Sponsored by:

JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-06-2001/jw-0601-soap.html

Clean up your wire protocol with SOAP, Part 3

Create SOAP services in Apache SOAP with JavaScript

By Tarak Modi, JavaWorld.com, 06/01/01

So you're a Java developer trying to stay on the cutting edge? With all the uncertainty in the software industry, you want to know what the future of the Web holds and, more importantly, how you can leverage your years of Java expertise to its maximum. In searching for these answers, you won't have to look too far before you stumble upon SOAP.

SOAP (Simple Object Access Protocol) -- a wire protocol that uses XML for data encoding -- takes Java's platform independence and portability to a higher level of interoperability. In Part 2 of this four-part series on SOAP I introduced you to Apache SOAP, an implementation of the SOAP specification that easily creates SOAP-based applications. Using that implementation, you built two simple HelloWorld services and clients that invoked the `sayHelloTo()` method on those services. You learned that though creating SOAP services is quite simple, the client must complete most of the work. It must set up the `Call` object, call the `invoke()` method on it, and then interpret the `Response` object that returns as a result of the invocation.

Creating SOAP services using Apache SOAP can be even easier. The only prerequisite: you must know one of the prescribed scripting languages. Luckily for me -- and most other Java developers -- one of those languages is JavaScript. That's right, you can create SOAP services in Apache SOAP using JavaScript. In this article I'll show you how.

Read the whole series on SOAP:

- Part 1: An introduction to SOAP basics
- Part 2: Use Apache SOAP to create SOAP-based applications
- Part 3: Create SOAP services in Apache SOAP with JavaScript
- Part 4: Dynamic proxies make Apache SOAP client development easy

Rebuild Apache SOAP

Apache SOAP's scripting support is built on top of the Bean Scripting Framework (BSF). Originally developed by IBM and now published as an open source project, BSF lets Java programs run scripts written in other languages and allows other scripting languages to use existing Java classes. Apache SOAP uses the former capability of BSF. The standard binaries that you download from Apache do not support scripting. More specifically, `soap.jar` does not contain the class `org.apache.soap.server.InvokeBSF`, which is the coupling/interface point between Apache SOAP and BSF. Knowing that not everyone using SOAP would have the required BSF jar and a scripting engine installed on their machine, the developers of Apache SOAP left out the scripting support in `soap.jar`. Since you are interested in scripting your SOAP service, you must rebuild the source to include the `InvokeBSF` class.

First you must download the source (`soap-src-2.0.zip`) from http://xml.apache.org/dist/soap/. Now unzip that file into the same directory where you installed Apache SOAP. In my case that is the root directory of my E drive. You should end up with an `src` subdirectory under the `soap_2-0` directory that contains all the source code for Apache SOAP. Before you can rebuild that source, you must download the required BSF jar. You will find one at ftp://ftp.mozilla.org/pub/js/. Use it in conjunction with Mozilla's JavaScript engine Rhino; Rhino is available in a zip file at http://www.mozilla.org/rhino/download.html. I unzipped the file into the root directory of my E drive, which resulted in the creation of an `E:\rhino` subdirectory that contains Rhino. You are interested in `js.jar` from this installation.

Next you need the tool that will actually perform the rebuild: Ant, another software project by Apache. A Java-based tool, Ant is actually part of the same project -- called Jakarta -- that created the Web server Tomcat. Ant is unique in that all the build information such as build targets, dependencies, and so forth are specified as XML configuration files. In addition, Ant is extensible. See Resources for articles on harnessing the power of Ant. You also can download Ant from Resources and unzip the file (which I've placed in the root directory of my C drive).

Now execute the following commands from your Apache SOAP installation directory:

```
set
CLASSPATH=E:\jakarta-tomcat\lib\servlet.jar;E:\xerces-1_2_0\xerces.jar;E:\s
oap-2_0\lib\bsf.jar
C:\build\bin\ant
```

Since the above code doesn't specify an XML configuration file, the Ant batch file will look for a file called `build.xml` in the current directory -- in my case `E:\soap_2-0`. Apache SOAP provides that file for you. Open the file and you will see that the `InvokeBSF` class only compiles if the class `com.ibm.bsf.BSFManager` is in the classpath. That is why I put `bsf.jar`, which contains the `BSFManager` class, in the classpath. Copy the newly built `soap.jar` file from the `build\lib` subdirectory into the `lib` subdirectory (I recommend renaming and saving the original `soap.jar` file). Finally, add `bsf.jar` and `js.jar` to your Web server's classpath.

That's it! You are now ready to start scripting SOAP services.

JavaScripting the HelloWorld application

Let's rewrite Part 2's HelloWorld service in the form of JavaScript. The complete service is shown below:

```
function sayHelloTo(name)
{
      return "Hello " + name + ", How are you?";
}
```

Could it get any easier? However, don't let the simplicity and ease fool you. You can actually perform pretty powerful tasks in the service. For example, you can access any of the standard Java classes from your script. Check out the code below for a modified version of the script that prints the server's date:

```
function sayHelloTo(name)
{
      var today = new java.util.Date();
      java.lang.System.out.println("Today is " +
today.toString());
      return "Hello " + name + ", How are you?";
}
```

You can also retrieve any of your own Java classes by importing the class you want to access. For example, let's modify the script to use the name `JavaBean`:

```
importClass(Packages.hello.Name);
function sayHelloTo(name)
{
      var today = new java.util.Date();
      java.lang.System.out.println("Today is " +
today.toString());
      var beanName = new Name();
      beanName.setName("John");

java.lang.System.out.println(beanName.getName());
      return "Hello " + name + ", How are you?";
}
```

Deploy the service

Before you can use the service's JavaScript version, you need to deploy it. As you learned from Part 2, there are two ways to deploy a service in Apache SOAP: use the Web-based UI or use the command line utility. Let's take a look at both approaches.

Use the Web-based administration tool

To use the administration tool, point your browser to http://localhost:8080/apache-soap/admin. Now click on the Deploy button in the left frame. Remember, you use the ID field to set the object ID; the SOAP infrastructure uses the object ID to tie an RPC (remote procedure call) request to a SOAP service. All Apache SOAP services must have an object ID that is unique among all the deployed services for that server. Set the ID to *urn:Hello*, which is the same ID that you used for your services in Part 2.

Set the Scope field to *application*. To recap, the Scope field defines the lifetime of the service instance serving the invocation request (refer to Part 2 for more information).

You need to set the Methods field to a white-space-delimited list of method names that invoke on the service being deployed. Our example service only supports one method, `sayHelloTo()`. Since you will use JavaScript instead of Java (as you did in Part 2) to implement the service, you'll set the Provider Type field to *script*. As such, you won't need to provide values for the Java Provider fields -- that is, the Provider Class and Static fields. But you will need to set values for the Script Provider fields. Select *JavaScript (Rhino)* as the Script Language. Because you'll provide the actual script itself in the Script textbox, keep the Script Filename blank. In the Script textbox, copy the following:

```
importClass(Packages.hello.Name);
function sayHelloTo(name)
{
      var today = new java.util.Date();
      java.lang.System.out.println("Today is " +
today.toString());
      var beanName = new Name();
      beanName.setName("John");

java.lang.System.out.println(beanName.getName());
      return "Hello " + name + ", How are you?";
}
```

Now scroll down to the bottom of the browser window and click the Deploy button just below the form (not the Deploy button in the left frame). To verify that the service deployed, click the List button in the left frame. The urn:Hello service should show up as one of the services in the list. Click on it and check that all the information matches what you just entered.

Use the command line utility

To deploy a service from the command line, all the deployment information must be placed in an XML deployment descriptor file. Below you will find the XML deployment descriptor file that I use to deploy the service:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
id="urn:Hello">
  <isd:provider type="script" scope="Application" methods="sayHelloTo">
    <isd:script language="javascript">
      importClass(Packages.hello.Name);
      function sayHelloTo(name)
      {
            var today = new java.util.Date();
            java.lang.System.out.println("Today is "
+ today.toString());
            var beanName = new Name();
            beanName.setName("John");

java.lang.System.out.println(beanName.getName());
            return "Hello " + name + ", How are
you?";
      }
    </isd:script>
  </isd:provider>
```

```
</isd:service>
```

The major difference here (from what was used in Part 2) is that you've set the provider type to `script` instead of `java`. As a result, instead of specifying a Java class, you specify the script to use.

Make sure you start your Web server before you try to deploy the service. The code below shows how to deploy the service:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/apache-soap/servlet/rpcrouter deploy
DeploymentDescriptor.xml
```

`DeploymentDescriptor.xml` is the name of the file that contains the deployment XML shown above. To verify that the server has been successfully deployed, try the following:

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/apache-soap/servlet/rpcrouter query urn:Hello
```

You should receive the same XML contained in `DeploymentDescriptor.xml`.

Test it out

You will use the same client program as you did in Part 2 (`Client.java`) to test the HelloWorld service. You can use the same client program because the client is completely oblivious to what language the server is written in. As long as the server understands a SOAP request and can return a SOAP response, the client does not care how the service is implemented. As a refresher, here's the batch file I use to run `hello.Client`:

```
set
CLASSPATH=E:\soap-2_0\samples\;E:\soap-2_0\lib\soap.jar;E:\xerces-1_2_0\xer
ces.jar
java hello.Client Tarak
```

Observe the console window of your Web server and you will see the current date and the output `John` each time you run the client.

Until next time

In this article I introduced you to the support that Apache's SOAP implementation provides for scripting languages. Why is this so significant? Try to analyze why Web development is so popular. In my opinion, a key reason is that Web development has matured to the point that almost anyone can create sophisticated Webpages using simple scripting languages such as HTML and JavaScript. Similarly, on the server side of Web development, one can use JavaServer Pages, another easy-to-learn, yet powerful scripting language. I contend that the same logic applies to SOAP development as well. If SOAP is to ever hit the mainstream and gain critical mass, it must be simplified. And that is exactly what Apache SOAP has done by adding scripting support; Apache has significantly broadened the range of developers that can create SOAP-based services.

But remember there is another audience that needs to be considered as well: the client developers, i.e., the developers who invoke the SOAP-based services. As I've mentioned before, client developers in Apache SOAP end up with a comparatively raw deal since they must perform extra work that they normally would not have to do. So in the next and final part of this series, I will present you with a framework based on the newly introduced dynamic proxy classes in version 1.3 of the Java 2 Platform that will make creating clients just as easy and intuitive as creating SOAP services.

About the author

A certified Java programmer, Tarak Modi is an architect at North Highland, which specializes in management and technology consulting. He has a master's degree in computer engineering and an MBA concentrating in information systems. He has several

years of experience working with C++ and Java and technologies such as DCOM, CORBA, RMI, and EJB. He has written articles for leading software magazines including JavaWorld and is currently working on a book on Java Message Service with Manning Publications. To find out more about Tarak, his articles, and upcoming book, please visit his Website at http://tmodi.home.att.net/.

All contents copyright 1995-2010 Java World, Inc. http://www.javaworld.com