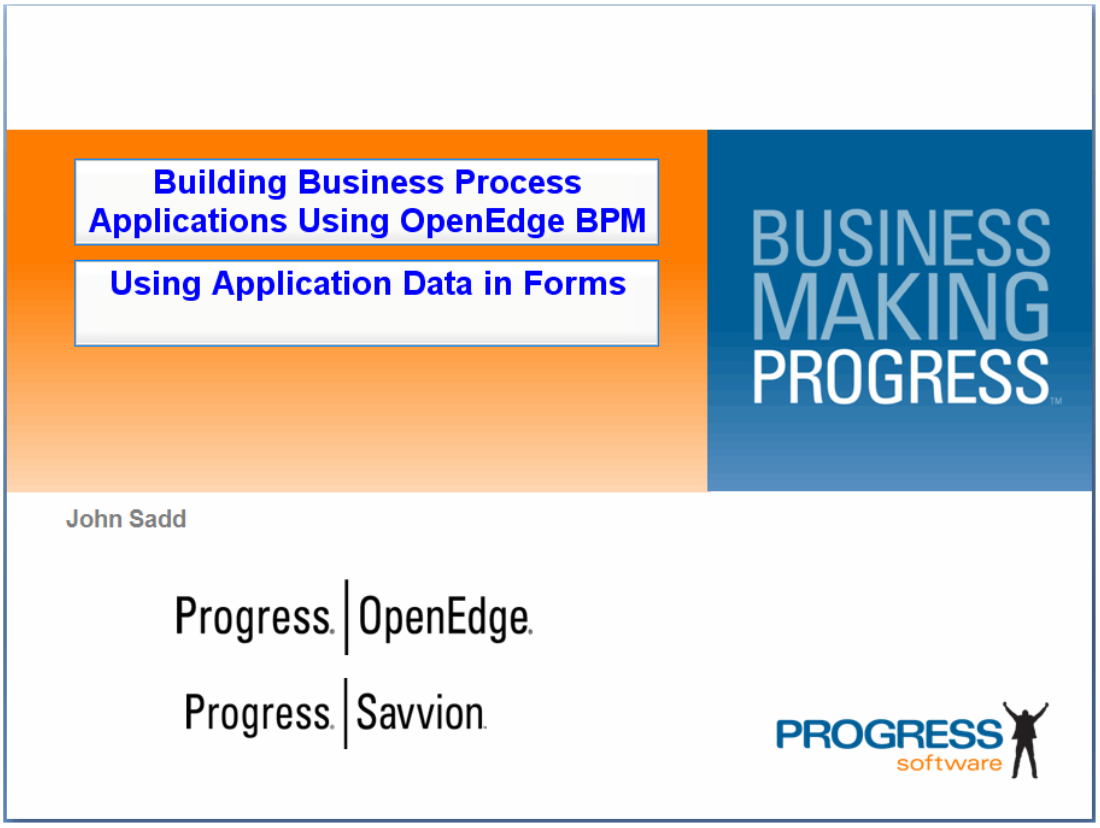
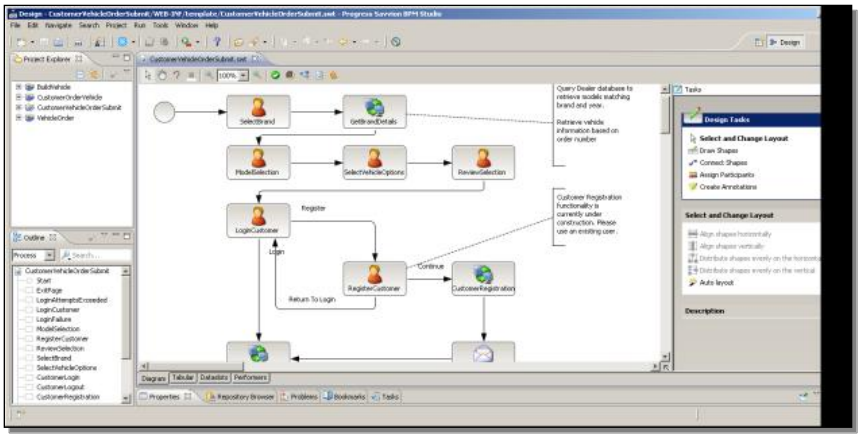
John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
July 2011



DISCLAIMER

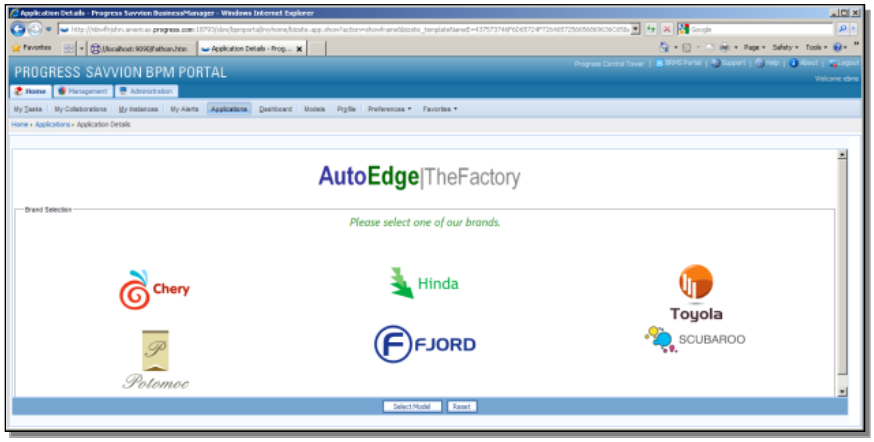
This document accompanies the two-part video which introduces you to the basics of how to populate fields in your forms with data retrieved from an application. The details on how to construct the actual calls out to the application are covered in a later session on using Web services. Here I just show how Savvion's controls can be assigned values at runtime, and how you can construct a flow that alternates between forms in the presentation and the calls that get data for them and pass the data from one form to the next.

The model below shows a sequence of steps in the Factory application that has been used before in videos and papers in this series:

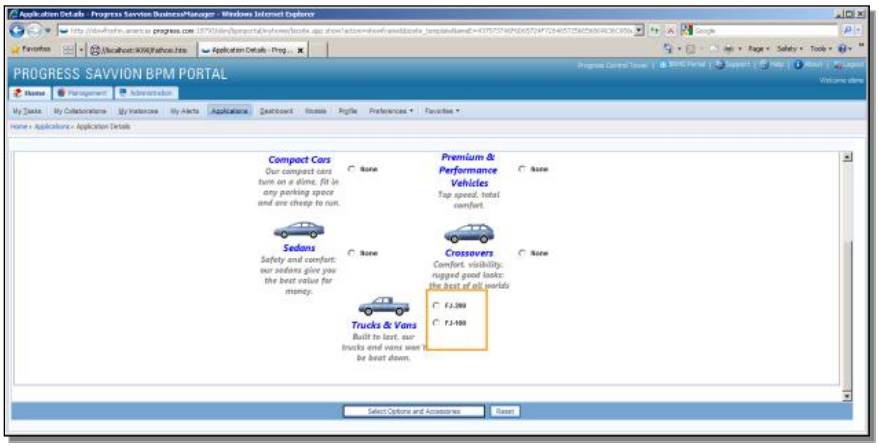


It's called a **Presentation Flow**, a special type of sub-process of a larger process that represents a series of forms that an individual user goes through to accomplish one of the major steps in the Savvion process, in this case, a customer placing an order for a vehicle.

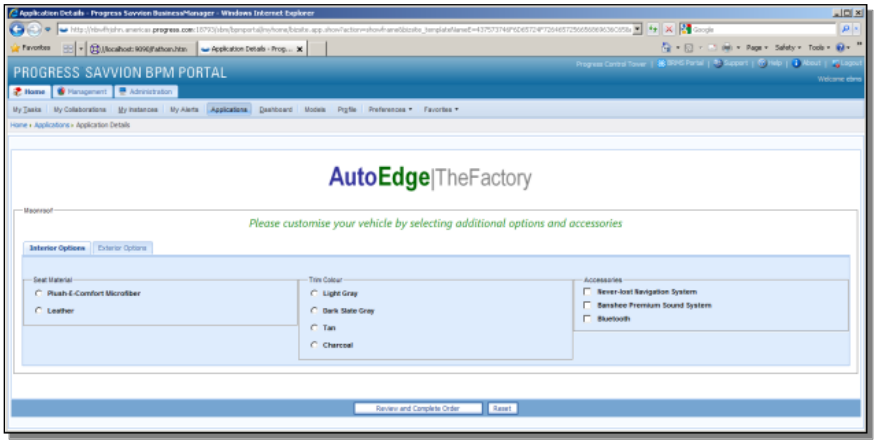
Just to review how this looks when this part of the process runs, I can start up an instance of the **CustomerOrderVehicle** process, and in the first form that comes up, the customer selects a brand:



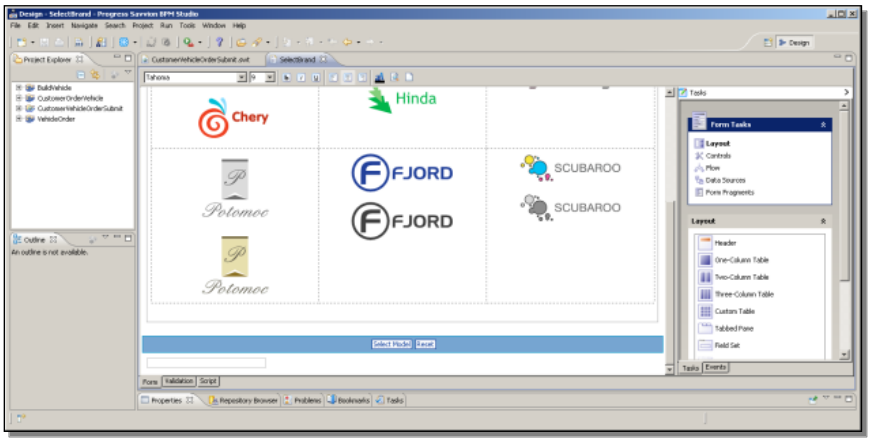
When the brand is selected, all the other brands are effectively grayed out. In the next step the user sees all the available models for the selected brand. (As you can see, there isn't much data in the sample database for some models yet.)



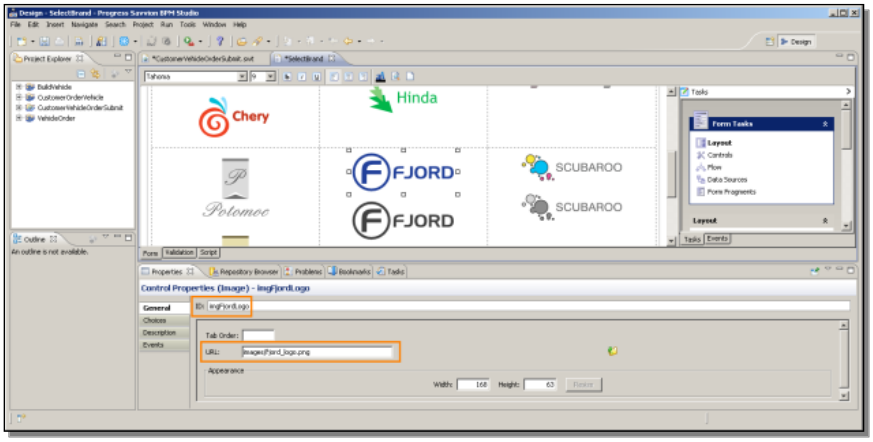
For truck models in this case I just have two choices. This is the key to what I'm covering in this paper: Where did these two values for Fjord truck models come from, and how did the process developer use them to assign the values for this radio set? That's the question this paper answers. Continuing through the Flow, the customer selects a model and proceeds, and goes on to the next step of selecting options, which are also dynamically populated from data retrieved through the OpenEdge application that is executing in the background:



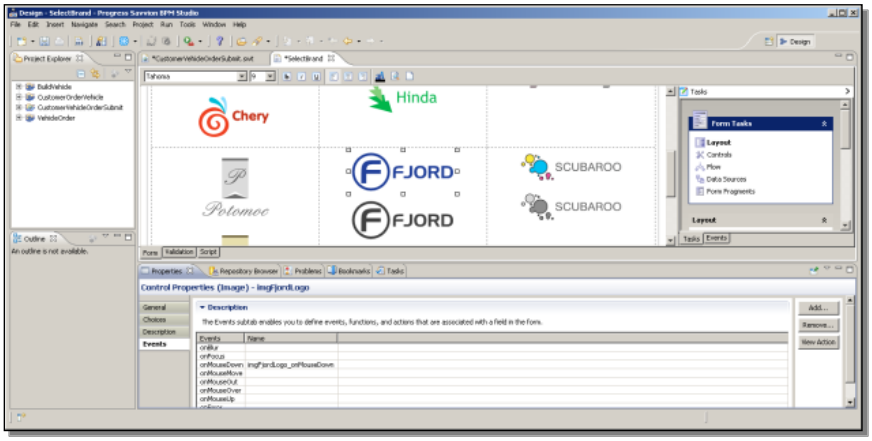
That's as far as I need to go in the Presentation Flow. Let me drill down into a few of the design details. The first step that has a visible form is called **SelectBrand**. I can open the form for that step, and take a look at the images in the form. You can see that there are actually two images for each brand, one color, and one black and white:



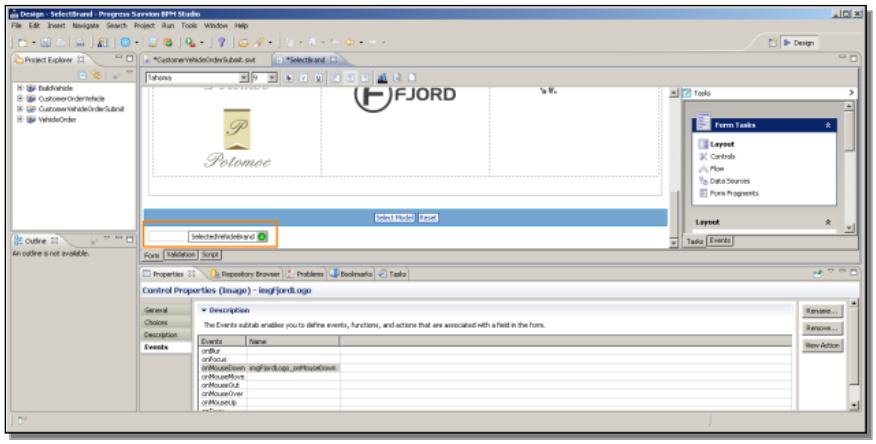
If I select the color image for Fjord, I can drag the Properties view back up to see what's defined for it:



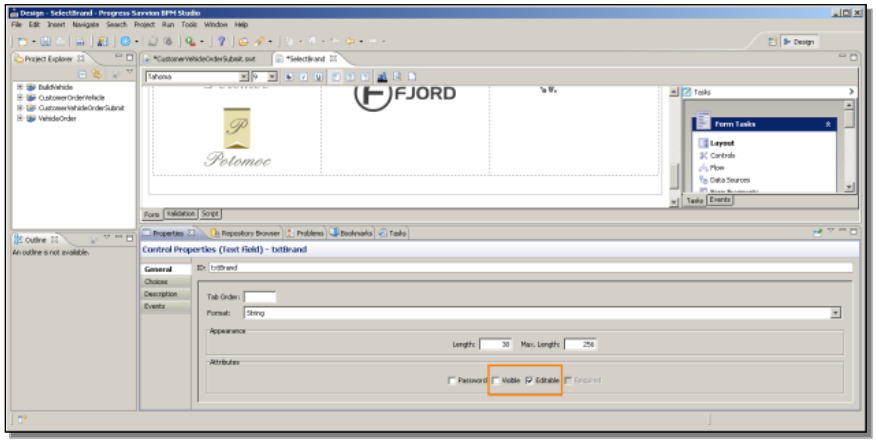
The **ID** is the name of the image control in the form. The **URL** shows the path to the actual file name for the image. What's new here is that there is an event defined for the image, so the image is really functioning as a button control. Selecting the **Events** tab in the properties, you see that there's an event defined for **onMouseDown**, the click event:



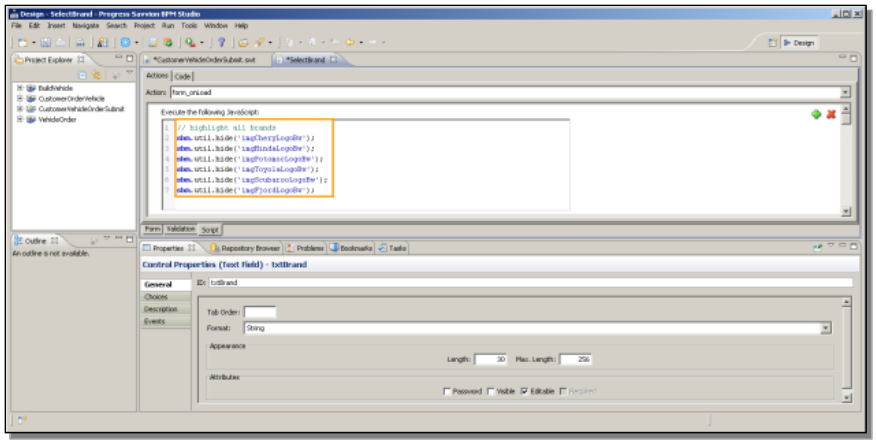
Before looking at the code for that event, there is another interesting element on the form, a text field under the footer:



It is bound to the Dataslot called **SelectedVehicleBrand**. The field didn't actually appear before when I ran the process. If I select the field and look at its properties, you can see that it's called **txtBrand**, and it's a **String** field. You can also see that its **Visible** property is off:



This is a hidden field, set in the background to hold a value that gets passed from form to form, which the user doesn't see. Let's look at the code for the form now by selecting the **Script** tab. The code that gets run when the form is first loaded is the **onLoad** event.

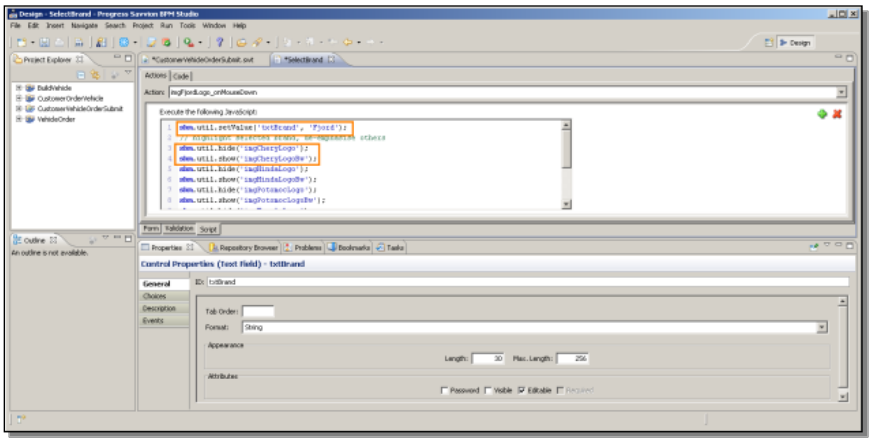


You can see that the code – and this is JavaScript – is executing a **hide** function that's in a library called **sbm.util**:

```
// highlight all brands
sbm.util.hide('imgCheryLogoBw');
sbm.util.hide('imgHindaLogoBw');
sbm.util.hide('imgPotomocLogoBw');
sbm.util.hide('imgToyolaLogoBw');
sbm.util.hide('imgScubarooLogoBw');
sbm.util.hide('imgFjordLogoBw');
```

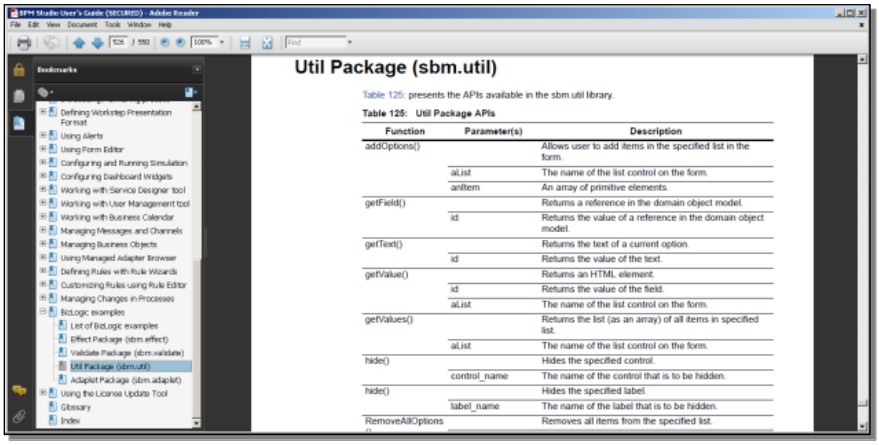
There's a color and a black and white image for each brand, and this little block of code is hiding the black and white images when the form first comes up. Now look at the code that gets run when the customer clicks on one of the visible color images, like the Fjord image:

```
sbm.util.setValue('txtBrand', 'Fjord');
// highlight selected brand, de-emphasise others
sbm.util.hide('imgCheryLogo');
sbm.util.show('imgCheryLogoBw');
sbm.util.hide('imgHindaLogo');
sbm.util.show('imgHindaLogoBw');
sbm.util.hide('imgPotomocLogo');
sbm.util.show('imgPotomocLogoBw');
sbm.util.hide('imgToyolaLogo');
sbm.util.show('imgToyolaLogoBw');
sbm.util.show('imgFjordLogo');
sbm.util.hide('imgFjordLogoBw');
sbm.util.hide('imgScubarooLogo');
sbm.util.show('imgScubarooLogoBw');
```

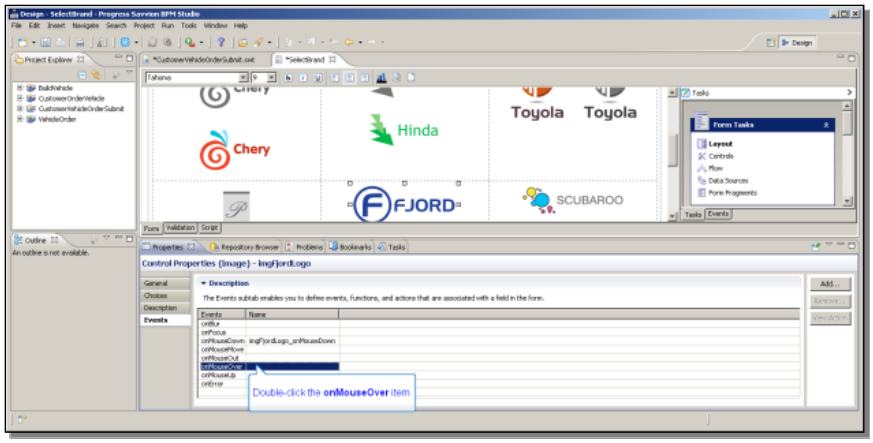


Remember that the form has a hidden field at the bottom named txtBrand. The first highlighted line is setting that field to **Fjord**, in preparation for passing that value on to the next step in the process. Then it hides all the color images for the other brands and shows their black and white images instead. Even though the design window for the form shows the color and black and white images stacked above one another, at runtime the form layout is done dynamically based on what's visible, so the black and white images just appear in place of the color ones, and there's no flashing or jumping around of the display.

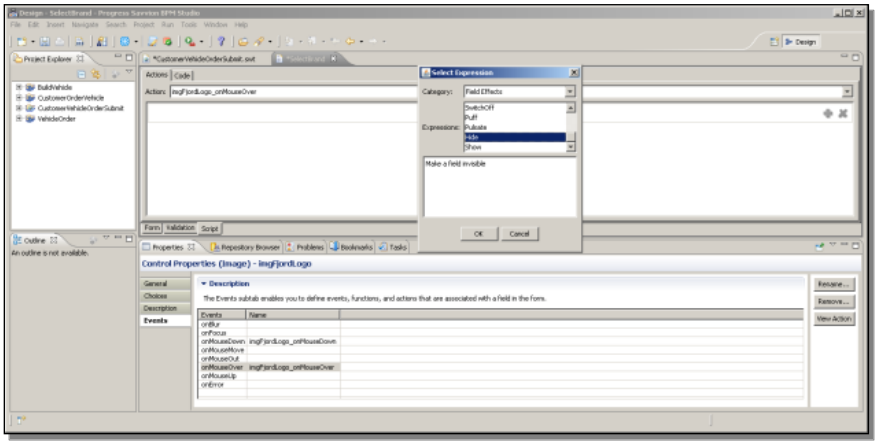
Below is a screenshot of the **BPM Studio Users Guide** in the documentation for Savvion. Just to make sure you know where to look to find all of the built-in function libraries,I can select the **Util** package down in the appendices at the end of the book, and there you will find a summary of what all the **sbm.util** functions do. All of this information is available for you when you start to write event handlers of your own:



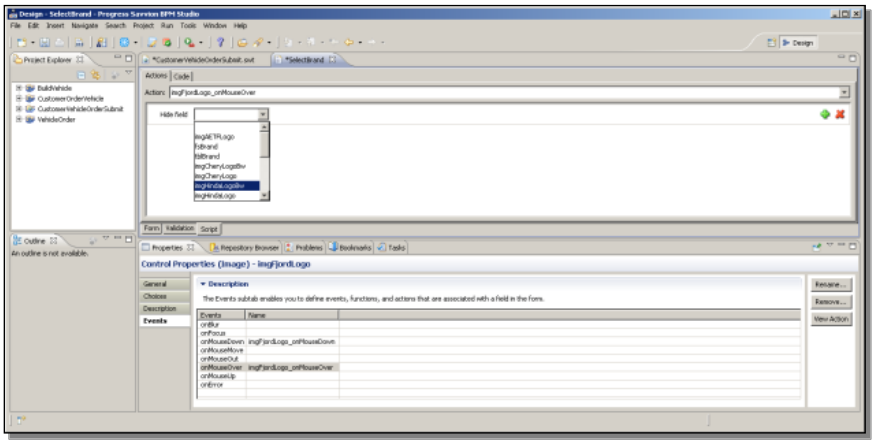
Back in the design window for the form, I want to give you a quick example of how you can put event handlers like this together yourself. Using the color Fjord icon again, I go back into the events for it, and this time I define a **mouseOver** event. I do that by double-clicking on the event:



I get a default event handler name that I can accept or rename as I wish. Then I press the **View Action** button. Since there's no event handler code for this event yet, I just see a message that says, "**Click here to add action**", so I do that. Immediately you can see that there are many types of actions that you can define handlers for. The **Select Expression** dialog shown below is there to help you define common actions in event handlers without having to write any code at all. In addition to **Data Operations** like setting or copying a field value, there are expressions you can define for the effects supported for a field, for a widget, and so forth. I select **Field Effects**, and from the different effects an object like the image might support, I select **Hide**.



Now the event handler UI prompts me to pick a control, in this case any of the controls on the form, so I select the black and white Hinda Logo.



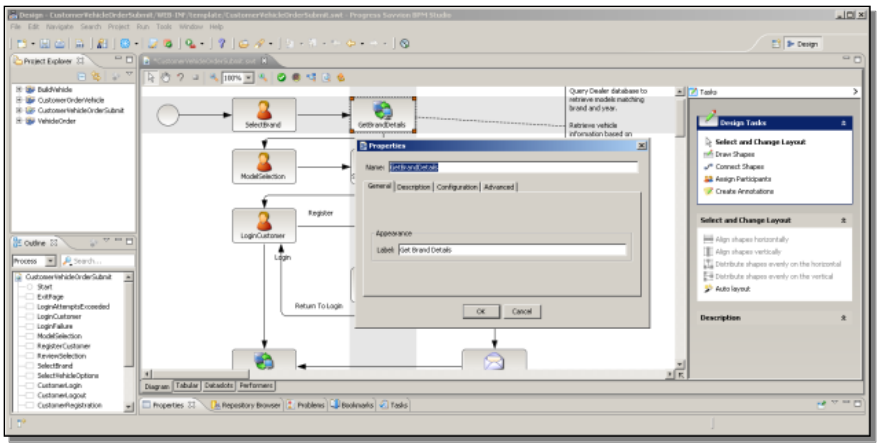
The result is that when I select the Fjord brand, and its color logo is displayed and all the black and white logos are shown for other brands, and then I mouse over the Fjord logo, the Hinda logo will disappear.

Now to make it come back, I define a similar handler for **mouseOut** of the Fjord image. Instead of Hide, this time I select the **Show** action, so when I mouse out of the Fjord image, the Hinda logo will appear again.

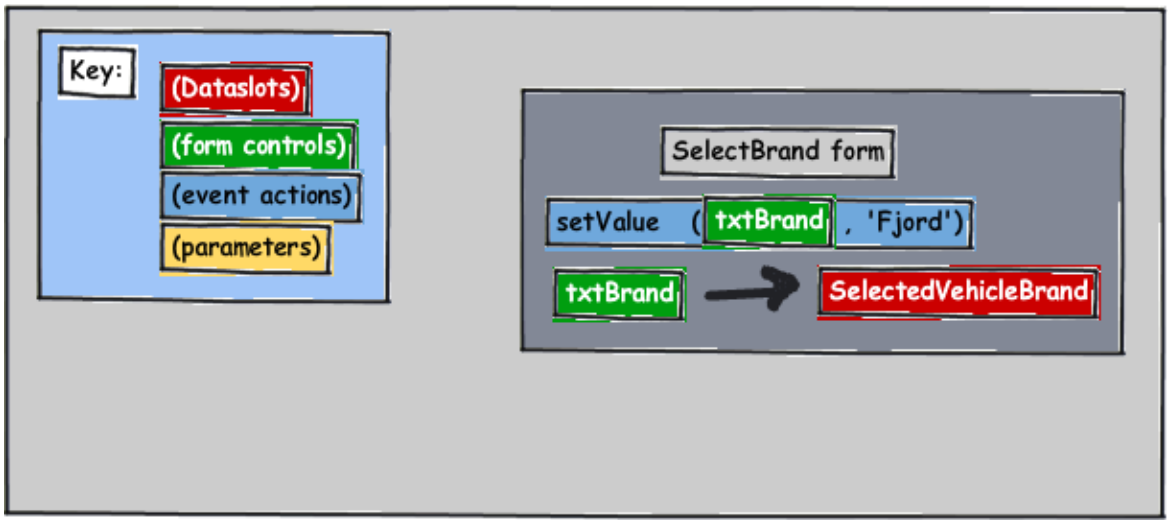
Now I can go back to the form, and close it to save the changes I just made, and then save the Flow that it's a part of. As I mentioned before, the flow is a special type of subprocess that gets deployed to the server on its own. I click the **Deployment** button to deploy the flow. Once I do that and start up another application instance, I can show the effect of the new event handlers I just defined. Once I get to the Select Brand form, and select the Fjord button, mousing over that image makes the Hinda logo disappear and then reappear.

This quick demo of how you can define event handlers of your own shows you that the design interface provides a lot of guidance for you so that in many cases you don't actually have to write any code at all.

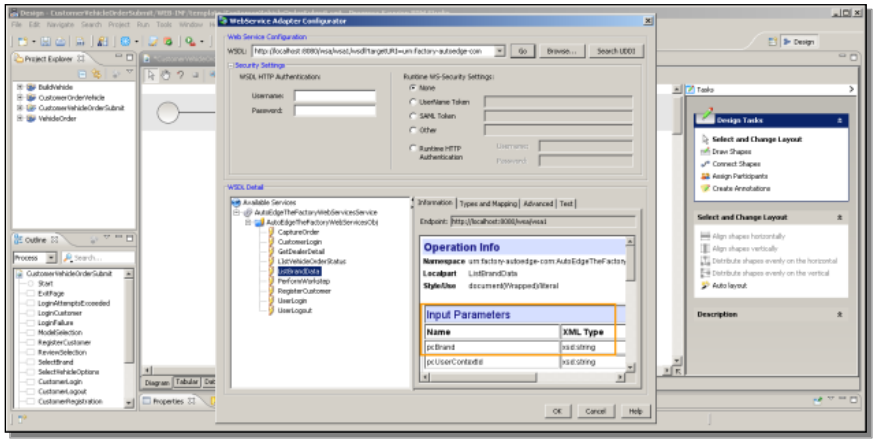
Now take a look at the next step in the flow, called **GetBrandDetails**. This one doesn't have an image of a person on it in the process model diagram, so it's not a step executed by a person. In its properties, I see the **Configuration** tab; clicking on that will tell me what kind of step it is.



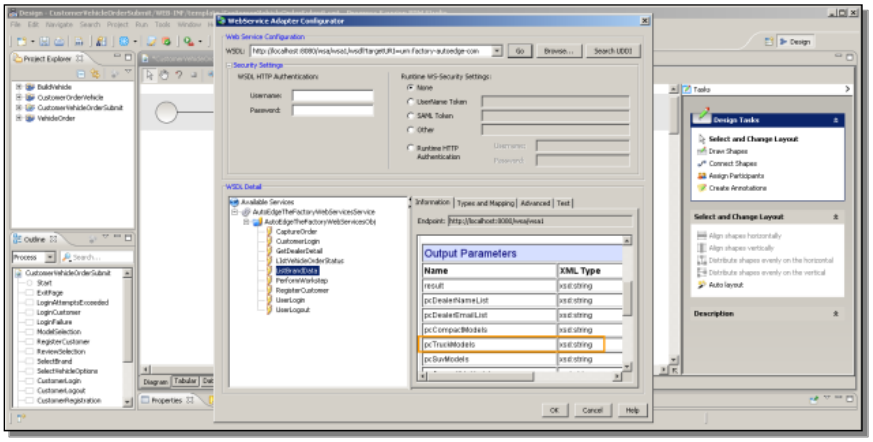
First let me review how I got here. The first form in the flow has an action that sets a hidden field called **txtBrand** to the value Fjord if I click on the Fjord image. Then it binds that variable to the Dataslot **SelectedVehicleBrand**:



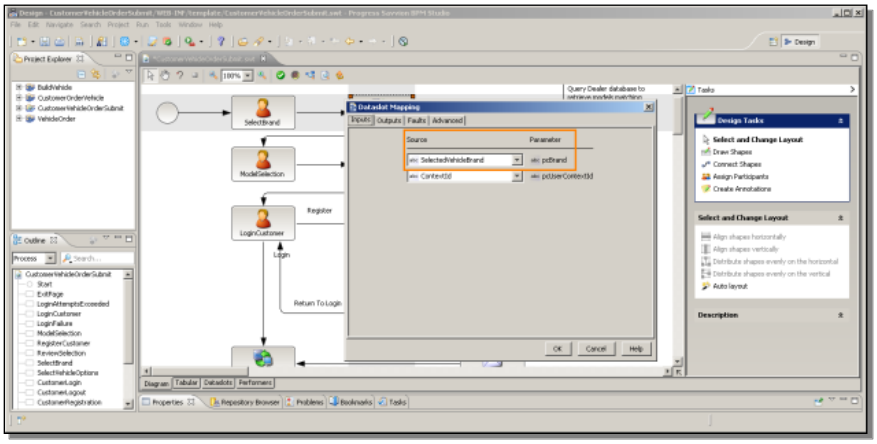
Now let's look at the **Configuration** tab to see what this step does. Clicking the **Configure** button, I see that this step is defined as a Web service adapter, in this case, one that calls out to an entry point in an OpenEdge application. One of the input parameters is called **pcBrand**:



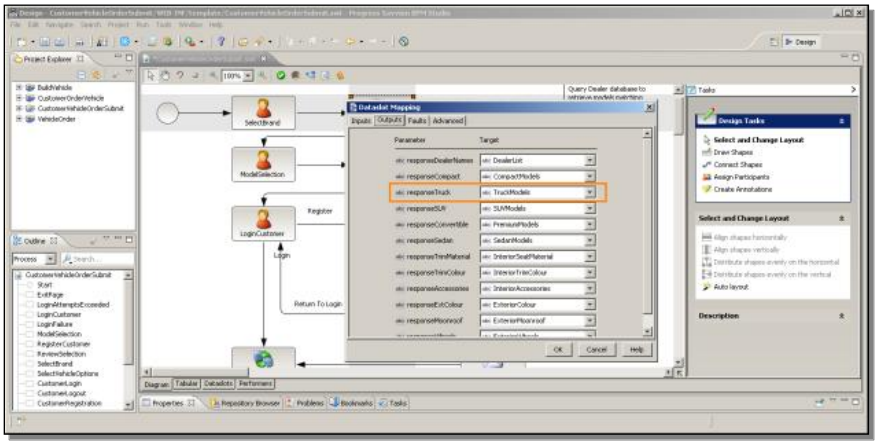
If I look at the output parameters, I see a number of parameters including one called **pcTruckModels**, which is where the list of Fjord truck models like **FJ-100** comes from:



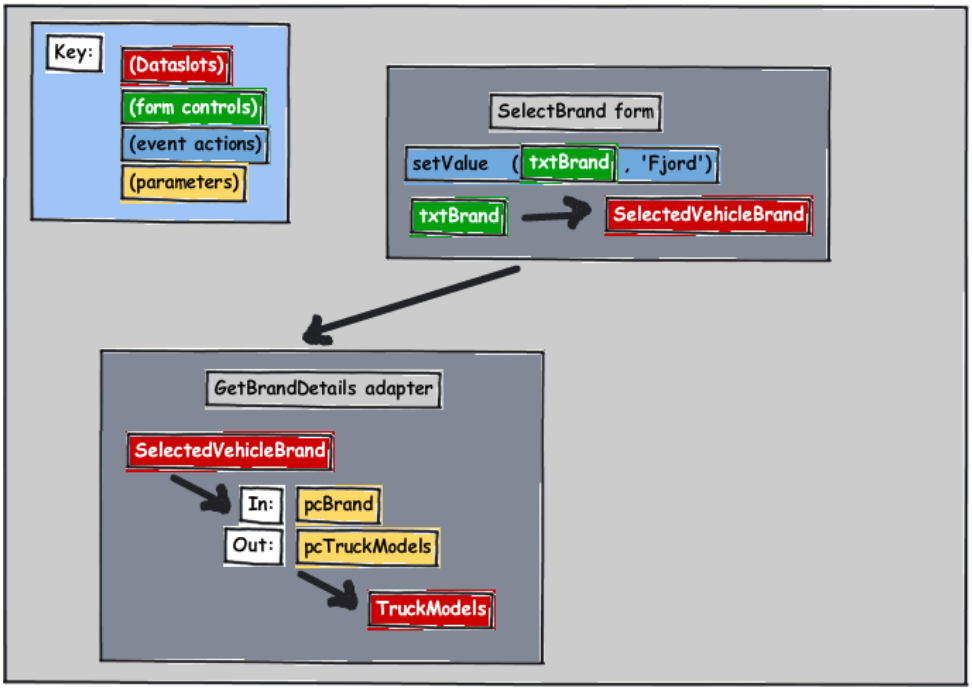
When I OK this dialog, I'm placed into the **Dataslot mapping**. This is where I'm able to pass values from one form to this step through variables and dataslots, and from here on to the next step as well:



Remember that the **SelectBrand** step has an **onChoose** event that places the brand represented by an image into the hidden field called **txtBrand**. You also saw that txtBrand is bound to a dataslot called **SelectedVehicleBrand**. This is where the process is using that value. In this step, it takes the Dataslot value, which is available to all steps in the process, and maps that to the input parameter **pcBrand** that goes out to the web service call:



For reasons I won't go into in this paper, the output parameters from the service come back with names beginning with **response**, so for example, **responseTruck** is the output from the OpenEdge application call with all the available truck models for the selected brand that was passed in. In the screenshot above you can see that is is mapped in turn to a Target dataslot called **TruckModels**. This diagram will help review the steps in the process:



The **SelectBrand** form set the **SelectedVehicleBrand** Dataslot. Now the Web service adapter is using that value to set the input parameter to a call out to

OpenEdge that returns a list of truck models, among other things, and that in turn sets another Dataslot called **TruckModels**. Take a look at just a bit of the actual ABL code that's being called here. On the OpenEdge side, it's a procedure called **service_branddata.p**, which is part of the Web service proxy that provides access to the ABL procedures from Savvion. You can see the **pcBrand** input parameter along with the **contextID**, as well as all the output parameters with various kinds of information for whatever brand was passed in:

```
define input  parameter pcBrand as character no-undo.
define input  parameter pcUserContextId as longchar no-undo.

define output parameter pcDealerNameList as longchar no-undo.
define output parameter pcCompactModels as longchar no-undo.
define output parameter pcTruckModels as longchar no-undo.
define output parameter pcSuvModels as longchar no-undo.
```

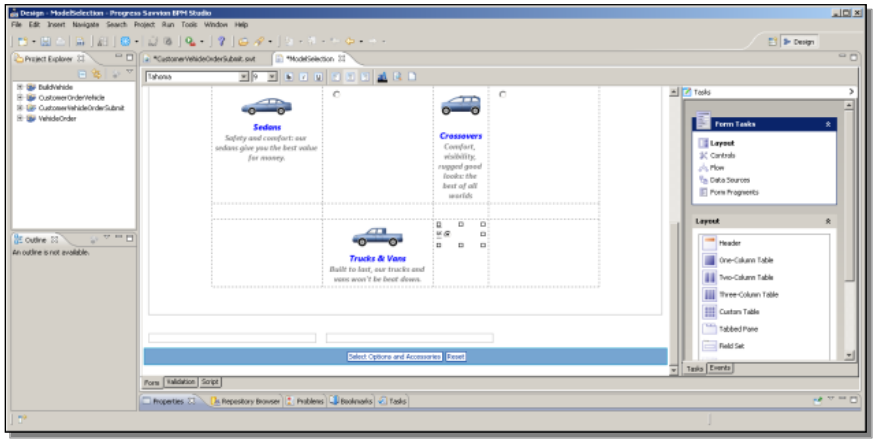
Later in the same procedure is code where the data in the ProDataSet that holds all the values for the brand is converted into a JSON object with **selected** and **value** and **label** properties, which is what a Savvion widget like the radio set expects:

```
    cOptions = ''.
    hQuery:get-first().
    do while hBuffer:available:
            cOptions = cOptions + ', ~{ '
                     + '~"selected~" : false, '
                     + '~"value~" : ~"' + SanitiseString(hBuffer::ItemId) + '~", '
                     + '~"label~" : ~"' + SanitiseString(hBuffer::Description) +
                       '~"'
                     + ' ~}'.
        hQuery:get-next().
    end.
```

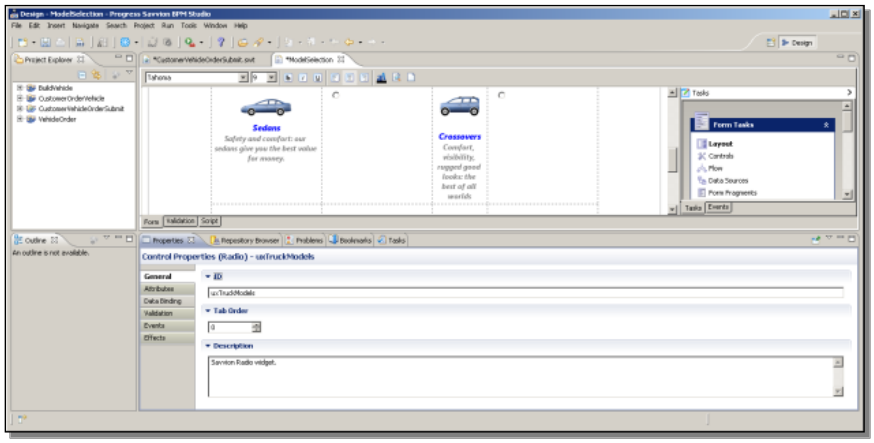
The **value** property in the JSON example below (truncated from its full length) represents the unique character string key that the OpenEdge application uses to identify each row in the database:

```
<json>
[ { "selected" : false, "value" : "fd1603…", "label" : "FJ-200" },
  { "selected" : false, "value" : "fd1604…", "label" : "FJ-100" }
]
</json>
```

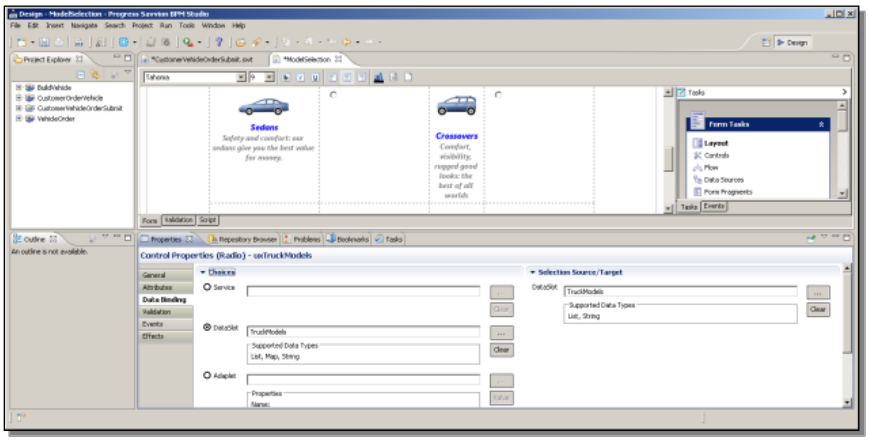
To continue, I open the next user-driven step in the process, called **ModelSelection**, and open its form. This is the form where model names are displayed, and here under Truck Models, take a look at the control. It is bound to the **TruckModels** dataslot, which was populated by output values from the Web service in the previous step.



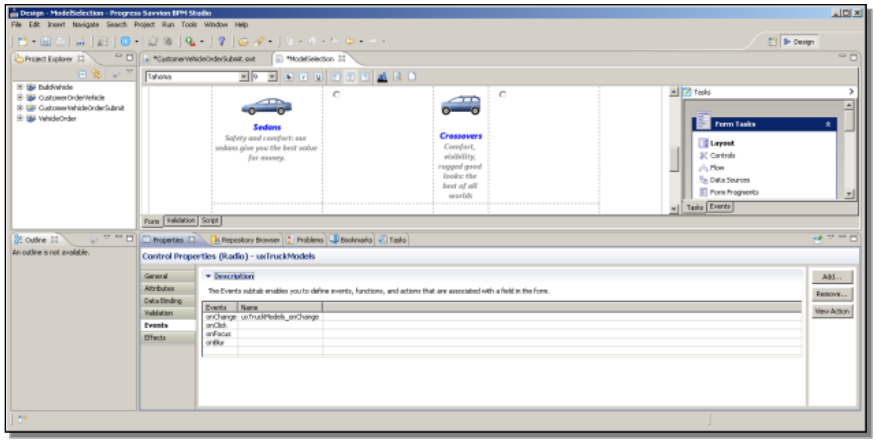
Looking at its properties, you see that its name is **uxTruckModels**, and it's a radio set widget:



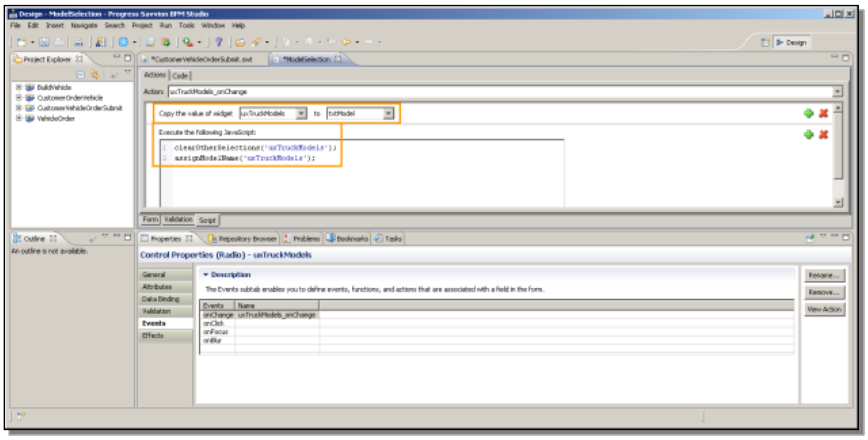
Under **Data Binding**, you can see that the radio set values are bound to the **TruckModels** Dataslot. When this form is realized, this radio set is populated with the values from TruckModels, which in turn was bound to an output parameter from the Web service call:



If I now look at the control's **Events** tab, there's an **onChange** event:



Selecting one of the radio set values fires this event. Selecting **View Action** again shows two actions defined:



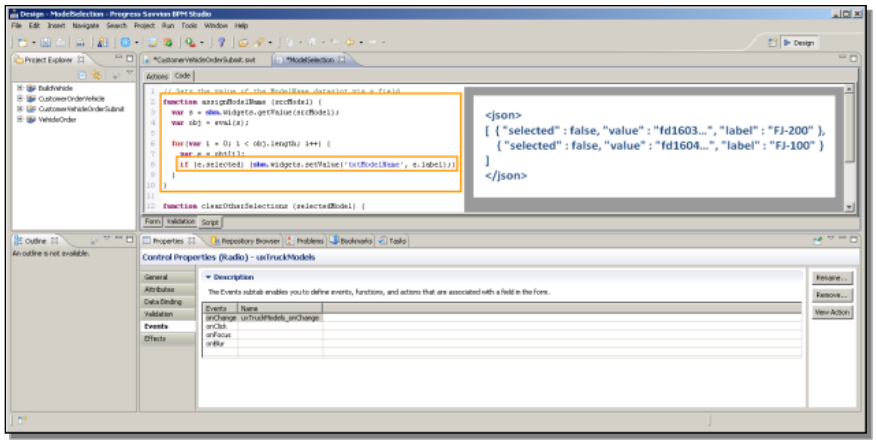
The first, highlighted as "Copy the value of widget…", defined through the kind of dropdowns and expression choices I showed you earlier, tells Savvion to copy the value of the **uxTruckModels** control to another variable called **txtModel**.

The second action is actually a bit of handwritten JavaScript code, executing two user-defined functions, both of which take the chosen radio set value as input:

```
clearOtherSelections('uxTruckModels');
assignModelName('uxTruckModels');
```

If I select the **Code** tab I can actually look at the code for those functions. Here's the first, **assignModelName**:

```
// Sets the value of the ModelName dataslot via a field
function assignModelName (srcModel) {
  var s = sbm.widgets.getValue(srcModel);
  var obj = eval(s);
  for(var i = 0; i < obj.length; i++) {
    var e = obj[i];
    if (e.selected) {sbm.widgets.setValue('txtModelName', e.label);}
  }
}
```



The object-oriented view of the data makes this look a bit complicated, but basically the code has to walk through all the possible values of the radio set control as a JSON object like the example superimposed on the screenshot above, and find the one whose selected property is set. That choice's **label** property in turn is written into a simple string variable called **txtModelName**. Note that this code is using another Savvion library called **sbm.widgets**, which you can find in the same appendix I showed you earlier in the **BPM Studio Users Guide**.

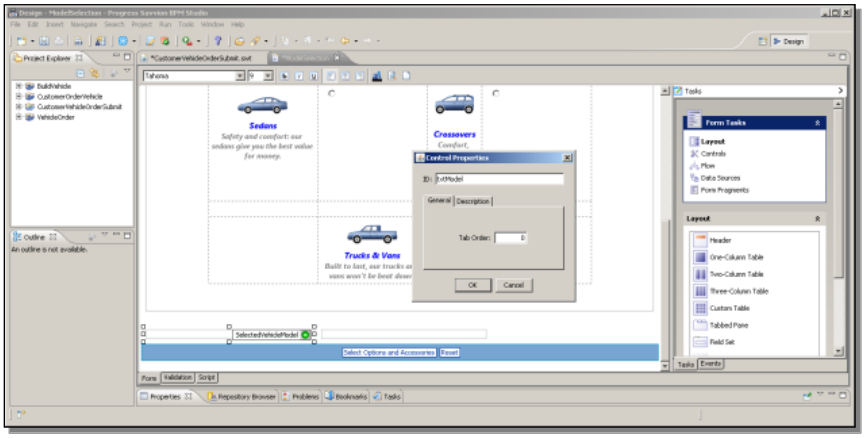
The other function clears any previous selection that was made for another model type, such as Sedan or Crossover, by setting its selectedIndex property to minus 1.

```
function clearOtherSelections (selectedModel) {
  if (selectedModel !== 'uxCompactModels')
        {document.getElementById('uxCompactModels').selectedIndex = -1;}
  if (selectedModel !== 'uxSedanModels')
        {document.getElementById('uxSedanModels').selectedIndex = -1;}
  if (selectedModel !== 'uxSUVModels')
        {document.getElementById('uxSUVModels').selectedIndex = -1;}
  if (selectedModel !== 'uxTruckModels')
        {document.getElementById('uxTruckModels').selectedIndex = -1;}
  if (selectedModel !== 'uxPremiumModels')
        {document.getElementById('uxPremiumModels').selectedIndex = -1;}
}
```

This is in case the customer changed his mind and selected one model type and then another. You can see that the code uses the standard HTML convention of **document.getElementById** to locate each radio set control within the form.

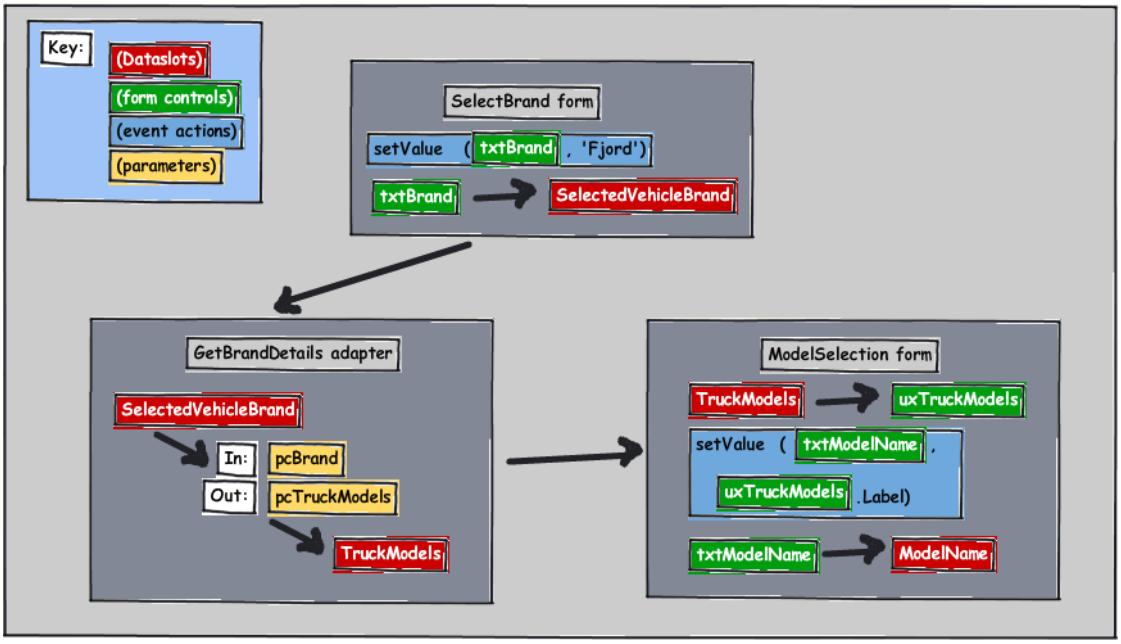
So overall, looking at the event handlers for just these steps has introduced you to several of the different ways you can define event handlers for Savvion forms, sometimes just using predefined categories of expression types that are offered to you, sometimes by writing your own JavaScript code.

Finally, the screenshot below shows the two hidden fields that this form uses. Double-click on the first one, which is bound to the **SelectedVehicleModel** dataslot, you see that this is the one named **txtModel**, referenced in the code we just looked at:



The second one, bound to the **ModelName** dataslot -- which in turn is passed on to a later step in the process -- is the one named **txtModelName**.

One final look at a diagram to help you understand the data flow here:



This user form **ModelSelection** starts with the **TruckModels** dataslot value, a JSON object holding a list of all the Fjord trucks, and uses it to set the radio set for the new form. It then sets the hidden field **txtModelName** to the label of the selected model, which in turn is mapped to the dataslot **ModelName**, which is used in a later step in the flow.

My goal in this paper has been to show you just enough about the very powerful ability to pass data values from one step in a process to another, and to define flexible event handlers for control events, to get you started. As with all of these videos and papers, you should take what you've been introduced to here and follow up on your own in your investigation of all the kinds of ways you can use data retrieved from your application to populate and control the steps in your Savvion process. This can enable you to build an application process model that at runtime is

closely tied to all the data and logic that's already built into your OpenEdge ABL application.