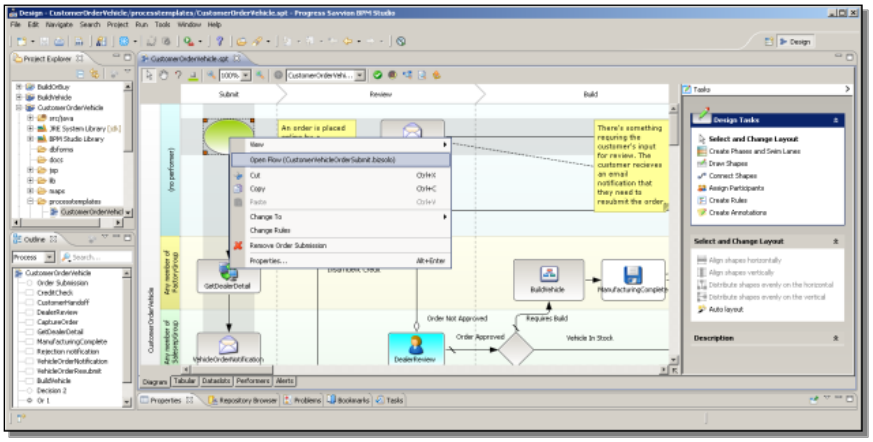
CALLING AN OPENEDGE WEB SERVICE

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
August 2011

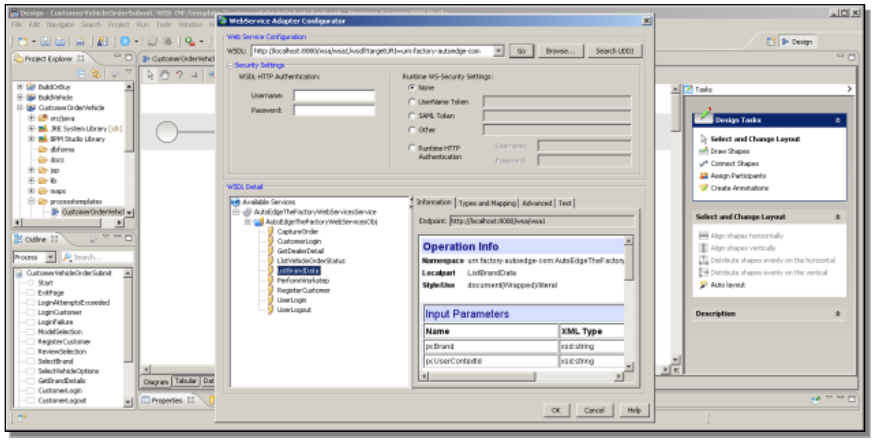
DISCLAIMER

In an earlier video and paper in this OEBPM series, ***Using Application Data in Forms***, I walked through how data gets passed between forms and Web service calls in a Progress Savvion presentation flow, but I postponed actually talking about how to set up the Web service call itself. It that presentation, I was focusing on how to map Dataslot values and form field values to and from parameters to other calls. In this session I'll get into the details of how to construct the Web service call itself out to OpenEdge. If you haven't watched that other video, or read the paper, you should do that before continuing with this one.

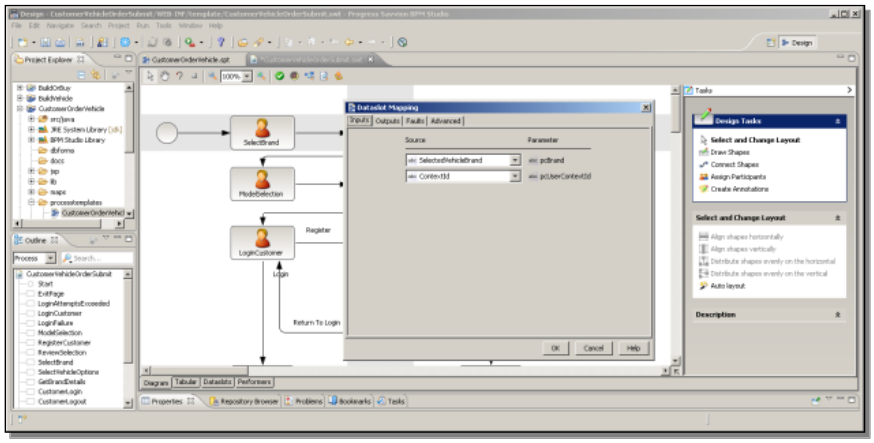
Starting in the main **AutoEdge | The Factory** process, **CustomerOrderVehicle**, if I look at its Start step, I can see once again that it's a Flow, a separate data entry sub-process, and open it:



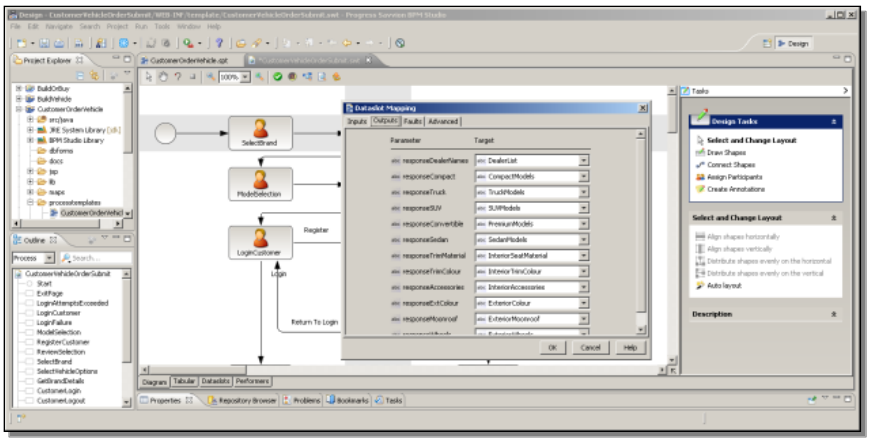
GetBrandDetails is the first workstep in the Flow that's an instance of a Web service adapter. Looking at its properties, and configuration information, I can see the input parameters that we looked at in that earlier presentation.



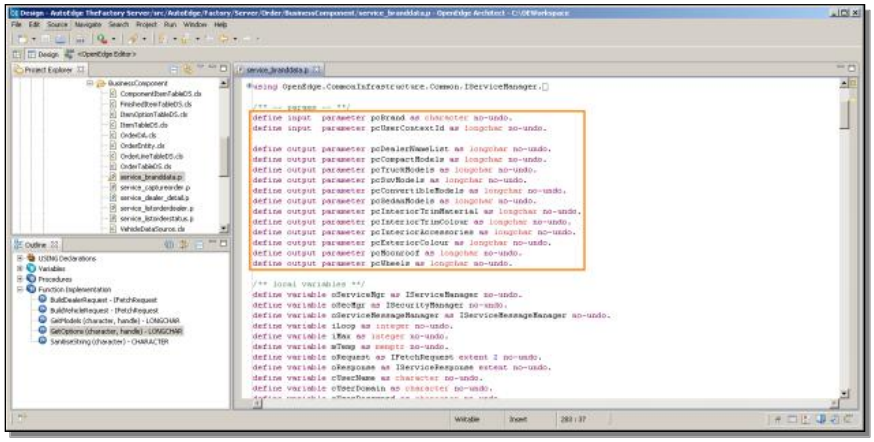
Clicking **OK**, I'm placed into the parameter mapping, for input parameters going into the call:



Selecting the **Outputs** tab shows the output parameters coming back that are mapped back to Savvion Dataslot values:

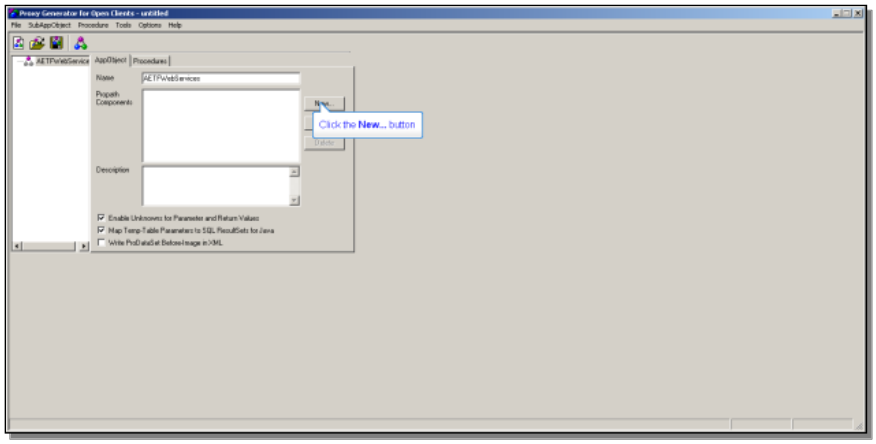


Here is what those parameters look like on the OpenEdge side, in the procedure **service_branddata.p**:

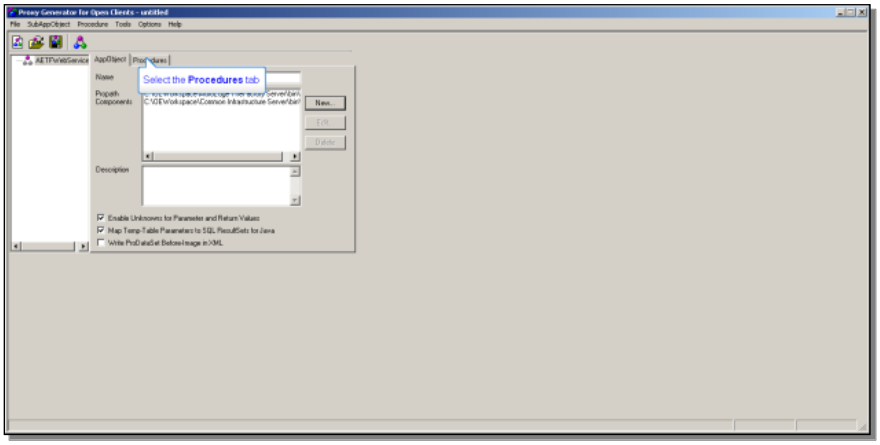


This is what has been set up in the **AutoEdge | The Factory** sample application flow. Now I want to go through how to recreate a piece of this call, so you can see how to do it yourself. Given an ABL procedure that you want to call out to, the first step is to go into ProxyGen to create a Web service proxy for the procedure.

I'll recreate a subset of the Web service call we just looked at, and call it **AETFWebServices**. The way the sample application is put together, all the Web service calls, which represent a number of different ABL procedures, are enclosed in a single proxy, which is one good way to organize them. First I just recreate the Propath components the Web service call requires:

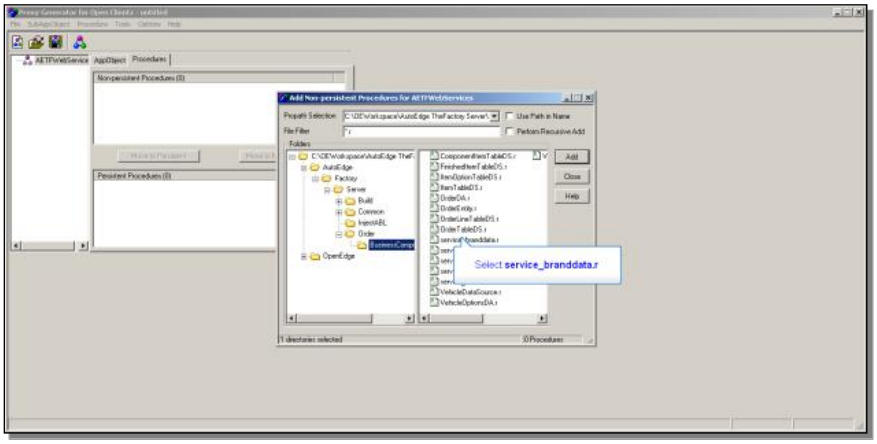


The first is the **bin** directory under the **AutoEdge The Factory Server** folder. The second, in the **OEWorkspace** Architect workspace I've set up, is the folder **Common Infrastructure Server**, and its **bin** directory.

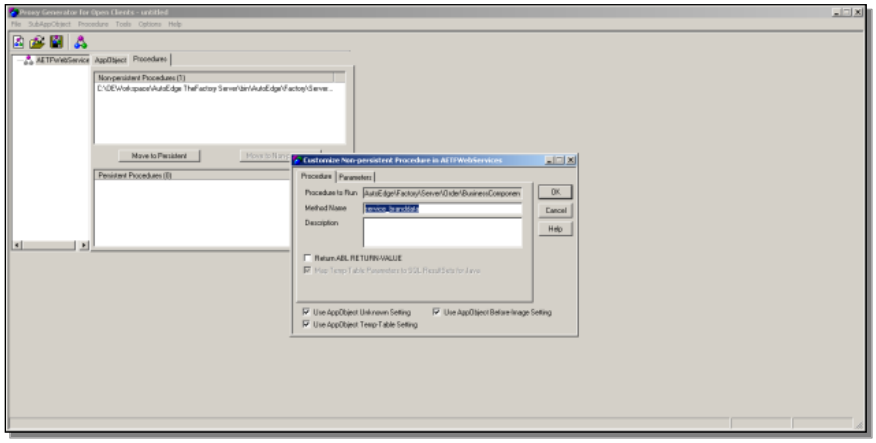


Now it's time to add procedures to the proxy. As I mentioned, the full sample application has a number of ABL procedures in its proxy, but I'll just add the one that is used in the **GetBrandDetails** call.

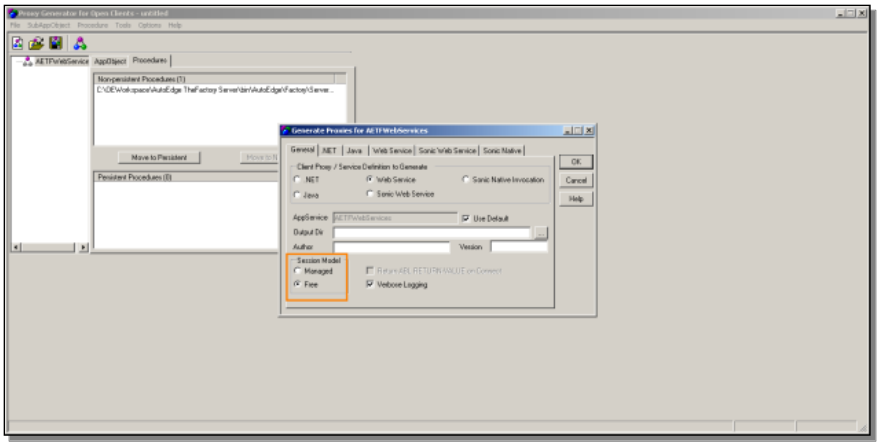
So I right-click, and add a **non-persistent procedure**. This corresponds to the state-free mode the AppServer runs in; I don't want to bind the AppServer by running a persistent procedure from the client. In the code structure for the sample app, it's under the **Factory -> Server -> Order** folder for all the procedures that support ordering a vehicle. Under that, **Business Component**, and the procedure whose parameters I showed you is **service_branddata.p**; to create the proxy you always need to identify the compiled .r file for it:



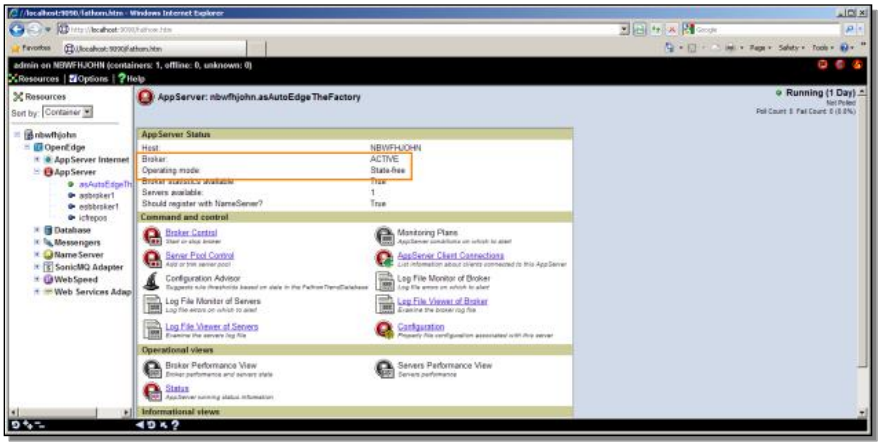
I click **Add** to add that as the one and only procedure in the proxy for now, and then **Close**. Now I'll make an adjustment to it that corresponds to how the Factory sample is organized. Right-clicking on the r-code file I just added, I select the **Customize**… option. One of the things it lets me do is to give the service call a method name that is different from the ABL procedure name, if that makes things more readable or intuitive. I rename it **ListBrandData**.



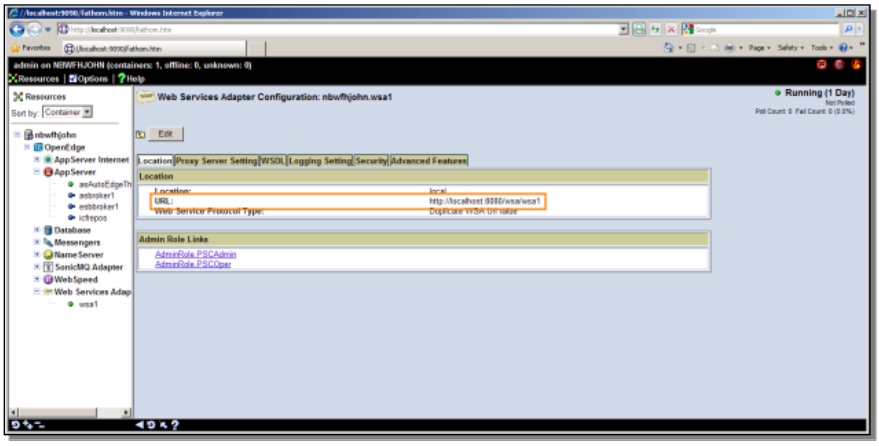
Now I press the **Generate** button, and the **Generate Proxies** dialog comes up to let me define all the particulars. I specify that this is a **Web service** proxy by selecting that radio set option. That enables the **Session Model** radio set, which I make sure is set to **Free**.



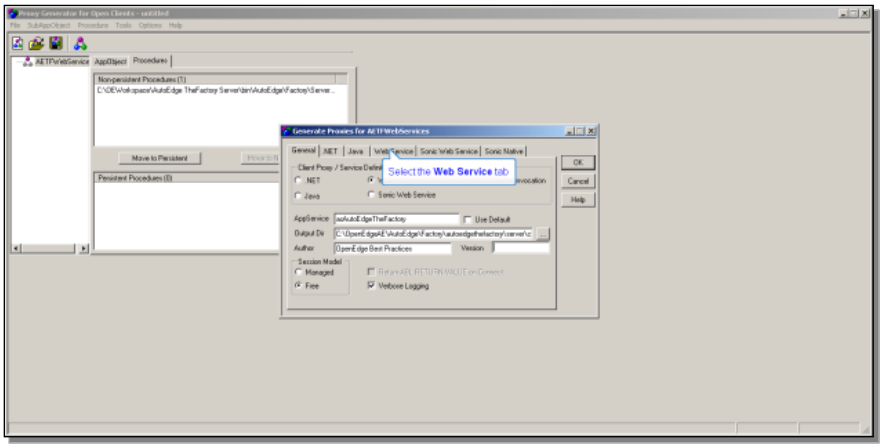
At this point I want to take another look at how things are set up in OpenEdge Explorer. First I'll look at the AppServer the application uses, and confirm that its name is **asAutoEdgeTheFactory**. It's **active** and it is **state-free**, so it's all set to go.



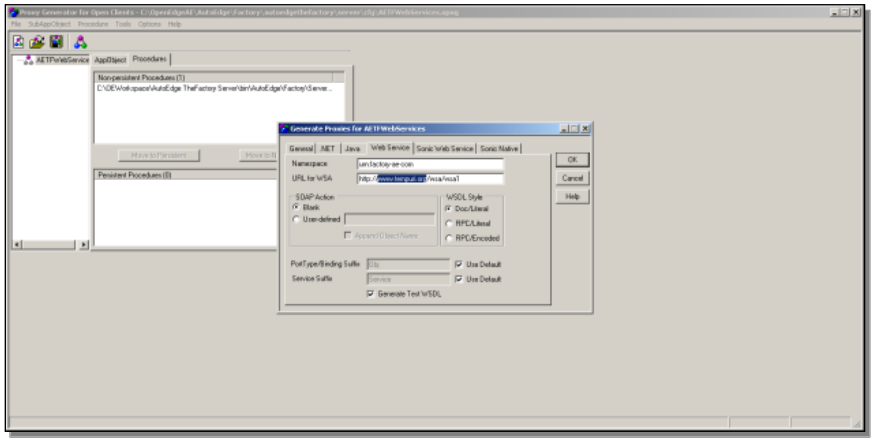
Now I look at the Web services adapter. In this case the application is using the default adapter **wsa1**. If I look at its configuration, I need to note the URL. It's just running on **localhost** in this instance, and because I'm using Tomcat, it's port **8080**.



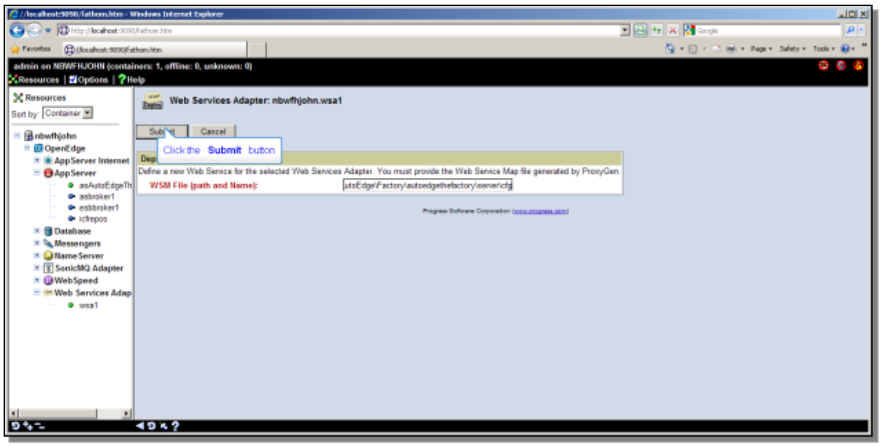
Back in Proxygen, I set the output directory for the proxy to the **autoedgethefactory** server **cfg** directory. I'm not using an AppServer named AETF Web services, which would be the default, so I need to rename that, and enter the AppServer name we just saw in Explorer. I enter an author value for documentation purposes, and indicate that I'm working with version 1.0.4 of the Factory sample application code:



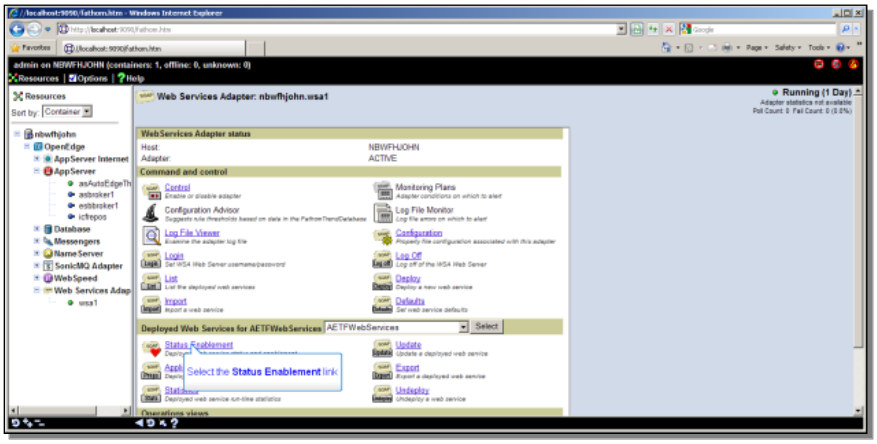
The namespace needs to be a distinctive qualifier for the factory services. In this case I use **factory-ae-com** as that distinguishing value. You'll see in a moment where I use that. The WSA pathname we also saw in Explorer a moment ago. That's **wsa/wsa1** on localhost port 8080.



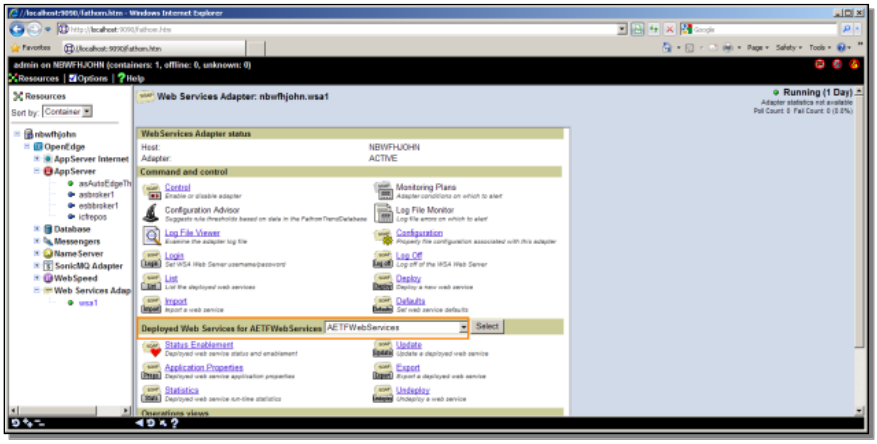
When I'm done defining the Web service proxy, and click **OK**, the proxy is generated into the **cfg** directory I named. Back in Explorer, I need to deploy that service. I enter the name of the **Web Service Map** file that ProxyGen created in the same cfg directory:



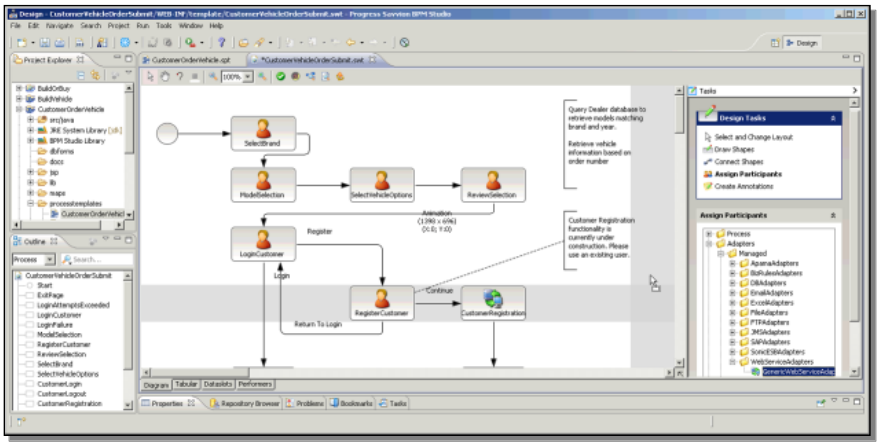
Then I deploy that file. Finally, I have to enable it, so I click **Status Enablement** and click **Enable**.



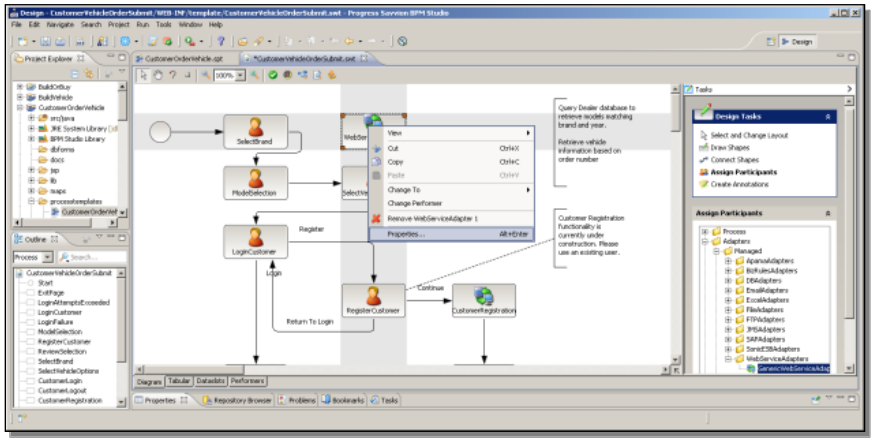
If I go back to the WSA summary, you can see that the Web service is active and deployed:



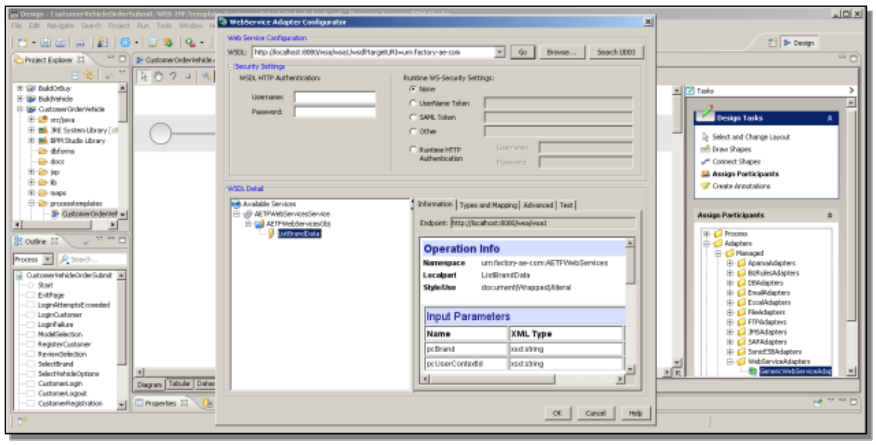
Now I want to recreate a piece of how this service is then enabled for access from BPM Studio. I just delete is the existing Web service adapter in the application and start over. Under **Assign Participants**, I can make the participant an adapter. Under that I select **Managed** adapters, then the **WebService adapter**, and the **Generic** one that comes with the product. I drag one of those onto the process diagram:



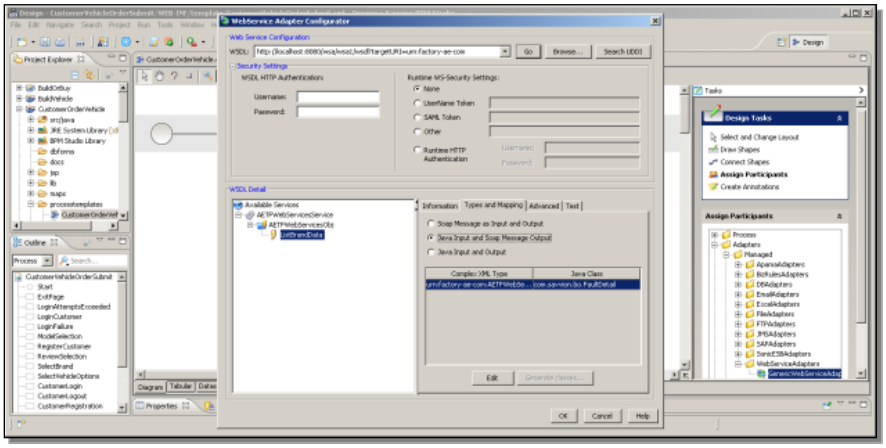
I drop it, and right click on it to set its Properties:



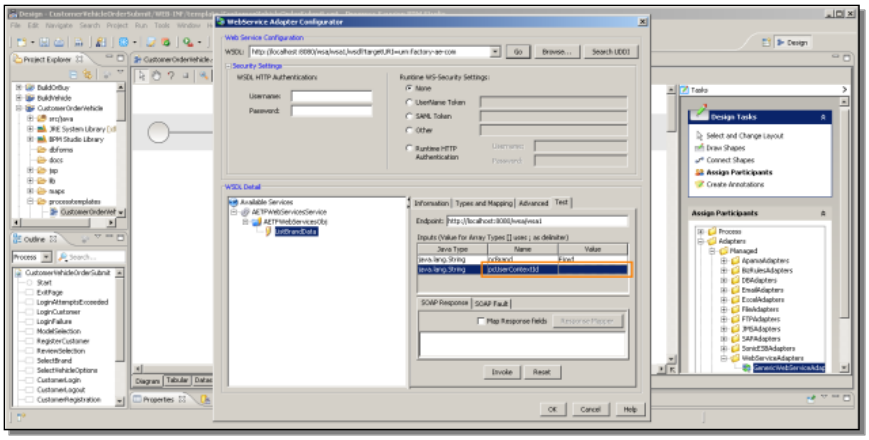
I name this variation **GetJustBrandDetails**, and in its configuration, the identifying information I need to enter here is the **WSDL** address, including the Namespace URN I specified in proxyGen, which allows Savvion to pull in the definition of the procedures in the proxy and their parameters. So when I enter that full WSDL and click **Go**, Savvion displays all the services in the **AETFWebServices** proxy, and for me, there's just one, **ListBrandData**, so I select it.



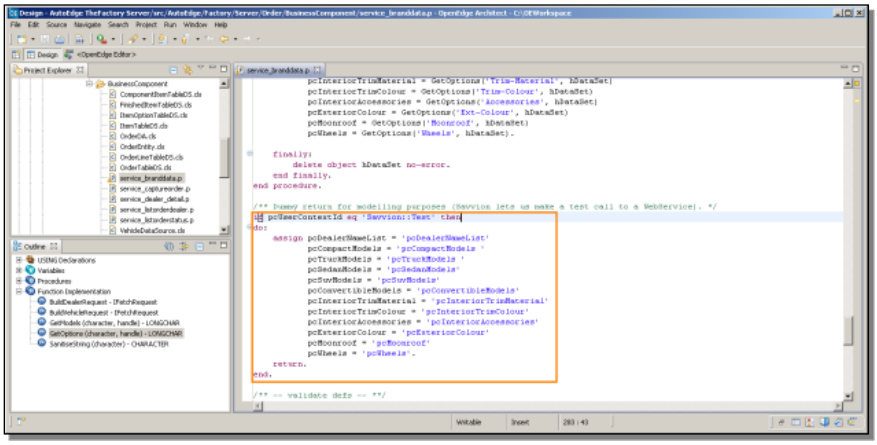
Now I need to map Dataslot values from the process to parameters into and out from the call. I select the **Types and Mapping** tab. The default is that data will be passed in both directions as Java, but in the current releases of the two products this format will not perform correctly. So It's necessary to select the radio set option labeled **Java input and SOAP message output**.



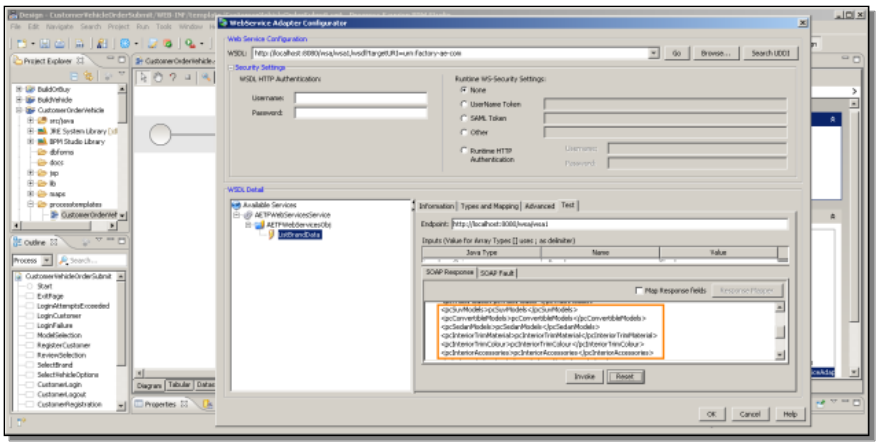
The dialog now displays a list of the input parameters to the Web service call, and lets me enter test values for them. So I enter Fjord as the value for **pcBrand**, and for the **contextID**, which identifies a particular client to the server, I will enter the special value **Savvion::Test**.



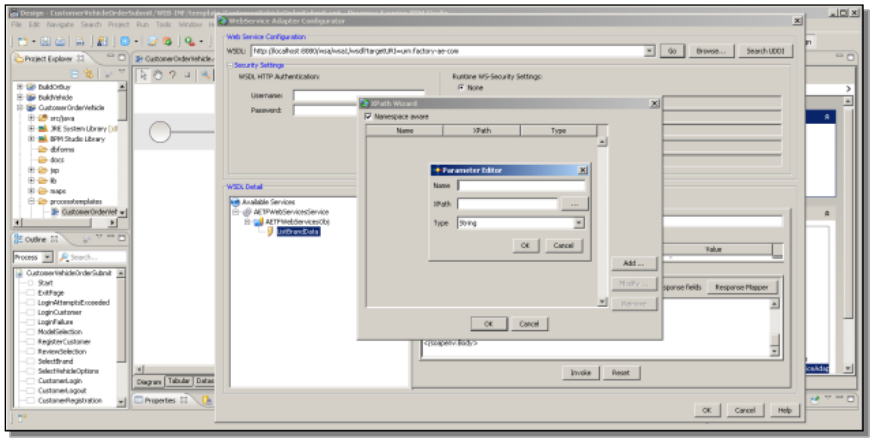
Let me show you why that works. In the ABL code for **service_branddata.p**, there's a block of code that checks for this special value, and if it's been passed in, the procedure constructs a response in which each output parameter is given a value that is actually the name of the parameter:



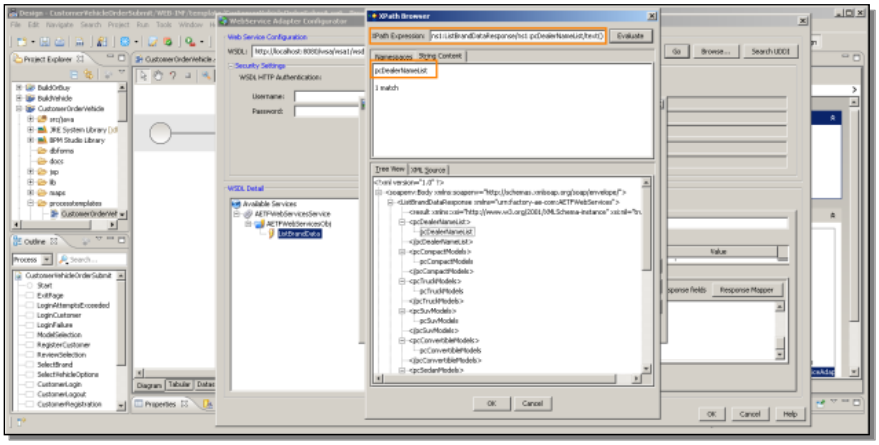
Let's look at how this comes back. To test the Web service call, I click the **Invoke** button. If I expand the dialog box some to see what comes back from the call, and then expand the display pane for the **SOAP Response**, you can see part of the response:



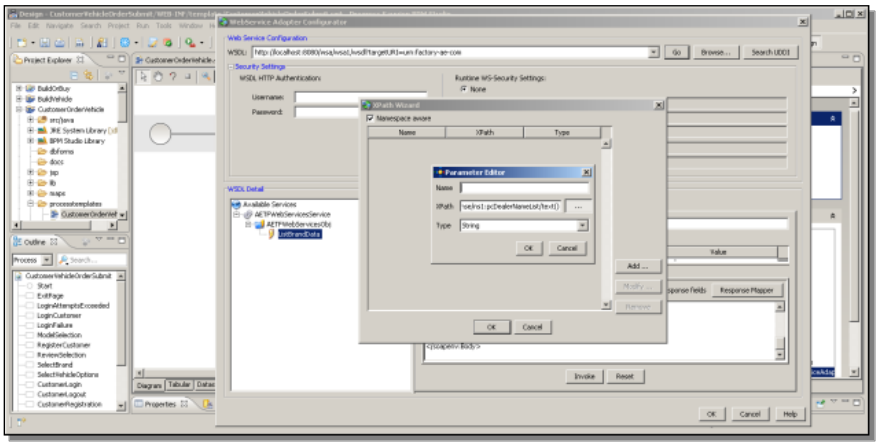
As you can see, it's an XML document in which the value of each node, corresponding to the service's output parameters, has a value that is the name of the node, that is, the name of the parameter. This is an arbitrary construction, but it makes it easy to do the next step in the mapping. Next I need to select the checkbox labeled **Map Response Fields**. Then I click the **Response Mapper** button. This places me into what's called the **XPath wizard**. I click **Add**, and I am able to provide a name, an Xpath expression, and a datatype for a parameter. It's perhaps easiest just to identify a piece of the XML data stream as an XPath expression and then give it a name, so I click on the ellipsis next to the XPath fill-in:



I then select the value for one of the output parameter nodes. This is why it is handy to construct sample data where the data value matches the name of the parameter you want to map it to.



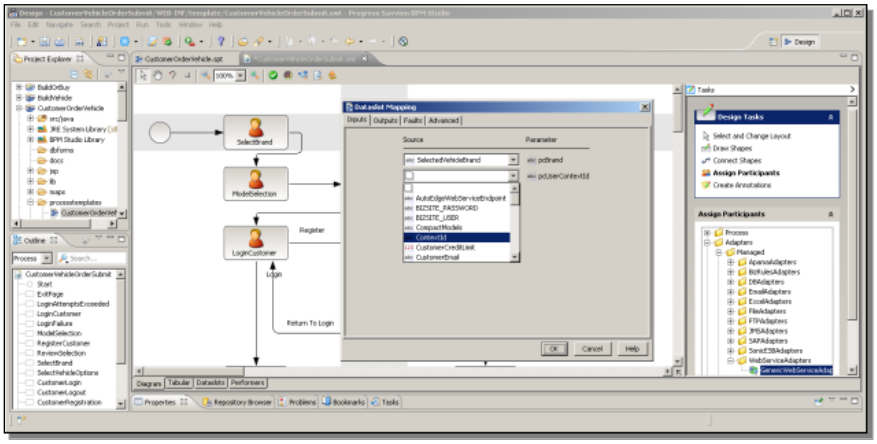
At the top you can see that the mapper has identified the value I selected, and generated an XPath expression that tells the Web service adapter how to locate whatever value occupies the spot where **pcDealerNameList** is in the sample data. I click **OK**, the XPath wizard dialog displays the XPath expression, and now I just need to give it a name and verify the basic datatype:



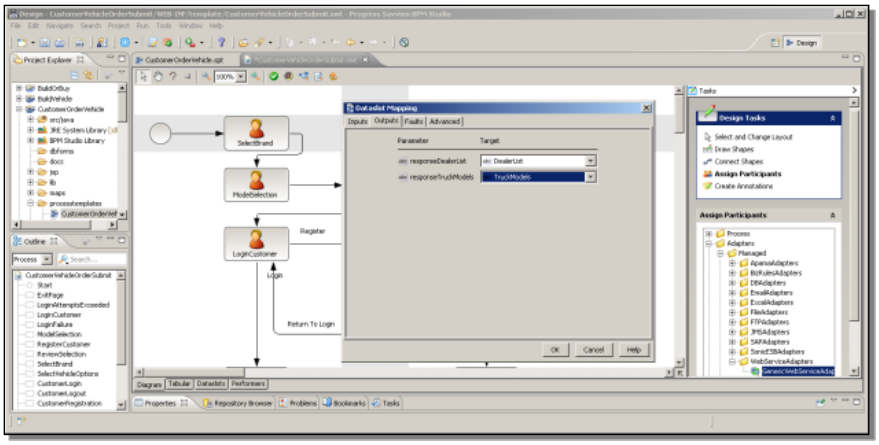
By convention, the Factory application uses names beginning with **response**; you've seen those when I reviewed that parameters for the Factory Web service I'm replacing. However, you can name the parameter anything you want, because the next step will be to map that parameter to a Dataslot in the business process.

I map a second parameter for data used by the next form in the Flow, the one that returns a list of the truck models available for the brand. I give the response parameter a name, and I'm done. And I'm also done with this part of the Web service configuration. When I click **OK**, I'm put into the Dataslot mapping process.

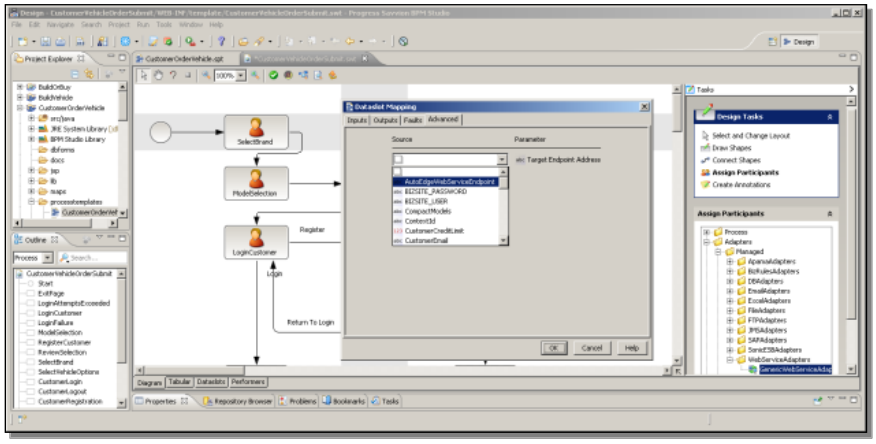
For each input parameter, I need to identify the application dataslot that is going to provide a value for the parameter, going into the Web service call. In this case the first one, mapped to **pcBrand**, is the one called **SelectedVehicleBrand**. This Dataslot value gets set when I pick an icon like the Fjord icon in that first form of the Flow. The second one is the Context ID that identifies a particular client The details of how you might establish that are beyond the scope of this talk, but it's a dataslot just named **ContextID**:



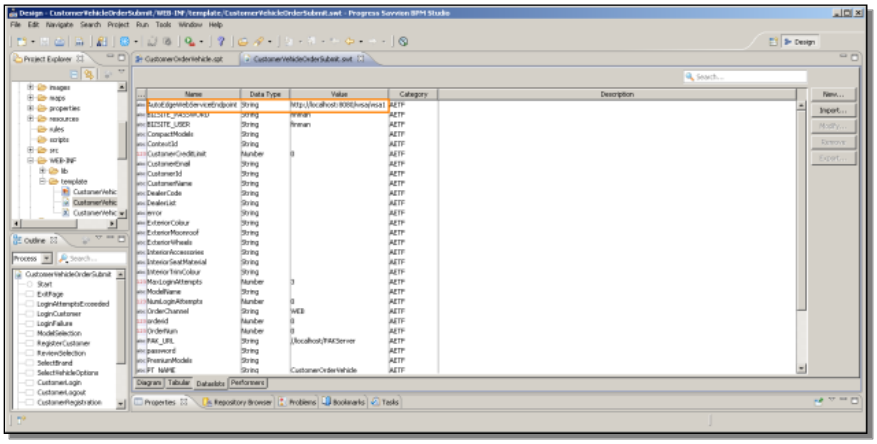
After I'm done with the input parameter mapping, I select the **Outputs** tab. Remember that I just created responses for two out of all the output parameters. The first maps to the **DealerList** Dataslot, and the second one to the **TruckModels** Dataslot:



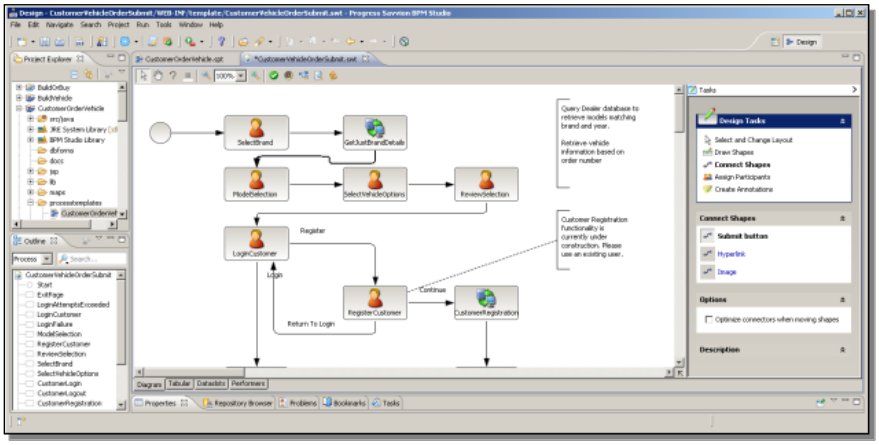
There's one other option you can set here, by selecting the **Advanced** tab. This allows you to map a Dataslot to what is labeled the **Target Endpoint Address**. This means the URL of the web service adapter that is the one actually used at runtime, in production, and lets you make this a value settable at runtime. In the Factory application there's a Dataslot set up to hold that value, called **AutoEdgeWebServiceEndpoint**:



This screenshot shows you that Dataslot to make clear what it is I'm saying here:

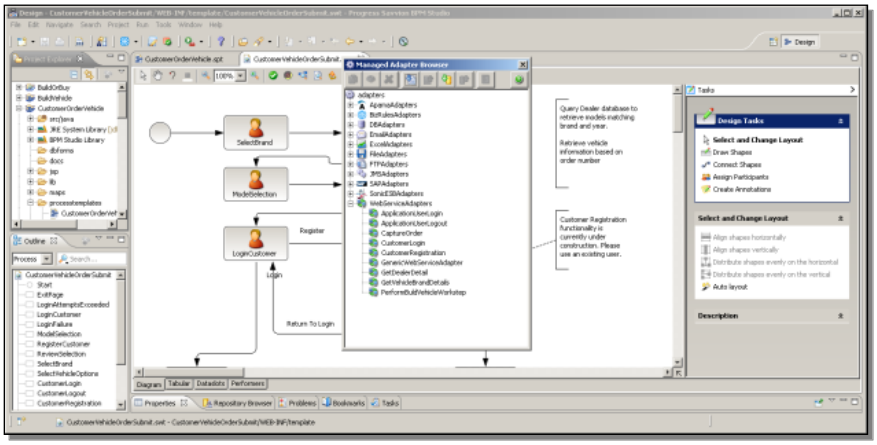


Here you can see the Endpoint Dataslot, which in this case is hard-coded to be the web service adapter that I set up for building and testing the model. If the application allowed this value to be set at runtime, then the endpoint of the actual wsa used would be more flexible. In any case, I'm done with the Web service configuration now. Since I replaced the workstep that represents the Web service call in the Flow, I have to reconnect it back into the diagram. I can select **Connect Shapes** or do **Control-Shift right-click** as I've done before, and re-establish that after the Web service call the **Model Selection** form comes up.



Now I've completed re-configuring this one Web service call to OpenEdge, and I can re-save the model with my new Web service adapter workstep.

I'll just give you one quick tip before I wrap up this presentation. Under the **Tools** menu in BPM Studio, you can select **Managed Adapters**, and use this to import or export Web services you've gone to the trouble of configuring, like this list of adapters for the Factory application, so you can easily reuse them.



Now I'm ready to re-deploy my application, and you're ready to connect up steps in a Progress Savvion business process with services in an ABL application that provide data and business logic support for your Savvion business process application.