SYBASE®

# WDSL and PowerBuilder 9

A Sybase White Paper by Berndt Hamboeck

technology

WHITE PAPER

## Table of Contents

## Overview

In my previous article "Apache Axis - The Third Generation SOAP Implementation" we saw the basic steps for implementing a Web service using Axis, EAServer and PowerBuilder 8. We are now ready to explore technologies that make it easier to use Web services that have already been deployed. Specifically, this article focuses on the Web Service Description Language (WSDL), which makes it possible for automated code-generation tools, like PowerBuilder 9 with its WebServices wizard, to simplify building clients for existing web services.

## What is WSDL?

Web Services Description Language is an XML-based language used to define Web services and describe how to access them. Fortunately, we do not need to learn all the details, because there are tools and implementations like Apache Axis that generate WSDL for us. This article gives you just enough WSDL knowledge to understand what's going on and to be able to tweak tool-generated WSDL files and maybe troubleshoot WSDL-related problems.

Interoperability between applications on various operating system platforms and programming languages is most often hindered because one system's "integer" may not be exactly the same as another system's "integer." Because different operating systems and programming languages have different definitions not only of what particular base (or primitive) data types are called, but also how they are expressed when sent out over the wire, those operating systems and programming languages cannot communicate with each other. To allow seamless cross-platform interoperability, there must be a mechanism by which the service consumer and the service provider agree to a common set of types and the textual representation of the data stored in them. The Web services description provides the framework through which the common data types may be defined.

In WSDL, the primary method of defining these shared data types is the W3C's XML Schema specification. WSDL is, however, capable of using any mechanism to define data types, and may actually leverage the type definition mechanisms of existing programming languages or data interchange standards. No matter what type definition mechanism is used, both the service consumer and the service provider must agree to it or the service description is useless. That is why the authors of the WSDL specification chose to use XML Schemas—they are completely platform neutral.

A WSDL document is a collection of one or more service definitions. The document contains a root XML element named definitions; this element contains the service definitions. The definitions element can also contain an optional targetNamespace attribute, which specifies the URI associated with the service definitions. WSDL uses namespaces and namespace IDs in the same way we've been using them in SOAP envelopes, so it's common to find a number of namespaces declared at the definitions level of the document.

WSDL defines services as collections of network endpoints. These endpoints, in WSDL terminology, are known as ports. WSDL separates the service ports and their associated messages from the network protocol that the service is bound to (the binding). The combination of a binding and a network address results in a port, and a service is a collection of those ports.

The WSDL specification defines the main document sections as follows:

- types: A container for data type definitions using some type system (such as XSD)
- message: An abstract, typed definition of the data being communicated
- operation: An abstract description of an action supported by the service
- portType: An abstract set of operations supported by one or more endpoints
- binding: A concrete protocol and data format specification for a particular port type
- port: A single endpoint defined as a combination of a binding and a network address
- service: A collection of related endpoints

We will later look at a WSDL document generated for our service. There is quite a bit to look at, considering that we have only a few service methods and no custom data type. As we will see, WSDL documents tend to be lengthy and hard to understand, even for extremely simple services. That's why you're best off letting a tool like AXIS generate the WSDL for you.

## Axis with PowerBuilder 9

A PowerBuilder application can now act as a client consuming a Web service that is accessed through the Internet without using any third party product (like we described in the previous article, "Apache Axis - The Third Generation SOAP Implementation"). Using SOAP and WSDL, a collection of functions published remotely as a single entity can now become part of our PowerBuilder application. A Web service accepts and responds to requests sent by applications or other Web services.

Invoking Web services through SOAP requires serialization and de-serialization of data types, and the building and parsing of XML-based SOAP messages. Using the new files *PBSoapClient90.pbd* and the *PBSoapClient90.dll*, the Web services client proxy performs these tasks for us, thereby eliminating the need to have extensive knowledge of the SOAP specification and schema, the XML Schema specification, or the WSDL specification and schema (however if you know these already it is not a fault).

## Custom Deployment with Axis - Introducing WSDD

JWS files are a great and quick way to get your classes out there as Web services, but they're not always the best choice. For one thing, you need the source code - there might be times when you want to expose a pre-existing class on your system without source. Also, the amount of configuration you can do as to how the service gets accessed is pretty limited - you can't specify custom type mappings, or control which Handlers get invoked when people are using your service. (note for the future : the Axis team, and the Java SOAP community at large, are thinking about ways to be able to embed this sort of metadata into your source files if desired)

To really use the flexibility available to you in Axis, you should become familiar with the Axis Web Service Deployment Descriptor (WSDD) format. A deployment descriptor contains many things you want to "deploy" into Axis - i.e. make available to the Axis engine. The most common thing to deploy is a Web service so let's start by taking a look at a deployment descriptor for our basic service:

**deploy.wsdd**

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<service name="PBCalculator" provider="java:RPC" >
<parameter name="className" value="AxisCalculator" />
<parameter name="allowedMethods" value="*" />
</service>
</deployment>
```

Pretty simple, really - the outermost element tells the engine that this is a WSDD deployment, and defines the "java" namespace. Then the service element actually defines the service for us. In this case, our provider is "java:RPC", which is built into Axis, and indicates a Java RPC service. The actual class that handles this is org.apache.axis.providers.java.RPCProvider.

We need to tell the RPCProvider that it should instantiate and call the correct class (e.g. AxisCalculator), and we do so by including <parameter> tags, giving the service one parameter to configure the class name, and another to tell the engine that any public method on that class may be called via SOAP (that's what the "*" means; we could also have restricted the SOAP-accessible methods by using a space or comma separated list of available method names).

## Using the AdminClient

Once we have this file, we need to send it to an Axis server in order to actually deploy the described service (do not forget to copy our Web service .class file from the last article into *%Jaguar%\Repository\WebApplication\Axis\WEB-INF\classes\AxisCalculator.class*. Note that you can also download the files from the Web site you obtained this article from). We do this with the AdminClient, or the "org.apache.axis.client.AdminClient" class. An invocation of the AdminClient looks like this:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

Note: Look at my setpath.bat file, which sets the classpath.

If you want to prove to yourself that the deployment really worked, use the AdminClient to get a listing of all the deployed components in the server:

```
java org.apache.axis.client.AdminClient list
```

When you make a service available using Axis, a unique URL is typically associated with that service. For JWS files, that URL is simply the path to the JWS file itself. For non-JWS services, this is usually the URL: http://localhost:8080/axis/services/<service-name> (in our case <service-name> is PBCalculator)

If you access the service URL in a browser, you'll see a message indicating that the endpoint is an Axis service, and that you should usually access it using SOAP. However, if you tack on "?wsdl" to the end of the URL, Axis will automatically generate a service description for the deployed service, and return it as XML in your browser (try it!).

The resulting description may be saved or used as input to proxy-generation, described next. You can give the WSDL-generation URL to your online partners, and they'll be able to use it to access your service with PowerBuilder, or with toolkits like .NET, SOAP::Lite, or any other software which supports using WSDL.

## The PowerBuilder 9 Client

We want PowerBuilder 9 to call our Web service. These are the steps we have to do to make this happen:

1. Create a proxy to the Web service by using a WSDL file
2. Use the proxy to call the Web service.
3. Test the request and response using tcpmon.

To accomplish step one we need a WSDL file. Axis supports WSDL in three ways:

- As we have already tried, we know that when we deploy a service in Axis, users may then access the service's URL with a standard web browser and by adding "?WSDL" to the end of the URL. They will then obtain an automatically-generated WSDL document which describes the service.
- There is a tool provided, called "WSDL2Java", which will build Java proxies and skeletons for services with WSDL descriptions,
- Another tool is provided, "Java2WSDL", which will build WSDL from Java classes.

We will choose option one. We will create a new Web service proxy, and select the “Web Service Proxy Wizard” icon from the Projects page in the “New” dialog box. The Web Service Proxy Wizard helps us create the proxy so we can use the Web service in PowerScript  In the wizard we specify:

- which WSDL file we want to access
- which service within the WSDL file we want to select

- which port or ports we want to use
- a prefix which is appended to a port name, and becomes the proxy name
- to which PowerBuilder library we want the proxy deployed

Fill in the values so that your project looks like this:

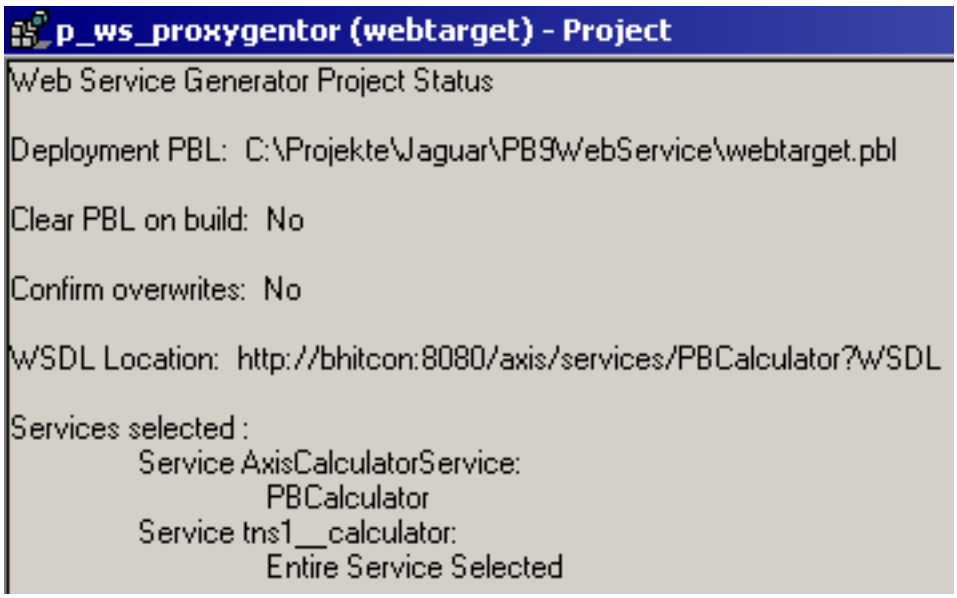


Figure 1: Proxy project

Execute the proxy and you will get some errors in this beta release. Change your p_pbcalculator.srx file so that it looks like the file below and import it into your library:

**p_pbcalculator.srx**

```
$PBExportHeader$p_PBCalculator.srx$PBExportComments$Proxy imported from
WSDL file via Soap Proxy generator.
global type p_PBCalculator from NonVisualObject
end type

type variables
Protected:
   string tns1 = "http://Axis"
   string intf = "http://localhost:8080/axis/services/PBCalculator"
   string impl = "http://localhost:8080/axis/services/PBCalculator-
impl"
   string SOAP_ENC = "http://schemas.xmlsoap.org/soap/encoding/"
   string wsdl = "http://schemas.xmlsoap.org/wsdl/"
   string wsdlsoap = "http://schemas.xmlsoap.org/wsdl/soap/"
   string xsd = "http://www.w3.org/2001/XMLSchema"

   string endpoint = "http://localhost:8080/axis/services/PBCalculator"
   tns1__calculator i__tns1__calculator
end variables

global p_PBCalculator p_PBCalculator

forward prototypes
public:
```

```
function long add (long in0, long in1) alias for "add([in] xsd:int in0,
[in] xsd:int in1) return xsd:int addReturn@add"

function tns1__calculator getComponent () alias for "getComponent()
return tns1:calculator return@getComponent"

function string getVersion () alias for "getVersion() return xsd:string
getVersionReturn@getVersion"

subroutine  main (string  in0[ ]) alias for "main([in] xsd:string[ ]
in0)@main"

function long subtract (long in0, long in1) alias for "subtract([in]
xsd:int in0, [in] xsd:int in1) return xsd:int subtractReturn@subtract"
end prototypes
```

Now comes step two. We want to invoke the Web service through our proxy object. To do this we need the PBSoapClient90.dll and PBSoapClient90.pbd files, which are installed in the Shared/PowerBuilder directory when you install PowerBuilder 9. When you create a Web service client application, you do not need to copy PBSoapClient90.dll to another location, but you do need to deploy it with the client executable in a directory in the application's search path. To add PBSoapClient90.pbd to the application's search path, right-click the client target in the System Tree and select Properties from the pop-up menu. In the current beta release, you need to type in the full path and name of the PBSoapClient90.pbd file.

The SoapConnection class is used to connect to the SOAP server that hosts the Web service you want to access. It has these public methods that we will use immediately:

```
CreateInstance(proxyObj,proxyName),
CreateInstance(proxyObj, proxyName, endpoint),
SetOptions(optionsString).
```

The following script shows a connection to a Web service on a SOAP server. It sets the connection properties using our tcpmon-URL (described later). If you did not set a URL, the one stored in the proxy would be used. Then the script creates an instance of the SoapConnection object, and enables logging so we set SoapLog (other connection options (for https) would be ‚UserID', ‚Password', or ‚Timeout'). Next we invoke our proxy object for the PBCalculator Web service. We would print out the return value or an error message if something were wrong.

**PowerBuilder 9 Beta3 Source Code:**

```
Long ll_ret
Long ll_v1 = 4, ll_v2 = 3, ll_ReturnVal
String str_proxy_name = "p_pbcalculator"
String ls_url = "http://localhost:8090/axis/services/PBCalculator"

SoapConnection lsc_Conn
p_pbcalculator lproxy_obj

lsc_Conn = create SoapConnection
lsc_Conn.SetOptions("SoapLog=~"C:\\soaplog.txt~"")

ll_ret = lsc_Conn.CreateInstance(lproxy_obj, str_proxy_name, ls_url)

IF ll_ret <> 0 THEN
      MessageBox("Error","CreateInstance failed " + String(ll_ret))
      return
```

```
END IF

Try

      ll_ReturnVal = lproxy_obj.add(ll_v1, ll_v2)
      MessageBox("add returned", &
            String(ll_v1) + "+" + String(ll_v1) + "=" +
String(ll_ReturnVal))

Catch ( SoapException e )

      MessageBox("Error","Exception: " + e.getmessage())

end try
Destroy lsc_Conn
```

## Using the Axis TCP Monitor (tcpmon)

And now to step three. We want to see what goes through the wire. We want to see what PowerBuilder 9 is going to send to Axis and what comes back. To do this we use another tool that comes with Axis: tcpmon. It's a great way to see what's going on with the SOAP messages.

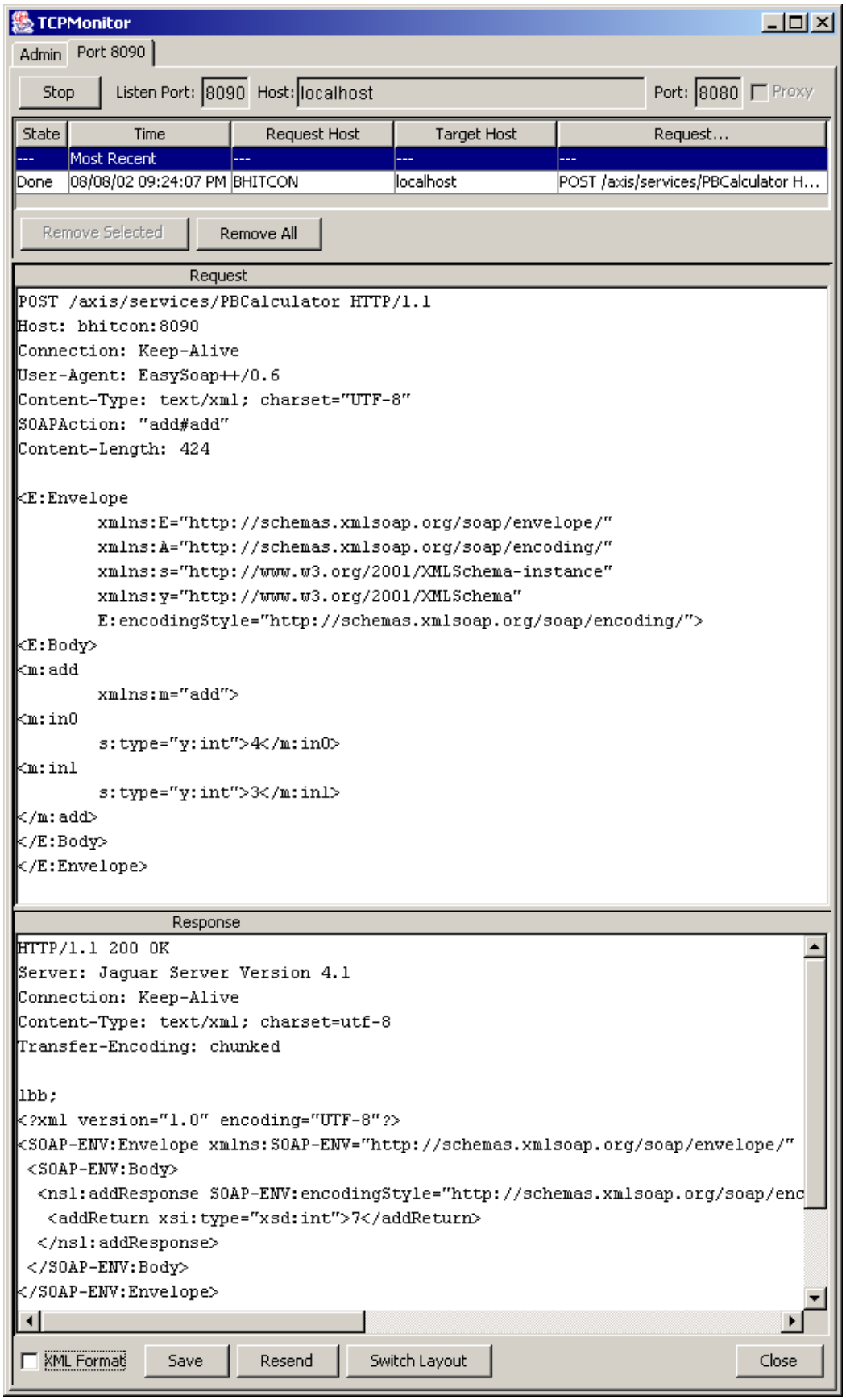
The included "tcpmon" utility can be found in the org.apache.axis.utils package. To run it from the command line, type:

```
javaw org.apache.axis.utils.tcpmon 8090 localhost 8080
```

This means that we select a local port which tcpmon will monitor for incoming connections (8090), a target host (localhost) where it will forward such connections, and the port number on the target machine that should be "tunneled" to (8080). Now each time a SOAP connection is made to the local port, you will see the request appear in the "Request" panel, and the response from the server in the "Response" panel. Tcpmon keeps a log of all request/response pairs, and allows you to view any particular pair by selecting an entry in the top panel. You may also remove selected entries, or all of them, or choose to save to a file for later viewing.

Now we start our application and we can see the request to EAServer and Axis and the response back to our PowerBuilder client. We should see a MessageBox with the correct value (4+3=7).

**Output from tcpmon: (next page)**


TCPMonitor
Admin
Port 8090
Stop
Listen Port: 8090
Host: localhost
Port: 8080
Proxy
State
Time
Request Host
Target Host
Request...
--- Most Recent --- --- ---
Done 08/08/02 09:24:07 PM BHITCON localhost POST /axis/services/PBCalculator H...
Remove Selected
Remove All
Request
POST /axis/services/PBCalculator HTTP/1.1
Host: bhitcon:8090
Connection: Keep-Alive
User-Agent: EasySoap++/0.6
Content-Type: text/xml; charset="UTF-8"
SOAPAction: "add#add"
Content-Length: 424
<E:Envelope
xmlns:E="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:A="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s="http://www.w3.org/2001/XMLSchema-instance"
xmlns:y="http://www.w3.org/2001/XMLSchema"
E:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<E:Body>
<m:add
xmlns:m="add">
<m:in0
s:type="y:int">4</m:in0>
<m:in1
s:type="y:int">3</m:in1>
</m:add>
</E:Body>
</E:Envelope>
Response
HTTP/1.1 200 OK
Server: Jaguar Server Version 4.1
Connection: Keep-Alive
Content-Type: text/xml; charset=utf-8
Transfer-Encoding: chunked
lbb;
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
<SOAP-ENV:Body>
<ns1:addResponse SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/enc
<addReturn xsi:type="xsd:int">7</addReturn>
</ns1:addResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
XML Format
Save
Resend
Switch Layout
Close

## Conclusion

PowerBuilder 9 is now a perfect tool to act as a Web Service client. EAServer with Axis and PowerBuilder 9 is the right combination for using the newest technologies in your companies. Download the beta of PowerBuilder 9 from http://www.sybase.com/betapb90program.

## About the Author

Berndt Hamboeck is a senior consultant for Sybase Austria. He is an EASAC, SCAPC8 and CSI and SCJP2.

**Listing 1: Setpath.bat**

```
REM SET JAGUAR=c:\sybase\EAServerSET
WEBAPP=%JAGUAR%\Repository\WebApplication\axis\WEB-INF\libSet
CLASSPATH=%CLASSPATH%;%JAGUAR%\java\lib\easclient.jar;Set
CLASSPATH=%CLASSPATH%;%JAGUAR%\html\classes;Set
CLASSPATH=%CLASSPATH%;%JAGUAR%\java\classes;Set
CLASSPATH=%CLASSPATH%;%JAGUAR%\java\classes\crimson.jar;Set
CLASSPATH=%CLASSPATH%;%WEBAPP%\axis.jar;Set
CLASSPATH=%CLASSPATH%;%WEBAPP%\commons-logging.jar;Set
CLASSPATH=%CLASSPATH%;%WEBAPP%\jaxrpc.jar;Set
CLASSPATH=%CLASSPATH%;%WEBAPP%\log4j-core.jar;Set
CLASSPATH=%CLASSPATH%;%WEBAPP%\saaj.jar;Set
CLASSPATH=%CLASSPATH%;%WEBAPP%\tt-bytecode.jar;Set
CLASSPATH=%CLASSPATH%;%WEBAPP%\wsdl4j.jar;.
```

**Sybase Incorporated**
**Worldwide Headquarters**
5000 Hacienda Drive
Dublin, CA 94568-7902, USA
Tel: 1-800-8-Sybase, Inc.
Fax: 1-510-922-3210
www.sybase.com