copy>
        <jar destfile="${dist.dir}/${ant.project.name}.war" basedir="${war.dir}" />
    </target>

</project>
```

**How to consume the web service using Metro web service stub**

**Capsule**

**1. Get the wsdl file and store in a directory.**
**2. Run the build script using the commands of wsimport to generate client stub and finally generate the jar file.**
**3. Put the generated jar file in the classpath and write a java program to test it.**

See the code below.

**client-build.xml**

```
<project name="simpleobjectwebservice_metro" default="create.client" basedir=".">

    <property name="config.dir"          value="${basedir}/config" />
    <property name="bin.dir"             value="${basedir}/bin" />
    <property name="lib.dir"             value="${basedir}/lib" />
```

```
        <property name="dist.dir"                    value="${basedir}/dist" />
        <property name="client.home.src"             value="${basedir}/generated_client_src"/>
        <property name="client.home.bin"             value="${basedir}/generated_client_bin"/>
        <property name="client.jar"                  value="${dist.dir}/${ant.project.name}.jar"/>
    <property name="war.dir"                         value="${basedir}/${ant.project.name}.war" />
    <property name="webinf.dir"                      value="${war.dir}/WEB-INF" />
    <property name="webinf.classes.dir"              value="${war.dir}/WEB-INF/classes" />
    <property name="webinf.lib.dir"                  value="${war.dir}/WEB-INF/lib" />
    <property name="webservice.jar"                  value="${dist.dir}/${ant.project.name}.jar" />
        <property name="wsdl.file.name"              value="${config.dir}/SimpleWebService.wsdl"
/>
    <property name="java.home"                       value="${env.JAVA_HOME}"/>
    <property environment="env"/>

        <path id="classpath">
                <pathelement location="${java.home}/../lib/tools.jar"/>
                <fileset dir="${lib.dir}">
                        <include name="**/*.jar" />
                </fileset>
        </path>

        <taskdef name="apt" classname="com.sun.tools.ws.ant.Apt">
                <classpath refid="classpath"/>
        </taskdef>

        <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
                <classpath refid="classpath"/>
        </taskdef>

        <target name="clean">
                <delete dir="${client.home.src}" />
                <delete dir="${client.home.bin}" />
                <delete dir="${dist.dir}" />
        </target>

        <target name="init" depends="clean">
                <mkdir dir="${client.home.src}" />
                <mkdir dir="${client.home.bin}" />
                <mkdir dir="${dist.dir}" />
        </target>

        <target name="generate.client" depends="init">
                <wsimport
           debug="true"
                        xendorsed="true"
           verbose="${verbose}"
           keep="true"
           destdir="${client.home.src}"
           package="com.ddlab.client.metro"
           wsdl="${wsdl.file.name}">
                </wsimport>
        </target>

        <target name="create.client" depends="generate.client">
                <copy todir="${client.home.bin}">
                        <fileset dir="${client.home.src}" includes="**/*.class"/>
                </copy>
```

```
        <jar destfile="${client.jar}" basedir="${client.home.bin}" />
    </target>

</project>
```

**TestWebServiceClientMetro.java**

```
import com.ddlab.client.metro.SimpleWebServiceImplService;

public class TestWebServiceClientMetro
{
    public static void main(String[] args) throws Exception
    {
        String response = new SimpleWebServiceImplService()
                .getSimpleWebServiceImplPort().getName("Debadatta Mishra");
        System.out.println("Response :::"+response);
    }

}
```

**NOTE**

1. Build the application with all the jar files available inside the lib.

2. While running from Eclipse, do not set the following jar files in the classpath
webservices-api.jar
webservices-extra-api.jar
webservices-extra.jar
webservices-rt.jar
webservices-tools.jar

Otherwise it throws the following exception.
Exception in thread "main" java.lang.NoSuchMethodError:
javax.xml.ws.WebFault.messageName()Ljava/lang/String;

# Webservice Technology JAX-WS

## How to create a web service using JAX-WS

**Capsule**

**1. Create a java class with the appropriate method and the parameters and provide the relevent annotation for creating web service.**
**2. Write an xml file called "sun-jaxws.xml" and provide the information for web service name, implementation class and the url pattern.**
**3. Configure the deployment descriptot "web.xml" by using the servlet "com.sun.xml.ws.transport.http.servlet.WSServlet" for JAX-WS.**
**4. Run the build script to package the web application for deployment.**

**NOTE**

Before building the application , follow the following steps for JDK 6, otherwise you will not be able to build the application.

1. Create a directory called "endorsed" inside **<JAVA_HOME>/jre/lib**. So the path looks like the following.
**F:\dev\jdk6\jre\lib\endorsed**
2. Copy the jar files "**jaxb-api.jar**" and "**jaxws-api.jar**" from JAX-WS and paste it in the above "**endorsed**" directory.

Finally the your **<JAVA_HOME>/jre/lib/endorsed** directory looks like the following.

**F:/dev/jdk6/jre/lib\endorsed/**
**| jaxb-api.jar**
**| jaxws-api.jar**

3. Finally build the application using Apache Ant from command propmt, do not run Ant script from Eclipse.

4. Deploy the application in Jboss5.1.0GA. It works.

See the code below.

**SimpleWebServiceImpl.java**

```
package com.ddlab.rnd.webservice.jaxws;
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class SimpleWebServiceImpl
{
	@WebMethod
	public String getName( String name ) throws Exception
	{
		System.out.println("Name from webservice client--->"+name);
		return "Name from server "+name;
	}
}
```

**sun-jaxws.xml**

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
  <endpoint
```

```
    name="SimpleWebService"
    implementation="com.ddlab.rnd.webservice.jaxws.SimpleWebServiceImpl"
    url-pattern="/SimpleWebService"/>
</endpoints>
```

**web.xml**

```
<web-app>
  <description>SimpleWebService</description>
  <display-name>SimpleWebService</display-name>
  <listener>
    <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>
  </listener>
  <servlet>
    <description>JAX-WS endpoint - ZodiacCalculatorService</description>
    <display-name>SimpleWebService</display-name>
    <servlet-name>SimpleWebService</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>SimpleWebService</servlet-name>
    <url-pattern>/SimpleWebService</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

**server-build.xml**

```
<!--
        This is a generic ant build script for building the server side code.
        Upon running the script, it will create a war file and you can
        deploy in any application server.
        Author : Debadatta Mishra
-->
<project name="SimpleWebService_jaxws" default="createwar" basedir=".">

        <property name="server.config.dir"          value="${basedir}/server-config" />
        <property name="src.dir"                    value="${basedir}/serversrc" />
        <property name="bin.dir"                    value="${basedir}/bin" />
        <property name="lib.dir"                    value="${basedir}/lib" />
        <property name="dist.dir"                   value="${basedir}/dist" />
        <property name="build.dir"                  value="${basedir}/build" />
        <property name="war.dir"                    value="${basedir}/${ant.project.name}.war" />
        <property name="webinf.dir"                 value="${war.dir}/WEB-INF" />
        <property name="webinf.classes.dir"         value="${war.dir}/WEB-INF/classes" />
        <property name="webinf.lib.dir"             value="${war.dir}/WEB-INF/lib" />
        <property name="webservice.jar"             value="${dist.dir}/${ant.project.name}.jar" />
        <property name="java.home"                  value="${env.JAVA_HOME}"/>
        <property environment="env"/>

        <!-- Setting the ClassPath -->
        <path id="classpath">
                <pathelement location="${java.home}/../lib/tools.jar"/>
                <fileset dir="${lib.dir}">
                        <include name="**/*.jar"/>
```

```
        </fileset>
</path>

<taskdef name="apt" classname="com.sun.tools.ws.ant.Apt">
        <classpath refid="classpath"/>
</taskdef>

<taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
        <classpath refid="classpath"/>
</taskdef>

<target name="clean">
        <delete dir="${dist.dir}" />
        <delete dir="${bin.dir}" />
        <delete dir="${war.dir}" />
        <delete dir="${build.dir}" />
</target>

<!-- Delete and create the required directories -->
<target name="init" depends="clean" >
        <mkdir dir="${bin.dir}" />
        <mkdir dir="${dist.dir}" />
        <mkdir dir="${war.dir}" />
        <mkdir dir="${build.dir}" />
        <mkdir dir="${webinf.dir}/classes" />
</target>

<!-- Compile the source and create the jar file -->
<target name="compile.src" depends="init">
        <javac srcdir="${src.dir}" destdir="${bin.dir}" debug="on" fork="yes">
                <classpath refid="classpath" />
        </javac>
        <jar destfile="${webservice.jar}" basedir="${bin.dir}" />
</target>

<!-- Generate the webservice -->
<target name="generate-service" depends="compile.src">
        <apt
   fork="true"
   debug="true"
   verbose="${verbose}"
   destdir="${build.dir}"
   sourcedestdir="${build.dir}"
   sourcepath="${src.dir}">
                <classpath>
                        <path refid="classpath"/>
                        <pathelement location="${src.dir}"/>
                </classpath>
                <option key="r" value="${build.dir}"/>

                <source dir="${src.dir}">
                        <include name="**/*.java"/>
                </source>
        </apt>
        <copy todir="${build.dir}/classes">
                <fileset dir="${build.dir}" includes="**/*.class"/>
        </copy>
```

```
        <jar destfile="${webservice.jar}" basedir="${build.dir}/classes" />
    </target>

    <!-- Creation of war file -->
    <target name="createwar" depends="generate-service">
        <copy todir="${webinf.dir}/lib" flatten="true">
            <fileset dir="${lib.dir}" includes="**/*.jar" />
        </copy>
        <copy todir="${webinf.dir}/lib" flatten="true">
            <fileset dir="${dist.dir}" includes="*.jar" />
        </copy>
        <copy todir="${webinf.dir}" flatten="true">
            <fileset dir="${server.config.dir}" includes="*.xml" />
        </copy>
        <jar destfile="${dist.dir}/${ant.project.name}.war" basedir="${war.dir}" />
    </target>

</project>
```

The following jar files are required for development.

jaxws-2.2.5/activation.jar
jaxws-2.2.5/FastInfoset.jar
jaxws-2.2.5/gmbal-api-only.jar
jaxws-2.2.5/ha-api.jar
jaxws-2.2.5/http.jar
jaxws-2.2.5/jaxb-api.jar
jaxws-2.2.5/jaxb-impl.jar
jaxws-2.2.5/jaxb-xjc.jar
jaxws-2.2.5/jaxws-api.jar
jaxws-2.2.5/jaxws-rt.jar
jaxws-2.2.5/jaxws-tools.jar
jaxws-2.2.5/jsr173_api.jar
jaxws-2.2.5/jsr181-api.jar
jaxws-2.2.5/jsr250-api.jar
jaxws-2.2.5/management-api.jar
jaxws-2.2.5/mimepull.jar
jaxws-2.2.5/policy.jar
jaxws-2.2.5/resolver.jar
jaxws-2.2.5/saaj-api.jar
jaxws-2.2.5/saaj-impl.jar
jaxws-2.2.5/stax-ex.jar
jaxws-2.2.5/streambuffer.jar
jaxws-2.2.5/woodstox.jar

**How to consume web service using JAX-WS client stub**

**Capsule**

**1. Store the wsdl file in a directory.**
**2. Run the build script using wsimport to generate client classes and finally create the jar file.**
**3. Put the above generated jar file in the class path and write a JAX-WS specific class to test.**

See the code below.

**client-build.xml**

```
<project name="simpleobjectwebservice_jaxws" default="create.client" basedir=".">
```

```
    <property name="config.dir"                 value="${basedir}/config" />
    <property name="bin.dir"                    value="${basedir}/bin" />
    <property name="lib.dir"                    value="${basedir}/lib" />
    <property name="dist.dir"                   value="${basedir}/dist" />
    <property name="client.home.src"            value="${basedir}/generated_client_src"/>
    <property name="client.home.bin"            value="${basedir}/generated_client_bin"/>
    <property name="client.jar"                 value="${dist.dir}/${ant.project.name}.jar"/>
  <property name="war.dir"                      value="${basedir}/${ant.project.name}.war" />
  <property name="webinf.dir"                   value="${war.dir}/WEB-INF" />
  <property name="webinf.classes.dir"           value="${war.dir}/WEB-INF/classes" />
  <property name="webinf.lib.dir"               value="${war.dir}/WEB-INF/lib" />
  <property name="webservice.jar"               value="${dist.dir}/${ant.project.name}.jar" />
    <property name="wsdl.file.name"             value="${config.dir}/SimpleWebService.wsdl"
/>
  <property name="java.home"                    value="${env.JAVA_HOME}"/>
  <property environment="env"/>

    <path id="classpath">
        <pathelement location="${java.home}/../lib/tools.jar"/>
        <fileset dir="${lib.dir}">
            <include name="**/*.jar" />
        </fileset>
    </path>

    <taskdef name="apt" classname="com.sun.tools.ws.ant.Apt">
        <classpath refid="classpath"/>
    </taskdef>

    <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
        <classpath refid="classpath"/>
    </taskdef>

    <target name="clean">
        <delete dir="${client.home.src}" />
        <delete dir="${client.home.bin}" />
        <delete dir="${dist.dir}" />
    </target>

    <target name="init" depends="clean">
        <mkdir dir="${client.home.src}" />
        <mkdir dir="${client.home.bin}" />
        <mkdir dir="${dist.dir}" />
    </target>

    <target name="generate.client" depends="init">
        <wsimport
      debug="true"
                xendorsed="true"
      verbose="${verbose}"
      keep="true"
      destdir="${client.home.src}"
      package="com.ddlab.client.jaxws"
      wsdl="${wsdl.file.name}">
        </wsimport>
    </target>
```

```
	<target name="create.client" depends="generate.client">
		<copy todir="${client.home.bin}">
			<fileset dir="${client.home.src}" includes="**/*.class"/>
		</copy>
		<jar destfile="${client.jar}" basedir="${client.home.bin}" />
	</target>

</project>
```

**TestWebServiceClient.java**

```
import com.ddlab.client.jaxws.SimpleWebServiceImpl;
import com.ddlab.client.jaxws.SimpleWebServiceImplService;

public class TestWebServiceClient
{
	public static void main(String[] args) throws Exception
	{
		String wsdlLocation = "http://localhost:8080/SimpleWebService_jaxws/SimpleWebService?wsdl";
		SimpleWebServiceImpl ss =  new
SimpleWebServiceImplService().getSimpleWebServiceImplPort();
		String response = ss.getName("I am the Test");
		System.out.println("Response :::"+response);
	}
}
```

**NOTE**

1. Build the application with all the jar files available inside the lib.

2. While running from Eclipse, do not set the following jar files in the classpath
jaxb-api.jar
jaxb-impl.jar
jaxb-xjc.jar
jaxws-api.jar
jaxws-rt.jar
jaxws-tools.jar

Otherwise it throws the following exception.
Exception in thread "main" java.lang.NoSuchMethodError:
javax.xml.ws.WebFault.messageName()Ljava/lang/String;

# Web service technology – Spring WS

**Web service creation using Spring WS framework**

You can also create a web service using Spring WS framework.

**Capsule**

**1. Create an interface with various methods and parameters. These methods will be the web service methods or operations.**
**2. Create a class that will implement the above interface. This is the implementation class.**
**3. Create an XML Schema using any tool in relation with the interface methods.**
**4. Write a class using Spring EndPoint extending AbstractDomPayloadEndpoint provide the implementation using xml parsing inside the method invokeInternal()**
**5. Configure web.xml file with the appropriate MessageDispatcherServlet of Spring for web service expose.**
**6. Configure the default schema in the configuration file.**
**7. Create -servlet.xml file called the IOC meta data configuration file**

**with the configuration for the following.**
**i). Configuration for the implementation class.**
**ii). Configuration for the End point and pass the implementation class as reference.**
**iii). Configure the default payload mapping.**
**iv). Configure the final web service bean with the parameters for the schema, location uri and port name**
**8. Build and deploy the application in J2EE server.**

See the code below.

**WebCalculatorService.java**

```
package com.ddlab.rnd.webservice.spring;

public interface WebCalculatorService
{
	public int add( int firstNo , int secondNo );
	public int sub( int firstNo , int secondNo );
}
```

**WebCalculatorServiceImpl.java**

```
package com.ddlab.rnd.webservice.spring;

public class WebCalculatorServiceImpl implements WebCalculatorService
{
	@Override
	public int add(int firstNo, int secondNo)
	{
		return firstNo+secondNo;
	}

	@Override
	public int sub(int firstNo, int secondNo)
	{
		return firstNo-secondNo;
	}

}
```

**WebCalculatorServiceEndPoint.java**

```
package com.ddlab.rnd.webservice.spring;
import java.io.StringWriter;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.springframework.ws.server.endpoint.AbstractDomPayloadEndpoint;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;

@SuppressWarnings("deprecation")
public class WebCalculatorServiceEndPoint extends AbstractDomPayloadEndpoint
```

```
{
	public static final String NAMESPACE = "http://www.ddlabinc.com";
	private WebCalculatorService webCalculator = null;

	public WebCalculatorService getWebCalculator() {
		return webCalculator;
	}

	public void setWebCalculator(WebCalculatorService webCalculator) {
		this.webCalculator = webCalculator;
	}

	public void showIncomingMessage( Document doc ) throws Exception
	{
		Transformer transformer = TransformerFactory.newInstance().newTransformer();
		transformer.setOutputProperty(OutputKeys.INDENT, "yes");

		//initialize StreamResult with File object to save to file
		StreamResult result = new StreamResult(new StringWriter());
		DOMSource source = new DOMSource(doc);
		transformer.transform(source, result);

		String xmlString = result.getWriter().toString();
		System.out.println("Incoming XML Message \n"+xmlString);

	}

	@Override
	protected Element invokeInternal(Element element, Document doc)throws Exception
	{
		Element responseElement = null;
		String requestText = element.getTextContent();
		System.out.println("Total Request Element :::"+requestText);
		try
		{
			showIncomingMessage(doc);
			System.out.println("Web service method called for web calculation ...");
			System.out.println("Element element------>"+element);
			System.out.println("Document doc------>"+doc);

			if( element !=  null )
			{
				String methodTagName = element.getTagName();
				System.out.println("Element Tag Name :::"+methodTagName);
				if( methodTagName.equals("addRequest"))
				{
					responseElement = handleAddRequest(element,doc);
				}
				else if( methodTagName.equals("subRequest"))
				{
					responseElement = handleSubRequest(element,doc);
				}
			}
		}
		catch (Exception e)
		{
			e.printStackTrace();
```

```
                throw e;
            }
            return responseElement;
        }

        private Element handleSubRequest(Element element,Document doc)
        {
            System.out.println("------------- handleSubRequest() ------------------------");
            Element responseElement = null;
            try
            {
                Node node = element.getChildNodes().item(0);
                NodeList childNodesList = node.getChildNodes();
                String firstNo = null;
                String secondNo = null;
                String result = "Some Testing Data Now ...";
                for( int i = 0 ; i < childNodesList.getLength() ; i++ )
                {
                    Node childNode = childNodesList.item(i);
                    if( childNode.getNodeType() == Node.ELEMENT_NODE )
                    {
                        Element childEl = (Element)childNode;
                        System.out.println("Element Tag Name :::"+childEl.getTagName());
                        if( childEl.getTagName().equals("FirstNo") ) firstNo =
childEl.getTextContent();
                        else if( childEl.getTagName().equals("SecondNo") ) secondNo =
childEl.getTextContent();
                        System.out.println(childEl.getTagName()+"--------
"+childEl.getTextContent());
                    }
                }
                System.out.println("FirstNo :::"+firstNo);
                System.out.println("SecondNo :::"+secondNo);
                result = String.valueOf(webCalculator.sub(Integer.parseInt(firstNo),
                        Integer.parseInt(secondNo)));
                responseElement = doc.createElementNS(
                        NAMESPACE, "subResponse");
                Text responseText = doc.createTextNode(result);
                responseElement.appendChild(responseText);
                System.out.println("Final Result :::"+result);
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }


            return responseElement;
        }

        private Element handleAddRequest(Element element , Document doc)
        {
            System.out.println("------------- handleAddRequest() ------------------------");
            Element responseElement = null;
            try
            {
                Node node = element.getChildNodes().item(0);
```

```
                        NodeList childNodesList = node.getChildNodes();

                        String firstNo = null;
                        String secondNo = null;
                        String result = "Some Testing Data Now ...";
                        for( int i = 0 ; i < childNodesList.getLength() ; i++ )
                        {
                                Node childNode = childNodesList.item(i);
                                if( childNode.getNodeType() == Node.ELEMENT_NODE )
                                {
                                        Element childEl = (Element)childNode;
                                        System.out.println("Element Tag Name :::"+childEl.getTagName());
                                        if( childEl.getTagName().equals("FirstNo") ) firstNo =
childEl.getTextContent();
                                        else if( childEl.getTagName().equals("SecondNo") ) secondNo =
childEl.getTextContent();
                                        System.out.println(childEl.getTagName()+"--------
"+childEl.getTextContent());
                                }
                        }
                        System.out.println("FirstNo :::"+firstNo);
                        System.out.println("SecondNo :::"+secondNo);
                        result = String.valueOf(webCalculator.add(Integer.parseInt(firstNo),
                                        Integer.parseInt(secondNo)));
                        System.out.println("Final Result :::"+result);
                        responseElement = doc.createElementNS(
                                        NAMESPACE, "addResponse");
                        Text responseText = doc.createTextNode(result);
                        responseElement.appendChild(responseText);
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
                return responseElement;
        }

}
```

**WebCalculator-schema.xsd**

```
<?xml version = "1.0" encoding = "utf-8"?>
<xs:schema xmlns="http://www.ddlabinc.com" targetNamespace="http://www.ddlabinc.com"
        xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
        attributeFormDefault="unqualified">
        <xs:element name="addRequest">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="Calculation" />
                        </xs:sequence>
                </xs:complexType>
```

```
        </xs:element>
        <xs:element name="subRequest">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="Calculation" />
        </xs:sequence>
      </xs:complexType>
   </xs:element>
        <xs:element name="Calculation">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="FirstNo" />
                                <xs:element ref="SecondNo" />
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="FirstNo" type="xs:string" />
        <xs:element name="SecondNo" type="xs:string" />
   <xs:element name="addResponse" type="xs:string" /> <!-- Response for the add method -->
   <xs:element name="subResponse" type="xs:string" /> <!-- Response for the sub method -->
</xs:schema>
```

**web.xml**

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <display-name>String Reversal Service Demo using Spring WebService</display-name>

   <servlet>
      <servlet-name>webcalculatorservice</servlet-name>
 <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
      <init-param>
         <param-name>transformWsdlLocations</param-name>
         <param-value>true</param-value>
      </init-param>
   </servlet>

   <servlet-mapping>
      <servlet-name>webcalculatorservice</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

**webcalculatorservice-servlet.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
        xmlns:sws="http://www.springframework.org/schema/web-services"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/web-services http://www.springframework.org/schema/web-
```

```
services/web-services-2.0.xsd
    http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-
context-3.0.xsd">

        <bean id="webcalculatorserviceimpl"
                class="com.ddlab.rnd.webservice.spring.WebCalculatorServiceImpl">
        </bean>

        <bean id="webcalculatorserviceEndpoint"
                class="com.ddlab.rnd.webservice.spring.WebCalculatorServiceEndPoint">
                <property name="webCalculator" ref="webcalculatorserviceimpl" />
        </bean>

        <bean id="payloadMapping"
                class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
                <property name="defaultEndpoint" ref="webcalculatorserviceEndpoint" />
        </bean>

        <bean id="webcalculatorSchema" class="org.springframework.xml.xsd.SimpleXsdSchema">
                <property name="xsd" value="/WEB-INF/WebCalculator-schema.xsd" />
        </bean>

   <!-- The following bean id is the webservice name -->
        <bean id="webcalculator"
                class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">
                <property name="schema" ref="webcalculatorSchema" />
                <property name="portTypeName" value="myport" /><!-- calculate, It is a name of your choice -->
                <property name="locationUri"
                        value="http://localhost:8080/webcalculatorservice/services" />
        </bean>

</beans>
```

**server-build.xml**

```
<!--
   Author : Debadatta Mishra
   This is a generic script to build and deploy Spring web service
-->
<project name="webcalculatorservice" basedir="." default="create.war">
        <property name="server.src.dir"                 value="${basedir}/serversrc" />
        <property name="lib.dir"                        value="${basedir}/lib" />
        <property name="build.dir"                      value="${basedir}/build"/>
        <property name="dist.dir"                       value="${basedir}/dist"/>
        <property name="web.dir"                        value="${basedir}/web"/>
        <property name="webinf.dir"                     value="${web.dir}/WEB-INF"/>
        <property name="jar.file.name"          value="${ant.project.name}.jar"/>
        <property name="war.file.name"                  value="${ant.project.name}.war"/>
        <property name="xsd.dir.name"           value="${basedir}/xsd"/>

        <!-- Create classpath setting -->
        <path id="build.classpath">
                <fileset dir="${lib.dir}">
                        <include name="**/*.jar"/>
                </fileset>
        </path>

        <!-- Delete directories -->
```

```
        <target name="clean">
                <delete dir="${build.dir}"/>
                <delete dir="${dist.dir}"/>
                <delete dir="${webinf.dir}/lib"/>
        </target>

    <!-- Delete and Create the required directories -->
    <target name="init">
            <delete dir="${build.dir}"/>
            <delete dir="${dist.dir}"/>
            <mkdir dir="${build.dir}"/>
            <mkdir dir="${dist.dir}"/>
            <mkdir dir="${build.dir}/classes"/>
    </target>

    <!-- Compile the services -->
    <target name="compile.service" depends="init">
            <javac debug="on"
      fork="true"
      destdir="${build.dir}/classes"
      srcdir="${server.src.dir}"
      classpathref="build.classpath">
            </javac>
            <jar destfile="${build.dir}/${jar.file.name}" >
    <fileset dir="${build.dir}/classes"/>
</jar>
    </target>

    <!-- Generate the Services -->
    <target name="generate.service" depends="compile.service">
            <copy toDir="${build.dir}/classes" failonerror="false">
                    <fileset dir="${basedir}/resources">
                            <include name="**/*.xml"/>
                    </fileset>
            </copy>
            <jar destfile="${build.dir}/${aar.file.name}" >
                    <fileset dir="${build.dir}/classes"/>
            </jar>
    </target>

    <!-- create the .war file to deploy into server -->
    <target name="create.war" depends="compile.service">
            <delete dir="${webinf.dir}/lib"/>
            <mkdir dir="${webinf.dir}/lib"/>
            <copy toDir="${webinf.dir}/lib" flatten="true">
                    <fileset dir="${lib.dir}" includes="**/*.jar" />
                    <fileset dir="${build.dir}" includes="**/*.jar" />
            </copy>
            <copy toDir="${webinf.dir}">
          <fileset dir="${xsd.dir.name}">
       <include name="**/*.xsd"/>
    </fileset>
            </copy>
            <jar destfile="${dist.dir}/${war.file.name}">
                    <fileset dir="${web.dir}"/>
            </jar>
    </target>
```

</project>

For the above application the following jar files are required.

commons-logging/commons-logging-1.1.1.jar
log4j/log4j-1.2.16.jar
servlet/servlet-api.jar
spring3.0/org.springframework.aop-3.0.5.RELEASE.jar
spring3.0/org.springframework.asm-3.0.5.RELEASE.jar
spring3.0/org.springframework.aspects-3.0.5.RELEASE.jar
spring3.0/org.springframework.beans-3.0.5.RELEASE.jar
spring3.0/org.springframework.context-3.0.5.RELEASE.jar
spring3.0/org.springframework.context.support-3.0.5.RELEASE.jar
spring3.0/org.springframework.core-3.0.5.RELEASE.jar
spring3.0/org.springframework.expression-3.0.5.RELEASE.jar
spring3.0/org.springframework.instrument-3.0.5.RELEASE.jar
spring3.0/org.springframework.instrument.tomcat-3.0.5.RELEASE.jar
spring3.0/org.springframework.jdbc-3.0.5.RELEASE.jar
spring3.0/org.springframework.jms-3.0.5.RELEASE.jar
spring3.0/org.springframework.orm-3.0.5.RELEASE.jar
spring3.0/org.springframework.oxm-3.0.5.RELEASE.jar
spring3.0/org.springframework.test-3.0.5.RELEASE.jar
spring3.0/org.springframework.transaction-3.0.5.RELEASE.jar
spring3.0/org.springframework.web-3.0.5.RELEASE.jar
pring3.0/org.springframework.web.portlet-3.0.5.RELEASE.jar
spring3.0/org.springframework.web.servlet-3.0.5.RELEASE.jar
spring3.0/org.springframework.web.struts-3.0.5.RELEASE.jar
springws2.0/spring-ws-2.0.2.RELEASE-all.jar
wsdl4j/wsdl4j-1.6.1.jar

**How to consume a web service using Spring WS framework**

To consume a web service using Spring, you have to manually or dynamically create an XML file and you have to connect to that service. See the code below.

**TesSpringWebService.java**

```
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.StringReader;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import org.springframework.ws.client.core.WebServiceTemplate;

public class TesSpringWebService  {
	public static String getXmlContents( String filePath ) throws Exception
	{
		String contents = null;
		File file = new File( filePath);
		byte[] buffer = new byte[(int)file.length()];
		InputStream inStream = new FileInputStream(file);
		inStream.read(buffer);
		contents = new String( buffer );
		inStream.close();
		return contents;
	}
	public static void main(String[] args) throws Exception
	{
		String wsdlUrl = "http://localhost:8080/webcalculatorservice/services/webcalculator.wsdl";
```

```
                String filePath = "sampledata/webservicerequest.xml";
                String xmlRequest = getXmlContents(filePath);
//              System.out.println(xmlRequest);
      StreamSource requestMessage = new StreamSource(new StringReader(xmlRequest));
      StreamResult responseMessage = new StreamResult(System.out);
      WebServiceTemplate template = new WebServiceTemplate();
      System.out.println("---->"+requestMessage.getReader());
      template.sendSourceAndReceiveToResult(wsdlUrl, requestMessage, responseMessage);
         }
}
```

The sample request XML file is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<addRequest
xmlns="http://www.ddlabinc.com"><Calculation><FirstNo>12</FirstNo><SecondNo>13</SecondNo></Calculat
ion></addRequest>
```

## Types of web service development

When creating Web services, there are two development styles: **Contract Last and Contract First**.
When using a contract-last approach, you start with the Java code, and let the Web service contract be generated from that. When using contract-first, you start with the WSDL contract, and use Java to implement said contract. “Code First or Contract First?”. This has been a hot topic in the Web services world for many years among the Web service developers. Some developers argue that the Contract First approach is the best and some others argue that the Code First approach is the best when it comes to developing Web services. Try to understand contract-first and contract-last from the view point of WSDL. In case of contract-first, first thing is the generation of WSDL file. In case of Code-first or contract-last approach , WSDL generation will happen later.

**Introduction**

When it comes to developing Web services, there are two main approaches: “**Code First**” and “**Contract First**”. This has been a hot topic in the Web services world for many years among the Web service developers. Some developers argue that the Contract First approach is the best and some others argue that the Code First approach is the best when it comes to developing Web services.

**Web Services Description Language (WSDL) is used to define a contract between a Web service and a client who is consuming the service**. **This contract acts as an agreement on all aspects of a Web service**. **So in the Contract First approach, the service provider defines the WSDL document first and develops the service according to it and publishes the WSDL after deploying the service inside a Web service engine**. **But in the Code First approach, Web service developer writes his business logic first and deploys it as a Web service inside a Web service engine. The engine generates the WSDL for the developer who finally publishes the generated WSDL.**

The core responsibility of a Web service engine is to support deployment of Web services. All well known Web service engines support both Contract First and Code First approaches. In order to support the above 2 approches, there are **WSDL to code and code to WSDL** tools which comes with such Web service engines. If the developer is trying to use the Code First approach, he just has to write his code and deploy in the engine which internally generates the WSDL. Even if the developer is using the Contract First approach, he can use a WSDL to code tool to generate the skeleton of the service. Then he can fill up the business logic and deploy the service with the WSDL.

WSO2 Web Services Application Server (WSO2 WSAS) is one such well known Web service engine, using which you can develop your services easily. WSO2 Web Services Application Server (WSO2 WSAS) supports deployment of many different service types. Through those, WSO2 Web Services Application Server (WSO2 WSAS) supports both Code First and Contract First approaches in many ways. It is extremely important to have a knowledge on these approaches if you are developing long running business applications. Here, I'm going to discuss the topic “Contract First or Code First”, in the context of WSO2 Web Services Application Server (WSO2 WSAS) 3.x.

Code First Approach

**In the Code First approach, development of the service is started from a code. So the developer can write a**

**service without knowing anything about WSDL.** When it comes to WSO2 Web Services Application Server (WSO2 WSAS), what he has to do is to write his business logic as a set of java classes and deploy it in WSO2 Web Services Application Server (WSO2 WSAS). WSO2 Web Services Application Server (WSO2 WSAS) will generate the WSDL file and make his code a Web service.
There are four simple mechanisms in WSO2 Web Services Application Server (WSO2 WSAS) to support Code First approach. In all these mechanisms, WSDL is generated by WSO2 Web Services Application Server (WSO2 WSAS).
POJO services – Converts a single Java class (without packages) into a Web service
Jar Services – Converts a set of Java classes (packaged as a Jar) into one or more Web services
AAR Code First Services – Converts a set of Java classes with a services.xml file into one or more Web services. In this case, you don't have to include a WSDL file in the AAR archive.
Spring Services – Converts Spring beans into Web services
EJB Services – Converts Enterprise Java Beans (EJBs) into Web services
JAX-WS Code First services – Converts a set of annotated (JAX-WS 2.0 [3]) Java classes into a Web Service.
**The Code First approach is simple, less time consuming and easy to use for people who are not familiar with Web services standards because WSDL is somewhat hard to understand and use for a beginner. And also, Code First is a good approach to convert a legacy code into a Web service.** Jar Service and AAR Services in WSO2 Web Services Application Server (WSO2 WSAS) can be used for such purposes. But this approach provides less control over the service contract as the WSDL is generated by WSO2 Web Services Application Server (WSO2 WSAS). If someone changes the service code, WSDL will be changed accordingly. And also the generated WSDL can have minor differences in different versions of WSO2 Web Services Application Server (WSO2 WSAS). Normally a WSDL definition is a contract between the service provider and the client. Therefore, such time to time changes in the WSDL should not be allowed. In addition to that, performance of Code First services are less in WSO2 Web Services Application Server (WSO2 WSAS) compared to Contract First services. This also becomes a disadvantage if the application is performance critical.
Contract First Approach

**In the Contract First approach, development of the service is started from a WSDL definition, which becomes a contract between the service provider and the client. In this approach,** WSDL definition is defined using Web services standards and the service implementation is done after that according to this contract. However, as mentioned in the introduction, manually implementing the service according to a WSDL definition is not very easy and it's time consuming. Therefore, WSO2 Web Services Application Server (WSO2 WSAS) provides the WSDL2Java tool which can generate the service code. So what you have to do is to generate server side code and fill the business logic and deploy the service. If you are using Contract First JAX-WS services, you can generate code using wsimport tool which comes with JDK. There are two simple mechanisms in WSO2 Web Services Application Server (WSO2 WSAS) to support Contract First approach. AAR Contract First Services – Here you can have your service classes, WSDL file, services.xml file and custom Message Receivers in the AAR archive. JAX-WS Contract First Services – Here you can have your annotated service classes and the WSDL file in the Jar archive. **Contract First approach is more complex compared to Code First approach**. And also it requires an in depth understanding of Web services standards. But it is the recommended approach in the Web services world. In this approach, the contract between the service provider and the client is kept constant. Therefore even if the business logic of the service implementation changes, it doesn't affect the client. Most importantly, Contract First approach can provide better performance in the context of WSO2 Web Services Application Server (WSO2 WSAS). When you generate code for your WSDL document, you can specify the appropriate data binding framework which suites your application. And also WSO2 Web Services Application Server (WSO2 WSAS) code generator can generate a custom Message Receiver for you and it helps a lot to improve performance of your service. In simple terms, the service is optimized according to the WSDL.
Code First or Contract First?

Now we know about both approaches and how to use both approaches in WSO2 Web Services Application Server (WSO2 WSAS). So which approach is the best? The answer depends on the service developer's requirement. When it comes to testing purposes or converting legacy code into Web services, Code First approach is far more convenient. But for a long running, performance critical business application, Contract First approach can be highly recommended.

Even if you are not familiar with Web services standards, still there is a way to achieve the advantages of Contract First development using WSO2 Web Services Application Server (WSO2 WSAS). In that case, you can initially

write the service in the Code First manner and get the generated WSDL contract from WSO2 Web Services Application Server (WSO2 WSAS). You can use the Java2WSDL tool which comes with WSO2 Web Services Application Server (WSO2 WSAS) to generate the WSDL. Then if you want, you can modify the generated WSDL according to your requirements. Now you have the contract. Therefore, this contract can be used to generate code and create a service using the Contract First approach.

Finally, I can summarize the advantages and disadvantages of Code First and Contract First approaches as follows. Please note that, as I mentioned above, this is in the context of WSO2 Web Services Application Server (WSO2 WSAS). But some points applies generally as well.

**Code First or Contract-Last**

**Advantages**
Easy to use
Simple and less time consuming
No need of in-depth knowledge on WSDL
Useful at development stage
Can be used to make Web services out of legacy systems

**Disadvantages**
Less performance
Can't guarantee the integrity of the WSDL

**Contract First**

**Advantages**
Can achieve better performance
Integrity of the WSDL is assured
Useful for long running business critical service development

**Disadvantages**
Complex than Code First
Need a better understanding on WSDL

When it comes to developing Web services, it is always better to consider the requirements and select the approach which suites the best. This will always allow you to find the best solution for the existing problem. Specially, if the services you are developing is business critical and log running ones, it is almost a must to take the best approach to gain the maximum outcome.

**Some Other Thoughts**
When implementing Web services, there are two alternative development paths you can take: the code-first approach and the contract-first approach. This article drills down into the contract-first approach in detail and shows how the Axis2 code generator can be used to generate code for the server side with the contract-first approach.
**Why Not Code First?**
Usually when developing Web services, developers like to code the business logic first and then expose that logic as a Web service. This is convenient because developers' core competency is the particular programming language they use. It also happens to be much more convenient for exposing existing programs as Web services. However, the code-first approach has several drawbacks. The developer has less control over the process of exposing code as services. A change to the code may mean a regeneration of the publicly visible Web services interfaces, and often it is difficult to agree on such a generated interface from a business perspective. The client programs often are generated using the
service's WSDL file; if the service WSDL file is likely to change, then the point of having generated clients becomes less obvious. The code for the service process is likely to change between service frameworks and even framework versions, and it becomes difficult to maintain a single interface across versions. It is true that by using

annotations the impact of some of these issues can be lessened. Annotations help the developer take control of the process of exposing code. However, there is no such
thing as a generic annotation scheme to make it universally applicable over multiple languages and multiple frameworks!

**"Contract First" - What is So Special About It?**

As opposed to code first, the contract-first approach takes the contract as the primary artifact. The "contract" in a Web service interaction obviously is the WSDL document; therefore, in the contract-first approach, the focus is on creating the WSDL file and the associated XML schema. The WSDL file and the schema clearly define the message formats, the operation and interface names, and other relevant information for a complete Web service interaction, and can be agreed on by multiple parties. Almost all major Web service frameworks allow service generation from WSDLs, and it becomes easier for the service implementer too; because the major portion of the code is generated, only the necessary business logic would need to be filled in. It should be noted that the contract-first approach has its own
problems, the most notable one being the need for WSDL and schema expertise. One can argue whether the Web service implementers would need to do anything with WSDL since the primary requirement of the WSDL is in providing a description of the service rather than providing the service itself. This would have been a major problem in earlier times, but now some very good visual
tools are available, both free and commercial, allowing easy construction of WSDLs. The Resources section provides links to one such free WSDL editor.

References

http://wso2.org/library/articles/code-first-or-contract-first-wso2-wsas
http://today.java.net/pub/a/today/2006/08/08/contract-first-web-services-with-axis2.html

**Web service Creation and Consumption using Apache CXF – By Debadatta Mishra**

First of all you have understand that Apache CXF works well with Spring.

**Web service Creation**

Step by Step by process to create a web service using Apache CXF

1. Create an Interface having methods for Webservice invocation and Write the annotation called **@WebService**.
2. Write an implementation class for the above interface by providing the annotation like below.

**@WebService(endpointInterface = "com.ddlab.ws.cxf.SimpleWebService",serviceName = "simplewebservice")**

3. Write an XML file called "**cxf-servlet.xml**" which should be inside the directory "WEB-INF". This xml file contains information about the interface name and implementation class which you have written. It also contains the url or address for the service.
4. Write the web.xml file by providing CXF servlet ie "**org.apache.cxf.transport.servlet.CXFServlet**".
5. Write the build script and deploy the application in application server.

Let us see the complete code details.

**SimpleWebService.java**

```
package com.ddlab.ws.cxf;
import javax.jws.WebService;

@WebService
public interface SimpleWebService
{
	public String getName( String name );
}
```

**SimpleWebServiceImpl.java**

```
package com.ddlab.ws.cxf;
import javax.jws.WebService;

@WebService(endpointInterface = "com.ddlab.ws.cxf.SimpleWebService",
serviceName = "simplewebservice")
public class SimpleWebServiceImpl implements SimpleWebService
{
	public String getName( String name )
	{
		System.out.println("Name from webservice client--->"+name);
		return "Name from server "+name;
	}
}
```

**cxf-servlet.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd">
 <jaxws:server id="jaxwsService" serviceClass="com.ddlab.ws.cxf.SimpleWebService"
address="/simplewebservice">
	<jaxws:serviceBean>
		<bean class="com.ddlab.ws.cxf.SimpleWebServiceImpl" />
	</jaxws:serviceBean>
 </jaxws:server>
</beans>
```

**web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <display-name>cxf</display-name>

   <servlet>
      <servlet-name>cxf</servlet-name>
      <display-name>cxf</display-name>
      <description>Apache CXF Endpoint</description>
      <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>cxf</servlet-name>
      <url-pattern>/services/*</url-pattern>
   </servlet-mapping>

   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>

</web-app>
```

**server-build.xml**

```
<!--
   Author : Debadatta Mishra
   A generic script to create a web service.
   This script helps to create .war file as a web service
   to deploy in J2EE server.
-->

<project name="pojoservice" basedir="." default="create.war">

        <property name="server.src.dir"          value="${basedir}/src" />
        <property name="cfx.home"                value="E:/devsoft/webserviceframeworks/apache-cxf-
2.6.1"/>
        <property name="cfx.lib.dir"             value="${cfx.home}/lib"/>
        <property name="wsdl.dir"                value="${basedir}/wsdl"/>
        <property name="wsdl.file.name"          value="${wsdl.dir}/${ant.project.name}.wsdl"/>
        <property name="build.dir"               value="${basedir}/build"/>
        <property name="dist.dir"                value="${basedir}/dist"/>
        <property name="web.dir"                 value="${basedir}/web"/>
        <property name="webinf.dir"              value="${web.dir}/WEB-INF"/>
        <property name="service.class.name"      value="com.ddlab.ws.cxf.SimpleWebService"/>
        <property name="war.file.name"           value="${ant.project.name}.war"/>
        <property name="service.name"            value="simplewebservice"/>


        <!-- Create classpath setting -->
        <path id="cxf.classpath">
                <pathelement location="${build.dir}/classes"/>

                <fileset dir="${cfx.lib.dir}">
                        <include name="**/*.jar"/>
                </fileset>

        </path>

        <!-- Delete and Create the required directories -->
```

```
<target name="clean">
        <delete dir="${build.dir}"/>
        <delete dir="${dist.dir}"/>
        <delete dir="${wsdl.dir}"/>
        <delete dir="${webinf.dir}/lib"/>
</target>

<target name="init" depends="clean">
        <mkdir dir="${wsdl.dir}"/>
        <mkdir dir="${build.dir}"/>
        <mkdir dir="${dist.dir}"/>
        <mkdir dir="${build.dir}/classes"/>
        <mkdir dir="${webinf.dir}/lib"/>
</target>

<!-- Compile the services -->
<target name="compile.service" depends="init">
        <javac debug="on"
  fork="true"
  destdir="${build.dir}/classes"
  srcdir="${server.src.dir}"
  classpathref="cxf.classpath">
        </javac>
        <jar destfile="${dist.dir}/${ant.project.name}.jar">
                <fileset dir="${build.dir}/classes"/>
        </jar>
</target>

<!-- Generate WSDL as per Apache CXF -->
<target name="java2WSDL" depends="compile.service">
        <java classname="org.apache.cxf.tools.java2ws.JavaToWS" fork="true">
                <arg value="-wsdl"/>
                <arg value="-o"/>
                <arg value="${wsdl.file.name}"/>
                <arg value="-classdir"/>
                <arg value="${build.dir}/classes"/>
                <arg value="${service.class.name}"/>
                <classpath>
                        <path refid="cxf.classpath"/>
                </classpath>
        </java>
</target>

<target name="create.war" depends="java2WSDL">
        <copy toDir="${webinf.dir}/lib" flatten="true">
                <fileset dir="${cfx.lib.dir}">
                        <include name="**/*.jar"/>
                </fileset>
                <fileset dir="${dist.dir}">
                        <include name="*.jar"/>
                </fileset>
        </copy>

        <copy todir="${build.dir}/${war.file.name}">
                <fileset dir="${web.dir}"/>
        </copy>
        <jar destfile="${dist.dir}/${war.file.name}" basedir="${web.dir}">
                <manifest>
                        <attribute name="Manifest-Version"                      value="1.0" />
                        <attribute name="DisableIBMJAXWSEngine"                 value="true" />
                        <attribute name="Author"                                        value="Debadatta
Mishra" />
```

```
                </manifest>
            </jar>
            <echo file="${basedir}/ReadMe/url.txt"
                message="http://localhost:8080/${ant.project.name}/services/${service.name}?wsdl"/>
        </target>
</project>
```

Once you run the build script, you will get the war file and you have to deploy the war file in any application server. In the above case , the url of the web service is given below.
**http://localhost:8080/pojoservice/services/simplewebservice?wsdl**

**Web service Consumption**

1. Download the WSDL url to a location called wsdl in your project directory.
2. Run the build script to invoke wsdl2java of apache CXF service.
3. It will generate the stub classes, compile and create a jar file for the above generated stub classes.
4. Use the above generated jar file in your classe apart from other CXF jar files.
5. Write test class to invoke.

Let use see the code structure below.

```
<!--
   Author : Debadatta Mishra
   A generic script to create a web service.
   This script helps to create .war file as a web service
   to deploy in J2EE server.
-->

<project name="pojoservice" basedir="." default="compile.service">

        <property name="cfx.home"                   value="E:/devsoft/webserviceframeworks/apache-cxf-
2.6.1"/>
        <property name="cfx.lib.dir"            value="${cfx.home}/lib"/>
        <property name="wsdl.dir"                   value="${basedir}/wsdl"/>
        <property name="wsdl.file.name"         value="${wsdl.dir}/${ant.project.name}.wsdl"/>
        <property name="build.dir"                  value="${basedir}/build"/>
        <property name="gen.src.dir"        value="${basedir}/gensrc"/>
        <property name="dist.dir"                   value="${basedir}/dist"/>
        <property name="service.name"           value="simplewebservice"/>

        <!-- Create classpath setting -->
        <path id="cxf.classpath">
            <fileset dir="${cfx.lib.dir}">
                <include name="**/*.jar"/>
            </fileset>
        </path>

        <!-- Delete and Create the required directories -->
        <target name="clean">
            <delete dir="${build.dir}"/>
            <delete dir="${dist.dir}"/>
            <delete dir="${gen.src.dir}"/>
        </target>

        <target name="init" depends="clean">
            <mkdir dir="${build.dir}"/>
            <mkdir dir="${dist.dir}"/>
            <mkdir dir="${gen.src.dir}"/>
        </target>

        <!-- Generate WSDL2JAVA as per Apache CXF -->
  <target name="cxfWSDLToJava" depends="init">
    <java classname="org.apache.cxf.tools.wsdlto.WSDLToJava" fork="true">
```

```
      <arg value="-client"/>
         <arg value="-p"/>
         <arg value="com.ddlab.rnd.ws.cxf"/>
      <arg value="-d"/>
      <arg value="${gen.src.dir}"/>
      <arg value="${wsdl.file.name}"/>
      <classpath>
        <path refid="cxf.classpath"/>
      </classpath>
    </java>
   </target>

        <!-- Compile the services -->
        <target name="compile.service" depends="cxfWSDLToJava">
                <javac debug="on"
         fork="true"
         destdir="${build.dir}"
         srcdir="${gen.src.dir}"
         classpathref="cxf.classpath">
                </javac>
                <jar destfile="${dist.dir}/${ant.project.name}.jar">
                        <fileset dir="${build.dir}"/>
                </jar>
        </target>
</project>

import java.net.URL;
import com.ddlab.rnd.ws.cxf.SimpleWebServiceService;
public class TestCXFWebserviceClient
{
        public static void main(String[] args) throws Exception
        {
                String endPointUrl = "http://localhost:8080/pojoservice/services/simplewebservice?wsdl";
                SimpleWebServiceService service = new SimpleWebServiceService(new URL(endPointUrl));
                String responseStr = service.getSimpleWebServicePort().getName("Deba");
                System.out.println(responseStr);
        }

}
```

## XML-RPC and Apache XML-RPC - By Debadatta Mishra

### What is XML-RPC

XML-RPC is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism. **Always remeber that it is a protocol like soap**.
XML-RPC, the protocol, was created in 1998 by Dave Winer of UserLand Software and Microsoft.
XML-RPC works by sending a HTTP request to a server implementing the protocol. The client in that case is typically software wanting to call a single method of a remote system. Multiple input parameters can be passed to the remote method, one return value is returned. The parameter types allow nesting of parameters into maps and lists, thus larger structures can be transported. Therefore XML-RPC can be used to transport objects or structures both as input and as output parameters.

Examples

An example of a typical **XML-RPC request** would be:

```
<?xml version="1.0"?>
<methodCall>
 <methodName>examples.getStateName</methodName>
 <params>
  <param>
    <value><i4>40</i4></value>
  </param>
 </params>
</methodCall>
```

**An example of a typical XML-RPC response would be:**

```
<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
    <value><string>South Dakota</string></value>
  </param>
 </params>
</methodResponse>
```

**A typical XML-RPC fault would be:**

```
<?xml version="1.0"?>
<methodResponse>
 <fault>
  <value>
   <struct>
    <member>
     <name>faultCode</name>
     <value><int>4</int></value>
    </member>
    <member>
     <name>faultString</name>
     <value><string>Too many parameters.</string></value>
    </member>
   </struct>
  </value>
 </fault>
</methodResponse>
```

## Data types

Common datatypes are converted into their XML equivalents with example values shown below:

| Name | Tag Example | Description |
|---|---|---|

| | | |
|---|---|---|
| array | <array><br><data><br><value><i4>1404</i4></value><br><value><string>Something here</string></value><br><value><i4>1</i4></value><br></data><br></array> | Array of values, storing no keys |
| base64 | <base64>eW91IGNhbid0IHJlYWQgdGhpcyE=</base64> | Base64-encoded binary data |
| boolean | <boolean>1</boolean> | Boolean logical value (0 or 1) |
| date/time | <dateTime.iso8601>19980717T14:08:55</dateTime.iso8601> | Date and time in ISO 8601 format |
| double | <double>-12.53</double> | Double precision floating point number |
| integer | <i4>42</i4><br>or<br><int>42</int> | Whole number, integer |
| string | <string>Hello world!</string> | String of characters. Must follow XML encoding. |
| struct | <struct><br><member><br><name>foo</name><br><value><i4>1</i4></value><br></member><br><member><br><name>bar</name><br><value><i4>2</i4></value><br></member><br></struct> | Associative array |
| nil | <nil/> | Discriminated null value; an XML-RPC extension |

## Criticism

Critics of XML-RPC argue that RPC calls can be made with plain XML, and that XML-RPC does not add any value over XML. Both XML-RPC and XML require an application-level data model, such as which field names are defined in the XML schema or the parameter names in XML-RPC. Furthermore, XML-RPC uses about 4 times the number of bytes compared to plain XML to encode the same objects, which is itself bloated compared to JSON.

## Apache XML-RPC

Apache XML-RPC is a Java implementation of XML-RPC, a popular protocol that uses XML over HTTP to implement remote procedure call. Follow the step by step process to create XML-RPC for both service and client side. We want to create a Calculat using XML RPC.

**1. Write simple class called "MyCalculator"**
**2. Define the property file called "XmlRpcServlet.properties". This file must be placed inside a package called "org.apache.xmlrpc.webserver". This file should**
**contain the name and the complete cannonical class name.**
**3. Define the deployment descriptor ie web.xml with the XML-RPC servlet "org.apache.xmlrpc.webserver.XmlRpcServlet".**
**4. Build your application using Apache Ant and deploy as war file.**

**5. Invoke the url by sending XML-RPC request XML to the url.**
**6. Make sure that the url will be like this.**
**http://localhost:8080/mycalculator/mycalculator.**
**It means it will like this "http://<IP ADDRESS>:<PORT>/<APPLICATION_NAME or war file>/<URL PATTERN defined in web.xml>"**

Let us see the code structure below.

```
package com.ddlab.rnd.xmlrpc.server;
public class MyXMLRPCCalculator
{
	public String add( String firstNo , String secondNo ) throws Exception
	{
		String result = null;
		try
		{
			int no1 = Integer.parseInt(firstNo);
			int no2 = Integer.parseInt(secondNo);
			System.out.println("First No : "+no1);
			System.out.println("Second No : "+no2);
			result = String.valueOf((no1+no2));
			System.out.println("Result : "+result);
		}
		catch (NumberFormatException nfe)
		{
			nfe.printStackTrace();
		}
		catch( Exception e )
		{
			System.out.println("UnExpected Server Exception ...");
			e.printStackTrace();
			throw new Exception("UnExpected Server Exception ...");
		}
		return result;
	}
}
```

**XmlRpcServlet.properties**

```
MyXMLRPCCalculator=com.ddlab.rnd.xmlrpc.server.MyXMLRPCCalculator
```

**This above file must be placed inside a package called "org.apache.xmlrpc.webserver"**

**web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
	http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>xmlrpc</display-name>

  <servlet>
    <servlet-name>XmlRpcServlet</servlet-name>
    <servlet-class>org.apache.xmlrpc.webserver.XmlRpcServlet</servlet-class>
    <init-param>
      <param-name>enabledForExtensions</param-name>
      <param-value>true</param-value>
      <description>
        Sets, whether the servlet supports vendor extensions for XML-RPC.
      </description>
    </init-param>
```

```
   </servlet>
   <servlet-mapping>
      <servlet-name>XmlRpcServlet</servlet-name>
      <url-pattern>/mycalculator</url-pattern>
   </servlet-mapping>
</web-app>
```

The followings are the test classes.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.URL;
import java.net.URLConnection;

public class TestPostXMLPRC
{
        private static String getRequestXML( String filePath ) throws Exception
        {
                File file = new File(filePath);
                byte[] buffer = new byte[(int)file.length()];
                InputStream in = new FileInputStream(file);
                in.read(buffer);
                return new String(buffer);
        }

        public static void main(String[] args) throws Exception
        {
                String data = getRequestXML("data/XMLRPCRequest.xml");
                URL url = new URL("http://localhost:8080/mycalculator/mycalculator");
                URLConnection conn = url.openConnection();
                conn.setDoOutput(true);
                OutputStreamWriter wr = new OutputStreamWriter(conn.getOutputStream());
                wr.write(data);
                 wr.flush();

           // Get the response
           BufferedReader rd = new BufferedReader(new
        InputStreamReader(conn.getInputStream()));
           String line;
           while ((line = rd.readLine()) != null) {
              System.out.println(line);
           }
           wr.close();
           rd.close();
        }
}

import java.net.URL;
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;
import org.apache.xmlrpc.client.XmlRpcCommonsTransportFactory;

public class TestXMLRPCClient
```

```
{
	public static void main(String[] args) throws Exception
	{
		XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
		config.setServerURL(new URL("http://localhost:8080/mycalculator/mycalculator"));
		config.setEnabledForExtensions(true);
		config.setConnectionTimeout(60 * 1000);
		config.setReplyTimeout(60 * 1000);

		XmlRpcClient client = new XmlRpcClient();
		// use Commons HttpClient as transport
		client.setTransportFactory(
				new XmlRpcCommonsTransportFactory(client));
		// set configuration
		client.setConfig(config);

		/*
		 * The structure of request XML will be like this
		 * <?xml version="1.0"?>
			<methodCall>
				<methodName>MyXMLRPCCalculator.add</methodName>
				<params>
				  <param>
				    <value><string>11</string></value>
				  </param>
					<param>
				    <value><string>22</string></value>
				  </param>
				 </params>
				</methodCall>
		 */

		Object[] params = new Object[]
				{ new String("11"), new String("44") };
		String result = (String) client.execute("MyXMLRPCCalculator.add", params);
		System.out.println("Final Result = " + result);
	}
}
```

The request and response xml files are given below.

**Request XML File**

```
<?xml version="1.0"?>
<methodCall>
 <methodName>MyXMLRPCCalculator.add</methodName>
 <params>
  <param>
    <value><string>40</string></value>
  </param>
  <param>
    <value><string>40</string></value>
  </param>
 </params>
</methodCall>
```

**Response XML File**

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse xmlns:ex="http://ws.apache.org/xmlrpc/namespaces/extensions">
 <params>
  <param>
```

```
    <value>33</value>
   </param>
  </params>
</methodResponse>
```

**The following jar files are used.**

**commons-codec-1.6.jar**
**commons-httpclient-3.1.jar**
**commons-logging-1.1.jar**
**servlet-api.jar**
**ws-commons-util-1.0.2.jar**
**xmlrpc-client-3.1.3.jar**
**xmlrpc-common-3.1.3.jar**
**xmlrpc-server-3.1.3.jar**

You can also use any http posting tools like **TCPMON from Apache**.