PINNACLE

XML Developer

XML Solutions for Client/Server, Web-Based, and Business-to-Business Systems

Connecting Web Services: IBM and Java to VBScript and Microsoft

Zoran Zaev

DOWNLOAD

This month, Zoran Zaev builds a Web Service in Java using IBM Web Services Toolkit, and then calls it from VBScript. This article shows the kinds of problems that you can run into as you use VBScript to call Java-based services.

IN my article "Connecting Web Services" in the October 2001 issue of *XML Developer*, I showed how to create a Web Service with Visual Basic and call it using three scripting languages: JScript, VBScript, and Perl. This month, I'll be using the IBM Web Services Toolkit, but I'll call the Service from Microsoft's VBScript. Keep in mind that *implementing* Web Services in different languages (and platforms) isn't difficult, as long as you have some familiarity with the particular language and the platform, and some knowledge of XML and SOAP. However, *connecting* different systems can be a challenge. I'll look into some of these interoperability challenges, particularly as they relate to using VBScript to call a Java-based Web Service.

Web Services offer a standard way of connecting systems based on different environments, across operating systems, languages, or object models. This doesn't mean that you should never use Web Services to connect systems

Continues on page 3

December 2001

Volume 2, Number 12

DOWNLOAD

Accompanying files available online at http://www.xmldevelopernewsletter.com

Connecting Web Services...

Continued from page 1

based on the same programming language or object model. Similar systems can benefit from the inherent loose coupling of the Web Services model. Using the same tools everywhere can be a good long-term decision, as it can make your system more easily interoperable in the future.

In this article, I'll create the same simple Web Service that I created in my October article. This time, I'll use:

- Java as my programming language
- IBM Web Services Toolkit 2.4 (IBM WSTK) as my development platform
- IBM Web Sphere (based on Apache's Web server) as my Web server

That's the server side of the problem. On the client side, I'll connect to this Web Service using VBScript implemented in the Windows Scripting Host (WSH) client. In future articles, I'll show how to connect using a Java client.

All of the sample code for this article is in the Source Code file at www.xmldevelopernewsletter.com. I'll begin by showing how to set up the various tools to run the sample applications (or build your own).

Setting up the JDK and Toolkit

The current version of the IBM Web Services Toolkit (IBM WSTK) is version 2.4 and contains several components. One component is the Apache SOAP Toolkit 2.2, which is also available as an individual download from the Apache organization (www.apache.org). The IBM Web Services Toolkit comes in Windows 2000 and Linux versions. On the Java language side, it requires Java 2 (for instance, JDK 1.3 or higher) from either IBM or Sun. Some of the Toolkit's utilities require either Internet Explorer 5 and above, or Netscape 4.5 and above. In terms of the Toolkit's production Web server support, the Toolkit can use any of the IBM Embedded WebSphere (which is included in the Toolkit), the IBM WebSphere Application Server 3.5/4.0, or Apache Tomcat (available free of charge from www.apache.org).

For development purposes, you can use the IBM Embedded WebSphere 3.5 Web server, which is a scaled-down version of the real production version server (it's really to be used only with the Toolkit). For production-level hosting of your service, you should use the IBM WebSphere Application Server or Apache Tomcat Web server. Finally, the IBM WSTK contains some other packages, including:

- The WSDL (Web Services Description Language) package for Java (WSDL4J 0.8)
- Apache Xerces 1.4.0, the XML processor for Java that provides DOM level 2 and SAX 2 functionality for the Java programming environment (XML4J 3.2.0)
- Lotus XSL–Java 2.2 that contains Apache Xalan 2.2, the XSLT processor for Java
- UDDI support libraries for Java (IBM UDDI4J 1.0.3)

The installation is fairly easy, although I have to say that it takes longer than the Microsoft SOAP Toolkit. At the beginning, you'll want to make sure that you have the Java SDK 2 (JDK 1.3 or above). If you don't have it, you can download it from IBM at www-106.ibm.com/developerworks/java/jdk/index.html or from Sun at http://java.sun.com/products/?frontpage-main. I'll be using Sun's version of Java SDK, the JDK 1.3.1 (http://java.sun.com/j2se/1.3/). I'll set up this sample application on a Windows 2000 machine, but you can use other operating systems (Linux, for example). Once you download the JDK, simply run the downloaded file and follow the instructions displayed.

In terms of post-installation configuration, you may want to set the PATH variable of your system to the path of the bin file for the JDK (for example, "C:\jdk1.3.1_01\bin") in order to make it easier on yourself when invoking the Java 2 SDK executables (for instance, javac.exe, java.exe). For more information on the Windows installation procedures for the JDK, go to http://java.sun.com/j2se/1.3/install-windows.html.

You'll also want to set the CLASSPATH environment variable in order to make it easy to compile and run Java

classes (the CLASSPATH setting tells Java where to look for user classes—for more information on setting the CLASSPATH, see http://java.sun.com/j2se/1.3/docs/tooldocs/win32/classpath.html). If you're installing the JDK on Linux, you can find installation help at http://java.sun.com/j2se/1.3/install-linux-sdk.html, and for SOLARIS installation, see http://java.sun.com/j2se/1.3/install-solaris.html. Not having the proper CLASSPATH can result in many hours of troubleshooting, so keep a close eye on this configuration step.

The CLASSPATH on my system is the following:

```
.;c:\wstk-2.4\soap;c:\wstk-2.4\soap\lib\soap.jar;
c:\wstk-2.4\lib\xerces.jar;c:\wstk-2.4\soap\lib\
activation.jar;c:\wstk-2.4\soap\lib\mail.jar;
c:\wstk-2.4\lib\bsf.jar;c:\wstk-2.4\lib\wstk.jar
```

The very first entry is "." (a period), which represents the current directory. Most of the other paths point to locations within my SOAP or IBM's WSTK installation directory. Not all of these settings are necessary for running the examples in this article, but it would be convenient for you to compare these settings with your own, in case you run into some errors that may be related to CLASSPATH settings.

Now that you've set up the Java SDK, you can continue with the installation of the IBM WSTK. If you don't have the Toolkit, you can download it from www.alphaworks.ibm.com/tech/webservicestoolkit. You can start by running the installation program and following the configuration instructions. The installation program will set the environment variable JAVA_HOME and point it to the location of your JDK. When this screen appears, make sure that you have the correct path. The IBM WSTK will ask you whether you want a "Typical," "Custom," or "Full" installation. "Typical" installation will be sufficient to run the demo, but you're welcome to experiment with the other options if you like. Once the installation is finished, a configuration screen will be displayed, allowing you to configure the Web server and the UDDI Registry that you'd like to use. For the sample application, keep the defaults: "Embedded WebSphere" at the address of "localhost:8080" (8080 is the port number, which in this case is different from the typical HTTP port number 80, so 8080 must be added to the URL). Also, you can keep the default IBM UDDI Test Registry settings, except that you'll have to enter a username and password (you can enter any username and password at this time because I won't use UDDI this month). If you have another Web server (for instance, Apache), you can experiment with using the other settings on this configuration screen.

You can run the Embedded WebSphere Web server by going into the "bin" directory of your IBM WSTK (for example, C:\wstk-2.4\bin) and typing "WebSphere start" at the command prompt. You can verify that the Web server is running by going to your Web browser and typing URL http://localhost:8080/soap/index.html. If you want to use another Web server (for instance, Apache Tomcat), you can follow the Help documents included with the IBM WSTK for assistance with the configuration.

The server-side code

As in my October article, the sample Web Service is a job submittal service, part of a sample Job Bank system. In this scenario, business partners can submit new job postings to the Web Service. Of course, this Job Bank is a sample application, and in a real-life scenario, you'll have more complex Web Services (and likely more than one Web Service). Additionally, security will be applied to the Service so that only business processes can submit a new job posting.

Now, I'll create the Java class file that I'll expose in this Web Service. This is just like creating any other Java class file and requires no special references to the IBM WSTK:

```
import java.util.*;

public class wsJobBank {

  public static void main (String[] args) {
    wsJobBank oTestWS = new wsJobBank();
    String msg = oTestWS.JobAdd(1,"1/1/2000",
      "Project Manager","Desc...",
      (double)70000.0);
    System.out.println("TEST: The following " +
      "message was returned by the Web service");
    System.out.println("  " + msg);
  }
```

In this first section of the code, I created the wsJobBank class and placed some test code within the main method. I can run this Java file from the command line and have this main method test the JobAdd method that follows.

This is the code for the JobAdd method that will be exposed as the Web Service:

```
  public String JobAdd (
    int companyID,
    String expireDate,
    String jobTitle,
    String jobDesc,
    double salaryAmount) {
    if (companyID == 1) {
      return "Your job with title: " +
        jobTitle + " and description: " +
        jobDesc + " was received " +
        "successfully on " +
        new Date().toString() +
        ". The salary requested is: " +
        salaryAmount + " and the expiration" +
        "date of this posting is: " +
        expireDate + ".";
    } else {
      return "Your company is not allowed" +
        " to post jobs to our Job Bank.";
    }
  }
}
```

Once the code is entered into the wsJobBank.java file,

it can be compiled with:

```
javac wsJobBank.java
```

You can run the compiled code with:

```
java wsJobBank
```

This procedure will execute the main method within the wsJobBank class, which, in return, will call the JobAdd method, providing it with some test values and displaying the result.

As you can see, there's no mention of Web Services anywhere in this code. That's because the classes and methods that get exposed via Web Services don't require any special support for Web Services within them. This means that you can expose any existing Java class that you have as a Web Service, a great feature.

After preparing my code, I moved the Java source file and compiled class into my SOAP directory (for example, the C:\wstk-2.4\soap directory). You can try another location if you like, just keep your CLASSPATH settings in mind.

Using the WSDL utility

Once the server side of the Java-based Web Service has been created, compiled, and successfully tested, you can run the WSDL utility. This utility will help you create:

- The WSDL files for this Web Service
- The additional deployment descriptor document needed for the deployment of this Web Service

The alternative to using the WSDL utility is to create the WSDL files and deployment descriptor file by hand, which is much more time-consuming (and a method I won't cover in this article).

You can start the WSDL wizard by executing the wsdlgen.bat file found within the bin folder of the IBM WSTK installation directory. The first screen will ask you to specify the type of entity that you want to expose as a Web Service. The options are Java Class, EJB (Enterprise JavaBeans) JAR file, or a COM dispatch interface (found on the Windows platform). You'll need to select Java Class in order to expose the Java Class JobBank that you just created. For the Class Name you must specify wsJobBank, and for Classpath use C:\wstk-2.4\soap (or whatever folder you've placed your Java Class file into). The Output Filename will default to wsJobBank_Service.wsdl. I've also modified the WSDL URL prefix in order to match the directory location where I'll put the WSDL files that will be created by the WSDL Utility. I put the WSDL files in a folder below the SOAP folder within the Web server space (for example, exposed onto the Web server). The specific location, in this case, is C:\wstk-2.4\soap\webapps\soap\wsdl\wsJobBank_Service.wsdl. You can see the rest of the settings in Figure 1.

Once you click the Next button, you'll be shown another screen with a list of the methods and operations found in the Java Class. At that point, you'll select the JobAdd method. Notice that some methods listed will have a colored dot within the status column. This shows that those methods may have complex data types that the WSDL generator may not be able to recognize. If you select an operation with a colored dot in the status column, you'll likely have to modify the description of the function in the WSDL file manually in order to support the operation. The JobAdd method from the wsJobBank Java Class doesn't have a complex method, and it won't have a colored dot in its status column. Once you've selected the operation that you want to expose via the Web Service, click the Next button and then the Finish button.

The WSDL utility will create three files:

- A WSDL file that includes the term "interface" in its name and contains the abstract port type definition of the Web Service
- A WSDL file that contains the concrete implementation information of the Web Service
- An XML file called DeploymentDescriptor.xml that can be used for linking the Web server listener and the Java Class

This leads to our first interoperability issue. The wsJobBank_Service.wsdl file imports the wsJobBank_Service-interface.wsdl file. However, Microsoft clients (like the WSH VBScript client that I'm using in this article) don't currently understand this directive. After running the WSDL utility, you'll need to modify the wsJobBank_Service.wsdl file. You'll have to

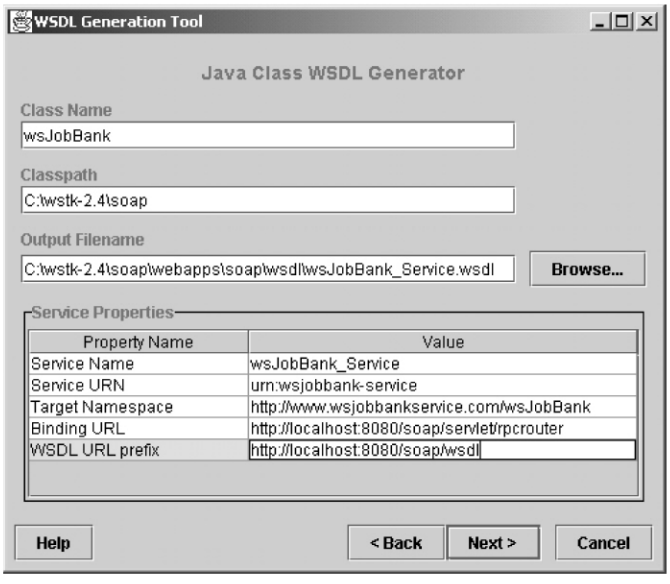

Figure 1. Java Class WSDL generator.

copy the entire content of the wsJobBank_Service-interface.wsdl file (everything from the <MESSAGE> tag to the </BINDING> tag) and paste it within the wsJobBank_Service.wsdl file where the directive appears.

If you want to trace the messages being passed between the client and the server using utilities like the Trace Utility included in the Microsoft SOAP Toolkit or the tcptunnel.bat utility provided with the IBM WSTK, you can modify the <SOAP:Address> element's location attribute to point to a different port number (8888, for instance). After doing that, set your tool to listen to this port to catch the messages and forward the request to the Web server port (typically 8080 for the IBM Embedded WebSphere or Apache Web servers).

Now, you can go to http://localhost:8080/soap/admin/ and deploy your Web Service. Click the Deploy button found on the left of the screen. You'll be presented with a Deploy Service form that you'll need to fill out. For the ID field, you can use urn:wsjobbank-service; set the scope to "Request." The method will be JobAdd, the provider "Java", and, under the Java Provider section, enter wsJobBank for the provider class. The rest of the form, in this situation, can be left at its default values.

As an alternative approach, you can deploy this Web Service using a command line interface and the "DeploymentDescriptor.xml" file:

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

You can use the "deploy.bat" file that I've included in the Source Code file.

Connecting the client

As I discussed in my October article, implementing a VB SOAP client in the Microsoft environment can be done using either:

- a high-level API that takes advantage of the information in the WSDL files; or
- a low-level API that doesn't use the WSDL files.

This leads to the second interoperability problem. Microsoft's high-level API creates SOAP messages without adding "xsi:type" attributes within the method call parameters. In other words, Microsoft's automatically generated SOAP messages don't specify the type of the parameters, instead letting the service rely on its WSDL file to specify data type.

Java Web Service servers, on the other hand, need to know the type of each of the parameters in the SOAP call. In the latest version of the Apache SOAP v.2.2, which is included with the IBM WSTK 2.4, there's some improvement in this area. However, there are still some outstanding issues that I'll cover more in upcoming articles (including handling dates, manually specifying deserializers on the Java side, using SOAPMappingRegistry on the Java client side, and so on). While many of these interoperability challenges will likely be addressed with upcoming versions of the Web Services Toolkit implementations, even at the current time there are workarounds that address these items.

One of the ways to handle this problem is to implement the client using the Microsoft low-level API. This approach will allow you to specify the type of each parameter manually. This method takes a little more code, and it bypasses using the WSDL files created during the development of the Web Service.

The VBScript code starts by printing a couple of messages on the screen, and then it sets the few important variables and objects that are used by the Microsoft low-level API:

```
Dim Connector
Dim Serializer
Dim Reader
Set Connector = _
   CreateObject ("MSSOAP.HttpConnector")
Set Serializer = _
   CreateObject ("MSSOAP.SoapSerializer")
Set Reader = CreateObject ("MSSOAP.SoapReader")

URL = _
   "http://localhost:8888/soap/servlet/rpcrouter"
ENC = "http://schemas.xmlsoap.org/soap/encoding/"
XSI = "http://www.w3.org/2001/XMLSchema-instance"
XSD = "http://www.w3.org/2001/XMLSchema"
URI = "urn:wsjobbank-service"
Method = "JobAdd"
wscript.echo chr(13) & chr(13) & chr(10) & _
   "SOAP Client Using Windows Scripting Host" & _
   " (WSH)" & chr(13) & chr(10)
wscript.echo "Reply from Java Web Server:" & _
   chr(13) & chr(10)
```

The Connector object handles the HTTP binding for the SOAP protocol. The Serializer object handles the manual creation of the SOAP message. You can see how I create the envelope first, and then the <BODY> tag with all of its sub-elements that contain the method call and all of the required parameters. The parameter types are specified in the XML:

```
Connector.Property("EndPointURL") = URL
Connector.Connect
Connector.Property("SoapAction") = URI & "#" & _
   Method
Connector.BeginMessage
Serializer.Init Connector.InputStream
Serializer.startEnvelope , ENC
Serializer.SoapNamespace "xsi", XSI
Serializer.SoapNamespace "SOAP-ENC", ENC
Serializer.SoapNamespace "xsd", XSD
Serializer.startBody
Serializer.startElement Method, URI, , "method"

Serializer.startElement "companyID"
Serializer.SoapAttribute "type",,"xsd:int","xsi"
Serializer.writeString "1"
Serializer.endElement
Serializer.startElement "expireDate"
Serializer.SoapAttribute "type",,"xsd:string",_
   "xsi"
Serializer.writeString "12/31/2001 11:15:00 AM"
Serializer.endElement
Serializer.startElement "jobTitle"
```

Continues on page 9

Connecting Web Services...

Continued from page 6

```
Serializer.SoapAttribute "type",,"xsd:string",_
   "xsi"
Serializer.writeString "XML Project Manager"
Serializer.endElement
Serializer.startElement "jobDesc"
Serializer.SoapAttribute "type",,"xsd:string",_
   "xsi"
Serializer.writeString _
   "Must have 10 years of technical "
Serializer.endElement
Serializer.startElement "salaryAmount"
Serializer.SoapAttribute "type",,"xsd:double",_
   "xsi"
Serializer.writeString "80000"
Serializer.endElement
```

Finally, I close the message body and envelope, and use the Reader object to read the SOAP message. This passes the SOAP message to the Service and receives either the result or a SOAP fault:

```
Serializer.endElement
Serializer.endBody
Serializer.endEnvelope
Connector.EndMessage
Reader.Load Connector.OutputStream
If Not Reader.Fault Is Nothing Then
    wscript.echo Reader.faultstring.Text
Else
    wscript.echo Reader.RPCResult.text
End If
```

I've put the VBScript code in the file JobAddVBSLow.vbs. You can run this file, from the command prompt, with:

```
cscript JobAddVBSLow.vbs
```

That's all for this month. In future articles, I'll continue with the coverage of Web Services in various environments. I'll return to Java, particularly to the design and setup of the Java-based Web Service client and the use of the WSDL within Java. I'll look at some other interoperability challenges and approaches to working around them. ▲

DOWNLOAD FILENAME.ZIP at www.xmldevelopernewsletter.com

Zoran Zaev works as the president and chief software architect for Xsynthesis, a software and technology services company in the Washington, D.C., area. He enjoys helping others realize the potential of technology, and when he isn't working, he spends considerable time writing articles such as this one and books (for example, he co-authored *Professional XML Web Services* and *Professional XML 2nd Edition* with Wrox Press). Zoran's research interests include complex systems that often involve XML, highly distributed architectures, and systems integration, as well as the application of these concepts in newer areas such as biotechnology. When not programming or thinking of exciting system architectures, you'll find Zoran traveling, reading, and exploring various learning opportunities. zzaev@yahoo.com.