CREATING AND DEPLOYING ABL WEB SERVICES

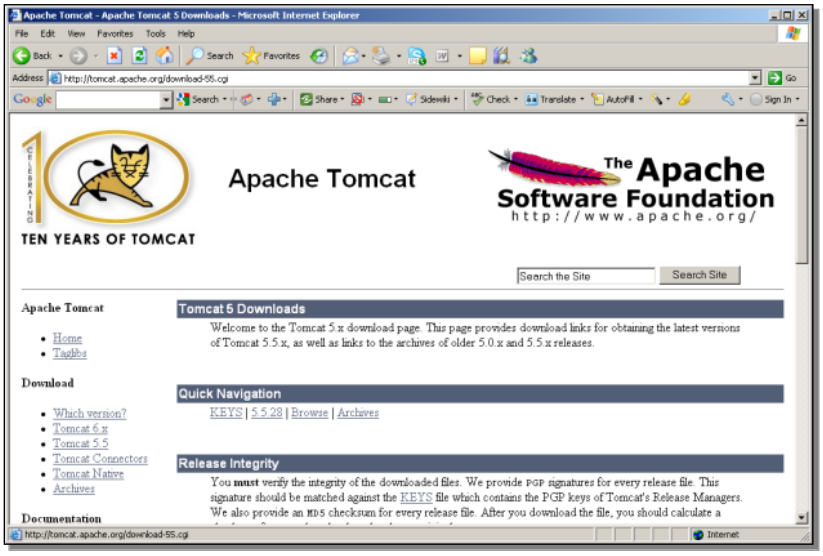
John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
August 2010

DISCLAIMER

This document accompanies two of the presentations that present different aspects of extending an OpenEdge application to take advantage of today's tools for building Rich Internet Applications. The first of the two sessions, entitled ***Installing and Configuring the Tomcat Web Server and the OpenEdge Web Services Adapter*** shows how to install the Apache Tomcat Web Server to use with the OpenEdge WebServices Adapter, in preparation for running OpenEdge procedures as services from other applications. The second session, on ***Deploying and Running an ABL Procedure as a Web Service,*** completes the process of exposing an ABL procedure as a Web service for access from other applications.

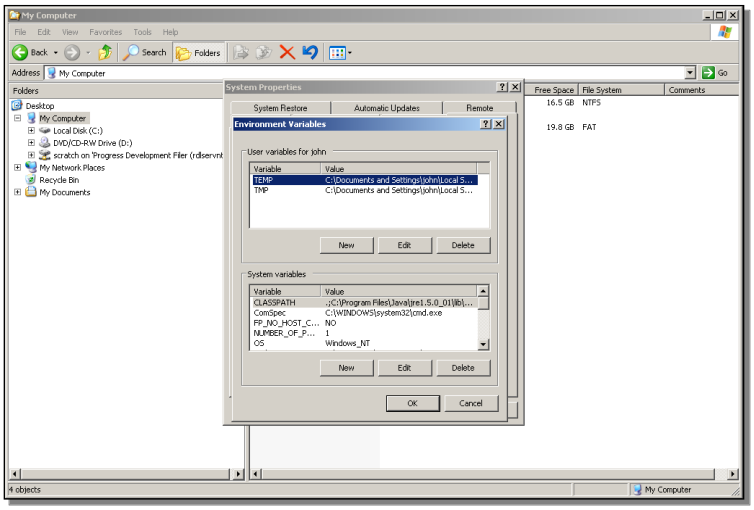
OpenEdge Web services are one way to invoke ABL data management and business logic procedures from a separately defined user interface. There are many choices you can make among third party Web servers that act as a Java Servlet Engine, which can then host the OpenEdge Web Services Adapter to allow other applications to run ABL procedures as Web services. One of the most popular is Tomcat, one of the Apache projects. And since OpenEdge now ships with a script to help you configure Tomcat, this document and the presentation it accompanies show you how to install Tomcat and use that OpenEdge install script to get you started.

At the URL **`apache.tomcat.org`**, you can select the latest Tomcat release from the Downloads area.

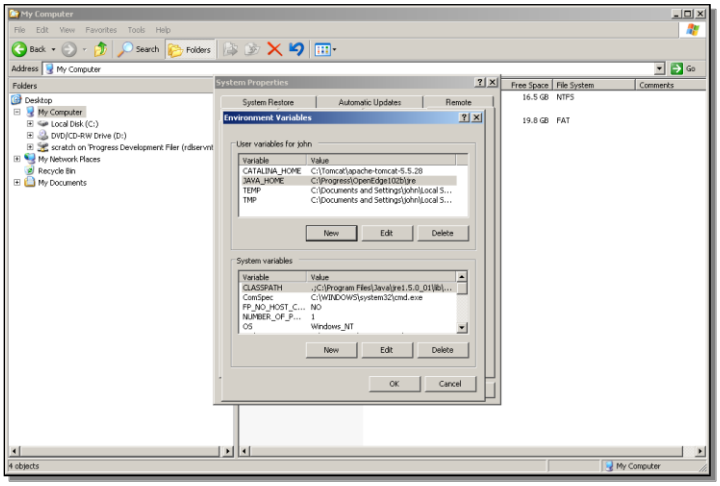


If you scroll down to see the download options, you can select the zip file from the Core list, and save that zip file to a folder on your machine where you want to install Tomcat.

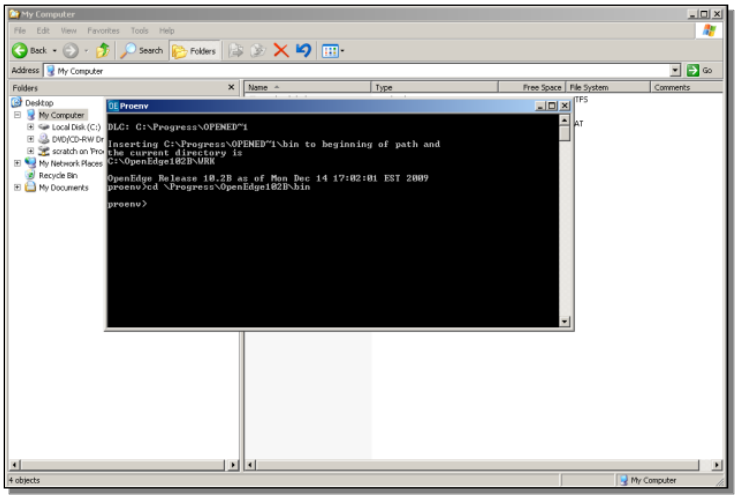
Once the download is done, you need to define a couple of environment variables for the configuration script to use. In Windows Explorer, you can right-click on the **My Computer** node, select **Properties**, and then the **Advanced** tab, and **Environment Variables**.



The first new one you need is to establish the home directory for the Tomcat install. The code name, if you will, for Tomcat, which is also the name of a batch file you use to start and stop the web server, is *catalina*, so the script identifies the home directory as **`CATALINA_HOME`**. You assign this the value of the folder under your Tomcat installation directory where you will extract the zip file. The second new environment variable establishes the **`JAVA_HOME`** directory for the configuration file to use. And that is the **`jre`** sub-directory under wherever you installed OpenEdge.

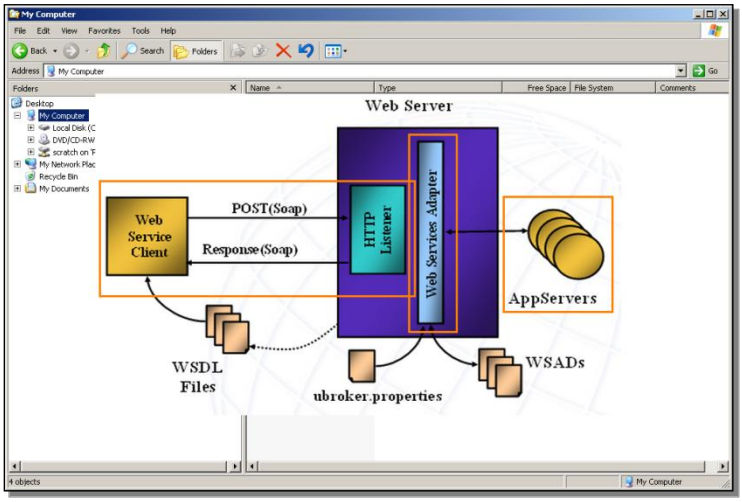


That takes care of the two new environment variables the script needs. Once that's done you can extract the contents of the zip file. After that has completed, you need to open a batch window, which can be a **`proenv`** window as shown here, to run the script from. Change to the **`bin`** directory under the OpenEdge install.

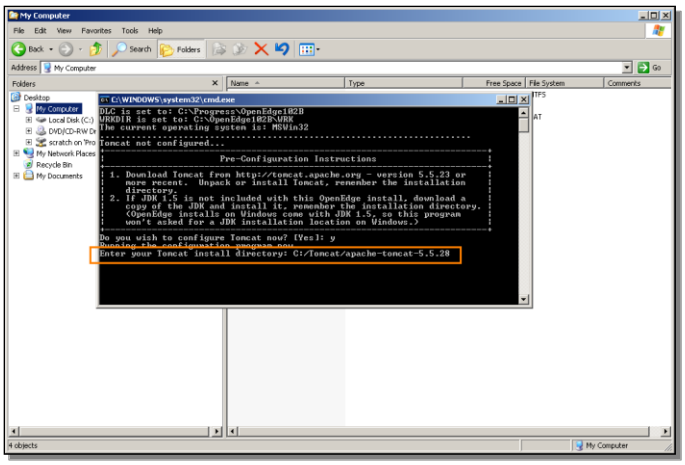


Enter **`OE_TC`**, for OpenEdge Tomcat. This is the name of the **`.bat`** file shipped with OpenEdge that does the configuration for you. Note that this script also operates on the Tomcat install zip file directly, if it hasn't been extracted already. The script does all the work to configure Tomcat as your Web server, including establishing the OpenEdge Web Services Adapter (WSA) as a Web application in Tomcat.

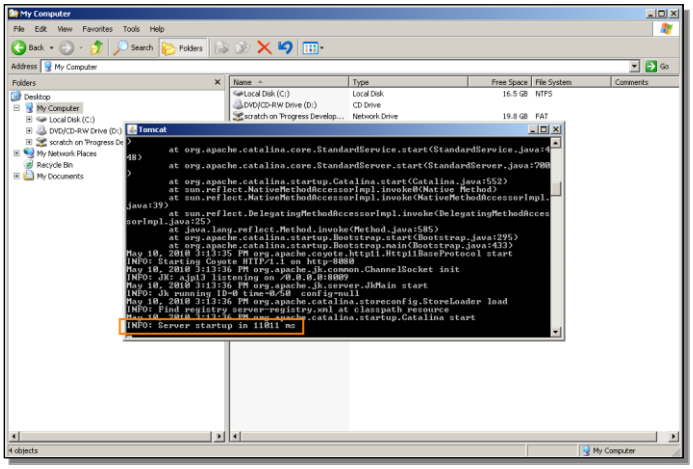
The WSA is a program supplied with OpenEdge that runs as a Java Servlet in the Web server. It provides the connectivity between the OpenEdge AppServer and the procedures.



The script does all the work to configure Tomcat as your Web server, including standard Web service requests that others will use to access your ABL procedures. After confirming that you want to configure Tomcat now, enter the directory where you downloaded the Tomcat zip file.

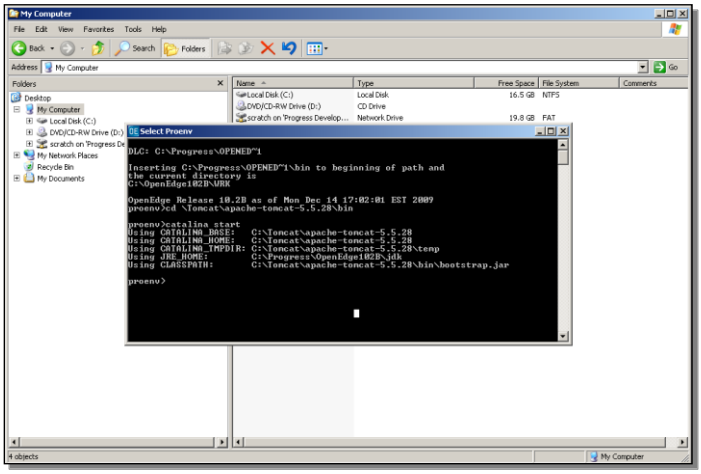


Once the configuration script runs to completion, switch to the **`bin`** directory under the Tomcat install directory. The **`.bat`** file that starts and stops Tomcat is called **`catalina.bat`**, so to start the web server, you type **`catalina start`**. The Web server startup information comes up in a separate window, and after a few seconds, it should display **`INFO: Server startup in (so many) milliseconds`**.



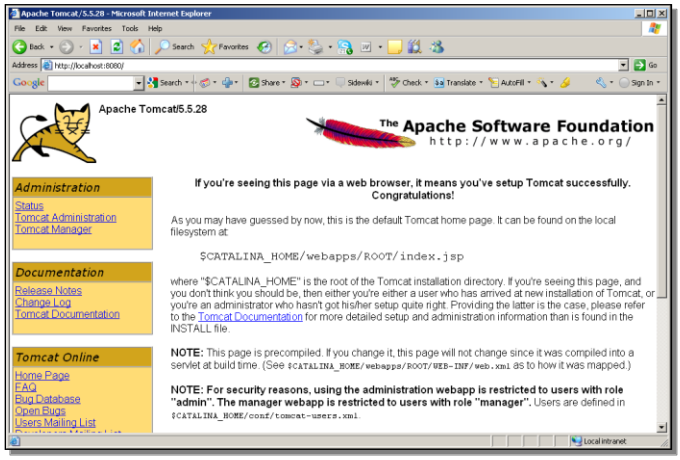
You can minimize that window, but don't close it for as long as you want the Web server to run.

Back in the **proenv** window, you can see that the **catalina** script has run to completion and confirmed some of the relevant environment variables.

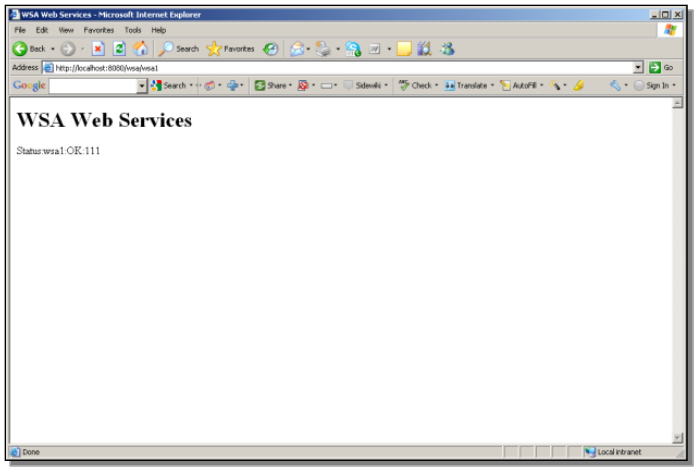


The **oe_tc** script that comes with OpenEdge is also designed to set up Tomcat for use as the Web server for WebSpeed as well, but the Apache HTTP Server is a more popular choice for WebSpeed, so this session doesn't go into the WebSpeed side of the configuration script, and the presentations in this series that focus on WebSpeed use the Apache Web server.

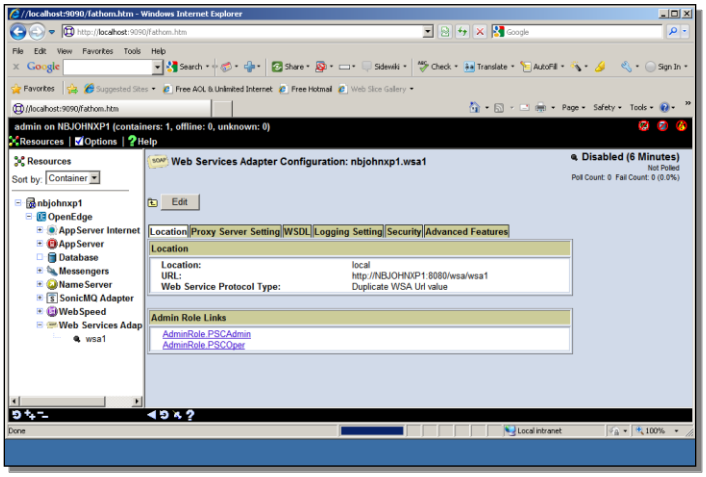
In any case, you can quickly verify that Tomcat is running. Unless you configure it otherwise, it runs on port 8080, so if you enter that port number on **localhost** as a URL, you see a confirmation page downloaded from the Apache site that the web server is set up successfully:



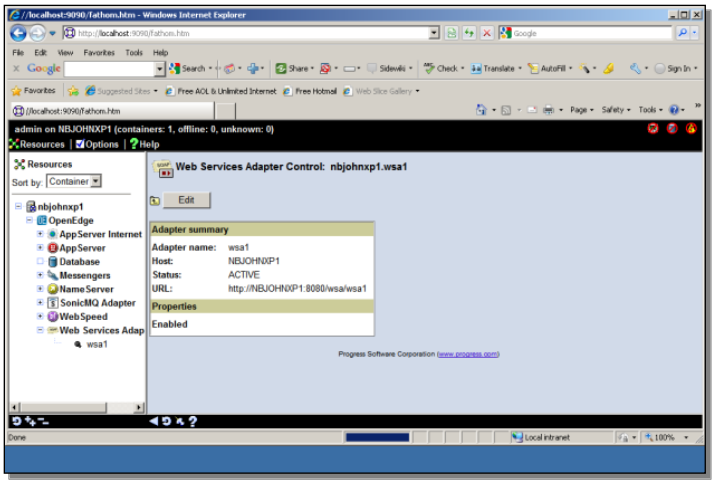
If you then add **`wsa`** and the default OpenEdge Web Services Adapter **`wsa1`** to that URL, you get a confirmation message from the WSA that you're communicating with it as well:

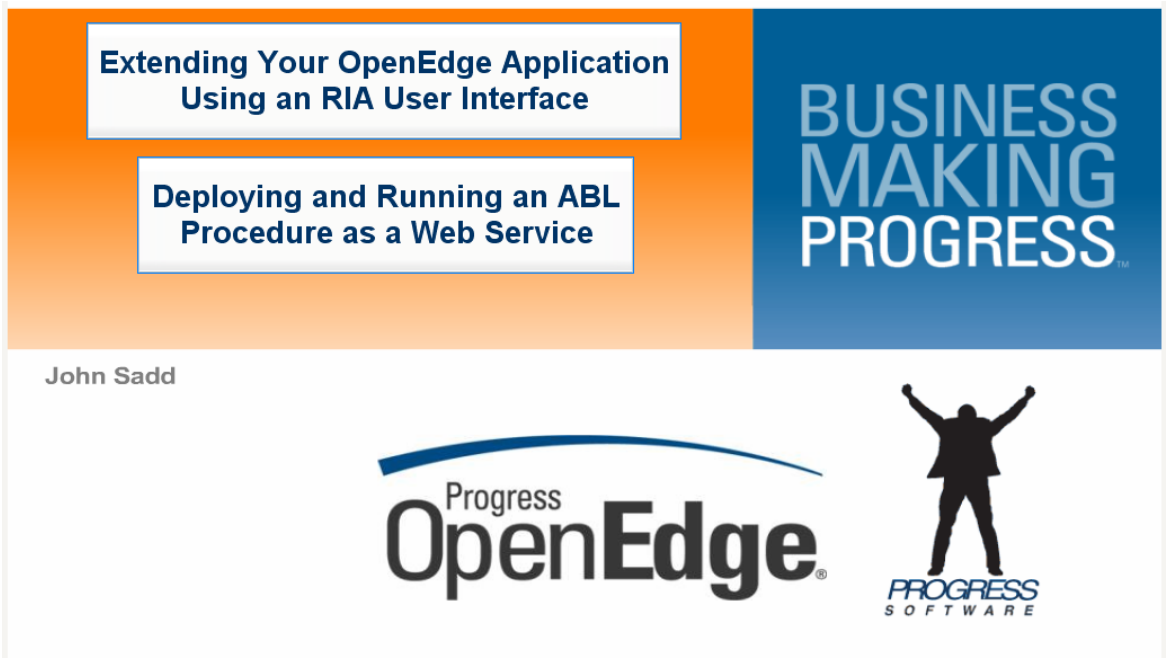


Now that Tomcat is installed and configured, you can go back into OpenEdge Explorer and take a look at the properties for the default Web Services Adapter, **`wsa1`**. If you expand the Web Services Adapter group for OpenEdge and select **`wsa1`**, then under **Configuration**, you can see the URL used to access the WSA.



You can verify that it's set to 8080, or whatever port number your web server uses, and also set more advanced properties. Back on the main **`wsa1`** page, you can then select the **Control** link, and confirm that the WSA is enabled and active.



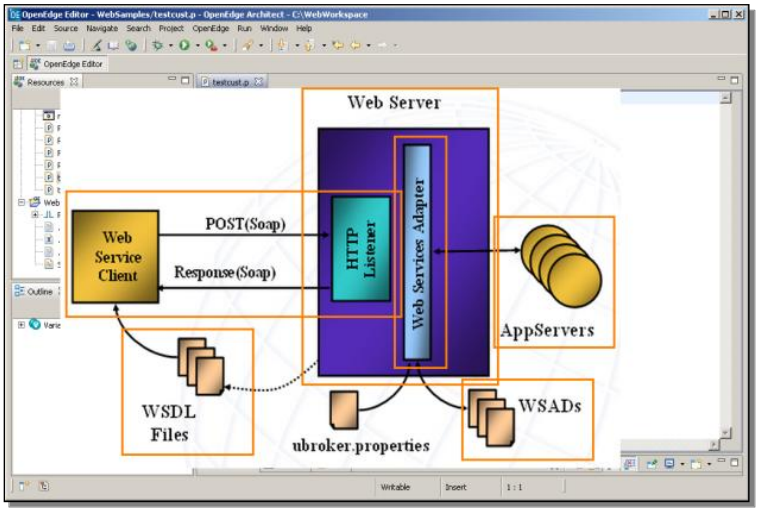


In the second part of this two-part session I show how to deploy an ABL procedure as a Web Service and invoke it from another ABL test procedure. I start with about the simplest possible ABL procedure, called **testcust.p**. This is also used in a similar example from another session to verify that the OpenEdge default AppServer is running properly.

```
/* TestCust.p:
   Test procedure for AppServer and WebServices call. 10/07 */

DEFINE INPUT  PARAMETER pCustNum  AS INTEGER   NO-UNDO.
DEFINE OUTPUT PARAMETER pCustName AS CHARACTER NO-UNDO.

DEFINE NEW GLOBAL SHARED VARIABLE giCallCount AS INTEGER NO-UNDO.

giCallCount = giCallCount + 1.
IF pCustNum = 1 THEN
DO:
    pCustName = "Test Name".
    RETURN "Success!".
END.
ELSE DO:
    pCustName = "No Name".
    RETURN "Failure!".
END.
```

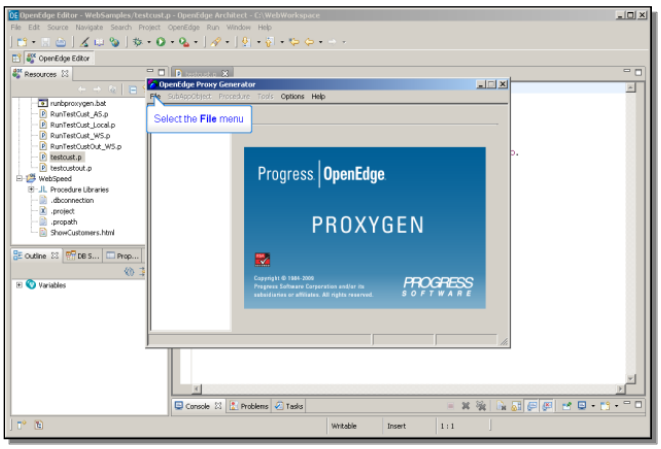
It takes a customer number as input, and passes back a string as if it were a customer name from the database, along with a **RETURN-VALUE** from the ABL **RETURN** statement.

So how do you set this up to be called as a Web service? First, you need to make your ABL procedures accessible through an OpenEdge AppServer to make them callable as Web services.

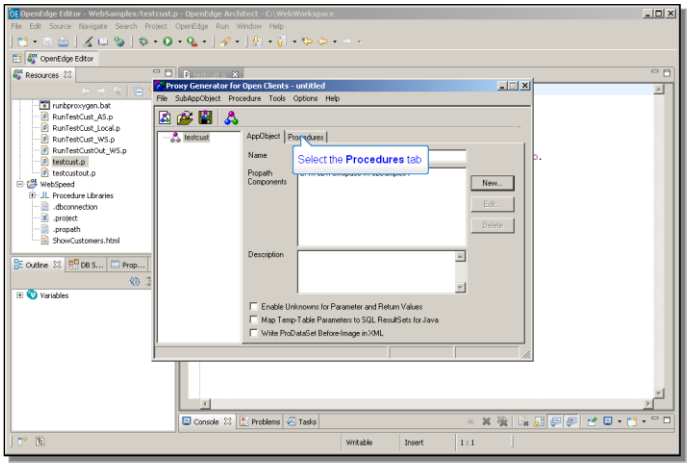


The AppServer is designed to handle requests that come directly from OpenEdge clients. To adapt it to receive requests as Web services, you need to insert a Web server into the mix. As shown earlier, this example uses the Apache Tomcat Web server. The Web server holds the Web Services Adapter (WSA), a program supplied with OpenEdge that runs as a Java Servlet in the Web server. The OpenEdge Proxy Generator, ProxyGen for short, generates the Web Service Application Descriptor (WSAD) and a test WSDL (Web Service Descriptor Language document) to use in deploying and testing the service. At runtime, a request comes in from a Web service client into the Web server, in the form of a SOAP request. The request is routed to the AppServer and your ABL procedure, and output returned to the caller as another SOAP message. All the message construction and formatting is taken care of for you.

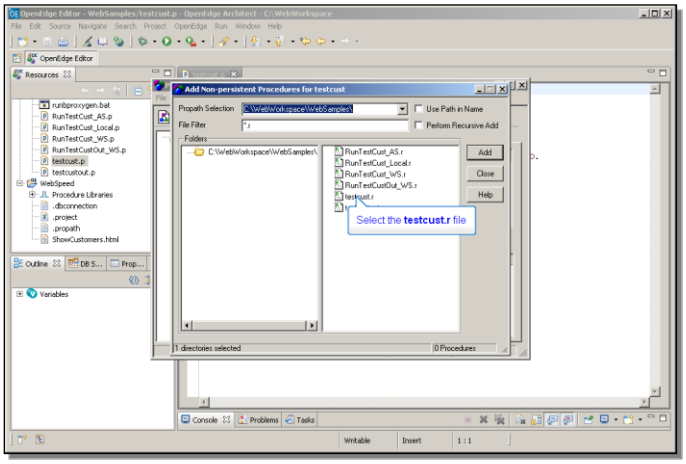
You start ProxyGen from the OpenEdge Start group, and under the **`File`** menu, select **`New`**.



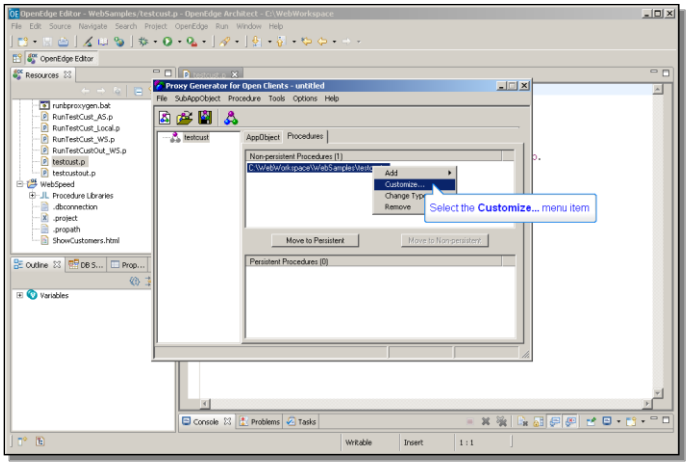
What you're defining is called an *application*, and for a non-persistent procedure -- one without multiple named internal entry points -- you define an **AppObject**. The example uses one called **`testcust`**. You need to adjust the Propath so that the compiled ABL code will be found, which is here in the **`WebSamples`** folder. Next you need to define what Procedures are part of the AppObject.



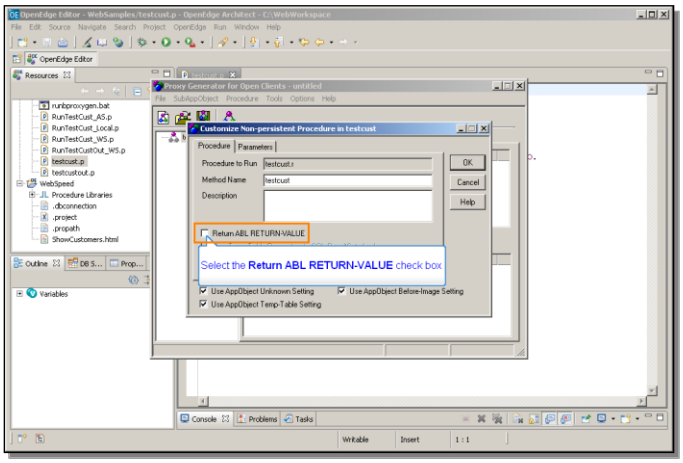
There's only one, and it's a non-persistent procedure, so you select **Add**, and the r-code for the test procedure:



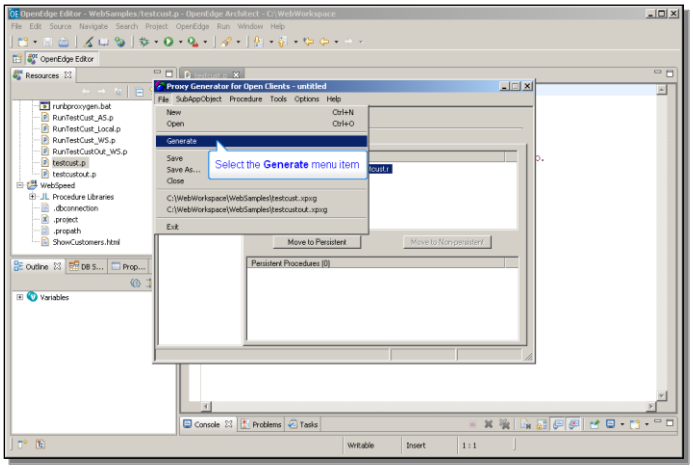
You add that to the AppObject. One more thing you need to do in this example is allow for the ABL **RETURN-VALUE** that **testcust.p** uses. This is under the **Customize** option for the procedure.



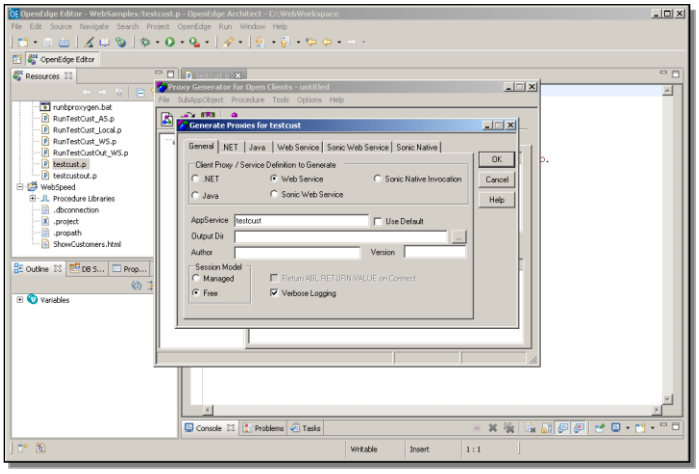
Check on the option to return the ABL **`RETURN-VALUE`**. It can't be returned implicitly, as it is from one ABL procedure to another procedure that calls it directly, so it will be passed as an additional output parameter called **`result`**.



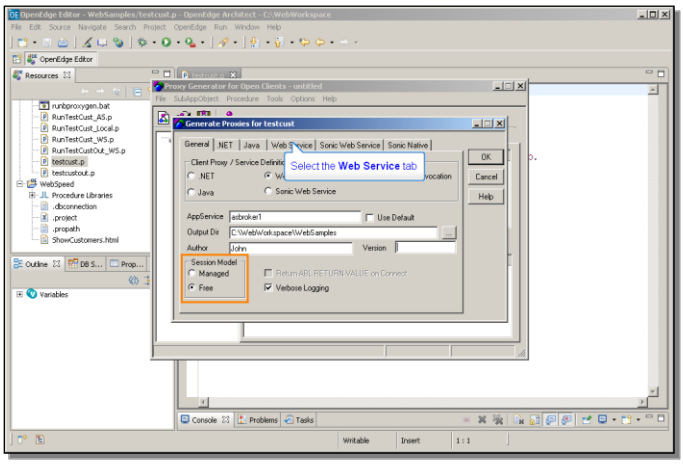
Now you need to generate the proxy for this new AppObject.



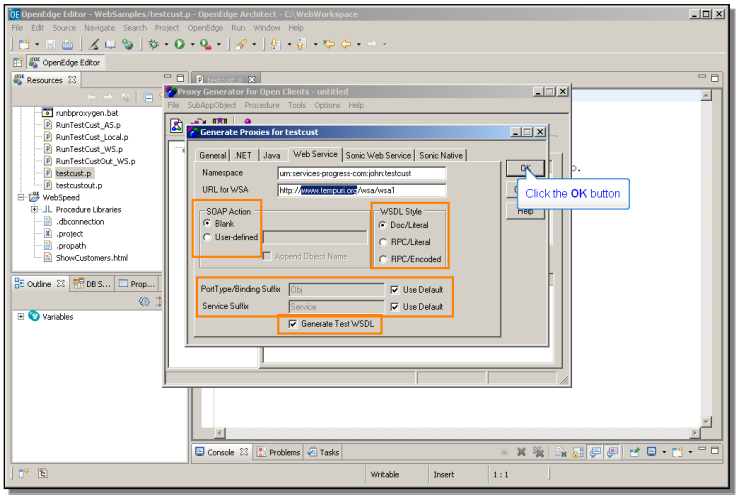
This creates all the pieces reviewed in the diagram of what a Web service call requires. ProxyGen can create proxies for many different kinds of external access to an OpenEdge session, but this one is a Web service.



The procedure is deployed where the standard AppServer, **`asbroker1`**, can run it, so you need to specify that here. You tell ProxyGen where the generated output should go, and enter any values for the **Author** and **Version**. You want the service to run without binding the AppServer session it gets run on, so make sure the **Session Model** is set to **Session-Free**.



Now you enter information specific to the Web Service proxy type. Every service has to have a **namespace** that uniquely identifies it. In this example I use one that identifies it as a Progress service, under my username. This becomes part of the identification of the service when it's called. You also need to specify the **URL** that identifies the port number the Web server will be running on.



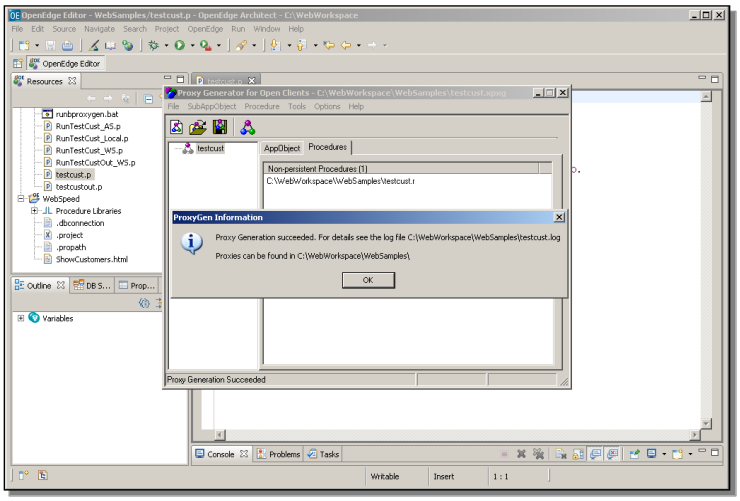
Since this example uses Tomcat as the Web server, that runs on port 8080. And the service will be deployed on the default Web Services Adapter for OpenEdge, **`wsa1`**.

There's no specific **SOAP Action** needed, so that can be left set to **`Blank`**.

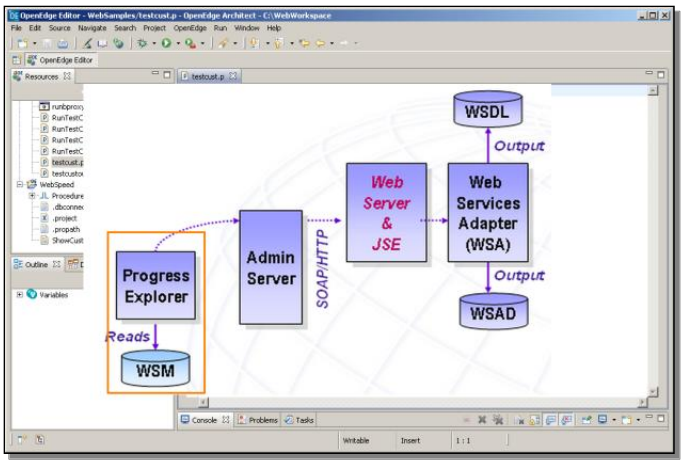
The **WSDL Style** tells ProxyGen which of several XML formats to use for the parameters to the request. The **`Doc/Literal`** style uses a standard XML format to represent all of the parameters as a single XML element. If you choose this style, ProxyGen does the necessary formatting for you, and the Web service consumer is informed that this is the style to expect, so there's no work for you to do. This is generally the recommended choice.

By default ProxyGen furnishes a suffix of **`Obj`** foreach **PortType** name -- representing a set of related operations that the Web service supports -- and **`Service`** for each service name, so you can leave those as they are.

The **Generate Test WSDL** option gives you a WSDL definition document at development time that you can inspect (using the WSDL Analyzer). You tell ProxyGen where you want it to save the files it creates, and it's done.

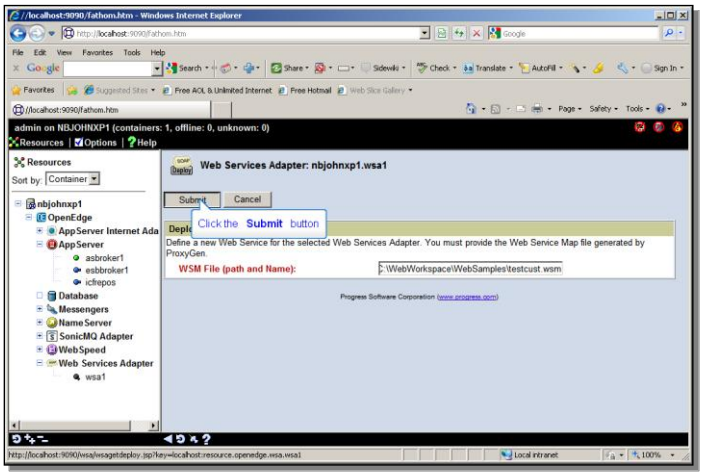


So that's the job ProxyGen does, creating the descriptor files the Web server will use to route a request to the right AppServer and the right ABL procedure. OpenEdge Explorer reads the Web Service Mapping file that is part of what ProxyGen created, and uses this to deploy the procedure as a service.

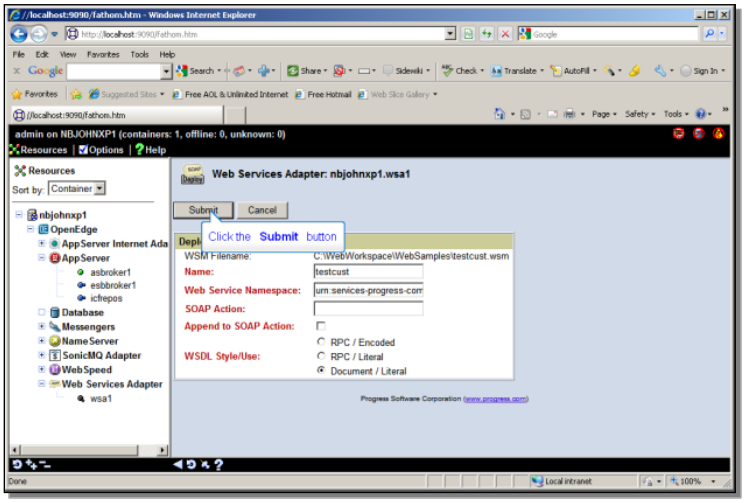


To examine the AppServer and Web Services Adapter, open OpenEdge Explorer from the OpenEdge start menu, expand the OpenEdge group, and the list of AppServers. The example specified in ProxyGen that it would be accessing **`testcust.p`** as a service from the default broker, **`asbroker1`**, so you can just confirm that it's active. If it isn't, you could activate it here.

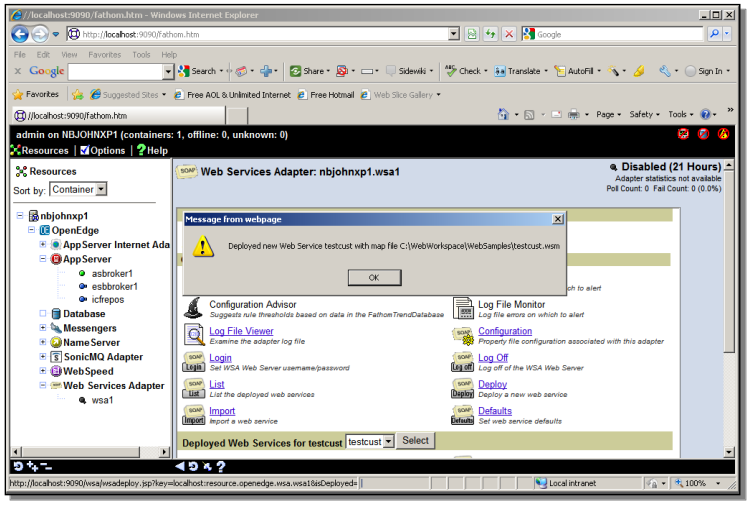
Under the Web Services Adapter group, you can select the default Adapter, **`wsa1`**, and activate it if needed. Next you need to deploy the new Web service to **`wsa1`**. To do this you select the **Deploy** link. The one value you have to fill in is the Web Service Mapping (WSM) file name that came from ProxyGen. There's no Browse button to use to locate the file, so you have to type in the pathname, and submit the change.



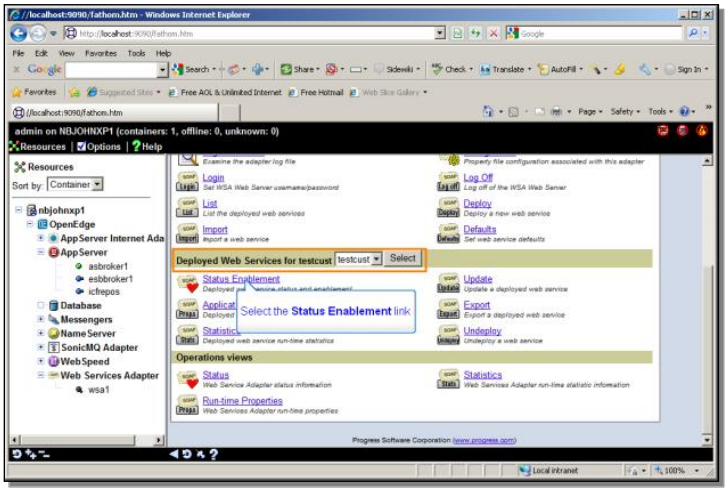
You can then review the deployment information, which is everything that was specified in ProxyGen.



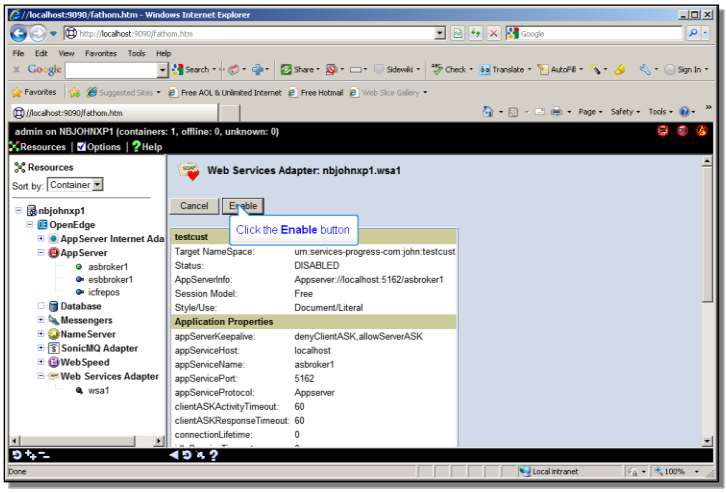
Pressing **Submit** again, you see that Explorer confirms that the service has been deployed, so it's ready to enable.



Scrolling down in the **wsa1** page, you can see a list of all the deployed services. If there's more than one deployed service, you can select one from the dropdown list and click **Select**. Then you click the **Status Enablement** link.



You then click **Enable** to enable it to be run as a Web service.

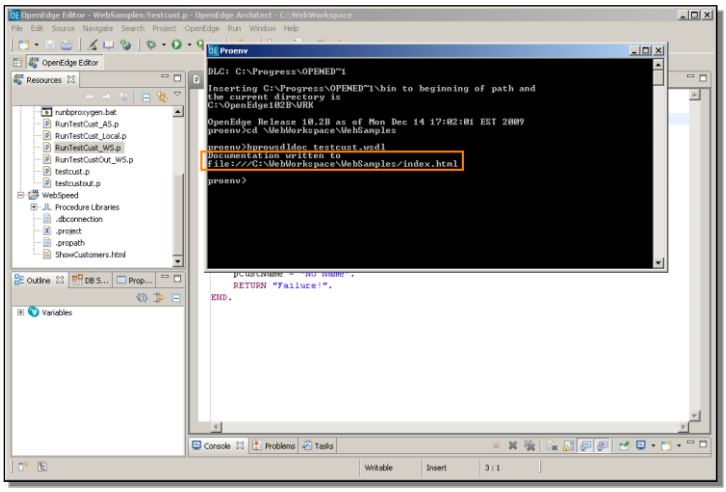


This sequence shows the role Explorer plays: to read the output from ProxyGen and use that to set up the procedure to be run on an AppServer, accessed through a Web server using the OpenEdge Web Services Adapter.

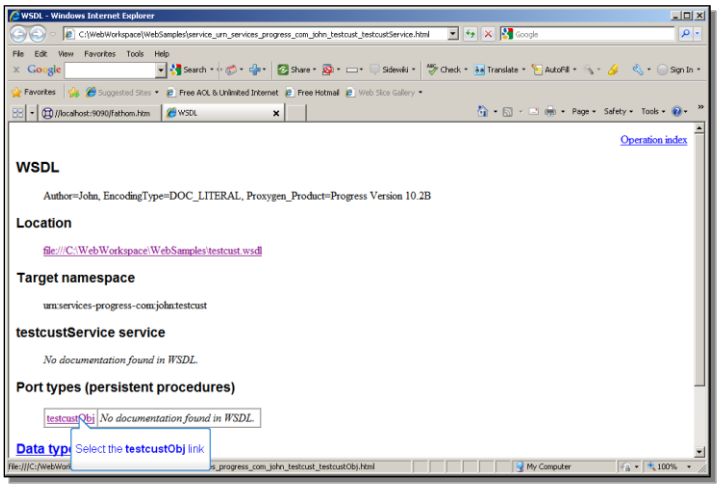
Now I'll show a simple example of how to get help in writing a procedure that will invoke the Web service. Of course, you wouldn't normally run an ABL procedure from another ABL procedure as a Web service – you could just run it directly on the AppServer – but this example does that just to test that it works. In ProxyGen, I had checked the option that said to generate a test WSDL, the descriptor document. You can use the test WSDL to help code a procedure that runs the Web service.

To access the test WSDL document, you can start a **`proenv`** window from OpenEdge, and set your directory to where all the generated proxy elements are. There's an OpenEdge utility to interpret a WSDL document for you and generate sample code and documentation, and it's called: **`b`**-for-batch **`pro`**-for –progress **`wsdl`** and **`doc`** for documentation, **`bprowsdldoc`**, and it takes the WSDL document that came out of ProxyGen as an argument. This is the same tool you would use to analyze the WSDL that describes a non-OpenEdge service that you wanted to run from an ABL

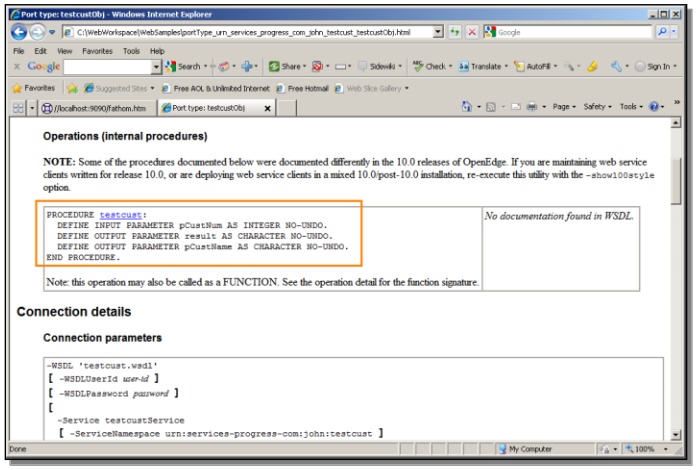
application. The utility generates a set of HTML files that provide detailed documentation of the service for you. The top-level file is always named **`index.html`**, so if you generate more than one of these for different services, make sure to put them into different directories.



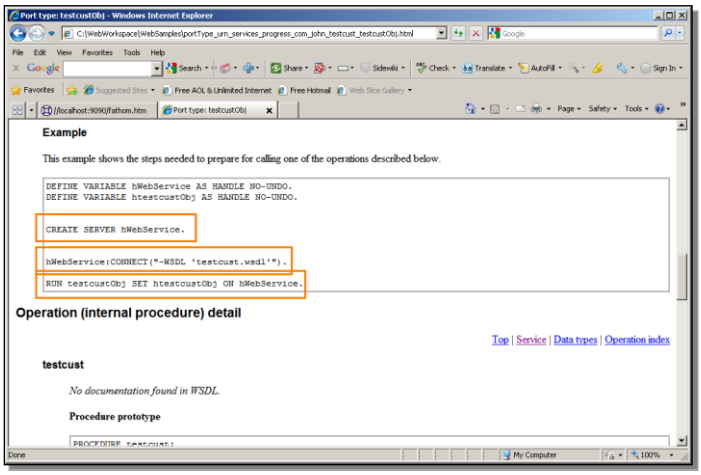
If you open **`index.html`**, it comes up in your browser. Remember that the call that's defined in my application is called a **Port Type**, and it has **`Obj`** as a generated extension.



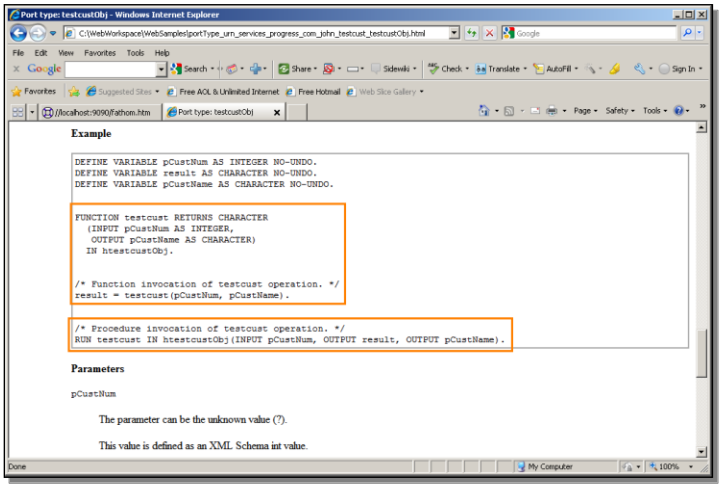
If you select the **`testcustObj`** link, you see detailed documentation for the service. Scrolling down, there's an example of how the procedure **`testcust`** would look as an internal procedure in my **AppObject**, with its parameters. The **`result`** output parameter represents the ABL **`RETURN-VALUE`** that I specified. And remember that in reality, **`testcust`** is an external .p, so calling it won't bind the AppServer session it runs on; it's just represented as an entry point in the Web service.



Scrolling down further, there are sample ABL statements for creating a server instance to connect to the Web service, connecting to the service, and running the **`testcust`** entry on that server. All this is very parallel to making an ordinary AppServer call.



Looking down further, the example shows both the code you would write to invoke the service as if it were a function and as if it were an internal procedure, even though it's actually a stand-alone .p.



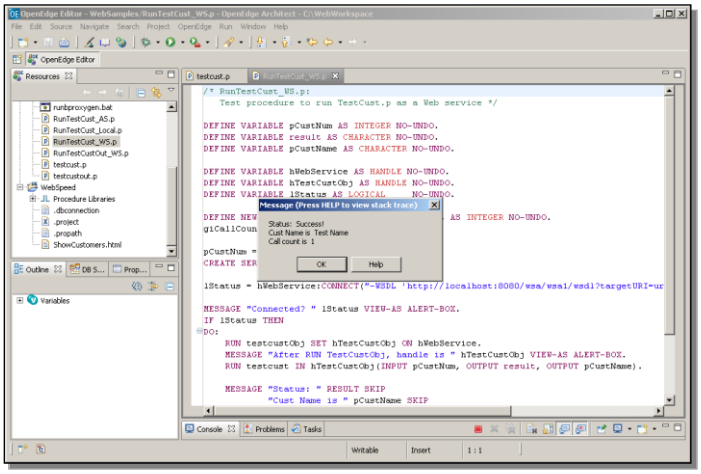
So based on sample code like what the WSDL documenter tool generates, you can create a test procedure such as the following to run **testcust** from an OpenEdge client:

```
/* RunTestCust_WS.p:
   Test procedure to run TestCust.p as a Web service */

DEFINE VARIABLE pCustNum AS INTEGER NO-UNDO.
DEFINE VARIABLE result AS CHARACTER NO-UNDO.
DEFINE VARIABLE pCustName AS CHARACTER NO-UNDO.

DEFINE VARIABLE hWebService AS HANDLE NO-UNDO.
DEFINE VARIABLE hTestCustObj AS HANDLE NO-UNDO.
DEFINE VARIABLE lStatus AS LOGICAL     NO-UNDO.

DEFINE NEW GLOBAL SHARED VARIABLE giCallCount AS INTEGER NO-UNDO.
giCallCount = 1.

pCustNum = 1.
```

Here's the **CREATE SERVER** statement, and the **CONNECT** statement with the full identifier for the service in its namespace.

```
CREATE SERVER hWebService.

lStatus = hWebService:CONNECT("-WSDL
'http://localhost:8080/wsa/wsa1/wsdl?targetURI=urn:services-progress-
com:john:testcust'").

MESSAGE "Connected? " lStatus VIEW-AS ALERT-BOX.
IF lStatus THEN
DO:
    RUN testcustObj SET hTestCustObj ON hWebService.
    MESSAGE "After RUN TestCustObj, handle is " hTestCustObj VIEW-AS ALERT-BOX.
    RUN testcust IN hTestCustObj(INPUT pCustNum, OUTPUT result, OUTPUT pCustName).

    MESSAGE "Status: " RESULT SKIP
            "Cust Name is " pCustName SKIP
            "Call count is " giCallCount
            VIEW-AS ALERT-BOX.
    hWebService:DISCONNECT().
END.
DELETE OBJECT hTestCustObj NO-ERROR.
DELETE OBJECT hWebService NO-ERROR.
```

When you run the test procedure, you get connected to the Web service, then see a message confirming that you got a valid handle back, representing the running instance of the service, and then get back the expected output values:



The call count of 1 verifies that my calling procedure and the called procedure are running in separate OpenEdge sessions, so they're looking at different copies of the global variable that's incremented in both places.

In conclusion, although there are a number of steps involved in setting up a procedure to be invoked as a Web service, ProxyGen does most of the work for you, and OpenEdge Explorer lets you easily deploy the services you create on an OpenEdge Web Services Adapter. Once they're set up, any other application running in any language on any platform can invoke them over the Internet. That's the value of our Web services support in OpenEdge.