RIA SERVICES INVOKE OPERATIONS

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
February 2011

DISCLAIMER

The videos and papers in this series on using Silverlight with RIA Services have shown examples of passing a ProDataSet as a parameter and converting it to an entity class that can provide data to a Silverlight user interface. This paper and the two-part video it accompanies show you an example of how to call an ABL procedure that doesn't pass a ProDataSet or temp-table, but instead passes one or more scalar or array values as input and output parameters. Earlier examples in the series show the **`Query`** and **`Update`** attributes as annotations in the DomainService class. The **`Query`** example just has a ProDataSet as an ABL **`OUTPUT`** parameter. The AppObject proxy expresses this as an ADO.NET DataSet, and the DomainService query method itself defines it as an enumerable collection of **`CustOrder`** objects, as shown here:

```
[Query]
public IEnumerable<CustOrder> GetCustOrders()
{
    try
    {
        _appObj.GetCustOrders(out CustOrderDS);
       …
```

An example such as that one could be extended to have one or more **`INPUT`** parameters as well, specifying filter criteria for example. But the **`Query`** form can't have any output parameters other than the DataSet, or alternatively a temp-table.

The **`Update`** example in the RIA Services series passes in just a changed row to return to the server, as shown here:

```
[Update]
public void UpdateCustOrder(CustOrder changeRow)
{
    DataRow updateRow = FindCustomer(changeRow.CustNum);
    if (updateRow != null)
    {
        updateRow["CustNum"] = changeRow.CustNum;
        …
    }
}
```

The associated **Submit** method passes this as an ADO.NET change set back to the server. The **Update** and **Submit** form in fact doesn't allow additional parameters beyond the change set. As a reminder, the **Submit** method code below shows both the **Submit** call with the object-oriented change set parameter and then the AppObject proxy call that passes the DataSet form of that change set through the proxy to the ABL **UpdateCustOrders** procedure:

```
        public override bool Submit(ChangeSet changeSet)
        {
            bool result = true;
            try
            {
                foreach (ChangeSetEntry changeRow in changeSet.ChangeSetEntries)
                {
                    if (changeRow.Entity is CustOrder)
                    {
                        CustOrder origEntity = (CustOrder)changeRow.OriginalEntity;
                        DataRow newRow = CreateCustOrder(origEntity);
                        CustOrderDS.Tables["ttCustomer"].Rows.Add(newRow);
                        newRow.AcceptChanges();
                    }
                }

                result = base.Submit(changeSet);
                if (changeSet.HasError)
                    return false;

                _appObj.UpdateCustOrders(ref CustOrderDS);
            }

        }
```

There is also a general purpose attribute named **Invoke** for making a call that doesn't involve an entity, but needs to pass some combination of input and output parameters, which can be scalar values or arrays of scalar values. That's what is covered in this paper and the two-part video that concludes the RIA Services series.

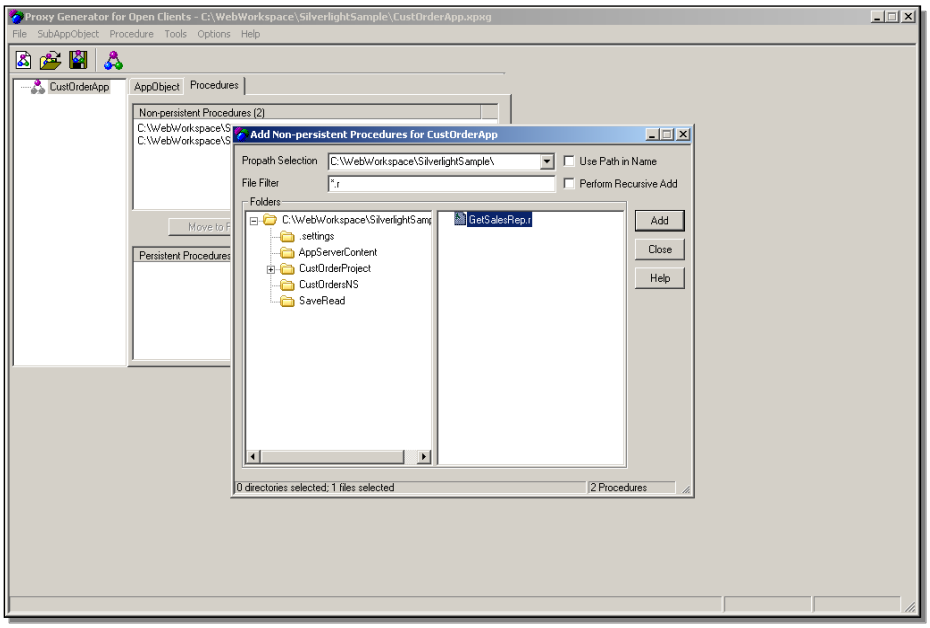
As in the earlier code samples in the series, this one defines an ABL service as a stand-alone non-persistent procedure, which takes a customer number as input, retrieves the corresponding SalesRep record from the database, and returns the SalesRep name as output:

```
/*------------------------------------------------------------------------
    File        : GetSalesRep.p

    Notes       : Returns the SalesRep RepName for the given customer number.
  ----------------------------------------------------------------------*/

DEFINE INPUT  PARAMETER piCustNum      AS INTEGER.
DEFINE OUTPUT PARAMETER pcSalesRepName AS CHARACTER.

FIND Customer WHERE Customer.CustNum = piCustNum.
FIND Salesrep WHERE Salesrep.SalesRep = Customer.SalesRep.
pcSalesRepName = Salesrep.RepName.
```

This is a somewhat simplified case. If your ABL procedure has a single **OUTPUT** parameter, and doesn't need to return the ABL **RETURN-VALUE** to the client, then ProxyGen maps the one **OUTPUT** parameter to the return value of the **Invoke** operation in the DomainService class, which this paper shows how to define. The

procedure could still have one or more **INPUT** parameters, and the process would be the same. However, if the procedure has multiple **INPUT-OUTPUT** or **OUTPUT** parameters, then there's an additional code construction that's needed, and an example of that is shown in the second of the two videos, and described in the second part of this paper.

So again, the newly-created the ABL procedure needs to be added to the same proxy as the other procedures that represent services on the Customer and Order tables. Opening the proxy that was built and extended before, I can add this next procedure to it. Like the others, it's in the AppServerContent folder, because that's what gets published to the AppServer's ProPath. Selecting the .r file for the new procedure, I add that to the proxy:



Once again I re-generate the proxy, and now I have a new version with three separate procedures defined.

Next I have to add a reference to it in the DomainService class. Back in Visual Studio, I re-open the **CustOrderDomainService** class. Down at the bottom of the file, I add support code for this first non-entity-passing method, which will run **GetSalesRep** in the proxy, and which as I said uses an attribute named **Invoke** to signal the behavior that needs to be supported. The **Invoke** attribute is expressed as an annotation in brackets, and the new public method has the same name as the ABL procedure that it's going to run. (This is just a naming convention I use for this example; there's no need to use the same name, and in fact you might prefer not to.)

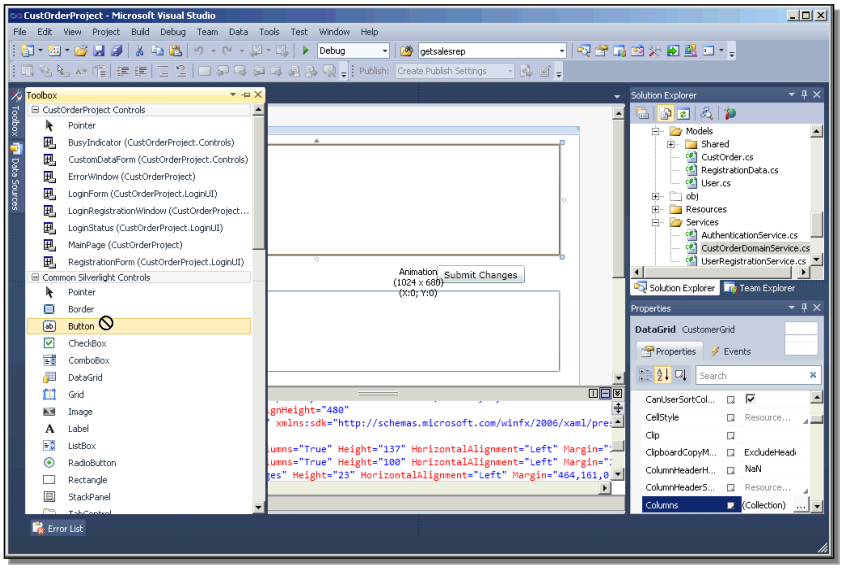
```
        [Invoke]
        public string GetSalesRep(int inputCustNum)
        {
```

The method just takes the input parameter that will be passed to the ABL procedure, the customer number. I need to define a variable to hold the output parameter that comes back from the ABL procedure. Then in the same AppObject instance used by

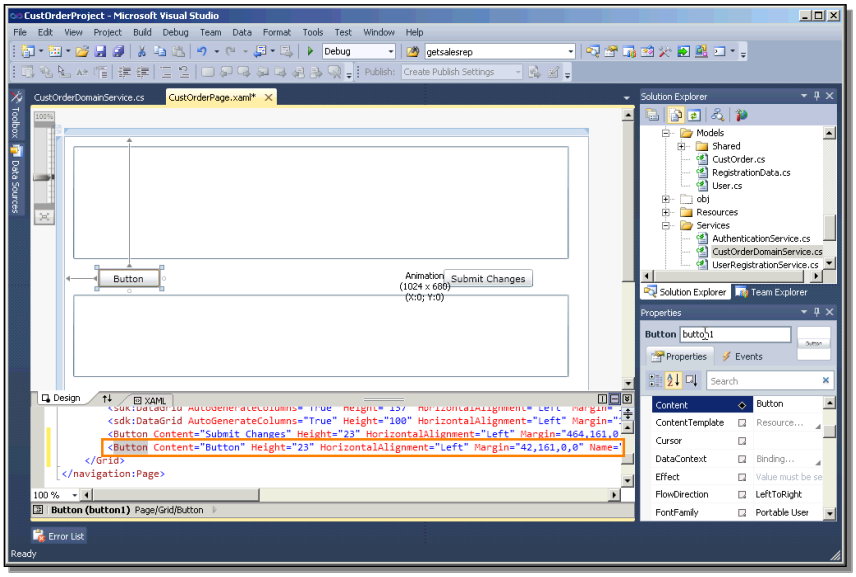
other methods in the DomainService, the **`GetSalesRep`** method needs to invoke the ABL procedure through the proxy. This call takes the parameters of the ABL procedure: the customer number as input and the SalesRep name as output. So the DomainService method returns the SalesRep name as its return value:

```
            string outRepname;
            _appObj.GetSalesRep(inputCustNum, out outRepname);
            return outRepname;
        }
```

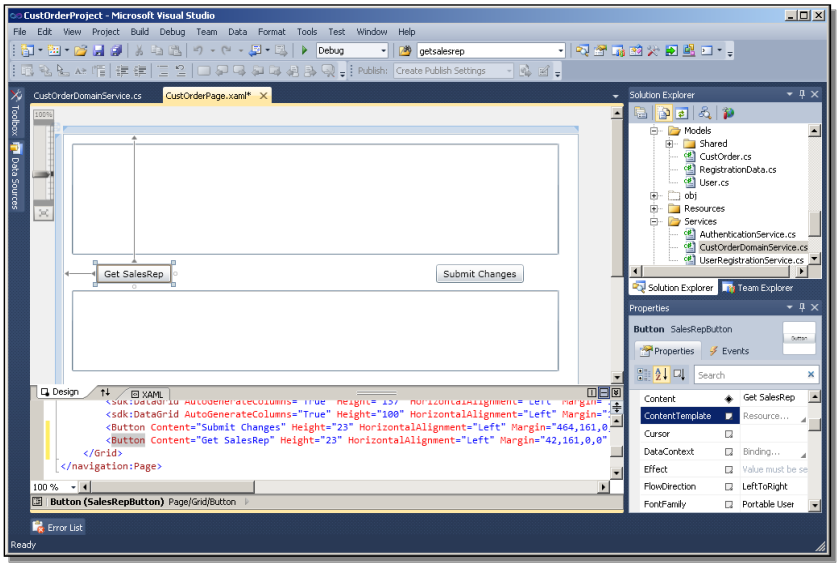
That's the end of the new method in the DomainService. After saving and compiling the DomainService, I do a Build to regenerate the DomainContext class in the solution. Once that's done, I need to update the user interface to display the SalesRepName that's returned from OpenEdge. Back in the XAML for the **CustOrderPage**, I need to add a button that will trigger a request to retrieve the SalesRep name. In the Toolbox, I select a Button control:



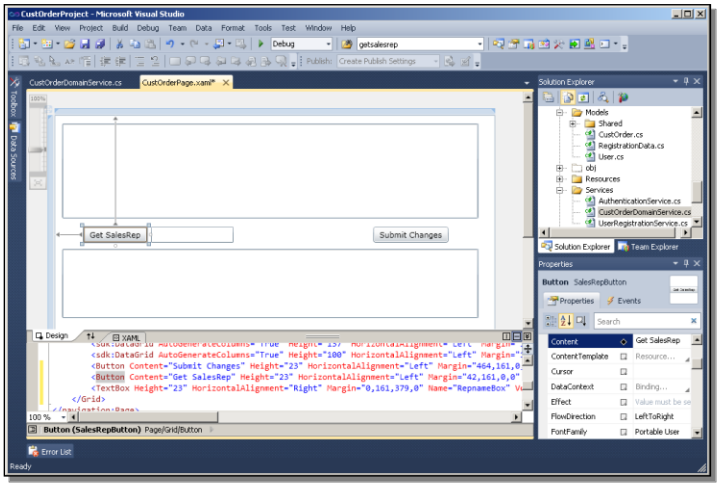
I drag and drop that onto the page, and move it over to the left, underneath the Customer grid. The screenshot below shows that the XAML now includes the Button and reflects its initial property settings:



I change the button name to **SalesRepButton** in the Properties tab, and change the label – that's the **Content** property -- to **Get SalesRep**. I can resize the button as well to make room for the new label:

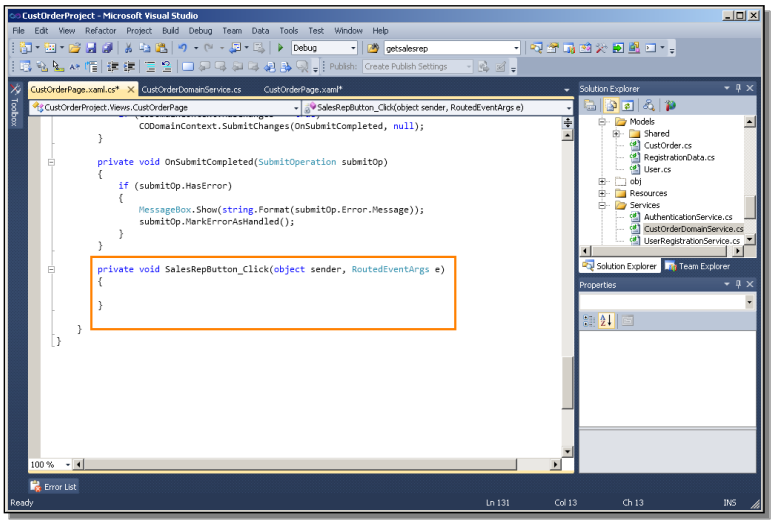


Now I want what ABL would call a fill-in field to display the SalesRep name that's retrieved. Once again in the Toolbox, I select a **TextBox** control, and drag that onto the page, and move it over next to the button. I name it **RepnameBox**.



The value displayed is the control's **Text** property. The supporting code for the UI sets that at runtime when a SalesRep name is retrieved for a Customer.

If I double-click on the button, I get the skeleton for a **Click** event handler for it. Here I want to retrieve the customer number from the currently selected row in the grid to use as the input parameter to my call:



I start with the same statement used in the **SelectionChanged** event that retrieves Orders for a selected customer. **`SelectedCust`** is a **`CustOrder`** entity from the **`CustomerGrid`**, and the current row is the **`SelectedItem`** property:

```
        private void SalesRepButton_Click(object sender, RoutedEventArgs e)
        {
            CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
```

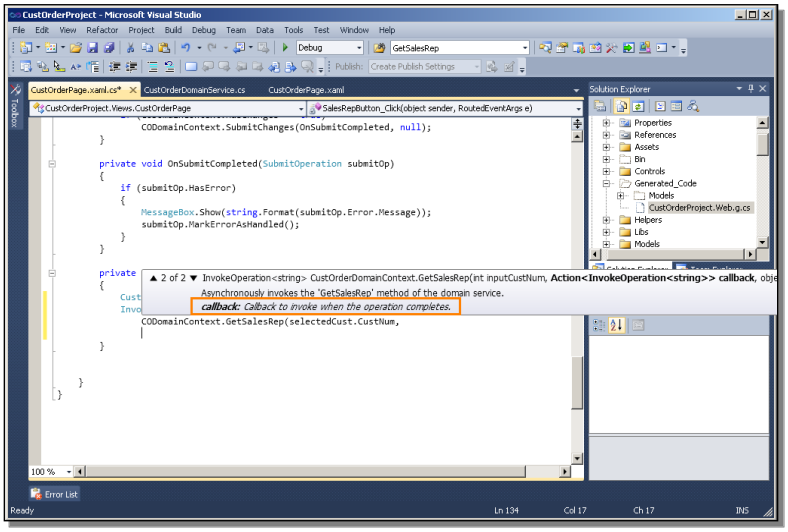
Now what comes next is a little tricky. Remember that the user interface C# support code doesn't make calls directly to the DomainService. It uses the generated DomainContext class as a client-side intermediary. If I open that class once again in the **Generated_Code** folder (which I exposed in an earlier video by pressing the

Show All Files button) you can take a look at the code generated to support **GetSalesRep**. Look at the comments for the **GetSalesRep** operation:

```
        /// <summary>
        /// Asynchronously invokes the 'GetSalesRep' method of the DomainService.
        /// </summary>
        /// <param name="inputCustNum">The value for the 'inputCustNum' parameter
        of this action.</param>
        /// <param name="callback">Callback to invoke when the operation
        completes.</param>
        /// <param name="userState">Value to pass to the callback.  It can be
        <c>null</c>.</param>
        /// <returns>An operation instance that can be used to manage the
        asynchronous request.</returns>
        public InvokeOperation<string> GetSalesRep(int inputCustNum,
          Action<InvokeOperation<string>> callback, object userState)
        {
            Dictionary<string, object> parameters = new Dictionary<string,
          object>();
            parameters.Add("inputCustNum", inputCustNum);
            this.ValidateMethod("GetSalesRep", parameters);
            return ((InvokeOperation<string>)(this.InvokeOperation("GetSalesRep",
          typeof(string), parameters, true, callback, userState)));
        }
```

First there's a reminder that calls to the DomainService are made asynchronously, so the code has to be a two-step process of making the request and then processing the response. The first parameter is my input parameter, the customer number. Then I have to name a callback method that's run when the request completes. Then there's an optional user state value that I'll just set to **null**; this is a pattern we've seen before. Most importantly, you can see that the call itself returns an **InvokeOperation** object.

Looking at this generated code in the DomainContext confirms what the call in the user interface support class needs to look like. I have to define an **InvokeOperation** object of type **string**, representing the one output parameter, which I call **invokeOp**. I assign that to the result of the method in the DomainContext called **GetSalesRep**. The first parameter to the method in the DomainContext is the customer number. To get to that, I start with the **SelectedCust** in the **CustomerGrid**, which is an instance of the **CustOrder** entity, and so has values for all the fields that are defined as properties in the entity class. I want the **CustNum** property, of course. You can see the code assist is reminding me that the next parameter is a callback method:



I call that callback **`OnSRB_Completed`** – SRB for **SalesRepButton**. I can just use **`null`** in place of a user state, and that's the end of the click event handler:

```
private void SalesRepButton_Click(object sender, RoutedEventArgs e)
{
    CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
    InvokeOperation<string> invokeOp =
            CODomainContext.GetSalesRep(selectedCust.CustNum,
              OnSRB_Completed, null);
}
```

Now I need to code the callback method, **`OnSRB_Completed`**, which takes the **`InvokeOperation`** object as input:

```
private void OnSRB_Completed(InvokeOperation<string> invokeOp)
{
    if (invokeOp.HasError)
    {
        MessageBox.Show(string.Format("Method Failed: {0}",
            invokeOp.Error.Message));
        invokeOp.MarkErrorAsHandled();
    }
    else
        RepnameBox.Text = invokeOp.Value;
}
```

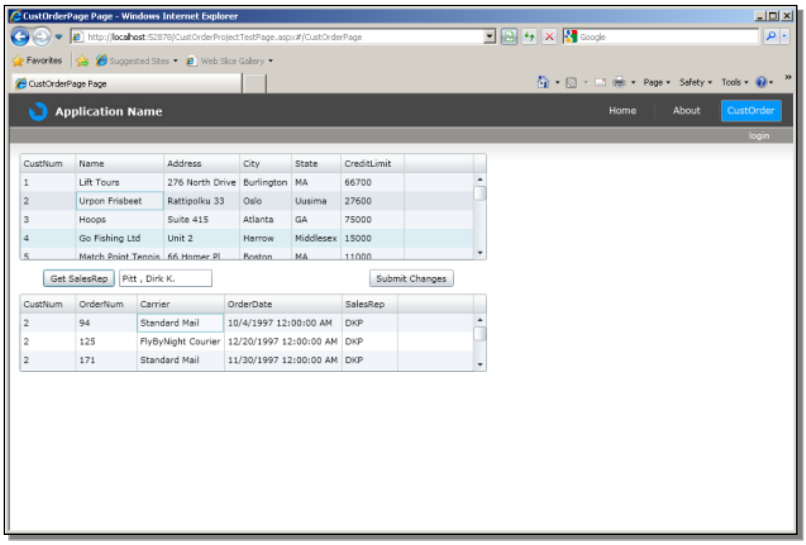
The method uses the same **`HasError`** check used in other callbacks like this one. Otherwise, if there's no error, all the code needs to do is assign the **`Text`** property of the **`RepnameBox`** TextBox to the **`Value`** property of the **`Invoke`** operation, which is a string. The **`Value`** property is the return value that the ABL procedure's output parameter is mapped to.

Just to clean up the display, I make a small change to the **`SelectionChanged`** event handler that gets fired every time the user selects a different customer in the grid. I clear the **`Text`** value of the TextBox, until the user clicks the **Get RepName** button again:

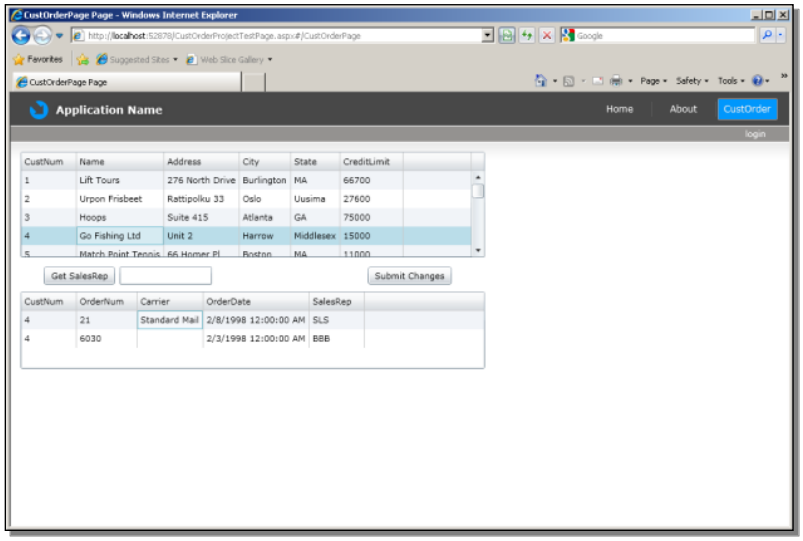
```
        private void CustomerGrid_SelectionChanged(object sender,
              SelectionChangedEventArgs e)
        {
            RepnameBox.Text = "";
            CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
            OrderGrid.ItemsSource = selectedCust.Orders;
        }
```

Of course, the code could trigger the **GetRepName** method from this **SelectionChanged** event handler, so that the value is immediately displayed whenever a new Customer row is selected, but that's just part of the user interface design, so feel free to make a change like that in your own examples.

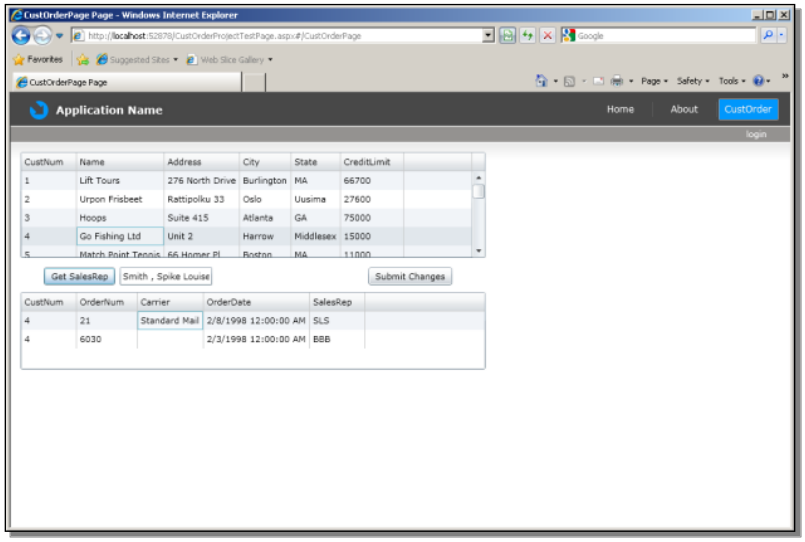
Now I can run the test page for the project. I select the CustOrder page, select a Customer, and click the Get SalesRep button. Here's the SalesRep name for that Customer:



I can select another Customer, and you can see that the SalesRep name was cleared by the line of code that I added to the **SelectionChanged** event handler:



So I click the button again, and I see the SalesRep for customer 4, so everything's working:



This example has shown you how to call an ABL procedure with any number of **`INPUT`** parameters, and exactly one non-entity **`OUTPUT`** parameter. As a review, and to allow you to copy code from this paper, here are the complete code elements used in this first Invoke example:

First, the ABL procedure:

```
/*------------------------------------------------------------------------
    File        : GetSalesRep.p

    Notes       : Returns the SalesRep RepName for the given customer number.
  ----------------------------------------------------------------------*/

DEFINE INPUT  PARAMETER piCustNum      AS INTEGER.
DEFINE OUTPUT PARAMETER pcSalesRepName AS CHARACTER.

FIND Customer WHERE Customer.CustNum = piCustNum.
FIND Salesrep WHERE Salesrep.SalesRep = Customer.SalesRep.
pcSalesRepName = Salesrep.RepName.
```

Next, the DomainService method:

```
        [Invoke]
        public string GetSalesRep(int inputCustNum)
        {
            string outRepname;
            _appObj.GetSalesRep(inputCustNum, out outRepname);
            return outRepname;
        }
```

Next, the click event handler for the new button in CustOrderPage.xaml.cs:

```
        private void SalesRepButton_Click(object sender, RoutedEventArgs e)
        {
            CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;

            InvokeOperation<string> invokeOp =
                    CODomainContext.GetSalesRep(selectedCust.CustNum,
                      OnSRB_Completed, null);
        }
```

Next the completion event handler for the client event handler:

```
        private void OnSRB_Completed(InvokeOperation<string> invokeOp)
        {
            if (invokeOp.HasError)
            {
                MessageBox.Show(string.Format("Method Failed: {0}",
                    invokeOp.Error.Message));
                invokeOp.MarkErrorAsHandled();
            }
            else
                RepnameBox.Text = invokeOp.Value;
        }
```

And finally, the one-line change to reset the **`Text`** property in the **`SelectionChanged`** event handler:

```
        private void CustomerGrid_SelectionChanged(object sender,
              SelectionChangedEventArgs e)
        {
            RepnameBox.Text = "";
            CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
            OrderGrid.ItemsSource = selectedCust.Orders;
        }
```

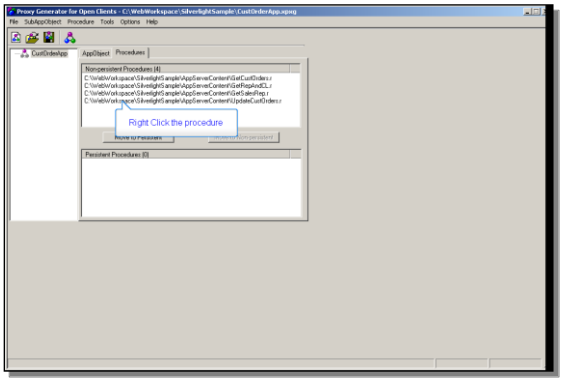
As I said at the outset, the first example is somewhat simplified, because it needs to return only a single scalar output parameter. In the second part of the paper I create a slightly more complex example to show how to return multiple output parameters as an object with two or more properties.

As before, I need to create a new stand-alone ABL procedure for a new service, which in this case goes beyond the previous one in that it takes the customer number as input, but also takes the **CreditLimit** from the client as an **INPUT-OUTPUT** parameter, and returns it, possibly modified, along with the **SalesRep** name. So in effect this procedure has two input parameters and two output parameters. I have also added a very simplistic bit of business logic here that says that if the **CreditLimit** hasn't been changed in the database, and if the customer **Balance** is less than half the **CreditLimit**, then the code increases the **CreditLimit** by twenty percent and returns that value:

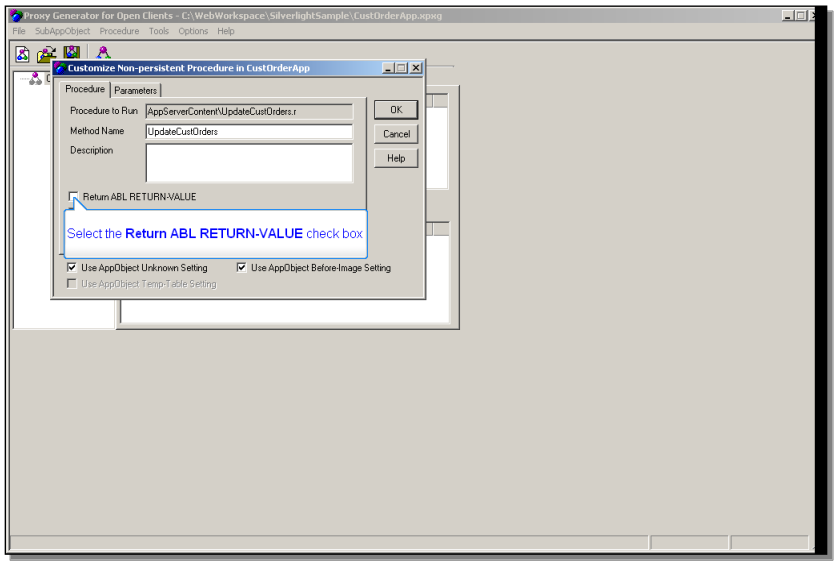
```
/*------------------------------------------------------------------------
    File        : GetRepAndCL.p

    Notes       : Returns the SalesRep RepName and adjusts CreditLimit.
  ----------------------------------------------------------------------*/

DEFINE INPUT         PARAMETER piCustNum      AS INTEGER.
DEFINE INPUT-OUTPUT  PARAMETER pdCreditLimit  AS DECIMAL.
DEFINE OUTPUT        PARAMETER pcSalesRepName AS CHARACTER.

FIND Customer WHERE Customer.CustNum = piCustNum.
IF (pdCreditLimit = Customer.CreditLimit) AND
 (Customer.Balance < (Customer.CreditLimit / 2)) THEN
  pdCreditLimit = Customer.CreditLimit * 1.2.

FIND Salesrep WHERE Salesrep.SalesRep = Customer.SalesRep.
pcSalesRepName = Salesrep.RepName.
```

After saving the new procedure to the AppServerContent directory, I need to add it to the same proxy as the other procedures in the series. I add the procedure, and now I have four entry points in the proxy, each a stand-alone ABL procedure.

Let me take a moment to make sure it's clear how you would return the ABL **RETURN-VALUE** to the client if you needed to. You can right-click on an ABL .r file in ProxyGen's **Procedures** tab, and select **Customize...**



The **Customize** dialog has an option to **Return ABL RETURN-VALUE**:



If you click that checkbox on, then the **`RETURN-VALUE`** is returned as a string with the name **`retValue`**, and you have to add that to the set of output parameters for the procedure. However, I'm not going to do that here.

When I click the **Generate** button, and then just **OK** the proxy properties dialog, I get a new version of the .NET proxy with four separate procedures defined.

Now I go back to the DomainService class I've been working with, to add support for the new procedure. As a reminder, here's the **`Invoke`** operation I added in the previous video.

```
        [Invoke]
        public string GetSalesRep(int inputCustNum)
        {
            string outRepname;
            _appObj.GetSalesRep(inputCustNum, out outRepname);
            return outRepname;
        }
```

Note that because the underlying ABL procedure that is being invoked here through the proxy just returns a single output parameter, and no ABL **`RETURN-VALUE`**, the DomainService method can just define its return type as the return type of the ABL parameter – **`CHARACTER`** or **`string`** in this case – and simply return that one value.

In this second example, it's not going to be quite that simple. The ABL procedure returns two values, the **`INPUT-OUTPUT`** **CreditLimit** and the output **SalesRep** name. To represent that in the DomainService, I need to define a class with a property for each of those values. The class then becomes the return type for the other DomainService methods.

So before I define the new **`Invoke`** method, I define the class that encapsulates all its output parameters. There's a rule that all DomainService operations must use

serializable types for parameters and return types, so I add the **Serializable** attribute as an annotation in front of the parameter class:

```
[Serializable]
public class GetRepAndCLParams
{
    public decimal inoutCreditLimit { get; set; }
    public string outRepname { get; set; }
}
```

The first property defined in the class represents the **CreditLimit**, the **INPUT-OUTPUT** parameter. The second one represents the **OUTPUT SalesRep** name. That's all that's needed in the class definition. Remember that because the call is asynchronous, the input parameters have already been passed to the request. Here I need to specify only the output parameters that come back to the completion event handler, including the output value for any input-output parameters.

Now I can define the class that invokes the ABL procedure through the proxy. Its return type is not a simple scalar type, but an object of the type I just defined. As I've done before, I give the method the same name as the ABL procedure, which is just my naming convention. There are two input parameters, the customer number and the current credit limit as held in the client:

```
[Invoke]
public GetRepAndCLParams GetRepAndCL
    (int inputCustNum, decimal inoutCreditLimit)
{
```

Because my return type is now a class, I have to create an instance of the class to return. The first two parameters are accounted for as input to this method, but I still need to define a local variable to hold the third parameter when it comes back.

```
GetRepAndCLParams outParams = new GetRepAndCLParams();
string outRepname;
```

Next I invoke **GetRepAndCL** in the running AppObject instance. The first parameter is the customer number. The second is the credit limit, which is passed using the **ref** mode in C# to correspond to what ABL terms **INPUT-OUTPUT**. The third parameter is the output parameter.

```
try
{
    _appObj.GetRepAndCL(inputCustNum, ref inoutCreditLimit,
        out outRepname);
```

Now I populate the parameter object I created, setting first the credit limit, in case that was modified on the server, and then the SalesRep name:

```
    outParams.inoutCreditLimit = inoutCreditLimit;
    outParams.outRepname = outRepname;
}
```

The standard **catch** block completes the additions to the DomainService class:

```
            catch (System.Exception e)
            {
                throw new Exception(e.Message);
            }
            return outParams;
        }
```

As I've noted elsewhere, my exception handling code in these examples is very minimal, to keep the code brief, but even this simple catch block is useful because it will at least return an exception message to the client.

I can compile what I've just added here, and then do a Build again, to make sure everything gets regenerated properly. Once that's done, I can take another look at the generated DomainContext class to confirm what the user interface support code needs to pass to it. As I did before, I can look at the comments to check what I need to pass:

```
        /// <summary>
        /// Asynchronously invokes the 'GetRepAndCL' method of the DomainService.
        /// </summary>
        /// <param name="inputCustNum">The value for the 'inputCustNum' parameter
        of this action.</param>
        /// <param name="inoutCreditLimit">The value for the 'inoutCreditLimit'
        parameter of this action.</param>
        /// <returns>An operation instance that can be used to manage the
        asynchronous request.</returns>
        public InvokeOperation<GetRepAndCLParams> GetRepAndCL(int inputCustNum,
          decimal inoutCreditLimit)
        {
            Dictionary<string, object> parameters = new Dictionary<string,
                object>();
            parameters.Add("inputCustNum", inputCustNum);
            parameters.Add("inoutCreditLimit", inoutCreditLimit);
            this.ValidateMethod("GetRepAndCL", parameters);
            return
          ((InvokeOperation<GetRepAndCLParams>)(this.InvokeOperation("GetRepAndCL",
              typeof(GetRepAndCLParams), parameters, true, null, null)));
        }
```

You can see that I just have to pass all the input parameters (including of course the input-output credit limit), followed by the callback method name, followed by a **null** for the user state. Remember that the call is asynchronous, so I pass all the inputs here and process the outputs later.

Having confirmed the signature of the call, I create a call to the DomainContext method in the user interface support code, **CustOrderPage.xaml.cs**. Starting with the click event handler for the Get SalesRep button, I can make some changes to run the new ABL procedure rather than the simpler one with the one **OUTPUT** parameter:

```
        private void SalesRepButton_Click(object sender, RoutedEventArgs e)
        {
            CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
```

First I add a simple sanity check here so that the code doesn't blow up if I click the button before I select a customer. (Be aware that my examples are very simplified in ways like this, so that I don't have to explain any more code than necessary.)

```
            if (selectedCust != null)
            {
```

Next I comment out the invoke operation for **GetSalesRep**, and in its place, I invoke the new one with two output parameters:

```
                /* InvokeOperation<string> invokeOp =
                    CODomainContext.GetSalesRep(selectedCust.CustNum,
                      OnSRB_Completed, null); */

                InvokeOperation<GetRepAndCLParams> invokeOp =
                    CODomainContext.GetRepAndCL(selectedCust.CustNum,
                        selectedCust.CreditLimit, OnSRB_CL_Completed, null);
            }
        }
```

Note that the type of the invoke operation is now my new parameters class. In the DomainContext, I invoke the **GetRepAndCL** method that I just looked at. The input parameters come from the currently selected Customer in the grid, the CustNum and the CreditLimit. I name my event handler for the completion event **OnSRB_CL_Completed**, adding **CL** for CreditLimit.

Next I implement the new completion event handler. It's instructive to compare it with the simpler method, which as a reminder is shown again here:

```
        private void OnSRB_Completed(InvokeOperation<string> invokeOp)
        {
            if (invokeOp.HasError)
            {
                MessageBox.Show(string.Format("Method Failed: {0}",
                    invokeOp.Error.Message));
                invokeOp.MarkErrorAsHandled();
            }
            else
                RepnameBox.Text = invokeOp.Value;
        }
```

First I add **CL** to the method name. Next I have to replace the **string** type with the parameter object type:

```
private void OnSRB_CL_Completed(InvokeOperation<GetRepAndCLParams> invokeOp)
        {
```

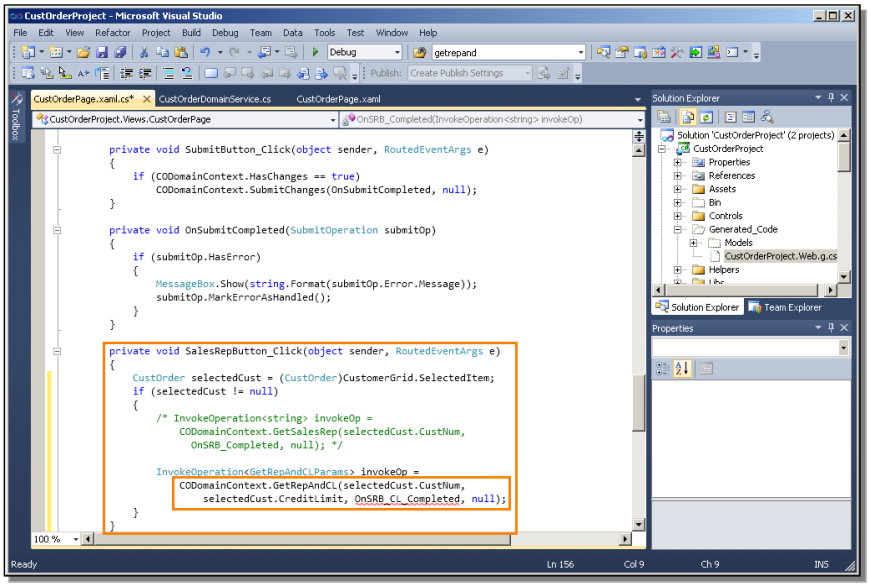
After the standard error check, instead of the **Value** parameter of the invoke operation being a single value, it's now an object with a property of its own for each property in the parameters class. So I set the **Text** property of the SalesRep name TextBox to that property:

```
            if (invokeOp.HasError)
            {
                MessageBox.Show(string.Format("Method Failed: {0}",
                       invokeOp.Error.Message));
                invokeOp.MarkErrorAsHandled();
            }
            else
            {
                RepnameBox.Text = invokeOp.Value.outRepname;
```

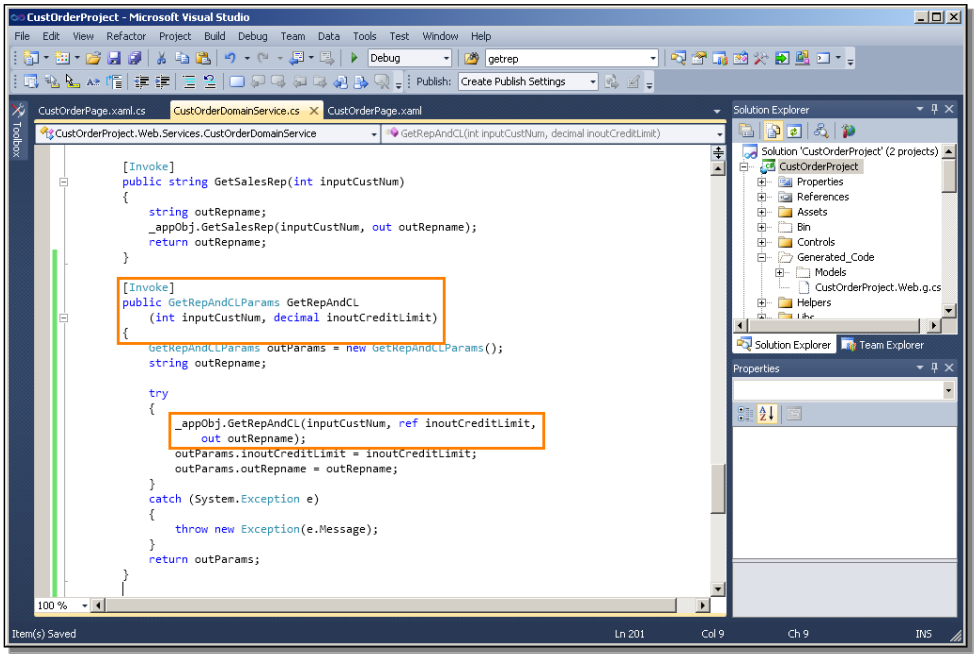
Next I need to re-display the CreditLimit value that's returned, in the grid. So I identify again the object that represents the currently selected row, the **`selectedItem`**. I set the CreditLimit cell in that row to the **`INPUT-OUTPUT`** value that was passed back from OpenEdge. Once again the Invoke operation's **`Value`** parameter is an object, which also holds the **`CreditLimit`** property:

```
                CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
                selectedCust.CreditLimit = invokeOp.Value.inoutCreditLimit;
            }
        }
```

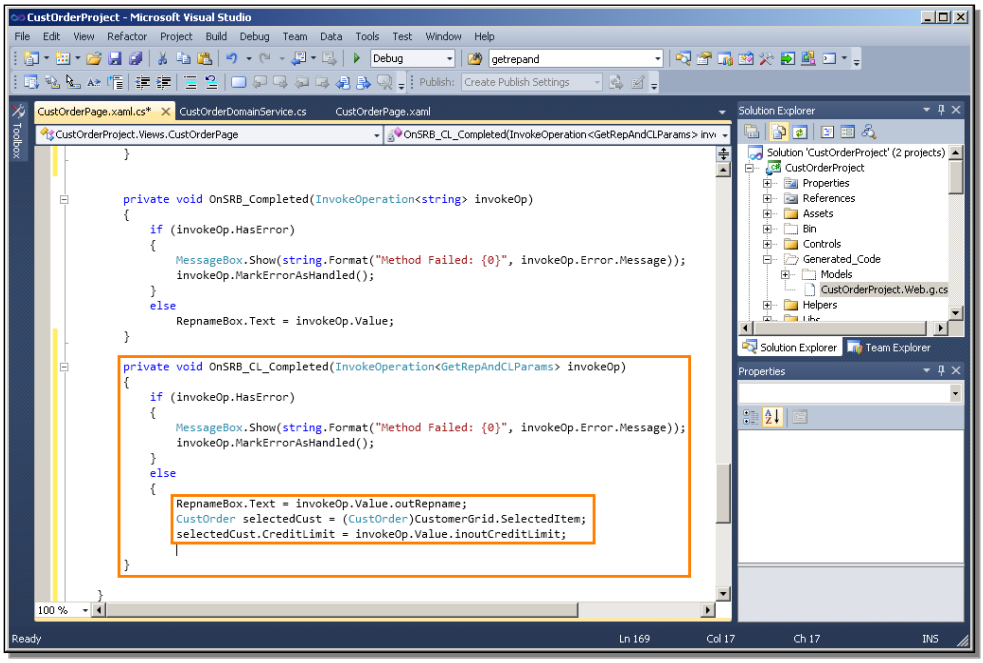
To review, when the user selects a Customer in the grid and clicks the Get SalesRep button, that click event fires the event handler in **CustOrderPage.xaml.cs**. The event handler runs the version of the **`GetRepAndCL`** method in the client's DomainContext class, passing the input parameters and the name of the completion method:



The DomainContext invokes the method that was just defined in the DomainService running on the Web server, asynchronously, which in turn invokes the ABL procedure via the AppObject proxy instance:



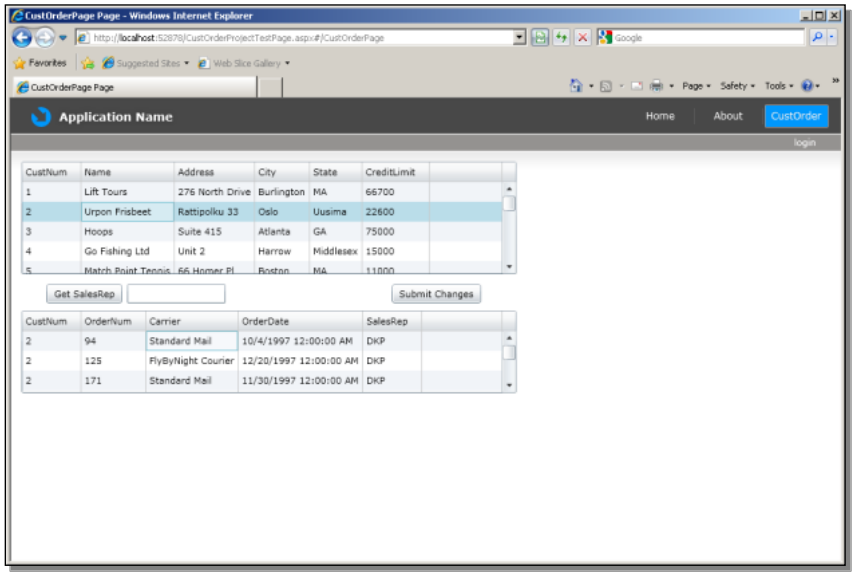
When the request completes, the new completion method **`OnSRB_CL_Completed`** runs, taking the invoke operation initiated by **`GetRepAndCL`** with its parameter object as input. The parameter object defines the output parameters, and the method copies the SalesRep name to its TextBox and the CreditLimit value back to the grid:



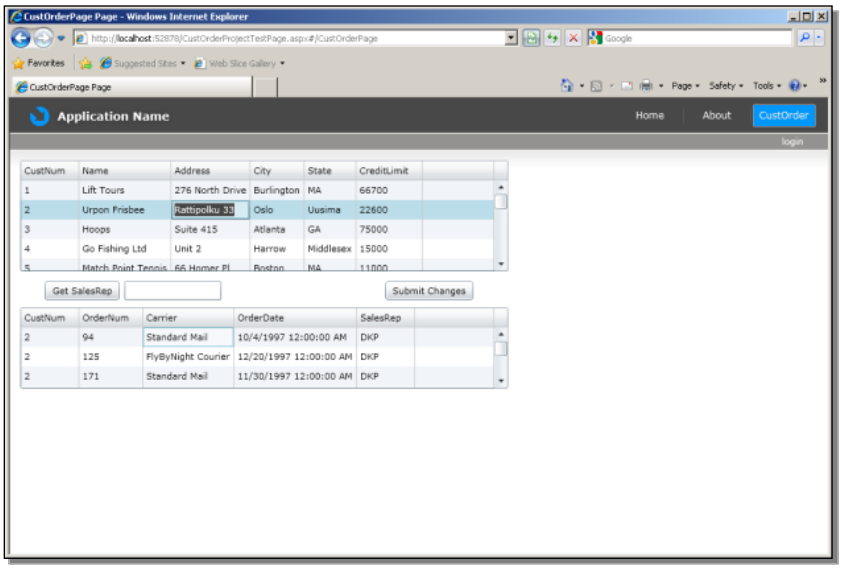
This example can reuse the Get SalesRep button and its TextBox, so there's no need to make any changes to the user interface itself.

If I run with the new methods compiled and and the project re-built, then when the test page comes up, I can select the CustOrder page, and Customer 2. I see the

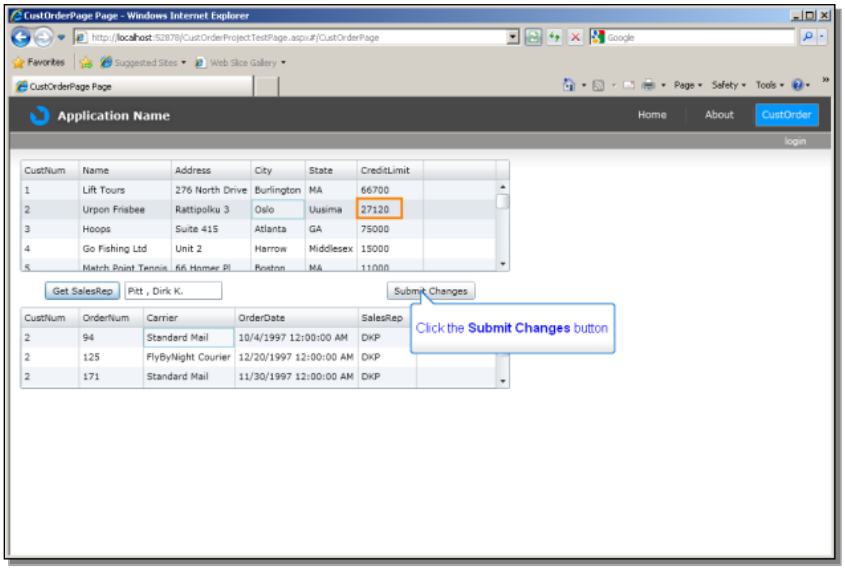
Orders that are retrieved by the other user interface event, **`SelectionChanged`** of the Customer grid:



Now I undo the little changes I made to this row in the update presentation, removing the **`t`** from the end of the name, and changing the street number back to **`3`**. Note the before value of the CreditLimit, **`22600`**:

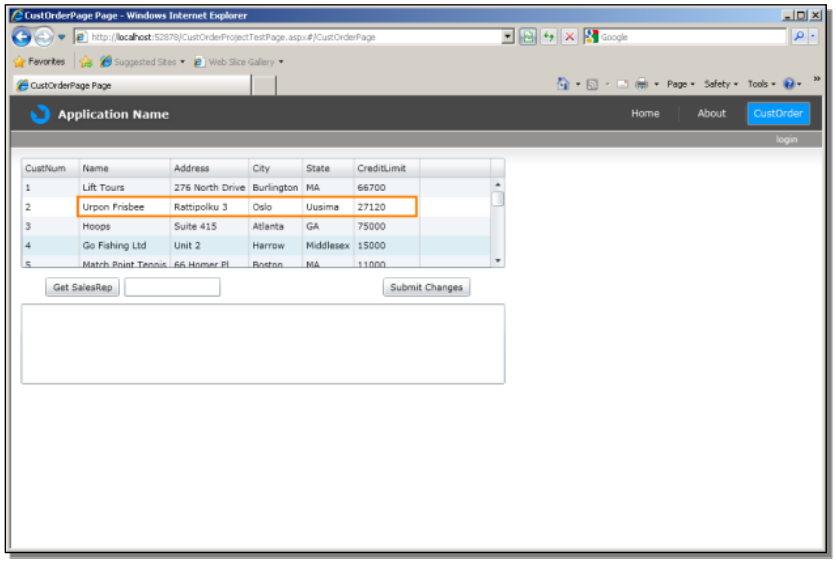


Now I click the Get SalesRep button. This invokes the new **`GetRepAndCL`** method, and in turn the new ABL procedure of the same name, which not only retrieves the SalesRep name but updates the credit limit if it satisfies the business logic on the backend:



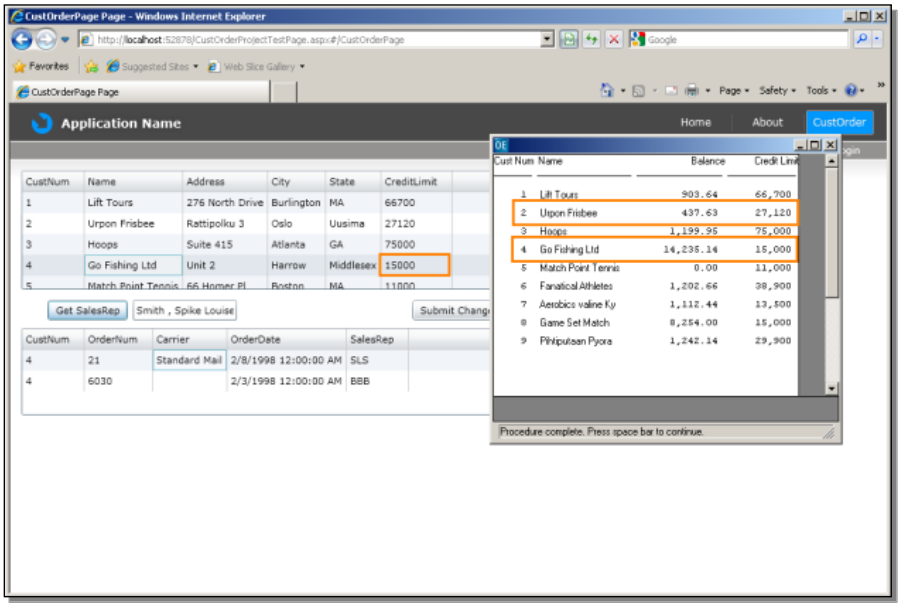
And after clicking the Get SalesRep button, the SalesRep name is displayed, and apparently the Balance for this Customer is less than half the CreditLimit, so the CreditLimit has been increased.

Now, the business logic didn't update the database, it just returned the new CreditLimit value to the client. If I now decide I want to save that new value, along with the other changes I made, I can press the Submit Changes button that was added to the application in the update example, and those changes are all sent back to the server in a change set and saved to the database. To verify that, I can get out of the CustOrder page, and then reselect it, and see that all the values for Customer 2 have been re-retrieved, confirming that they were saved to the database:



If I select Customer 4, and click the Get SalesRep button, I get the SalesRep but no change to the CreditLimit. The grid doesn't display the Balance value, but if I superimpose the little **`FOR EACH Customer`** result shown below, you can see that, whereas the Balance for Customer 2 satisfies the criterion of being less than half the

CreditLimit, Customer 4 does not, so there was no change to the CreditLimit value that was passed back. A more responsive user interface that makes all this clearer is left as a Silverlight exercise for the reader.



The complete code for all the pieces of this second example is repeated here. First, the ABL procedure:

```
/*------------------------------------------------------------------------
    File        : GetRepAndCL.p

    Notes       : Returns the SalesRep RepName and adjusts CreditLimit.
  ----------------------------------------------------------------------*/

DEFINE INPUT        PARAMETER piCustNum      AS INTEGER.
DEFINE INPUT-OUTPUT PARAMETER pdCreditLimit  AS DECIMAL.
DEFINE OUTPUT       PARAMETER pcSalesRepName AS CHARACTER.

FIND Customer WHERE Customer.CustNum = piCustNum.
IF (pdCreditLimit = Customer.CreditLimit) AND
 (Customer.Balance < (Customer.CreditLimit / 2)) THEN
  pdCreditLimit = Customer.CreditLimit * 1.2.

FIND Salesrep WHERE Salesrep.SalesRep = Customer.SalesRep.
pcSalesRepName = Salesrep.RepName.
```

Next, the parameter class that defines the output parameters in the DomainService class:

```
        [Serializable]
        public class GetRepAndCLParams
        {
            public decimal inoutCreditLimit { get; set; }
            public string outRepname { get; set; }
        }
```

Next, the Invoke method in the DomainService class that invokes the ABL procedure through the proxy:

```
[Invoke]
public GetRepAndCLParams GetRepAndCL
    (int inputCustNum, decimal inoutCreditLimit)
{
    GetRepAndCLParams outParams = new GetRepAndCLParams();
    string outRepname;

    try
    {
        _appObj.GetRepAndCL(inputCustNum, ref inoutCreditLimit,
            out outRepname);
        outParams.inoutCreditLimit = inoutCreditLimit;
        outParams.outRepname = outRepname;
    }
    catch (System.Exception e)
    {
        throw new Exception(e.Message);
    }
    return outParams;
}
```

And finally, the modified click event handler in CustOrderPage.xaml.cs, and its completion event handler:

```
private void SalesRepButton_Click(object sender, RoutedEventArgs e)
{
    CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
    if (selectedCust != null)
    {
        /* InvokeOperation<string> invokeOp =
            CODomainContext.GetSalesRep(selectedCust.CustNum,
              OnSRB_Completed, null); */

        InvokeOperation<GetRepAndCLParams> invokeOp =
            CODomainContext.GetRepAndCL(selectedCust.CustNum,
                selectedCust.CreditLimit, OnSRB_CL_Completed, null);
    }
}
```

```
private void OnSRB_CL_Completed(InvokeOperation<GetRepAndCLParams>
      invokeOp)
{
    if (invokeOp.HasError)
    {
        MessageBox.Show(string.Format("Method Failed: {0}",
              invokeOp.Error.Message));
        invokeOp.MarkErrorAsHandled();
    }
    else
    {
        RepnameBox.Text = invokeOp.Value.outRepname;
         CustOrder selectedCust = (CustOrder)CustomerGrid.SelectedItem;
        selectedCust.CreditLimit = invokeOp.Value.inoutCreditLimit;
    }
}
```

This brings me to the end of the series of presentations on using Silverlight with WCF RIA Services. I hope this information has been useful. Though there's much to learn about all the controls and capabilities that Silverlight supports, the basic thread of retrieving and updating data from an OpenEdge application via a set of proxies, entity class definitions, and DomaInService classes should be pretty clear to you. At this point the rest is up to you. Thanks for watching and reading.