Creating Axis 2 based Web Service with NetBeans 5.5

This tutorial is about developing and deploying Axis 2 Web Service with NetBeans on Tomcat. The tutorial itself is very similar to the last tutorial “Developing Axis 1.X Web Service with NetBeans”. Although the last tutorial was very simple as I do believe that Axis 1.X will be out of picture soon and developers will be encouraged to use Axis 2, which provides much more functionality, stability, compliance with standards and flexibility.

This tutorial is not about Axis 2. I will not cover the Axis 2 architecture in detail or different configuration options for Axis 2. This tutorial will help you to develop Axis 2 Web Services with NetBeans and try different configuration options yourself. I assume readers are familiar with Ant, Web Services and NetBeans and have desire to develop Axis 2 Web Service with minimum efforts.

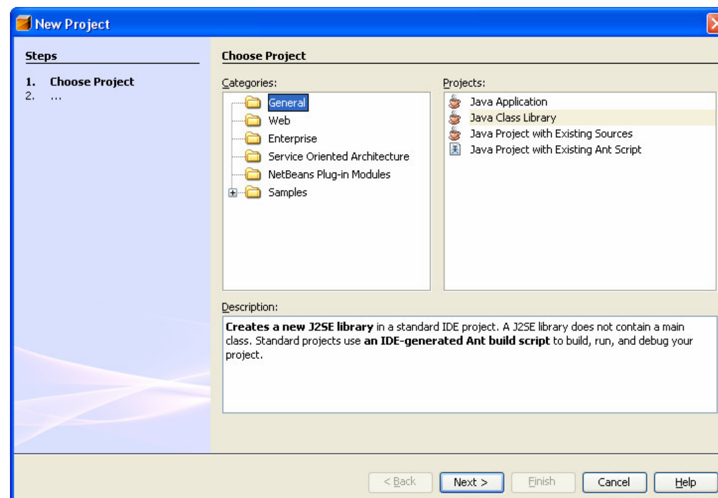
Pre-requisite for this tutorial are following:

- NetBeans 5.5 with J2SE1.5.
- Axis 2 1.1

Although the tutorial is using the above mentioned build and releases, but it doesn’t have any dependency on those specific releases. One can also use Axis2 1.0 or 0.9 with NetBeans 5.0.

Steps involved in creating Axis 1.X based Web Services with NetBeans are following:

1. Create a new project (File > New Project).
2. This time it is simple Java Application or even Java Library because Axis 2 Web Service doesn’t rely on web.xml and specific directory structure created by NetBeans for Web Application.
3. Select “General” from “Categories” and “Java Class Library” from “Projects” as shown below. The reason for choosing Java Class Library rather than Java Application is that we don’t need main class with main method for our Web Service.



4. Give any name to our Project. I am using the name “Axis2AdvancedTest” for this tutorial.
5. I am creating slightly interesting application than simple echo Web Service. We will create few data classes which are in fact Java Beans to be consumed in the Web Service.
6. Create three Java classes Name, Address and Person in the package “my.axis2.second.example.data”. The Person class creates an object based on the name and address object.
7. The Name class is very simple and is shown below:

```
package my.axis2.second.example.data;

public class Name {

    /** Creates a new instance of Name */
    public Name() {
    }

    /**
     * Holds value of property firstName.
     */
    private String firstName;

    /**
     * Getter for property firstName.
     * @return Value of property firstName.
     */
    public String getFirstName() {
        return this.firstName;
    }

    /**
     * Setter for property firstName.
     * @param firstName New value of property firstName.
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    /**
     * Holds value of property lastName.
     */
    private String lastName;

    /**
     * Getter for property lastName.
     * @return Value of property lastName.
     */
    public String getLastName() {
        return this.lastName;
    }
```

```
    /**
     * Setter for property lastName.
     * @param lastName New value of property lastName.
     */
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Name(String firstNameValue, String lastNameValue) {
        this.firstName = firstNameValue;
        this.lastName = lastNameValue;
    }

}
```

8. The Address class is also very simple and is shown below:

```
package my.axis2.second.example.data;

public class Address {

    /**
     * Holds value of property city.
     */
    private String city;

    /**
     * Holds value of property country.
     */
    private String country;

    /**
     * Holds value of property houseNumber.
     */
    private String houseNumber;

    /**
     * Holds value of property postCode.
     */
    private String postCode;

    /**
     * Holds value of property streetName.
     */
    private String streetName;

    /** Creates a new instance of Address */
    public Address() {}
```

```
public Address(String houseNumberValue, String streetNameValue,
        String cityValue, String countryValue,
            String postCodeValue) {
    this.houseNumber = houseNumberValue;
    this.streetName = streetNameValue;
    this.city = cityValue;
    this.country = countryValue;
}

/**
 * Getter for property houseNumber.
 * @return Value of property houseNumber.
 */
public String getHouseNumber() {
    return this.houseNumber;
}

/**
 * Setter for property houseNumber.
 * @param houseNumber New value of property houseNumber.
 */
public void setHouseNumber(String houseNumber) {
    this.houseNumber = houseNumber;
}

/**
 * Getter for property streetName.
 * @return Value of property streetName.
 */
public String getStreetName() {
    return this.streetName;
}

/**
 * Setter for property streetName.
 * @param streetName New value of property streetName.
 */
public void setStreetName(String streetName) {
    this.streetName = streetName;
}

/**
 * Getter for property city.
 * @return Value of property city.
 */
public String getCity() {
    return this.city;
}

/**
```

```
     * Setter for property city.
     * @param city New value of property city.
     */
    public void setCity(String city) {
        this.city = city;
    }

    /**
     * Getter for property country.
     * @return Value of property country.
     */
    public String getCountry() {
        return this.country;
    }

    /**
     * Setter for property country.
     * @param country New value of property country.
     */
    public void setCountry(String country) {
        this.country = country;
    }

    /**
     * Getter for property postCode.
     * @return Value of property postCode.
     */
    public String getPostCode() {
        return this.postCode;
    }

    /**
     * Setter for property postCode.
     * @param postCode New value of property postCode.
     */
    public void setPostCode(String postCode) {
        this.postCode = postCode;
    }
}
```

9. The Person class is slightly different and is shown below:

```
package my.axis2.second.example.data;

public class Person {

    /**
     * Holds value of property address.
     */
    private Address address;
```

```
    /**
     * Holds value of property name.
     */
    private Name name;

    /** Creates a new instance of Person */
    public Person() {}

    public Person(Name nameValue, Address addressValue) {
        this.name = nameValue;
        this.address = addressValue;
    }

    /**
     * Getter for property name.
     * @return Value of property name.
     */
    public Name getName() {
        return this.name;
    }

    /**
     * Setter for property name.
     * @param name New value of property name.
     */
    public void setName(Name name) {
        this.name = name;
    }

    /**
     * Getter for property address.
     * @return Value of property address.
     */
    public Address getAddress() {
        return this.address;
    }

    /**
     * Setter for property address.
     * @param address New value of property address.
     */
    public void setAddress(Address address) {
        this.address = address;
    }
}
```

10. Now we will create a Web Service Person Web Service which can create a Person and add it into the registry. The Web Service is in the package "my.axis2.second.example.service". Below is the implementation of our PersonService.

```
package my.axis2.second.example.service;

import my.axis2.second.example.data.*;

public class PersonService {

    /**
     * Holds value of property person.
     */
    private Person person = null;

    /**
     * Getter for property person.
     * @return Value of property person.
     */
    public Person getPerson() {
        return this.person;
    }

    public void createPersonFromObject(Person personValue) {
        this.person = personValue;
    }

    public void createPersonFromNameAddress(Name nameValue,
            Address addressValue) {
        person = new Person(nameValue, addressValue);
    }

}
```

11. This is the whole implementation we required to develop our Web Service. Technically we will not use the data classes and the better approach was to just create different interfaces. We need the data classes/interfaces only to generate the appropriate WSDL.
12. Select the project "Axis2AdvancedTest" in the Projects tab and right click. Select **Properties > Libraries > Add Library**.
13. Add Library will bring the list of libraries already registered with NetBeans, click on button "**Manage Libraries**". This will bring another window and there select "**New Library**".
14. Create a new Library called Axis2 and then add all jar files available in the Axis2_Home\\lib directory of Axis. Axis2_Home is the directory where you have unzipped the Axis2. In my case it is "F:\axis2-1.1.1\". Don't forget to add the newly created library to the project.
15. Time to edit the build.xml to add Axis2 based information in the Ant script. We will start with adding different properties specific to our Web Service.
16. Double click the build.xml and add following properties before the closing </project> tag. Each property is followed by the comment; which explains its role.

```
<!-- Location to put compiled classes, service.xml and WSDL -->
    <property name="build.dir" value="${basedir}/axis2Service"/>
```

```
    <property name="build.classes" value="${build.dir}/classes" />
    <property name="build.lib" value="${build.dir}/lib" />
    <property name="build.resources" value="${build.dir}/resources" />

<!- - The jar file for data to be used by the client -->
     <property value="XBeans-packaged.jar"
           name="xbeans.packaged.jar.name" />

    <!-- Web Service Name-->
    <property name="service.name" value="PersonWS"/>

    <!-- Web Service Namespace -->
    <property name="service.namespace"
          value="http://service.example.second.axis2.my"/>

    <!-- Web Service Data Namespace -->
    <property name="service.data.namespace"
          value="http://data.example.second.axis2.my"/>

    <!-- Java Package Name -->
    <property name="service.java.packageName"
          value="my.axis2.second.example.service"/>

    <!-- Physical Location of Java Implementation
    <property name="service.java.path" value="java\axis2\test\ws"/>
    -->

    <!-- Name of Java Implementation Class-->
    <property name="service.java.name" value="PersonService"/>

    <!-- **** Don't edit these ***-->
    <!-- Java Class Name with Package Information-->
    <property        name="service.class"
       value="${service.java.packageName}.${service.java.name}"/>

    <!-- Name of the WSDL, which will be Generated-->
    <property      name="service.wsdl"
       value="${service.name}.wsdl"/>
```

17. Most of the properties are similar to the properties used for the Axis 1.X tutorial. Few things to notice are that we are creating the Directory with the name “build.dir” where source code, java compiled classes, WSDL and Axis2 configuration files will be placed. In the case of Axis 1.X, we selected Web Application and NetBeans IDE was automatically creating “web” with various sub directories to hold different type of files.
18. To generate WSDL from our initial implementation and then create the stubs and skeletons from the WSDL we need to add Axis2 specific jar files in the class path. Following Ant target will add the corresponding jar files in the class path and validate the presence of required classes.

```
<!-- Axis Home -->
<property name="axis2.home" value="F:\axis2-1.1.1" />

<path id="base.libraries">
   <!-- Moved from Java 2 Wsdl -->
   <pathelement location="${build.classes}"/>
   <fileset dir="${axis2.home}\lib">
      <include name="*.jar" />
   </fileset>
</path>

<property name="axis2.classpath" value="base.libraries" />

<target name="validate">
   <available property="axis2.classpath" value="base.libraries"
      file="${axis2.home}\axis2-java2wsdl-1.1.1.jar"/>
</target>
```

19. Now we need to compile our Web Service implementation and data classes. In the case of Axis 1.X we did from the NetBeans IDE by selecting “Clean and Build” but this time I am calling the <compile> target.

```
<target name="compile.service" depends="validate">
   <antcall target="compile" />
</target>
```

20. After compiling the classes we need to generate the WSDL for our Web Service. The target shown below will generate the WSDL based on the properties already set previously.

```
<target name="generate.wsdl" depends="compile.service">
   <taskdef name="java2wsdl"
         classname="org.apache.ws.java2wsdl.Java2WSDLTask"
         classpathref="${axis2.classpath}"/>
   <java2wsdl outputfilename="${service.wsdl}"
          className="${service.class}"
          outputLocation="${build.dir}\wsdl"
          targetNamespace="${service.namespace}"
          servicename="${service.name}"
          schemaTargetNamespace="${service.data.namespace}">
   </java2wsdl>
</target>
```

21. Now from the generated WSDL we will create the Service Implementation and corresponding Data Bindings. I prefer starting Web Service development from the WSDL, but rather than writing WSDL from scratch it is always easier to write few interfaces and generate template WSDL. The template WSDL can be refined for better control. In this example we are not modifying the WSDL but in real world scenarios you may need to add existing XML Schemas in the WSDL for tight coupling.
22. The target shown below will generate the corresponding stubs and skeletons for us. I have added comments which are self explanatory. In the end of the target I am copying few configuration files generated as the result of WSDL2Java from default location to my preferred location. This has no

specific benefits other than just my personal preference and to make source code available in the IDE for modification. Axis2 doesn't have WSDD for deployment in fact it uses "services.xml" which will be automatically generated by WSDL2Java task.

```
<target name="generate.stub" depends="generate.wsdl">
        <delete dir="${build.dir}/classes" />
        <java classname="org.apache.axis2.wsdl.WSDL2Java"
                fork="true" classpathref="${axis2.classpath}">
            <!-- <classpath refid="axis.classpath"/> -->
            <!-- Location of the WSDL -->
            <arg value="-uri"/>
            <arg file="${build.dir}/wsdl/${service.wsdl}"/>

            <!-- Output Directory for generated source code -->
            <arg value="-o"/>
            <arg file="${build.dir}"/>

            <!-- Package structure for generated classes-->
            <arg value="-p"/>
            <arg value="${service.java.packageName}"/>

            <arg value="-ss"/>
            <arg value="-g"/>
            <arg value="-sd"/>
            <arg value="-t"/>

            <!-- Type of Data Bindings -->
            <arg value="-d"/>
            <arg value="xmlbeans"/>
        </java>
        <delete dir="${build.dir}/wsdl"/>

        <!-- Copying  source files to access from the IDE Source Packages -->
        <copy todir="${basedir}/src">
             <fileset dir="${build.dir}/src">
                <include name="**/**/*.java"/>
            </fileset>
        </copy>

        <!-- Copying  test files to access from the IDE Test Packages -->
        <copy todir="${basedir}/test">
             <fileset dir="${build.dir}/test">
                <include name="**/**/*.java"/>
            </fileset>
        </copy>
    </target>
```

23. Click the Projects tab and expand the Source Packages. Now it has more than two packages initially we have created. The number of files and packages generated as result of last task depends on the Data Binding preference. In this tutorial I have selected XMLBeans for data binding, so most of the files are

related to XML Beans. The most important Java Class is “PersonWSSkeleton” in the package “my.axis2.second.example.service”. This class is equivalent to XXXBindingImpl class of Axis 1.X and implements the business logic of our Web Service. Modify this class, for the sake of tutorial I am just printing the values passed to the Web Service operations.

```
package my.axis2.second.example.service;

/**
 *  PersonWSSkeleton java skeleton for the axisService
 */
public class PersonWSSkeleton {

    /**
     * Auto generated method signature
     *
     *  @param param0
     *
     */
    public void createPersonFromObject
            (my.axis2.second.example.data.CreatePersonFromObjectDocument
                    param0) {

        // Todo fill this with the necessary business logic

            System.out.println(param0.getCreatePersonFromObject().getPersonVa
            lue().getName().getFirstName());
    }

    /**
     * Auto generated method signature
     *
     *  @param param1
     *
     */
    public void createPersonFromNameAddress
            (my.axis2.second.example.data.CreatePersonFromNameAddressDocu
ment param1) {

        // Todo fill this with the necessary business logic

            System.out.println(param1.getCreatePersonFromNameAddress().getN
            ameValue().getFirstName());
    }

    /**
     * Auto generated method signature
     *
     *
     */
    public my.axis2.second.example.data.GetPersonResponseDocument
```

```
        getPerson() {

      // Todo fill this with the necessary business logic
      throw new java.lang.UnsupportedOperationException
        ("Please implement " + this.getClass().getName()
          + "#getPerson");
    }
  }
```

24. The next step is to create the Data Binding Jar files for the client application. Our Data Binding Jar files are based on XMLBeans.

```
<target name="jar.xbeans">
      <mkdir dir="${build.lib}" />
      <jar basedir="${build.dir}/resources"
        destfile="${build.lib}/${xbeans.packaged.jar.name}"
        excludes="**/services.xml, **/*.xsd" />
  </target>
```

25. Now we need to compile our Web Service Implementation which we modified in the Step 23. Before compiling the Web Service and other classes we will perform validation test; which is all optional. Validation test checks the presence of different jar files in the class path.

```
<target depends="jar.xbeans" name="pre.compile.test">
      <!-- Test the classpath for the availability of necesary classes   -->
      <available classpathref="${axis2.classpath}"
        property="xbeans.available"
        classname="org.apache.xmlbeans.XmlObject" />
      <available classpathref="${axis2.classpath}" property="stax.available"
        classname="javax.xml.stream.XMLStreamReader" />
      <available classpathref="${axis2.classpath}" property="axis2.available"
        classname="org.apache.axis2.engine.AxisEngine" />
      <condition property="true">
        <and>
          <isset property="xbeans.available" />
          <isset property="stax.available" />
          <isset property="axis2.available" />
        </and>
      </condition>
      <!-- Print out the availabilities  -->
      <echo message="XmlBeans Availability = ${xbeans.available}" />
      <echo message="Stax Availability = ${stax.available}" />
      <echo message="Axis2 Availability = ${axis2.available}" />
  </target>
```

26. Now we will compile the source code for our Web Service.

```
<target depends="pre.compile.test" name="compile.src" >
      <mkdir dir="${build.classes}"/>
      <javac debug="on" memoryMaximumSize="256m"
        memoryInitialSize="256m"
          fork="true" destdir="${build.classes}" srcdir="${build.dir}\src"
        classpathref="${axis2.classpath}">
        <classpath location="${build.lib}/${xbeans.packaged.jar.name}" />
      </javac>
```

```
</target>
```

27. After compiling the Web Service we will jar them with the extension .aar; which is the requirement of Axis2.

```
<target depends="compile.src" name="jar.server">
    <mkdir dir="${build.classes}/META-INF"/>
    <mkdir dir="${build.classes}/lib"/>
    <copy toDir="${build.classes}/META-INF" failonerror="false">
        <fileset dir="${build.dir}\resources">
            <include name="*.xml" />
            <include name="*.wsdl" />
            <include name="*.xsd" />
            <exclude name="**/schemaorg_apache_xmlbean/**" />
        </fileset>
    </copy>
    <copy file="${build.lib}/${xbeans.packaged.jar.name}"
        toDir="${build.classes}/lib" />
    <jar destfile="${build.lib}/${service.name}.aar">
        <fileset excludes="**/Test.class" dir="${build.classes}" />
    </jar>
</target>
```

28. Once creating archive file for the Web Service we will create the jar file for the client which will contain the Data Binding jar file created in Step 24 along with different XMLBeans generated Java Classes used as parameters for our Web Service operations.

```
<target name="jar.client">
    <jar destfile="${build.lib}/${service.name}-test-client.jar">
        <fileset dir="${build.classes}">
            <exclude name="**/META-INF/*.*" />
            <exclude name="**/lib/*.*" />
            <exclude name="**/*MessageReceiver.class" />
            <exclude name="**/*Skeleton.class" />
        </fileset>
        <fileset dir="${build.resources}">
            <exclude name="**/*.wsdl" />
            <exclude name="**/*.xsd" />
            <exclude name="**/*.xml" />
        </fileset>
    </jar>
</target>
```

29. There are two ways to test the Web Service. One way is to test the Web Service by deploying it on Tomcat which is already configured with Axis2 or to use the standalone simple HTTP Server bundled with Axis2. First we will test the Web Service using the built in server. To test the test the Web Service against bundled HTTP Server; we need to create “repository /services” in the build.dir and copy the .aar file in the services directory. The following Ant target accomplish the required tasks.

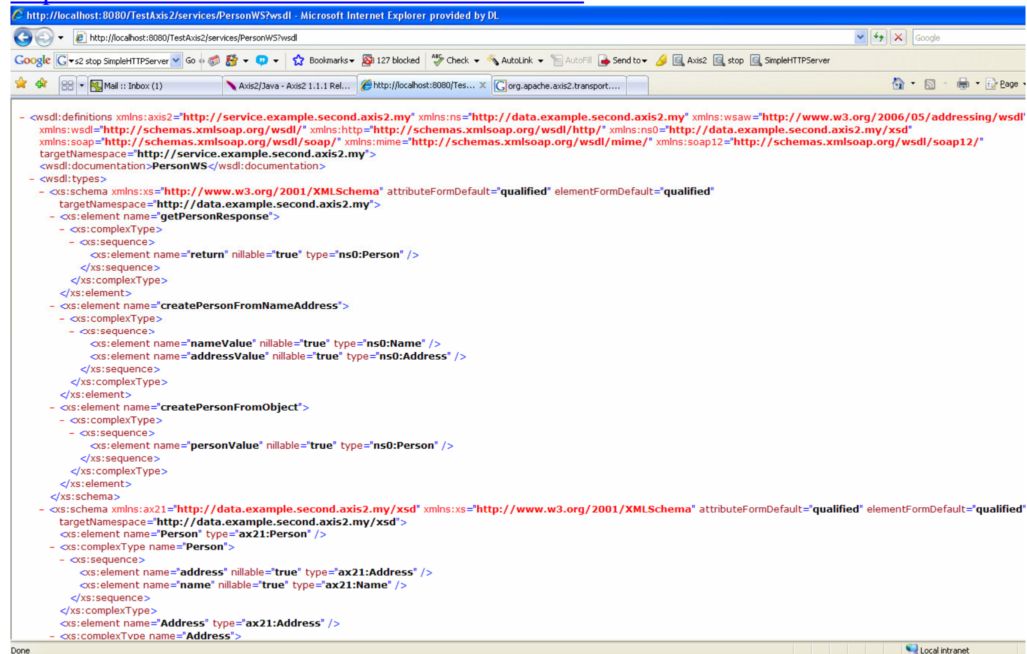
```
<target name="make. repository " >
    <mkdir dir="${build.dir}/ repository /"/>
    <mkdir dir="${build.dir}/ repository /services"/>
    <copy file="${build.lib}/${service.name}.aar"
```

```
        toDir="${build.dir}/ repository /services/"/>
    </target>
```

30. Now we need to run the server and pass the location of our “repository” directory as argument. **NOTE:** To run the bundled HTTP Server you need to set AXIS2_Home and JAVA_HOME. Check Axis2 documentation for more details.

```
<target name="start.server" depends="make.repository">
    <java fork="true"
        classname="org.apache.axis2.transport.http.SimpleHTTPServer"
        classpathref="${axis2.classpath}">
        <arg value="${build.dir}/repository"/>
    </java>
</target>
```

31. If everything goes well then you can check the deployed Web Service on the URL: http://localhost:8080/axis2/services/.
32. To deploy the Web Service on the Tomcat, install the Tomcat and Axis2 WAR (Web Archive) Distribution. Deploy the Axis2 WAR (Web Archive) Distribution on the Tomcat according to the instruction which is in fact unzip the war file and copy the unzipped war file call is “axis2” in the TOMCAT_HOME\webapps directory.
33. Copy the Service.aar file in the directory TOMCAT_HOME\webapps\axis2\WEB-INF\services. You can test the Web Services by running the Server and accessing the Web Services on the URL http://localhost:8080/axis2/services/listServices.



34. There are few better tricks to deploy Axis2 Web Services on the Tomcat which I will discuss in the next Tutorial.