UPDATING DATABASE DATA WITH RIA SERVICES

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
February 2011

Extending Your OpenEdge Application with an RIA User Interface

Using RIA Services with Silverlight : Updating Database Data

John Sadd

BUSINESS MAKING PROGRESS

Progress OpenEdge

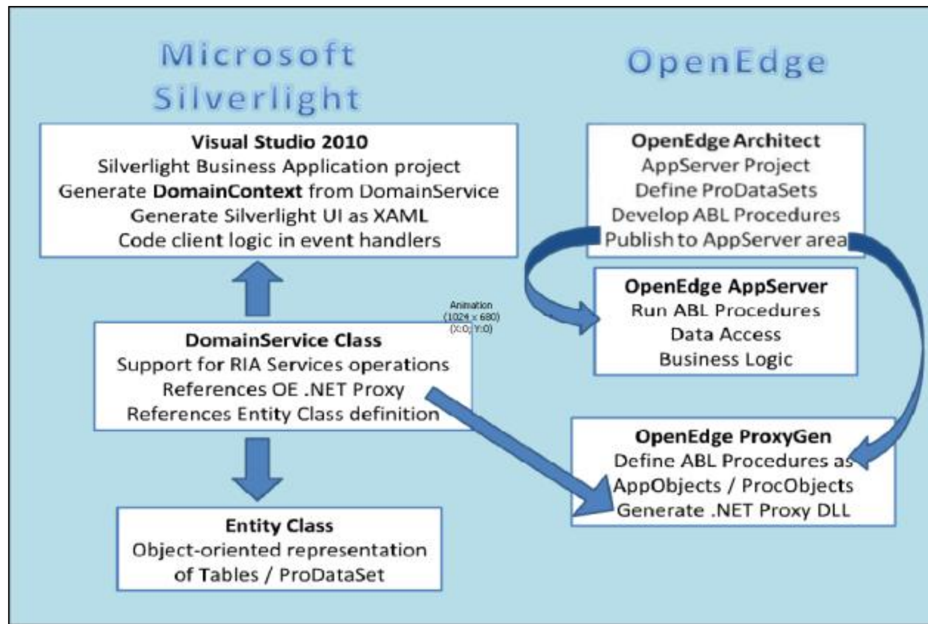
PROGRESS SOFTWARE

PROGRESS SOFTWARE

DISCLAIMER

Certain portions of this document contain information about Progress Software Corporation's plans for future product development and overall business strategies. Such information is proprietary and confidential to Progress Software Corporation and may be used by you solely in accordance with the terms and conditions specified in the PSDN Online (http://www.psdn.com) Terms of Use (http://psdn.progress.com/terms/index.ssp). Progress Software Corporation reserves the right, in its sole discretion, to modify or abandon without notice any of the plans described herein pertaining to future development and/or business development strategies. Any reference to third party software and/or features is intended for illustration purposes only. Progress Software Corporation does not endorse or sponsor such third parties or software.

This document continues the series that begins with the video series and document entitled *Using RIA Services with Silverlight.* Having gotten Customer and Order data all the way from OpenEdge and the AppServer to the Silverlight user interface, it's time to take a look at a simple update operation to return changes back to the database. I'll go through most of the same steps I've gone through before, as illustrated here:



I have a new ABL procedure to save customer updates back to the database, which will also run in my OpenEdge AppServer, as the GetCustOrders.p procedure does.

I'll edit the proxy to add this second procedure to the same AppObject used by the other examples.

I'll then have to make some changes to the data definitions in the entity class.

I'll need to make some substantial additions to the DomainService class, mostly to transform the object-oriented representation of a changed row into an ADO.NET dataset that the DomainService can then pass back to the proxy, which in turn runs the ABL procedure that applies the changes to the database.

Finally, I can regenerate the DomainContext class so that it includes support for the update methods. Then I'll add an update button to the user interface and an event handler to pass the event on to the rest of the supporting C# code.

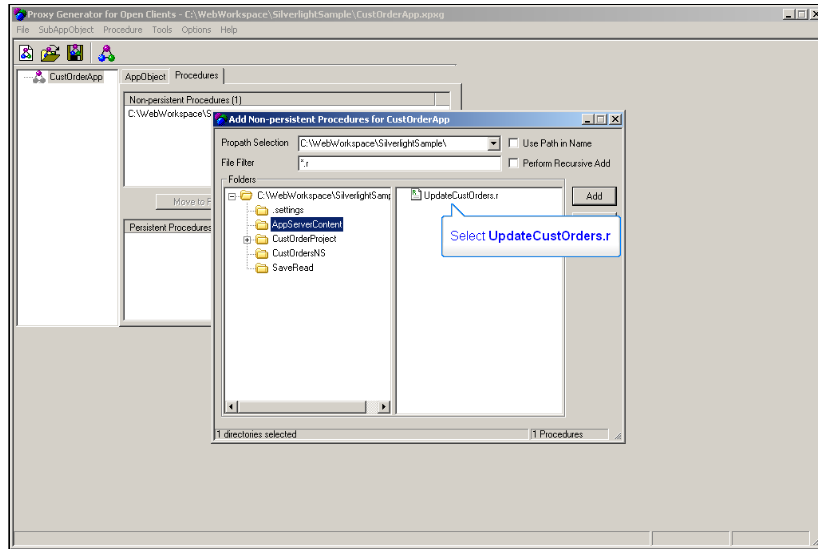
Remember that I'm building the ABL procedures so that each one -- each service in effect -- is a separate non-persistent procedure, so that the application doesn't do any AppServer binding.

The following code is a simple ABL update procedure, which takes the CustOrders ProDataSet as an **INPUT-OUTPUT** parameter and uses the ABL **SAVE-ROW-CHANGES** method to save any changes back to the database. I have added a couple of **MESSAGE** statements to confirm that the user interface sent back what I expect to the ABL procedure:

```
/*------------------------------------------------------------------------
    File        : UpdateCustOrders.p
    Notes       :
  ----------------------------------------------------------------------*/
    {DSCustOrders.i}

    DEFINE INPUT-OUTPUT PARAMETER DATASET FOR CustOrders.

    DEFINE DATA-SOURCE dsCust  FOR Customer.
    DEFINE DATA-SOURCE dsOrder FOR Order.

    BUFFER ttCustomer:ATTACH-DATA-SOURCE (DATA-SOURCE dsCust:HANDLE ).
    BUFFER ttOrder:ATTACH-DATA-SOURCE (DATA-SOURCE dsOrder:HANDLE ).
    OUTPUT TO "c:\AppServerDeploy\UpdateCustOrders.txt".
    MESSAGE "In UpdateCustOrders" SKIP.

    FOR EACH ttCustomerBefore TRANSACTION:
        MESSAGE "Found CustomerBefore " ttCustomerBefore.CustNum
                                        ttCustomerBefore.Name SKIP.
        FIND ttCustomer WHERE ttCustomer.CustNum = ttCustomerBefore.CustNum.
        MESSAGE "ttCustomer is " ttCustomer.CustNum ttCustomer.Name SKIP.
        BUFFER ttCustomerBefore:SAVE-ROW-CHANGES ().
    END.
    FOR EACH ttOrderBefore TRANSACTION:
        BUFFER ttOrderBefore:SAVE-ROW-CHANGES ().
    END.
```

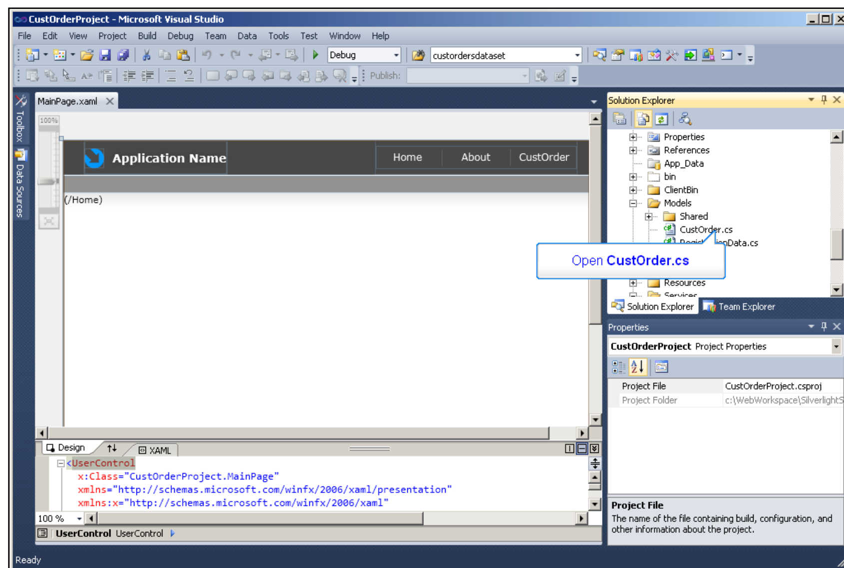
Here you can see that the procedure sends a message to a log file with the before and after records of the **CustNum** and **Name** fields. If I make a change to a Customer Name, I should see that in the log file. I save this new procedure in the **AppServerContent** directory, so my AppServer-enabled project in Architect copies the source and .r files to the **AppServerDeploy** directory in the AppServer's ProPath.

Back in ProxyGen, I reopen the **CustOrderApp** proxy project. Previously I had added **GetCustOrders.p** as the first procedure in this AppObject. Now I'm going to add the update procedure to the same AppObject. Selecting the **Procedures** tab, I right-click to add another non-persistent procedure. In the AppServerContent directory, I find **UpdateCustOrders.r**.



I add that to the proxy, and click the **Generate** button. I don't need to change any settings, and a new version of the proxy DLL is generated that contains references to both my procedures. That's all I need to do in ProxyGen.

Back in Visual Studio, I first need to re-open the entity class file that defines the parent and child tables as C# objects. That's **CustOrder.cs** in the Web project's **Models** folder:



I haven't changed my data definitions, so why do I need to change this file?

Well, it turns out that between Silverlight version 3 and Silverlight 4, Microsoft decided that it was inefficient to return all field values from the client back to the server when you do an update. So by default, when you ask for the before image of a row in the DomainService class, it's populated only with values marked as **Keys**. Even values that were actually changed aren't automatically populated. So in Silverlight 4, you have to prepend a new annotation to each and every property definition in the

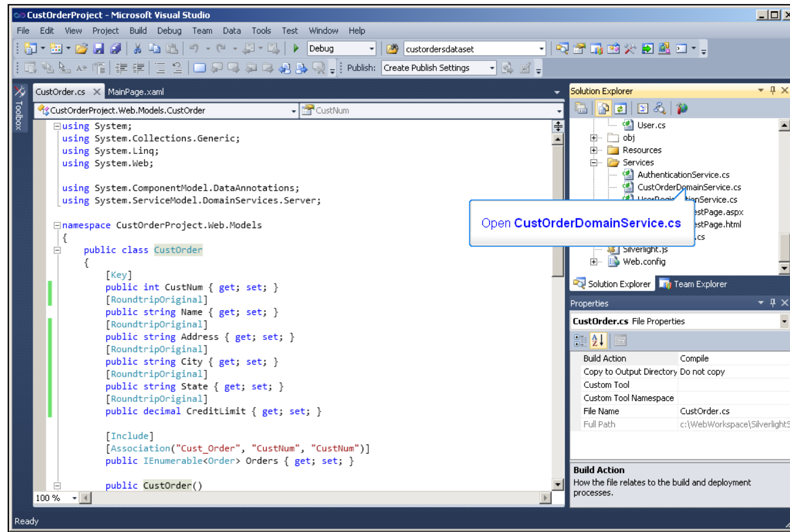
entity class, unless for some reason it's a field you would never want returned to the server. Keep in mind that the ABL **SAVE-ROW-CHANGES** method compares all the fields in the before table row with the values in the database to see if anyone has changed the database record after you read it, so you really have to return all the field values.

The new annotation is called **RoundtripOriginal**. Be sure to spell it with just capital **R** and capital **O**. The **CustNum** property, identified as a **Key**, will be populated automatically. so it's all the other properties you have to annotate, as shown here:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

using System.ComponentModel.DataAnnotations;
using System.ServiceModel.DomainServices.Server;

namespace CustOrderProject.Web.Models
{
    public class CustOrder
    {
        [Key]
        public int CustNum { get; set; }
        [RoundtripOriginal]
        public string Name { get; set; }
        [RoundtripOriginal]
        public string Address { get; set; }
        [RoundtripOriginal]
        public string City { get; set; }
        [RoundtripOriginal]
        public string State { get; set; }
        [RoundtripOriginal]
        public decimal CreditLimit { get; set; }

        [Include]
        [Association("Cust_Order", "CustNum", "CustNum")]
        public IEnumerable<Order> Orders { get; set; }

        public CustOrder()
        {
            Orders = new List<Order>();
        }


    }
    public class Order
    {
        [Key]
        public int OrderNum { get; set; }
        [Key]
        public int CustNum { get; set; }
        public System.DateTime OrderDate { get; set; }
        public string Carrier { get; set; }
        public string SalesRep { get; set; }
    }
}
```

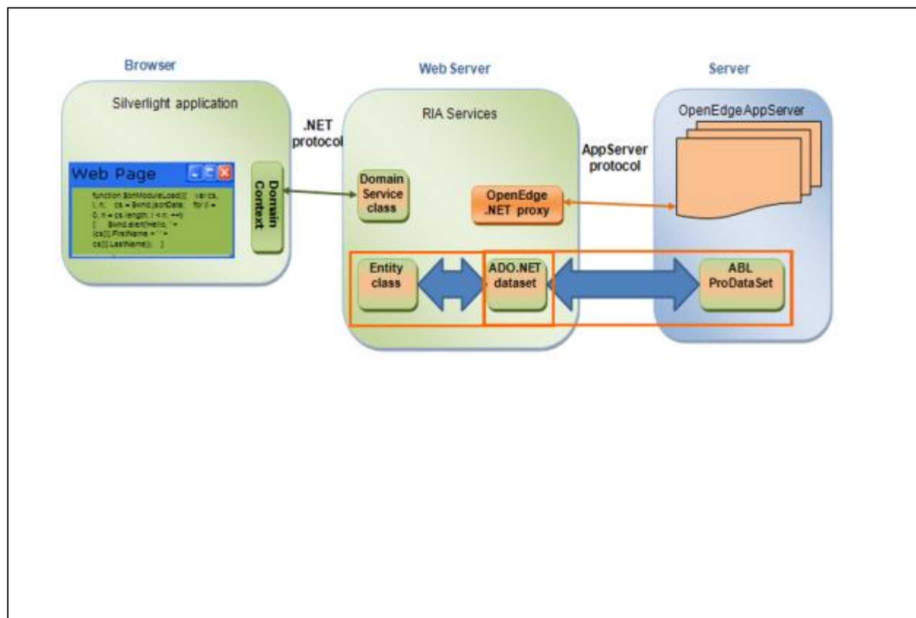
To keep the code in the example as simple as possible, I'm not allowing for changes to fields in the Order table, but you can make the same kinds of changes to those values as well, to support updating both Customers and Orders.

Now I need to start making the more significant changes to the DomainService class. I open the file as it stands after the first series of videos (described in the white paper *Using RIA Services with Silverlight)*, with all the code to support reading data:



I scroll down to the bottom to start adding several new methods to support update operations. The primary job these new methods have to do is to transform the object-oriented form of the Customer and Order data back into a ProDataSet.

Here's another version of the overall architecture diagram that illustrates these steps:



Remember that part of the job of the .NET proxy is to transform a ProDataSet that's passed as a parameter from an ABL procedure into the equivalent ADO.NET DataSet that .NET can use. The entity class is then an object-oriented representation of that DataSet. When the client passes an update back to the server, the code in the DomainService class needs to take the object representation of the before and after rows and turn them back into an ADO.NET DataSet, so that the proxy can in turn pass that back to OpenEdge as a ProDataSet with its before and after tables.

First I add a couple of little support methods to the DomainService class. The first is called **CreateCustOrder**. **CustOrder** is the top-level class that holds Customer field values and a collection of Orders. This method takes an entity representation of a CustOrder and create a DataTable row that is its equivalent:

```
        private DataRow CreateCustOrder(CustOrder entity)
        {
```

I define a **DataRow** called **newRow**. In the CustOrder dataset I identify the table corresponding to **ttCustomer**, and add the new row to that:

```
            DataRow newRow = CustOrderDS.Tables["ttCustomer"].NewRow();
```

When the **Submit** code is called in the DomainService to start the update process, the CustOrder DataSet starts out empty, so this will be the first row in it. I assign the **CustNum** field in the new row to the **CustNum** value from the entity:

```
            newRow["CustNum"] = entity.CustNum;
```

Then I do the same for the remaining fields, and return the new row:

```
            newRow["Name"] = entity.Name;
            newRow["Address"] = entity.Address;
            newRow["City"] = entity.City;
            newRow["State"] = entity.State;
            newRow["CreditLimit"] = entity.CreditLimit;

            return newRow;
        }
```

As with all this code, updates are enabled only for Customer; the code could be extended to do the same for Orders as well.

The next support method is called **FindCustomer**. After I've added a row to the initially empty CustOrder DataSet at the start of an update, the code needs to locate it to generate an after version of the row, and this method just finds it in the DataSet by its key value. In the simple case of submitting a single update to the server, there will only be one row in the table at this point:

```
        private DataRow FindCustomer(int custNum)
        {
            foreach (DataRow custrow in CustOrderDS.Tables["ttCustomer"].Rows)
                if (custrow.Field<int>("CustNum") == custNum)
                    return custrow;
            return null;
        }
```

The third support method is one that is called as part of the **Submit** operation. I have to identify it as an Update using the **[Update]** data annotation. The method itself takes a CustOrder object as a parameter. The combination of these two things, the

update annotation and the parameter, identifies what method is run in the DomainService to do an update of the CustOrder object:

```
        [Update]
        public void UpdateCustOrder(CustOrder changeRow)
        {
```

I use the **FindCustomer** method just created to return a DataRow that matches the CustNum key, which is used to create an update version of that row:

```
            DataRow updateRow = FindCustomer(changeRow.CustNum);
```

If there is a matching row, I create the update verson field by field. Now I've got the after version of all the field values:

```
            if (updateRow != null)
            {
                updateRow["CustNum"] = changeRow.CustNum;
                updateRow["Name"] = changeRow.Name;
                updateRow["Address"] = changeRow.Address;
                updateRow["City"] = changeRow.City;
                updateRow["State"] = changeRow.State;
                updateRow["CreditLimit"] = changeRow.CreditLimit;
            }
        }
```

Now I have support methods to build up the before and after rows in an ADO.NET DataSet based on an object representation of those rows. It's time to create the principal update support method, called **Submit**. This is an override of an inherited DomainService method, so as soon as I type the name I get a whole skeleton for the method, including the **changeSet** as a parameter, and a statement to invoke the inherited behavior.

```
public override bool Submit(ChangeSet changeSet)
        {
```

The first thing I add to that is a variable to hold the return value from other methods:

```
            bool result = true;
```

In the usual **try** block, I look at each changed row in the **changeSet** passed in:

```
            try
            {
                foreach (ChangeSetEntry changeRow in changeSet.ChangeSetEntries)
                {
```

This **changeSet** is constructed when I invoke **SubmitChanges** from the user interface support code, and is parallel to the rows in a modified DataSet -- either ADO.NET DataSet or ABL ProDataSet -- with before and after versions of modified rows, and

Inserts and Deletes as well if the code supported those operations. In principle this could be any number of updates, inserts, and deletes, but I'm just supporting updates to start with.

I next check the entity type of the row; there could be multiple different entities, like CustOrder and Order. If it's the top-level **CustOrder**, I'm prepared to deal with it. Remember that I'm leaving Order updates out of the code for now:

```
                    if (changeRow.Entity is CustOrder)
                    {
```

I need to extract the before version of the row, which in the object representation is called the **OriginalEntity**:

```
                        CustOrder origEntity = (CustOrder)changeRow.OriginalEntity;
```

Now, remember that I had to add the annotation **RoundtripOriginal** to every property in my entity class, because otherwise only the **Key** values would be populated. This is where that change comes into effect. If I hadn't added that annotation, my **OriginalEntity** would come back to me with only the CustNum field populated, because that's identified as a **Key**. All the other fields would be null, and if I passed that back to OpenEdge my **SAVE-ROW-CHANGES** method in the ABL update procedure wouldn't succeed. So adding that annotation to all the properties in the entity means that all the before values will be set in the object I get back here.

Now I'm going to use my local **CreateCustOrder** method to create a DataTable row based on this OriginalEntity. That's my before table row:

```
                        DataRow newRow = CreateCustOrder(origEntity);
```

Then I add that row to the DataSet.

```
                        CustOrderDS.Tables["ttCustomer"].Rows.Add(newRow);
```

I turn that into a before image in the DataSet by invoking **AcceptChanges**, so its data values aren't considered new and modified anymore. This is exactly equivalent to what you do to a row in a ProDataSet temp-table on the OpenEdge side, where there is an **ACCEPT-CHANGES** method as well:

```
                        newRow.AcceptChanges();
                    }
                }
```

Here is where the code invokes the standard **Submit** method that the local method is overriding:

```
                result = base.Submit(changeSet);
```

Now you need to understand the next bit of built-in behavior that happens when you submit changes. The inherited **Submit** method looks at the change set you pass to it, and for each change, it invokes another support method in your DomainService class. It does this by examining the data annotations in front of the local methods in your DomainService, along with their signatures.

I have a method annotated as an **[Update]**, which takes a **CustOrder** entity as a parameter. So when I invoke the base **Submit** method, it turns around and invokes **UpdateCustOrder** in my DomainService based not on the method name itself, but on the data annotation and the signature. And you remember that my **UpdateCustOrder** method creates the after table row from the **changeSet**. Back in the **Submit** method, I don't want to return after invoking the base version of **Submit**, because I'm not done yet. So I just save off the return value in my result variable.

If the **changeSet** has any errors I return false:

```
            if (changeSet.HasError)
                return false;
```

Now I've got an ADO.NET dataset with a representation of any changes that took place in the client that my .NET proxy will be able to accept.

Having transformed the change set data into a form the OpenEdge .NET proxy can in turn convert to a ProDataSet with change tables, the next action is to invoke the ABL update procedure via the proxy, using the AppObject:

```
            _appObj.UpdateCustOrders(ref CustOrderDS);
        }
```

Remember that even though each ABL procedure is a separate non-persistent .p, my one AppObject -- the proxy DLL that ProxyGen created -- can invoke any and all of them. In the first RIA Services example I put the AppObject instance into a cache so that it could be reused throughout a whole session, but in this case I'm still running the same AppObject instance as I was when I read the data, so I don't even need to retrieve it from the cache.

I pass the **CustOrder** DataSet as a parameter, and I pass it by reference because that corresponds to the **INPUT-OUTPUT** parameter on the OpenEdge side.

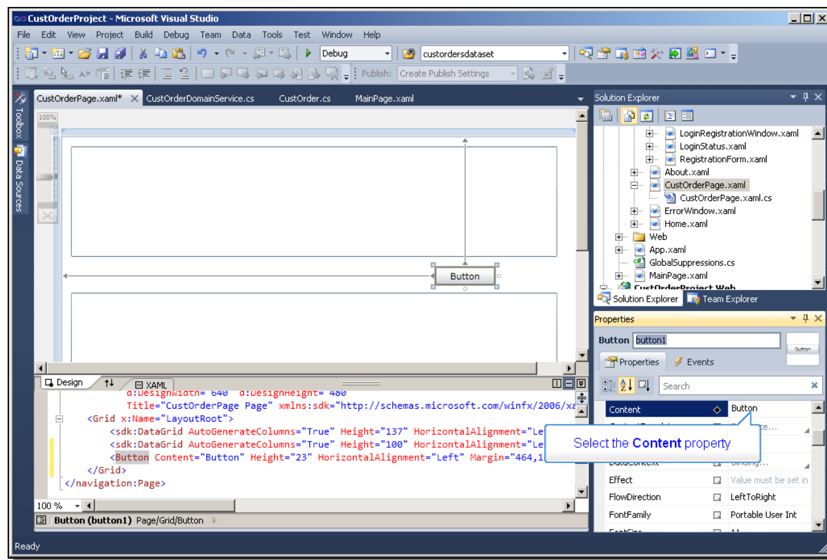
I've executed **UpdateCustOrders.p**, and my changes should be back in the OpenEdge database. As before, I just insert a simple catch block:

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

Now I can save and compile the DomainService class.

Next I need to add a Submit button to the user interface, so that when I make a change to a value in the datagrid, I can tell Silverlight to send it back to OpenEdge. Once again, I'm just enabling saving changes for the Customer grid, to simplify the code. By the way, the grid's **`IsReadOnly`** property is false by default, so my datagrids were always editable; I just wasn't prepared to do anything with any changes before.

I make room under the Customer grid for a new button. I drag a button from the Toolbox to my **`CustOrderPage`**, and call it **`SubmitButton`**, and change the label, which is the **`Content`** property, to **Submit Changes**:



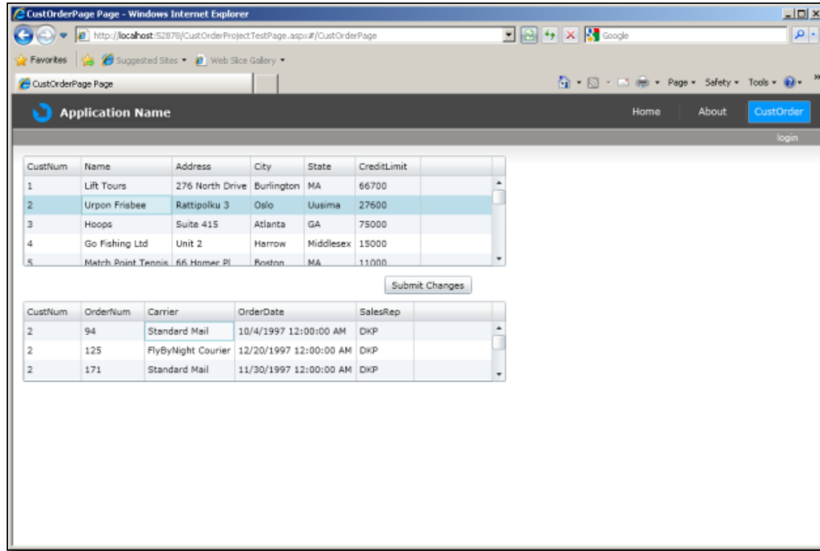
I resize the button to show the new label, and when I double-click on the button, I get a skeleton event handler for the **`Click`** event, the default event for a Button. All this event handler needs to do is check if the DomainContext has registered any changes in the user interface, and if so, to invoke the standard **`SubmitChanges`** method in the DomainContext. Remember that this will turn around and run **`Submit`** in my DomainService class:

```
        private void SubmitButton_Click(object sender, RoutedEventArgs e)
        {
            if (CODomainContext.HasChanges == true)
                CODomainContext.SubmitChanges(OnSubmitCompleted, null);
        }
```

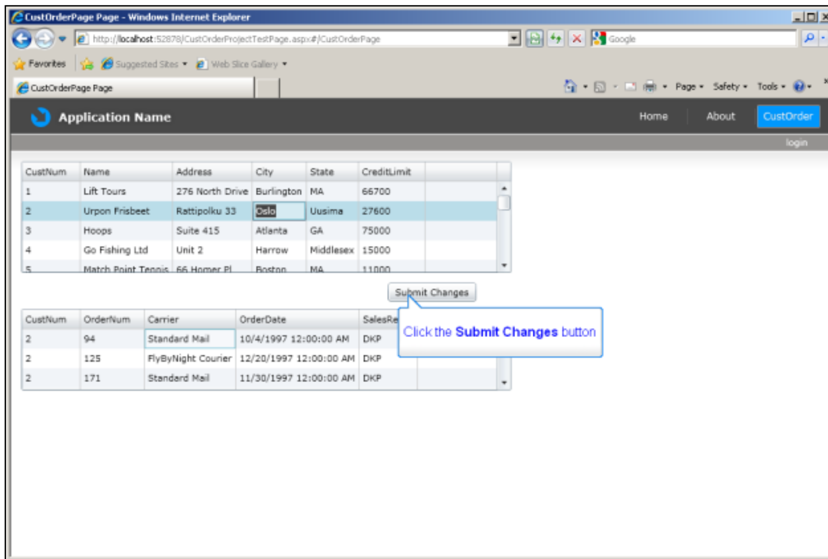
Here's the simple completion event method that checks for errors:

```
        private void OnSubmitCompleted(SubmitOperation submitOp)
        {
            if (submitOp.HasError)
            {
                MessageBox.Show(string.Format(submitOp.Error.Message));
                submitOp.MarkErrorAsHandled();
            }
        }
```

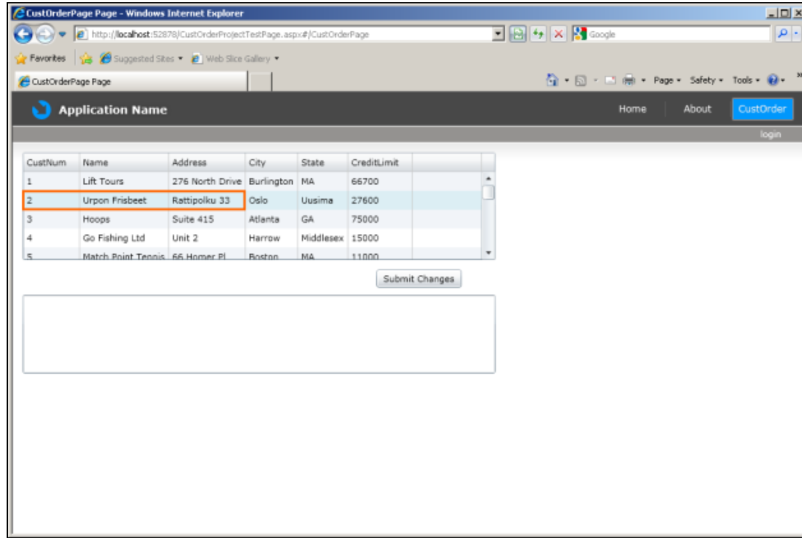
I do a build to get everything regenerated and synchronized. Now I can see what happens when I run the test page for the project. In the Home page I select the CustOrderPage, and then select a Customer name. That causes the Order grid to display the Orders for that Customer because of the **`SelectionChanged`** event defined in an earlier presentation:



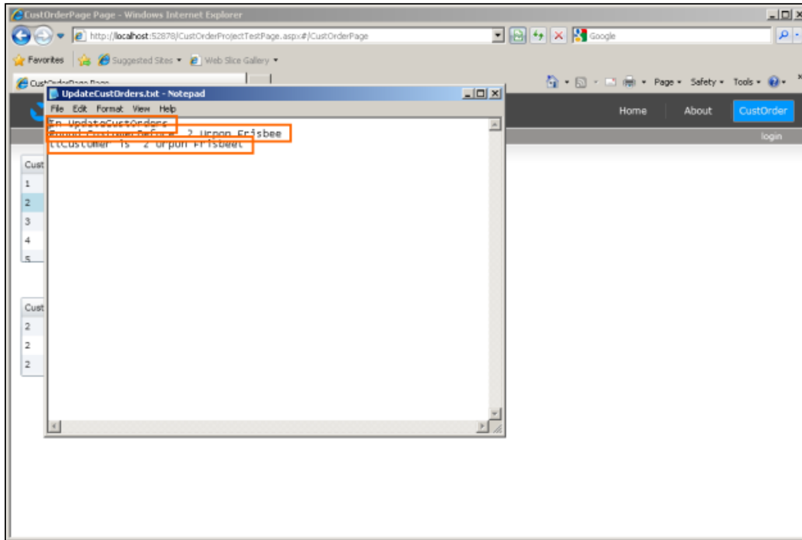
I add a **`t`** to the end of the Customer Name. In the Address field, I change the street number to **`33`**. Then I click the Submit Changes button:



Because this is the simplest of user interfaces, there's no specific affordance to confirm that the **`Submit`** succeeded. I can go back to the Home page, though, and then reselect the CustOrderPage. Because the data is re-retrieved from **GetCustOrders.p**, this shows that my changes went through to the OpenEdge database:



To get a little more reassurance that things worked as I expected, I can take a look at the log file that the **MESSAGE** statements in **UpdateCustOrders.p** were written to. The first message confirms that UpdateCustOrders.p is executing. The procedure found the **ttCustomerBefore** table row with the original value of the Name field, as well as the CustNum. And the **ttCustomer** table row, which has the changes, shows the updated value of the Name. (The message statement doesn't display the Address, though that was also modified.)



That's the end of the update sequence. As with the other parts of this series, there are clearly a number of steps involved. But given this concrete example, you should be able to adapt it to handle updates for your own data as well. All the DomainService code is fairly mechanical, and just deals with the data transformation and data binding from OpenEdge to .NET. The application-specific logic should all be in your ABL procedures on one end, and in your user interface design on the other.

To review, all of the new methods in **CustOrderDomainService.cs** are shown here:

```
        private DataRow CreateCustOrder(CustOrder entity)
        {
            DataRow newRow = CustOrderDS.Tables["ttCustomer"].NewRow();

            newRow["CustNum"] = entity.CustNum;
            newRow["Name"] = entity.Name;
            newRow["Address"] = entity.Address;
            newRow["City"] = entity.City;
            newRow["State"] = entity.State;
            newRow["CreditLimit"] = entity.CreditLimit;

            return newRow;
        }

        private DataRow FindCustomer(int custNum)
        {
            foreach (DataRow custrow in CustOrderDS.Tables["ttCustomer"].Rows)
                if (custrow.Field<int>("CustNum") == custNum)
                    return custrow;
            return null;
        }

        [Update]
        public void UpdateCustOrder(CustOrder changeRow)
        {
            DataRow updateRow = FindCustomer(changeRow.CustNum);
            if (updateRow != null)
            {
                updateRow["CustNum"] = changeRow.CustNum;
                updateRow["Name"] = changeRow.Name;
                updateRow["Address"] = changeRow.Address;
                updateRow["City"] = changeRow.City;
                updateRow["State"] = changeRow.State;
                updateRow["CreditLimit"] = changeRow.CreditLimit;
            }
        }

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

The event handler method and its completion method from **CustOrderPage.xaml.cs** are shown here:

```
private void SubmitButton_Click(object sender, RoutedEventArgs e)
{
    if (CODomainContext.HasChanges == true)
        CODomainContext.SubmitChanges(OnSubmitCompleted, null);
}

private void OnSubmitCompleted(SubmitOperation submitOp)
{
    if (submitOp.HasError)
    {
        MessageBox.Show(string.Format(submitOp.Error.Message));
        submitOp.MarkErrorAsHandled();
    }
}
```

If you combine these with the updated entity class **CustOrder.cs** shown in its entirety earlier in this document, and add the Submit Changes button to the user interface, you will have all the code needed to reproduce the example. Given your understanding of what all the code in the example is doing, you should be able to create similar examples that access your own application data using your own ABL procedures.