Web Services & Apache Axis

Dr. Dobb's Journal November, 2004

The right tools make the job easier

By Paul Tremblett

Paul is a member of the technical staff at AudioAudit Inc. He can be contacted at paul@tremblett.ca.

Apache Axis is one of a growing number of tools designed to simplify the development of web services. It attempts to do this by delivering a SOAP server, a simple administration client, an API, and client-side code generators that shield you from the details of the Simple Object Access Protocol (SOAP) and the Web Services Definition Language (WSDL). In this article, I use Apache Axis (http://ws.apache.org/axis/) to work with web services from both the client and server side.

A Working Web Service

For starters, take a look at the WSDL for a web service at http://tremblett.ca/axis/TemperatureConversion.jws?WSDL. Of course, what you see depends on the browser you're using. Here, I use Internet Explorer; see Figure 1. I'm presenting this screen for comparison with subsequent examples.

At the outset, you must have a properly configured operational application server. The Axis documentation recommends Jakarta Tomcat, but others have been reported to work without problems. The examples at tremblett.ca run under Resin 2.1.12 on Linux. For the examples in this article, I use Jakarta Tomcat running on Macintosh OS X 10.3.

Once you have satisfied the prerequisite, you can develop and deploy a web service by executing these three steps:

1. Download and extract the software. Point your browser to the Apache Software Foundation's Apache Axis Project (http://ws.apache.org/axis/), look for the latest stable build, and download it. The software is available as both a gzipped tarball and .zip file.
2. Deploy the Axis server. This step is easy once you know that the Axis server is a web application. It is not delivered as a WAR file but, if you examine the installation directory, you will see a webapps directory in which you will find the axis context (directory). If you follow the directory tree until you reach axis-1_1/webapps/axis/WEB-INF, you see the web.xml file you would expect to see in any web application. Examine its contents and you'll notice that it defines three servlets—AxisServlet, AdminServlet, and SOAPMonitorService. The first of these is the servlet invoked by Tomcat on incoming web-service requests (that is, those requests with a *.jws URL pattern). The other servlets are supplied for convenience. AdminServlet is commented out and you should probably leave it that way until you understand the security implications. The good news is that you really don't have to pay attention to any of this for now. All you have to do is deploy the web application by recursively copying webapps/axis from the distribution directory into the webapps directory of your application server. If you are using Jakarta Tomcat, it's likely to be /usr/local/tomcat/webapps. Because I'm using Mac OS X, my Tomcat is at /Library/Tomcat. After the software has been copied, assuming Tomcat is listening on port 8080, point your browser to http://localhost:8080/axis/ and you will see a display like Figure 2. Validate the server configuration by clicking on the link labeled "Validate the local

installation's configuration." This launches happyaxis.jsp. Follow the instructions until happyaxis.jsp reports that all required components can be located. Don't worry about the optional components.

3. Write eight lines of Java code:

```
public class TemperatureConversion {
public double f2c(double f) {
return (f- 32.) * 5. / 9.;
}
public double c2f(double c) {
return (c * 9. / 5.) + 32.;
}
}
```

This is a Java class that converts Fahrenheit to Celsius and vice versa. Save it in the webapps/axis directory but, instead of using a .java extension, name it TemperatureConversion.jws.

That's it. If you find it difficult to believe that creating and deploying a web service could possibly be that easy, point your browser at http://localhost:8080/axis/TemperatureConversion.jws?WSDL and compare what you see in the browser window to Figure 1. What's happening here is that the request is sent by Tomcat to AxisServlet because it matches the *.jws pattern and the servlet is generating the WSDL from the .jws file you just created.

Writing a Web-Service Client

You can access the web service using TemperatureConversionClient (Listing One) and three command-line arguments.

- The first is the host name (including the port) on which the web service is located.
- The second is either "f2c" or "c2f," depending on whether you want to convert from Fahrenheit to Celsius or vice versa. You will recognize these as the names of the methods in the Java code you just wrote. In web-service terms, they are the "operations."
- The third argument is the temperature to be converted.

Even though Listing One is approximately 50 lines of code, if you ignore the code required to validate the command-line arguments and format the output, you are left with less than 10 lines of code that perform the following actions:

- Create a *Service* object, which is Axis's Dynamic Invocation Implementation of the javax.xml.rpc.Service interface that is used to access a web service.
- Create an unconfigured *Call* object. When configured, this object lets you invoke a web service.
- Configure the *Call* object by adding an endpoint, operation, parameters, and return type.
- Invoke the web service.

You can run this form of the client to access the web service at tremblett.ca by typing:

```
java ca.tremblett.ddj.Temperature-
ConversionClient tremblett.ca f2c 98.6
```

The program responds with:

```
98.6 degrees Fahrenheit =
37.0 degrees Celsius.
```

Now, prove that the web service you developed in three steps works by typing:

```
java ca.tremblett.ddj.Temperature-
ConversionClient localhost:8080 f2c 98.6
```

Using TCPMon as a Debugging Tool

Sometimes software doesn't behave as anticipated on the first attempt; hence, debugging. In the case of web services, where half of the software is remotely located, sometimes the best you can do is to make sure that the SOAP messages your client is sending/receiving are correct. The best place to capture the SOAP message is on the wire between your application and the remote application. The Axis TCP Monitor (tcpmon) lets you do this (http://ws.apache.org/axis/java/apiDocs/org/apache/axis/utils/tcpmon.html). You start tcpmon by typing:

```
java org.apache.axis.utils.tcpmon
8888 targethost 8080
```

The command-line arguments instruct *tcpmon* to listen on port 8888 and forward all requests to *targethost* on port 8080. In other words, *tcpmon* acts as a proxy that displays the traffic it relays. Now, run TemperatureConversionClient:

```
java ca.tremblett.ddj.Temperature-
ConversionClient localhost:8888 f2c 98.6
```

The output is identical to the output produced when you ran it the first time. This time, however, the SOAP messages that are sent-to/received-from tremblett.ca are displayed as in Figure 3.

Can you see why it's best to pass the host and port from the command line? If it was hard coded, you have to modify the source code to debug it.

Simplifying the Client

The client works fine, but isn't written like a typical web-service client. In the absence of knowing what operations are available, you must rely on the WSDL that the service makes available, then generate Java code based upon the information it contains. The process of reading and parsing the XML in a WSDL file and generating Java source code derived from it is well defined and lends itself to automation. The tool Apache Axis uses for this automation is WSDL2Java. In addition to showing how WSDL2Java can generate code that supports a simplified client, I demonstrate interoperability by choosing a web service that was written by someone else.

The web site http://random.org/, operated by the Distributed Systems Group, Department of Computer Science, University of Dublin, Trinity College in Ireland, offers true random numbers to anyone on the Internet. Unlike random numbers generated by a pseudorandom-number generator, these random numbers are suitable for cryptographic use. One way this web site provides access to the random-number generator is via SOAP. This means you can write a Java client that invokes the service.

Start by simply passing the URL at which the WSDL is located to WSDL2Java:

```
java org.apache.axis.wsdl.WSDL2Java
http://random.org/RandomDotOrg.wsdl
```

WSDL2Java opens a connection to the specified URL, reads the XML it finds there, parses it, and creates the directory org/random/www/RandomDotOrg_wsdl. This directory, whose name is derived from the *targetNamespace* in the WSDL, contains four Java source files (available electronically; see "Resource Center," page 5):

- RandomDotOrg.java
- RandomDotOrgBindingStub.java
- RandomDotOrgLocator.java
- RandomDotOrgPortType.java

RandomDotOrgLocator.java extends org.apache.axis.Client.Service, which is Axis's JAXRPC Dynamic Invocation Interface implementation of the javax.xml.rpc.Service interface. A *Service* object acts as a factory for the following:

- Dynamic proxy for the target service endpoint.
- Instance of the type *javax.xml.rpc.Call* for the dynamic invocation of a remote operation on the target service endpoint.
- Instance of a generated stub class.

When you invoke the *getRandomDotOrgPort()* method of the *Server* class, the object that is returned is a *RandomDotOrgBindingStub*, which implements the methods defined by the RandomDotOrgPortType interface. As an applications programmer, you simply invoke the *lrand()* and *mrand()* methods defined by this interface in the same manner as you would invoke any Java method. The binding stub implementation of these methods creates a *Call* object and configures it by adding an endpoint, operation, parameters, and return type. It then uses the configured *Call* object to invoke the remote service. In other words, it does everything you did in the first client you wrote to validate the *TemperatureConversion* service, but shields you from the details. You can see that RandomClient (Listing Two) is cleaner and simpler.

When you run the client by typing java random.org.www.RandomDotOrg.RandomDotOrg_wsdl.RandomClient, it displays two random numbers generated by the web service.

If you are inclined to experiment, you can find listings of other public web services at http://xmethods.com/ and http://webservicex.net/.

Legacy Applications as Web Services

A considerable amount of Java code was developed before web services emerged. Any such legacy code can be accessed as a web service even if you do not have the source. Axis provides the tools to develop wrappers that enable the public methods in any Java class to be exposed as web services.

For instance, some time ago, I developed the *CanadaInfo* class (also available electronically), which provides information about each of the Canadian provinces and territories. To modernize this code by making it available as a web service, I start by using another Axis utility, Java2WSDL, which generates WSDL from a Java class file. Run it by typing:

```
java org.apache.axis.wsdl.Java2WSDL
-o CanadaInfo.wsdl
-l"http://localhost:8080/axis/services/
canadaInfo"
-n urn:canadaInfo
-p"CanadaInfo" urn:canadaInfo
ca.tremblett.ddj.CanadaInfo
```

The command-line options are: *-o*, the name of the WSDL file to generate; *-l*, the service location URL; *-n*, the target namespace; and *-p*, the package to namespace mapping. Run WSDL2Java using the WSDL file I just generated as input:

```
java org.apache.axis.wsdl.WSDL2Java
-o .
-d Session
-s
-p ca.tremblett.ddj.ws CanadaInfo.wsdl
```

The command-line options in this case are: *-o*, the output directory for the emitted files; *-d*, the deploy scope (Application, Request, Session); *-s*, emit server-side bindings; and *-p*, override all namespace to package mappings. Use the specified package name instead.

The program creates four files similar to those in the random-number example. I modify CanadaInfoSoapBindingImpl.java to tie it to the existing *CanadaInfo* class. Listings Three and Four are the original and modified versions, respectively. To compile everything, type:

```
javac ca/tremblett/ddj/ws/*.java.
```

Next, package it all into a JAR file by typing:

```
jar -cvf CanadaInfo.jar ca/tremblett/ddj/*.class ca/tremblett/ddj/ws/*.class
```

Now, move the JAR file into the library directory of the Axis web app:

```
mv CanadaInfo.jar /library/Tomcat/webapps/axis/WEB-INF/lib
```

Finally, use yet another Axis utility, AdminClient, to deploy the service:

```
java org.apache.axis.client.AdminClient ca/tremblett/ddj/ws/deploy.wsdd.
```

The file deploy.wsdd is generated by WSDL2Java.

After the service has been deployed, point your browser at http://localhost:8080/axis/ and click on the link labeled "View the list of deployed web services." The browser then looks like Figure 4. CanadaInfoClient (Listing Five) tests the service.

Conclusion

Developing web services by manually writing SOAP messages and WSDL is a tedious and error-prone process. APIs that help you construct SOAP messages by assembling all of their components are useful but still require lots of coding. Apache Axis delivers an environment that makes both of these approaches unnecessary.

DDJ

Listing One

```
package ca.tremblett.ddj;

import org.apache.axis.client.Call;
```

```
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.rpc.ParameterMode;
public class TemperatureConversionClient {
  public static void main(String [] args) throws Exception {
    if (args.length != 3) {
      System.err.println("Usage: " +
        "java TemperatureConversionClient url operation temp");
        System.exit(1);
    }

    if ((!"f2c".equals(args[1])) && (!"c2f".equals(args[1]))) {
      System.err.println(args[1] + " is not a valid operation");
      System.exit(1);
    }
    Double temp = null;
    try {
      temp = new Double(args[2]);
    }
    catch (NumberFormatException e) {
      System.err.println(args[2] + " is not a valid temperature");
      System.exit(1);
    }
    String endpoint = "http://" + args[0] +
      "/axis/TemperatureConversion.jws";
    Service service = new Service();
    Call call = (Call) service.createCall();
    call.setTargetEndpointAddress( new java.net.URL(endpoint) );
    call.setOperationName( args[1] );
    call.addParameter( "temp", XMLType.XSD_DOUBLE, ParameterMode.IN );
    call.setReturnType( XMLType.XSD_DOUBLE );
    System.out.println(temp.toString() + " degrees " +
      (("f2c".equals(args[1])) ? "Farenheit" : "Celsius") +
      " = " + (Double) call.invoke( new Object [] { temp }) +
      " degrees " + (("f2c".equals(args[1])) ?
      "Celsius" : "Farenheit"));
  }
}
```

Back to article

Listing Two

```
package org.random.www.RandomDotOrg_wsdl;

public class RandomClient {
  public static void main(String[] args) {
    org.random.www.RandomDotOrg_wsdl.RandomDotOrgLocator service =
      new org.random.www.RandomDotOrg_wsdl.RandomDotOrgLocator();
    try {
    org.random.www.RandomDotOrg_wsdl.RandomDotOrgPortType port =
      service.getRandomDotOrgPort();
    System.out.println("lrand48 returned " +
      port.lrand48());
    System.out.println("mrand48 returned " +
      port.mrand48());
    }
    catch (Exception e) {
        System.err.println("web service failed");
    }
```

```
  }
}
```

Back to article

Listing Three

```
/** CanadaInfoSoapBindingImpl.java
 * This file was auto-generated from WSDL by Apache Axis WSDL2Java emitter.
 */

package ca.tremblett.ddj.ws;

public class CanadaInfoSoapBindingImpl
    implements ca.tremblett.ddj.ws.CanadaInfo{

  public java.lang.String[] provinces()
      throws java.rmi.RemoteException {
    return null;
  }
  public java.lang.String[] provincesAndTerritories()
      throws java.rmi.RemoteException {
    return null;
  }
  public java.lang.String[] territories()
      throws java.rmi.RemoteException {
    return null;
  }
  public java.lang.String capital(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    return null;
  }
  public int population(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    return -3;
  }
  public java.lang.String premier(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    return null;
  }
  public int area(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    return -3;
  }
}
```

Back to article

Listing Four

```
/** CanadaInfoSoapBindingImpl.java
 * This file was auto-generated from WSDL by Apache Axis WSDL2Java emitter.
 */

package ca.tremblett.ddj.ws;

import ca.tremblett.ddj.CanadaInfo;

public class CanadaInfoSoapBindingImpl
      implements ca.tremblett.ddj.ws.CanadaInfo{
```

```
  public CanadaInfo ci = null;
  public CanadaInfoSoapBindingImpl() throws java.lang.Exception {
    ci = new CanadaInfo();
  }
  public java.lang.String[] provinces()
      throws java.rmi.RemoteException {
    return ci.provinces();
  }
  public java.lang.String[] provincesAndTerritories()
      throws java.rmi.RemoteException {
    return ci.provincesAndTerritories();
  }
  public java.lang.String[] territories()
      throws java.rmi.RemoteException {
    return ci.territories();
  }
  public java.lang.String capital(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    try {
      return ci.capital(provinceOrTerritory);
    }
    catch (java.lang.Exception e) {
      throw new java.rmi.RemoteException();
    }
  }
  public int population(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    try {
      return ci.population(provinceOrTerritory);
    }
    catch (java.lang.Exception e) {
      throw new java.rmi.RemoteException();
    }
  }
  public java.lang.String premier(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    try {
      return ci.premier(provinceOrTerritory);
    }
    catch (java.lang.Exception e) {
      throw new java.rmi.RemoteException();
    }
  }
  public int area(java.lang.String provinceOrTerritory)
      throws java.rmi.RemoteException {
    try {
      return ci.area(provinceOrTerritory);
    }
    catch (java.lang.Exception e) {
      throw new java.rmi.RemoteException();
    }
  }
}
```

Back to article

Listing Five

```
package ca.tremblett.ddj;

public class CanadaInfoClient {
  public static void main(String [] args) throws Exception {
```

```
    try {
      ca.tremblett.ddj.ws.CanadaInfoService service =
        new ca.tremblett.ddj.ws.CanadaInfoServiceLocator();
      ca.tremblett.ddj.ws.CanadaInfo ci = service.getcanadaInfo();
      System.out.println("Capital of Newfoundland is " + ci.capital("NL"));
    }
    catch (Exception e) {
      System.err.println("Web Service failed");
    }
  }
}
```