

Writing A JAX-RPC Client from WSDL

Suppose you know there's a Web service out there that exposes its description as a WSDL file. Maybe you located the WSDL file via a Google search, or perhaps you retrieved it from a UDDI registry.

Now your task is to write a Java client program that accesses that Web service. And you have decided to use Sun's Java Web Services Developer Pack (JWS DP) as your toolkit. Should be simple, right?

Well it's not that easy! At least, it's not quite as simple as it is in other Web services toolkits, like Apache Axis or the Microsoft .NET toolkit. But it's not impossible, either, as this tutorial will show.

I am assuming that you've already downloaded and installed the JWS DP version 1.1 and have run through the "Hello, World" JAX-RPC tutorial.

Shameless plug: If you want to learn more about Java Web services, XML or J2EE, consider an onsite training course from Descriptor Systems! See www.descriptor.com for details, or drop an email to Joel Barnum.

Writing a JAX-RPC Client from WSDL Stubs

Steps:

- _1. Your first job is to locate the WSDL for the service. As an example, we will use the Cape Science Airport Weather Service. The WSDL for this service is available from <http://live.capescience.com/wsd/AirportWeather.wsdl>. The service's home page is at <http://www.capescience.com/webservices/airportweather/index.shtml>.
- _2. Next you need to examine the WSDL to see if the service uses the Remote Procedure Call (RPC) model or is document oriented. In the WSDL, look for the *binding* element and look for the *soap:binding* element's *style* attribute. We are assuming RPC here (and it just so happens that the AirportWeather service is an RPC).
- _3. Next you need to write a file to configure the JWS DP *wscompile* tool that you will run in a moment. Create a file named **config.xml** with the following contents:

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

The main element of interest here is the *wsdl* element, which contains attributes that specify the location of the WSDL file and the package name that *wscompile* will use to generate Java stubs for the service.

The WSDL reference shown here is for a local file, but it could be a URL (i.e. we are assuming that you've already downloaded the WSDL file into the same directory where you created the config.xml file).

- _4. Now you can run *wscompile* to generate a Java stub that will handle all of the low-level SOAP protocol

for you. In a command prompt, change to the directory where you saved the config.xml file and enter:

```
wscompile -gen:client config.xml -keep
```

The *gen:client* flag tells wscompile to generate stubs and the *keep* switch tells it not to delete the source for the stubs and related classes. It can be helpful to examine the source, as we shall see.

- _5. The wscompile tool will create a package (and subdirectory) named **airport**. Look in this directory for all of the .java and .class files generated.

The most interesting for our purposes are the **Station.java** and **AirportWeather.java** files. Please examine these.

AirportWeather.java defines a Java interface that represents the service. It defines a *getStation* method that returns a remote reference, or stub, for the service.

Station.java defines a Java interface that represents the methods on the Web service. You should be able to match these methods with elements defined in the WSDL, specifically in the *portType* element.

- _6. Now you can write the client. Create a file named **Client.java** with the following contents:

```
import airport.*;

public class Client
{
    public static void main (String args[])
        throws Exception
    {
        AirportWeather_Impl service =
            new AirportWeather_Impl();
        Station stub = service.getStation();

        String temp = stub.getTemperature ( "KCID" );

        System.out.println ( temp );
    }
}
```

- _7. This client first creates an instance of the *AirportWeather_Impl* class, which implements the *AirportWeather* interface that represents the service. The code then retrieves a stub that represents the remote Web service and calls the *getTemperature* method, passing the station code for Cedar Rapids, Iowa (you can look up other station codes on the Cape Science web page).

- _8. Now you can build and run the client. To make this a bit easier, enter the following Ant tasks into a file named **build.xml**:

```
<?xml version="1.0"?>

<project default="run-client">

    <!-- define the destination directories -->
    <property name="jwsdp-home"
        value="c:/java/jwsdp-1.1"/>

    <property name="jaxrpc-home"
        value="${jwsdp-home}/jaxrpc-1.0.3"/>
```

```

<property name="saaaj-home"
  value="\${jwsdp-home}/saaaj-1.1.1"/>

<property name="jaxm-home"
  value="\${jwsdp-home}/jaxm-1.1.1"/>

<!-- setup the CLASSPATH -->
<path id="classpath">
  <fileset dir="\${jwsdp-home}/common/lib">
    <include name="*.jar" />
  </fileset>

  <fileset dir="\${jaxrpc-home}/lib">
    <include name="*.jar" />
  </fileset>

  <fileset dir="\${saaaj-home}/lib">
    <include name="*.jar" />
  </fileset>

  <fileset dir="\${jaxm-home}/lib">
    <include name="*.jar" />
  </fileset>

  <fileset dir="\${jwsdp-home}/jwsdp-shared/lib">
    <include name="*.jar" />
  </fileset>
</path>

<!-- define the "build" target -->
<target name="build">
  <javac srcdir=".">
    <classpath refid="classpath"/>
    <classpath>
      <pathelement location="."/>
    </classpath>
  </javac>
</target>

<!-- define the "run-client" target -->
<target name="run-client" depends="build">
  <java classname="Client" fork="true">
    <classpath refid="classpath"/>
    <classpath>
      <pathelement location="."/>
    </classpath>
  </java>
</target>

</project>

```

Note: You will need to adjust the *values* of the *jwsdp-home* and related properties to match where you installed the JWS DP.

_9. To compile and run your Web service client, enter:

```
ant -emacs
```

The *emacs* switch tells Ant to produce less verbose (and more readable) output.

You should see the current temperature at the airport station!