Connecting Web Services: Perl and Java

Zoran Zaev

DOWNLOAD

In this article, Zoran Zaev continues his exploration of the complexities of connecting Web Services and their clients. This month, Zoran builds a service in Perl and connects to it with Java.

A couple of months ago, I showed you how to create a Web Service with Visual Basic and then call it using the scripting languages JScript, VBScript, and Perl. Last month, I created the same Web Service in Java, using the IBM Web Services Toolkit, and then I called it from VBScript. This month, I'll work on another combination of languages: I'll create the Web Service in Perl and then call it from Java. As I've said in the past, implementing Web Services in different languages (and platforms) isn't very difficult, as long as you have some familiarity with the particular language and the platform, as well some knowledge of XML and SOAP. However, making sure that the different systems can connect to each other can be a bit more challenging. I'll continue looking into some of these interoperability difficulties, particularly as it relates to Perl and Java-based Web Services.

Web Services offer a standard way of connecting systems based on different environments, whether operating systems, languages, or object models. As I had mentioned in the previous articles, this isn't to say that you should never use Web Services to connect systems based on the same programming language or object model. Similar systems can, at times, benefit from the inherent loose coupling of the Web Services model. Building an application as a Web Service even in homogenous systems can also be a good long-term decision in order to make your system more interoperable in the future.

In this article, I'll re-create the same simple Web Service that I'd created in the past issues (a job bank service, where job postings can be added by business partners).

Server setup of the Perl Web Service

I used ActiveState's ActivePerl 5.6 (see http://aspn.activestate.com/ASPN/Downloads/ActivePerl), but you should be able to use any version of Perl 5.004 or later. The SOAP implementation that I'll use is SOAP::Lite, a Perl implementation by Paul Kulchenko. You can find both Win32 and UNIX versions of SOAP::Lite at www.soaplite.com, as well as installation instructions, documentation, and so on.

Once you've downloaded Perl and the SOAP::Lite module, you'll need to configure your Web server to run Perl when contacted (this step will vary depending on the particular Web server that you're using—check your Perl documentation for specific instructions on getting Perl to work with your Web server). In my case, I used IIS 5 with Windows 2000 (both the Professional and Server versions of Windows 2000 ought to work fine). The installation procedure of ActivePerl provides an option that will register the Perl executable with IIS. If you don't check this, you'll have to manually add this mapping to IIS.

Setup of the JDK and the IBM Web Services Toolkit

This installation is fairly easy. At the beginning, you want to make sure that you have the Java SDK 2 (JDK 1.3 or higher). If you don't, 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 use Sun's version of Java SDK, the JDK 1.3.1 (http://java.sun.com/j2se/1.3), and I'll set up this sample application on a Windows 2000 machine. You can use Linux, if you wish. Once you download the JDK, simply run the downloaded file and follow the instructions provided to you.

In terms of post-installation configuration, you may want to set the PATH variable of your system to "C:\jdk1.3.1_01\bin" or whatever the JDK's path is on your own computer. This will make it easier when you invoke the Java 2 SDK executables such as javac.exe and 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 may 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 easily result in many hours of troubleshooting, so do 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 in the CLASSPATH 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 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 whether you'd like to conduct a Typical, Custom, or Full installation. The Typical installation will be sufficient for my examples, but you're welcome to experiment with the other options. Once the installation is finished, a configuration screen will be presented to you, allowing you to configure the Web server and the UDDI Registry that you'd like to use. You can keep the default Web server and UDDI configuration options, as I won't be using the Embedded WebSphere Web server. Instead, I'll run Perl within IIS.

Web Services server code

I created a virtual directory within IIS to point to the location where my Perl files are stored (in this case, my path location to the Perl files is "C:\Inetpub\wsJobBankPerl," and the virtual directory alias is "wsJobBank"). Under IIS, you'd want to enable "Script and Executable" access to this directory in order for Perl to work properly.

Next, I created the Perl Web Service server. This is quite easy to do, by simply exposing the addJob subroutine from the JobBank package or class. The subroutine checks the companyID and, depending on its value, returns a different value. Of course, in a real-life scenario, you'd likely have code within this subroutine that handles some meaningful tasks (such as access to a database) and only then return a confirmation to the calling Web Services. The following is the addJobSrv.pl file, containing the Perl Web Service:

```
use SOAP::Transport::HTTP;
binmode STDIN;

SOAP::Transport::HTTP::CGI
  -> dispatch_to('JobBank')
  -> on_action(sub{return})
  -> handle;

package JobBank;

sub addJob {
  my ($className,$companyID,$expireDate,
    $jobTitle,$jobDesc,$salaryAmount) = @_;

  if ($companyID == 1) {
    return "Your job with title: " .
      $jobTitle . " and description: " .
      $jobDesc . " was received " .
      "successfully on " . localtime(time) .
      ". 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.";
  }

}
```

This Web Service server implementation is a CGI-based SOAP server. You may be aware that applications that run as CGI scripts don't perform well. So one of the alternatives is to run your Web Service scripts through the Perl ISAPI for IIS/ Windows environments or using persistent technologies like mod_perl. A more detailed discussion of these approaches would be too long to cover in this article and not strictly related to Web Services.

So, how about creating a Perl client that would call this Web Service? It's quite easy to create a simple Perl Web Services client. Perl being a typeless language, I don't have to bother setting up the specific types for each of the parameters that I'll pass to the Web Service (although this is possible, as described in my last article where the Perl Web Services client invoked the Microsoft SOAP server). With Perl , you can avoid specifying the parameter types and names, even when you don't have a WSDL file available for your Web Service. On the other hand, the Java and Microsoft clients could only be made as short (in terms of lines of code) if they used WSDL files. Here's the start of my Perl client:

```
use SOAP::Lite;

print "HTTP/1.0 201 Ok \n";
print "Content-type:text/html\n\n";
print "<HTML><HEAD><TITLE>JobBank:".
  "Add Jobs Via PERL";
```

```
print "</TITLE></HEAD><BODY>The Response ".
  "from the Web Service is: ";
print "<BR><BR>";
```

This code simply prints a few lines that would create the HTTP response and the beginning of the HTML page. In the next step, the code initiates the SOAP call and places a reference to the application into the $soap variable. After that, I obtain the result from the SOAP call and place it into the $result variable. Finally, I print the result (or the error code) onto the screen:

```
my $soap = SOAP::Lite
  -> uri('http://localhost/JobBank')
  -> proxy(
  'http://localhost:8080/wsJobBank/addJobSrv.pl');

my $result = $soap->addJob
  (1,"12/31/2001 11:15:00 AM",
  "XML Project Manager",
  "Must have 10 years experience",80000);

unless ($result->fault) {
  print $result->result();
} else {
  print join ', ',
  $result->faultcode,
  $result->faultstring;
}

print "</BODY></HTML>";
```

The proxy value that I specify refers to port 8080. You'll need to change this value and the server name to match your system. In fact, here I'd used port 8080 in order to capture the SOAP traffic using the Microsoft Trace Utility (you can use other tracing utilities, if you like). When connecting different systems via Web Services and SOAP, tracing utilities are most helpful. Here's an example of a SOAP message sent by my Perl client, as it was captured by the trace utility. The first lines specify the SOAP envelope and the various namespaces needed for it. However, the SOAP body is the part that's even more interesting from the perspective of interoperability:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENC=
"http://schemas.xmlsoap.org/soap/encoding/"
SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi=
"http://www.w3.org/1999/XMLSchema-instance"
xmlns:SOAP-ENV=
"http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd=
"http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <namesp1:addJob xmlns:namesp1=
      "http://localhost/JobBank">
    <c-gensym5 xsi:type="xsd:int">1</c-gensym5>
    <c-gensym7 xsi:type="xsd:string">
      12/31/2001 11:15:00 AM</c-gensym7>
    <c-gensym9 xsi:type="xsd:string">
      XML Project Manager</c-gensym9>
    <c-gensym11 xsi:type="xsd:string">
      Must have 10 years experience</c-gensym11>
    <c-gensym13 xsi:type="xsd:int">
      80000</c-gensym13>
    </namesp1:addJob>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this example, Perl created the various parameters by providing auto-generated names for them. Further-more, the Perl client had tried its best to guess the parameter types for each of the parameters, even though I hadn't provided data types. Analyzing the SOAP message and its structure reveals quite a bit of infor-mation about what the client is sending to the server side of the Web Service. As a debugging technique, it's often very helpful to compare SOAP messages that the server can successfully process with SOAP messages that don't work as expected. This is often the route to resolving seemingly impossible interoperability situations.

Using a Java client to call the Perl server

Creating the Java Web Services client isn't very difficult, although it does require a bit more work due to the fact that I'm not using a WSDL file as part of creating the call to the Perl Web Service. SOAP::Lite doesn't include a WSDL generator, and if I wanted to use a WSDL file, I would have had to generate it manually. I decided not to do that in this case and created the parameters manually in the Java Web Service client. Here's the relevant Java code. The code starts with the typical Java import statements, referencing the libraries that I need in order to make this Web Services call work.

```
import java.util.*;
import java.net.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import org.apache.soap.transport.
  http.SOAPHTTPConnection;

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

Next, I defined the client class that will handle the call. Here, I've implemented the Java Web Services client as a Java application. I could have chosen to implement it in a JSP/Servlet scenario, if I wanted to do so. The following lines deal mostly with the setup of the SOAPHTTPConnection object and the Call object. Among the few important items specified here are:

- The target URI that points the Java client to the class that ought to be invoked on the Web Services server end.
- The Web Service URL (also called Web Service end-point). In your environment, you'll want to modify

Continues on page 15

Connecting Web Services...

Continued from page 8

this address to match the server name and port number of your Perl Web Services server.

- The method name that's to be called (in this case, addJob).

```
SOAPHTTPConnection oHTTPConn =
      new SOAPHTTPConnection();
Call oCall = new Call();
oCall.setSOAPTransport(oHTTPConn);
SOAPMappingRegistry oSOAPMap =
      new SOAPMappingRegistry ();
StringDeserializer oStrDeserial =
      new StringDeserializer ();
oSOAPMap.mapTypes (Constants.NS_URI_SOAP_ENC,
      new QName ("", "Result"),
      null, null, oStrDeserial);
oCall.setSOAPMappingRegistry (oSOAPMap);
oCall.setEncodingStyleURI ("http://" +
      "schemas.xmlsoap.org/soap/encoding/");
oCall.setTargetObjectURI("http://" +
      "localhost/JobBank");
String strServiceURL =
      new String ("http://" +
      "localhost:8080/wsJobBank/addJobSrv.pl");
oCall.setMethodName("addJob");
URL oURL = new URL(strServiceURL);
```

The next code section is where I specified the various parameters, by manually stating the name, type, value, and their encoding type. One method of working around parameter types and challenges with accurate conversion of the parameters from one system to another is to do manual data type conversions and then transfer the data as a string. SOAP calls are XML, and XML is plain text. Even though this isn't the perfect way of doing things, it works just fine, at least for the short run. It provides a short-term way of handling interoperability issues while some of the toolkit's interoperability issues get cleared up. So, in this code, I decided to convert the date type into a string manually and send it as a string across the wire. The other data types in this call (string, integer, and double) worked without problems:

```
Vector oParIn = new Vector();
oParIn.addElement(
      new Parameter("intCompanyID",Integer.class,
        new Integer (1),
        Constants.NS_URI_SOAP_ENC));
oParIn.addElement(
      new Parameter("strExpireDate",String.class,
        new String ("12/31/2001 11:15:00 AM"),
        Constants.NS_URI_SOAP_ENC));
oParIn.addElement(
      new Parameter("strJobTitle", String.class,
        new String ("Project Manager"),
        Constants.NS_URI_SOAP_ENC));
oParIn.addElement(
      new Parameter("strJobDesc", String.class,
        new String ("Desc..."),
        Constants.NS_URI_SOAP_ENC));
oParIn.addElement(
      new Parameter("intSalary", Double.class,
        new Double (80000.0),
        Constants.NS_URI_SOAP_ENC));
oCall.setParams(oParIn);

System.out.println("\n\nAdding Jobs Using " +
      "Java Web Services Client \n" +
      "Request Submitted to " +
      "PERL Web Services Server\n\tAt Address:" +
      " " + strServiceURL + "\n");
```

Finally, I created the response object and populated it by calling the invoke method of the Call object. I checked for a potential Fault response and printed some information on the screen:

```
Response oResp;
    try {
      oResp = oCall.invoke(oURL, strServiceURL);
    } catch (SOAPException e) {
      System.out.println("SOAPException ( " +
      e.getFaultCode() + "): " + e.getMessage());
      return;
    }

    if (!oResp.generatedFault()) {
      Parameter oParResult =
        oResp.getReturnValue();
      Object value = oParResult.getValue();
      System.out.println (value);
    } else {
      Fault oFault = oResp.getFault ();
      System.out.println ("JAVA Web Service"+
        "Error!\n" + "Fault Code= " +
        oFault.getFaultCode() + "\n" +
        "Fault String = " +
        oFault.getFaultString());
    }
  }
}
```

That's all for this month. At this point, you should be getting a sense that connecting various systems via Web Services isn't a daunting task (as it may have appeared originally). Just as with any other technology, you need to know some of the specific issues and difficulties to watch for. The rest is no different from what you're used to doing when programming with any kind of system or environment. I'll continue with the coverage of con-necting Web Services in various environments in upcoming issues with yet another combination of environments. ▲

DOWNLOAD ZAEV.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's not 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.