USING RIA SERVICES WITH SILVERLIGHT

John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
January 2011

Extending Your OpenEdge Application with an RIA User Interface

Using RIA Services with Silverlight

John Sadd

BUSINESS MAKING PROGRESS™
Progress OpenEdge®
PROGRESS SOFTWARE

DISCLAIMER

This document accompanies a series of presentations introducing Microsoft's RIA Services and their use with the Silverlight user interface framework.

INTRODUCTION

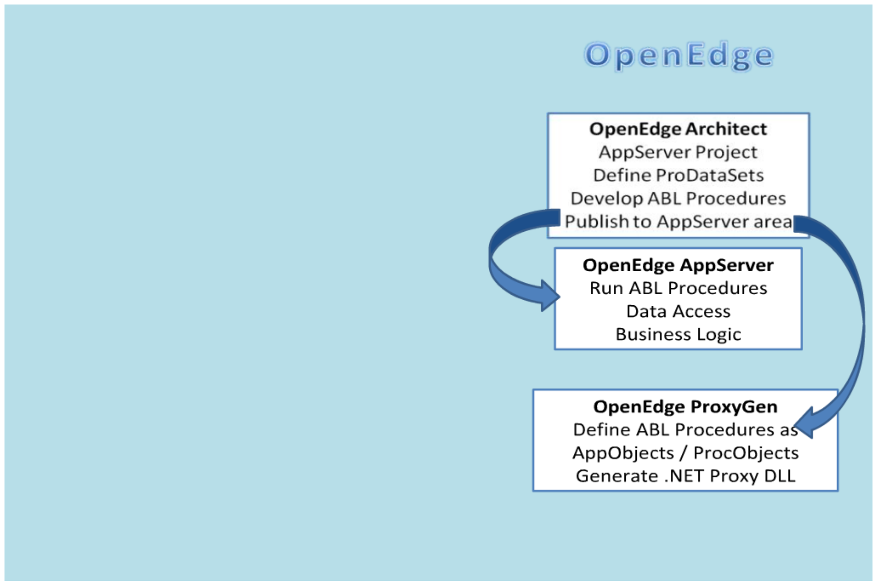
In the first video I show the overall architecture of the pieces you need to put together to create a Silverlight user interface and populate it with data using WCF RIA Services. I then give you a taste of the steps that I will go through in the remaining videos in the series to build those pieces.

There's another session in this RIA series that introduces Silverlight itself, and creates a simple form with a calendar control. But if you want to communicate with an OpenEdge backend, you need to be able to make calls to ABL procedures that return data. You can do that with Web services, but Microsoft has defined a supporting framework they call RIA Services to provide support for standard data read operations, and for updates, creates, and deletes, including building the kind of change set to return to the server that you may be familiar with from using ProDataSets or ADO DataSets. This series introduces you to how to use those services in OE 10.2B.

RIA Services provides a set of **ASP.NET** extensions to help you manage n-tier applications more simply than with Web services. All this is built on the **Windows Communication Foundation**, or **WCF**, which provides a unified programming model under .NET for building distributed service-oriented applications. The data definition part is built on the **ADO.NET Entity Framework**, which provides an object-oriented mapping from relational tables to the object-oriented model that the client application will expect. And finally, the Silverlight UI builds on the **Windows Presentation Foundation**, or **WPF**, which extends the browser capabilities to enable you to create a rich user interface that runs in the browser.

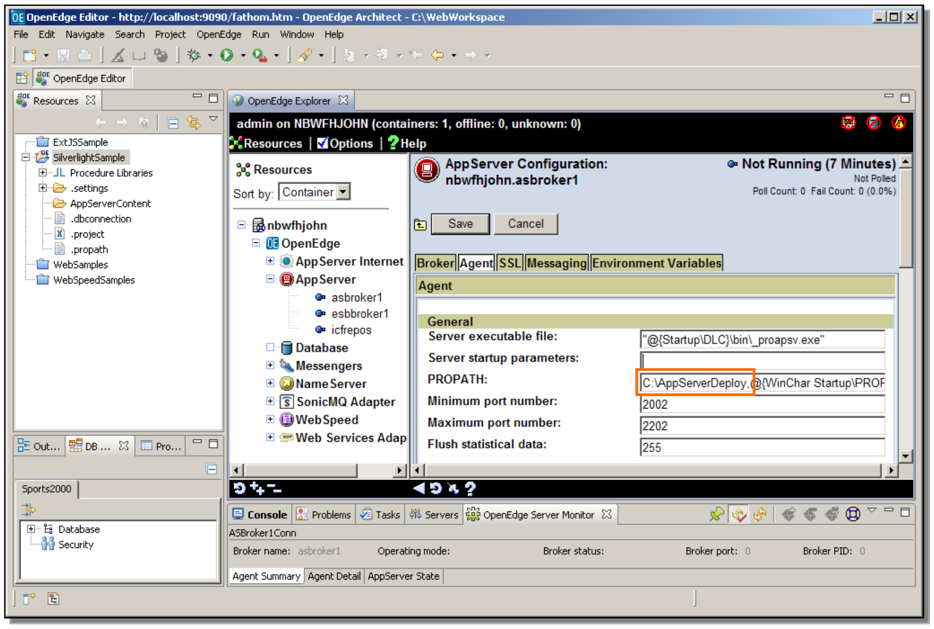
The problem with all this, of course, is that if you're an OpenEdge developer without a lot of experience working directly in .NET and other Microsoft technologies, then most of these terms may not mean a lot to you. So the purpose of this paper and the video series itself is to walk you through a concrete example of everything you have to do to connect OpenEdge on the backend with Silverlight on the front-end in OpenEdge 10.2B. When we're done, I hope you'll be able to use what you've learned to build

Silverlight Uis to some of your own data. I focus on all the steps involved in getting the data binding and data retrieval parts to work. Given that foundation, you should then be able to use and understand the Silverlight documentation to help you build the advanced UI you need for your application. This diagram summarizes the tools and technologies you need to use on the OpenEdge side to prepare for using Silverlight and RIA Services:



First, you'll want to use **OpenEdge Architect** to create a project that supports the **OpenEdge AppServer**, one that publishes content to the area where your AppServer runs. Then you can develop data definitions and application procedures that use those definitions, and publish those procedures where they can be accessed by your AppServer.

In part 2 of this paper and the second video in the RIA Services series I give you a refresher on how to create an AppServer-enabled project that will automatically publish ABL content to a folder where your test AppServer can find it. Your project can use a pre-defined connection to an AppServer on your machine to test with. And you can define which parts of your application get published to the AppServer's ProPath for testing. To do this yourself you'll need a running AppServer on your machine where the parts of Silverlight that run in the Web browser will be able to run your ABL procedures, with all their data access logic and validation and other business logic, so in the video I configure an AppServer in OpenEdge Explorer to make sure that it's running in state-free mode. State-free mode prevents me from locking up an AppServer session while my application is running, and lets me make each call to OpenEdge into a separate, discrete operation. And I make sure that my AppServer agents are configured to use the database connection my user interface will need to retrieve data, and to make sure that the AppServer's ProPath includes the directory where AppServerContent gets published, as shown here:



admin on NBWFHJOHN (containers: 1, offline: 0, unknown: 0)

AppServer Configuration: nbwfhjohn.asbroker1
Not Running (7 Minutes)
Not Polled
Poll Count: 0 Fail Count: 0 (0.0%)

Agent

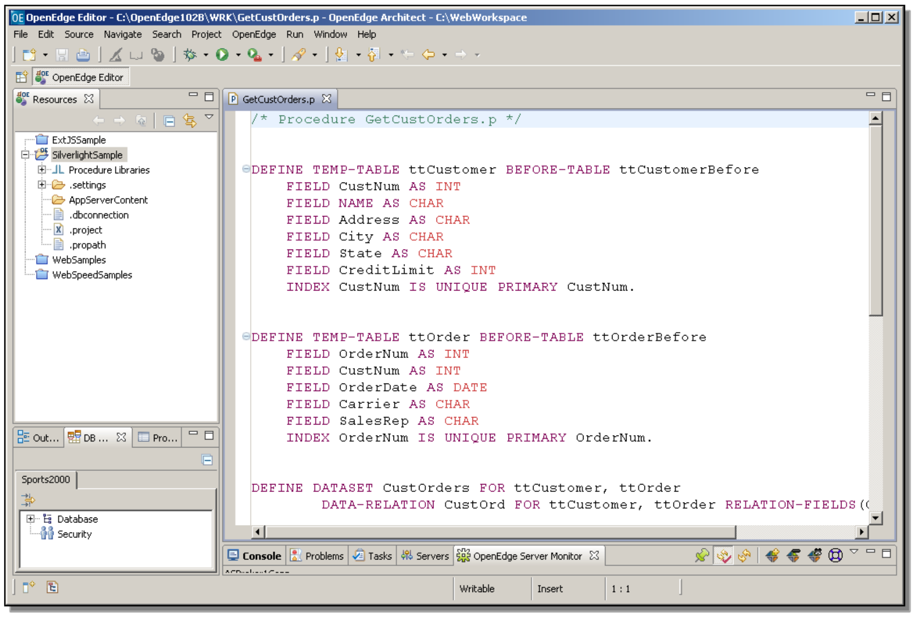
General

Server executable file:	"@{Startup\DLC}\bin_proapsv.exe"
Server startup parameters:	
PROPATH:	C:\AppServerDeploy;@{WinChar Startup\PROF
Minimum port number:	2002
Maximum port number:	2202
Flush statistical data:	255

ASBroker1Conn

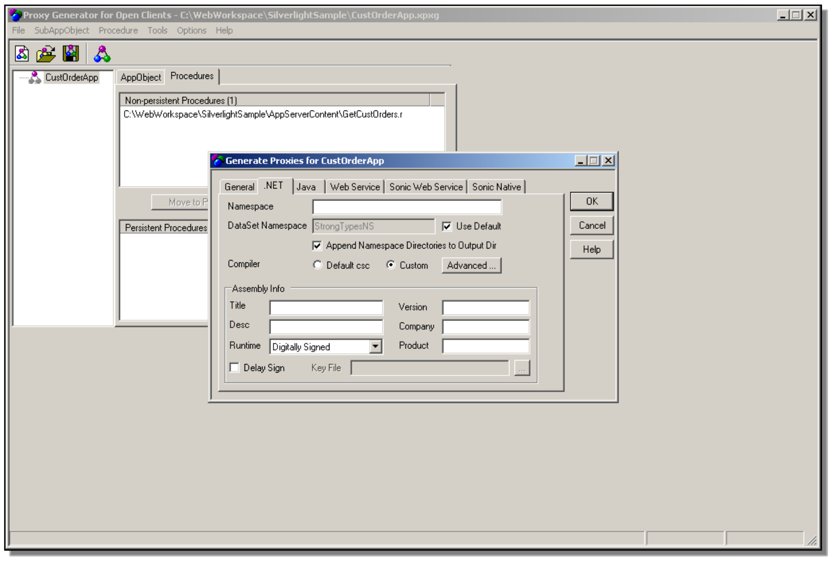
Broker name: asbroker1 Operating mode: Broker status: Broker port: 0 Broker PID: 0

Finally on the OpenEdge side, you need to generate a **.NET Open Client** proxy that describes your ABL procedures to .NET. My example uses the definition for a set of related temp-tables and a ProDataSet that combines them, as shown here:


OpenEdge Editor - C:\OpenEdge102B\WRK\GetCustOrders.p - OpenEdge Architect - C:\WebWorkspace
File Edit Source Navigate Search Project OpenEdge Run Window Help
OpenEdge Editor
Resources
ExtJSSample
SilverlightSample
Procedure Libraries
.settings
AppServerContent
.dbconnection
.project
.propath
WebSamples
WebSpeedSamples
GetCustOrders.p
Sports2000
Database
Security
Console Problems Tasks Servers OpenEdge Server Monitor
Writable
Insert
1 : 1

```
/* Procedure GetCustOrders.p */


DEFINE TEMP-TABLE ttCustomer BEFORE-TABLE ttCustomerBefore
    FIELD CustNum AS INT
    FIELD NAME AS CHAR
    FIELD Address AS CHAR
    FIELD City AS CHAR
    FIELD State AS CHAR
    FIELD CreditLimit AS INT
    INDEX CustNum IS UNIQUE PRIMARY CustNum.


DEFINE TEMP-TABLE ttOrder BEFORE-TABLE ttOrderBefore
    FIELD OrderNum AS INT
    FIELD CustNum AS INT
    FIELD OrderDate AS DATE
    FIELD Carrier AS CHAR
    FIELD SalesRep AS CHAR
    INDEX OrderNum IS UNIQUE PRIMARY OrderNum.


DEFINE DATASET CustOrders FOR ttCustomer, ttOrder
        DATA-RELATION CustOrd FOR ttCustomer, ttOrder RELATION-FIELDS (
```

I create a new proxy in **ProxyGen**, that describes a data read procedure, and later update procedures, for that data. Each ABL procedure becomes a part of an **AppObject** that I later reference from the Silverlight support code. I make them ABL procedures non-persistent .p's so that my AppServer can run state-free. The first is a procedure that returns a ProDataSet with a set of Customers and their Orders. When I generate the proxy, I define the AppServer to run on, and some properties specific to .NET, including a namespace to group all my related calls, as shown here:



Proxy Generator for Open Clients - C:\WebWorkspace\SilverlightSample\CustOrderApp.xpxg

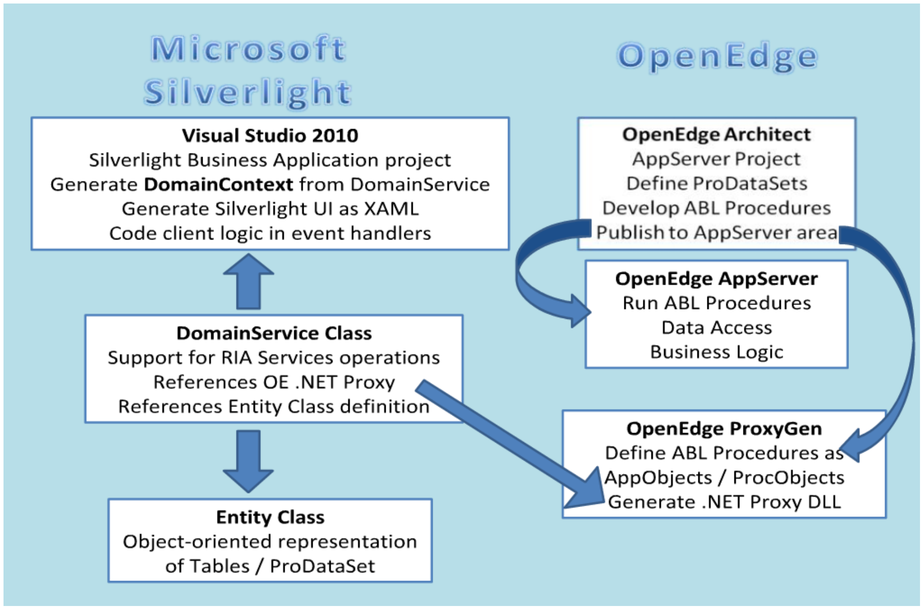
Non-persistent Procedures (1)
C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.r

Generate Proxies for CustOrderApp

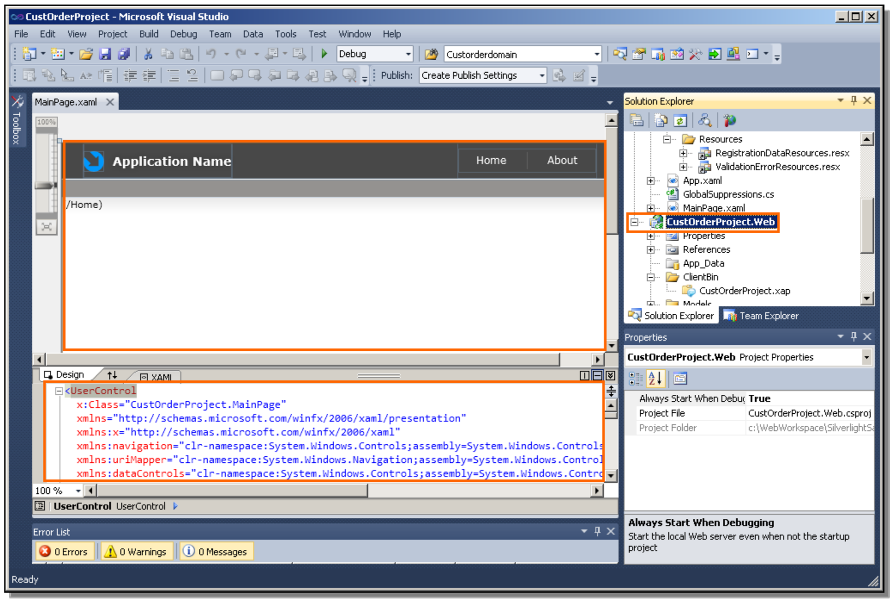
Namespace
DataSet Namespace StrongTypesNS ☑ Use Default
☑ Append Namespace Directories to Output Dir
Compiler ○ Default csc ● Custom
Assembly Info
Title
Version
Desc
Company
Runtime Digitally Signed
Product
☐ Delay Sign Key File

ProxyGen then creates a DLL that I can incorporate into the Silverlight application.

That summarizes the OpenEdge work, which should be pretty familiar to you. On the Silverlight side, you need to use **Visual Studio** to create a **Silverlight Business Application**, and that's what I show you in the third section of this paper and in part 3 of the video series. The work on the Microsoft side of the example is summarized on the left of this diagram:

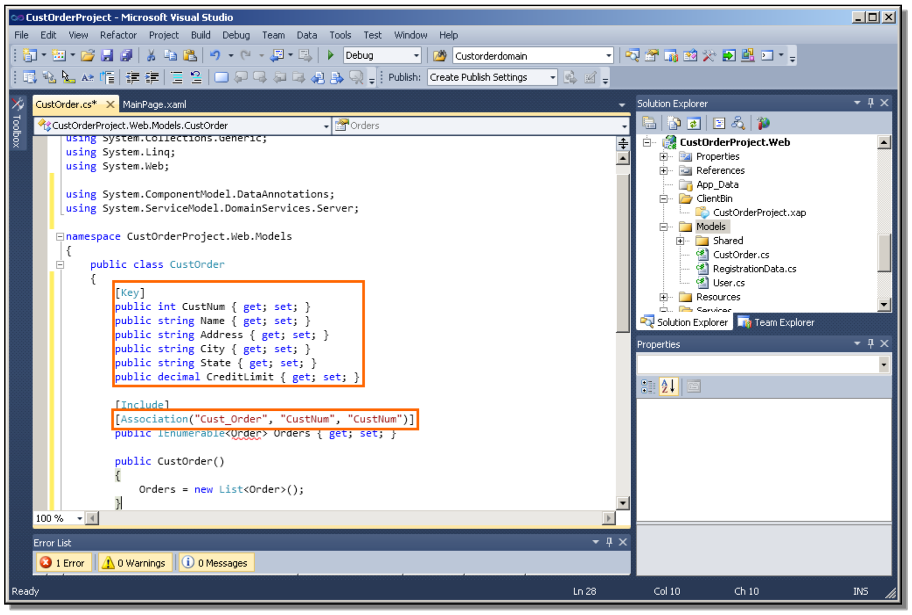


In my example I create a new Visual Studio project, using the pre-defined **Silverlight Business Application** template, which provides a lot of default behavior. It's a project to support reads and updates of my Customer and Order data. I explain a bit about what the Business Application template provides for you. Some of the visual and code elements are shown in this Visual Studio screenshot:



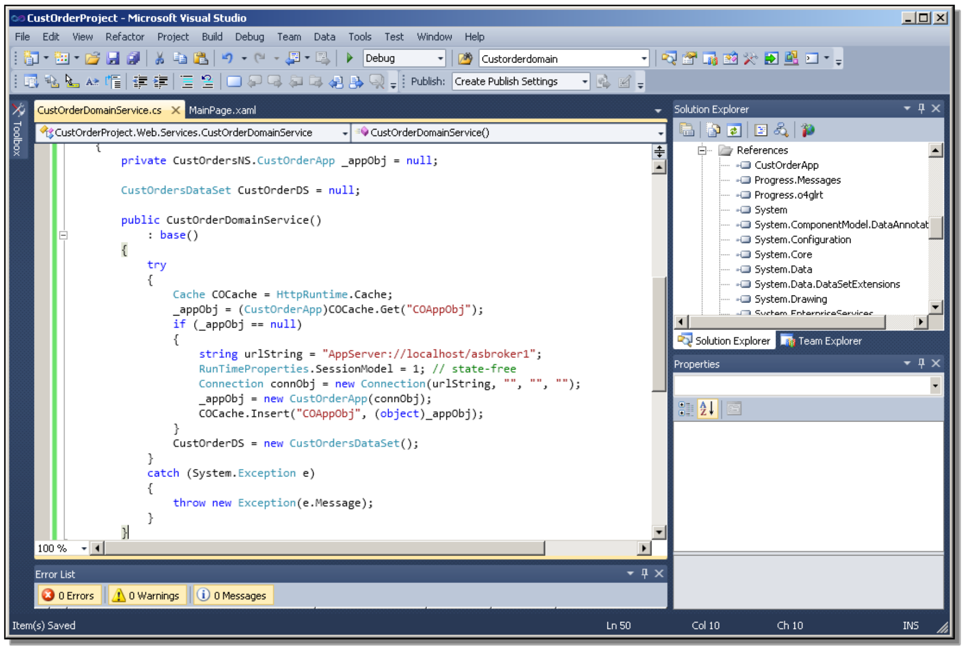
```
<UserControl
  x:Class="CustOrderProject.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls
  xmlns:uriMapper="clr-namespace:System.Windows.Navigation;assembly=System.Windows.Control
  xmlns:dataControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Contro
```

The first Silverlight element that needs to be defined in the project is called an **entity class**, an object-oriented representation of a ProDataSet being passed in from OpenEdge, and the tables that it contains. So in my example I add this to the Models defined in the project. The entity class file is one or more classes that you need to define in a .NET language like C#. In the example the entity class is an object representation of the data for each of the two relational tables in the ProDataSet. Visual Studio provides you with a starting point for the class, and in the third section of this paper and the corresponding video I explain to you just how to represent each table and each field in your tables, as well as the data-relation that defines how they're to be joined, so you wind up with your data defined in a class, like this:



```
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; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public decimal CreditLimit { get; set; }

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

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

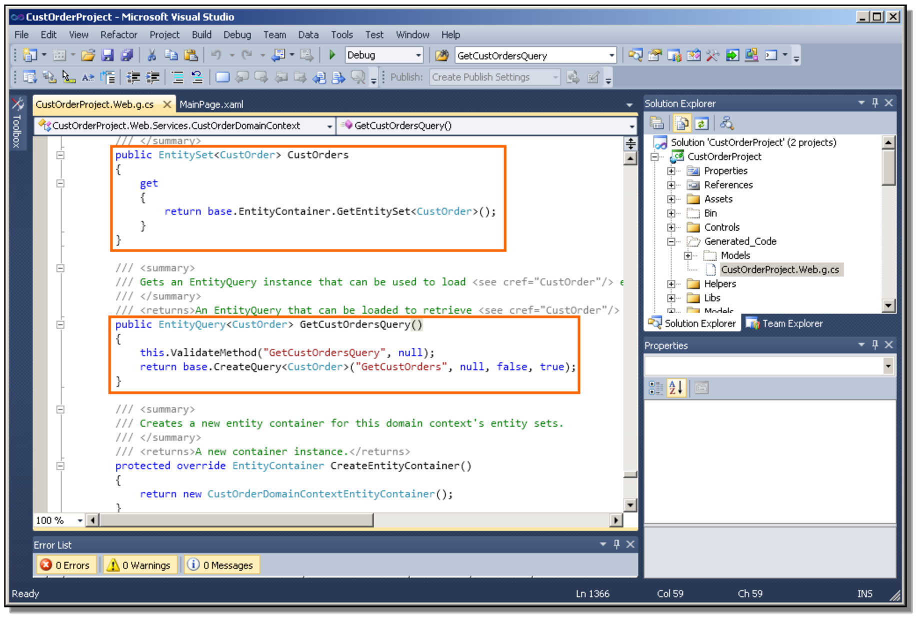
The next big piece of the project is the **DomainService** class. This supports structured operations to read, update, create, and delete data, providing standard service interface types for these operations that you can pass a ProDataSet into. In the fourth section of this paper and in parts 4 and 5 of the video series I create a DomainService in the Services folder of my project. There's a template for it that will get me started. This class runa in the Web browser and acta as the intermediary between the client and my proxy and ABL code on the backend. I add references to it for the DLL that ProxyGen generated, as well as other standard Progress DLLs. There will be a fair amount of C# code that I have to write to fill out the DomainService class, a bit of which is shown below, but I explain the code as I go along in enough detail that you should be able to follow suit with classes to support your own application data.


CustOrderProject - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Team Data Tools Test Window Help
Debug
Custorderdomain
Publish: Create Publish Settings
CustOrderDomainService.cs
MainPage.xaml
CustOrderProject.Web.Services.CustOrderDomainService
CustOrderDomainService()
Toolbox
Solution Explorer
References
CustOrderApp
Progress.Messages
Progress.o4glrt
System
System.ComponentModel.DataAnnotat
System.Configuration
System.Core
System.Data
System.Data.DataSetExtensions
System.Drawing
System.EnterpriseServices
Solution Explorer
Team Explorer
Properties
100 %
Error List
0 Errors
0 Warnings
0 Messages
Item(s) Saved
Ln 50
Col 10
Ch 10
INS

```
    {
        private CustOrdersNS.CustOrderApp _appObj = null;

        CustOrdersDataSet CustOrderDS = null;

        public CustOrderDomainService()
            : base()
        {
            try
            {
                Cache COCache = HttpRuntime.Cache;
                _appObj = (CustOrderApp)COCache.Get("COAppObj");
                if (_appObj == null)
                {
                    string urlString = "AppServer://localhost/asbroker1";
                    RunTimeProperties.SessionModel = 1; // state-free
                    Connection connObj = new Connection(urlString, "", "", "");
                    _appObj = new CustOrderApp(connObj);
                    COCache.Insert("COAppObj", (object)_appObj);
                }
                CustOrderDS = new CustOrdersDataSet();
            }
            catch (System.Exception e)
            {
                throw new Exception(e.Message);
            }
        }
```

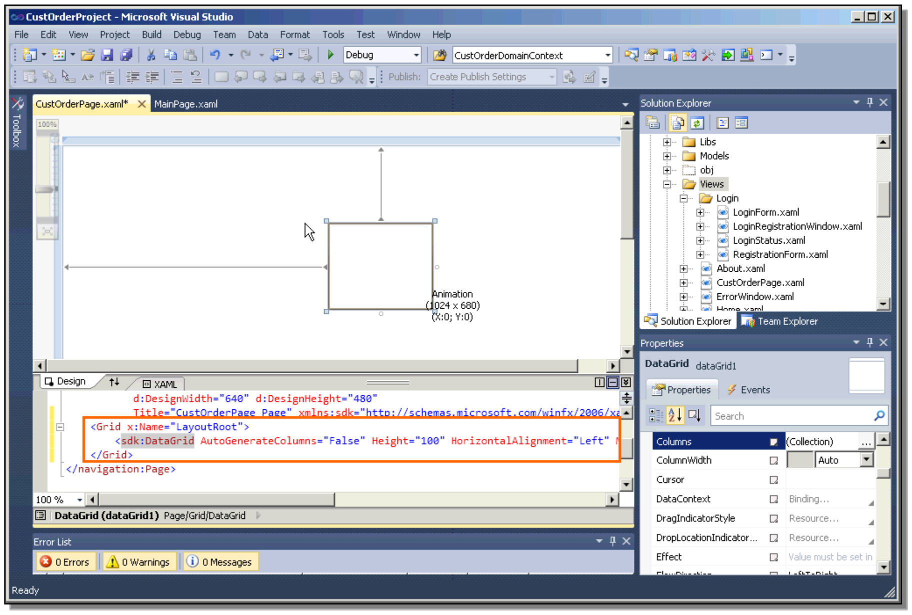
When you build your project with its DomainService, Visual Studio generates what's called the **DomainContext** class, which is the client proxy that makes calls to the DomainService in the Web server to provide support for all your client operations.

You can also use Visual Studio to build the user interface using its WYSIWYG designer, which generates the data definition code they call **XAML**. And you can code client logic like event handlers to respond to events that occur in the user interface. I explain all of this in the fifth section of this paper and in part 6 of the video series. In my example, I do a project build, which generates the DomainContext class. Then I show you some selected bits of that class, even though it's generated code you don't normally look at. There's lot of support for standard operations like authentication, and generated references to your data sets and operations like this query to get Customers and Orders, as shown here:



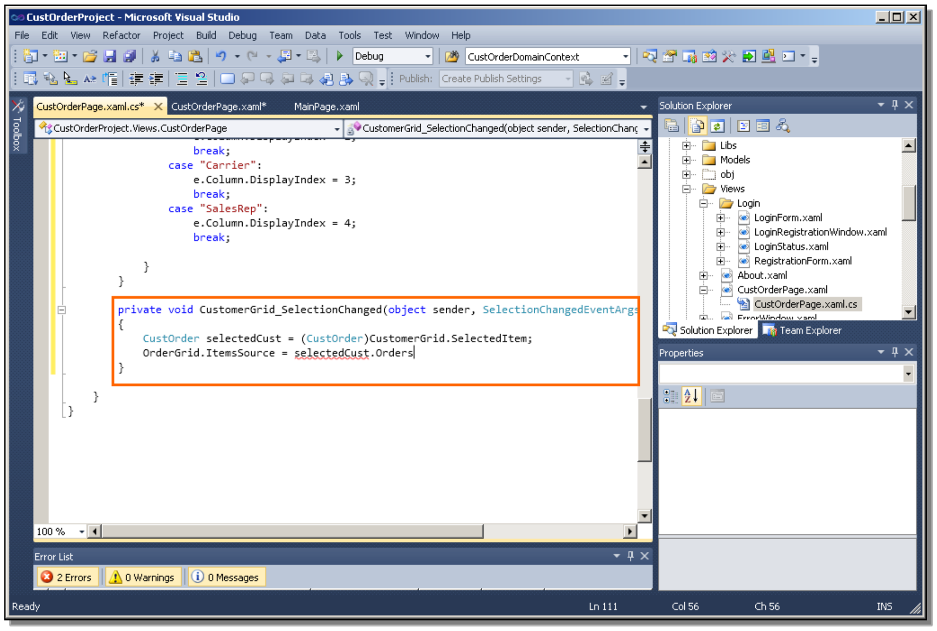
```
/// </summary>
public EntitySet<CustOrder> CustOrders
{
    get
    {
        return base.EntityContainer.GetEntitySet<CustOrder>();
    }
}

/// <summary>
/// Gets an EntityQuery instance that can be used to load <see cref="CustOrder"/> e
/// </summary>
/// <returns>An EntityQuery that can be loaded to retrieve <see cref="CustOrder"/>
public EntityQuery<CustOrder> GetCustOrdersQuery()
{
    this.ValidateMethod("GetCustOrdersQuery", null);
    return base.CreateQuery<CustOrder>("GetCustOrders", null, false, true);
}

/// <summary>
/// Creates a new entity container for this domain context's entity sets.
/// </summary>
/// <returns>A new container instance.</returns>
protected override EntityContainer CreateEntityContainer()
{
    return new CustOrderDomainContextEntityContainer();
}
```

Then on the user interface side, I show you how to edit the XAML to create a new button on the Main Page of the user interface, and then how to create a whole new page of your user interface as a new View in the project, built using the Silverlight Page template. Then I show you how to grab controls from the Toolbox, such as a datagrid, and drop them onto the UI, which generates the XAML you use as a starting point for your user interface definition:



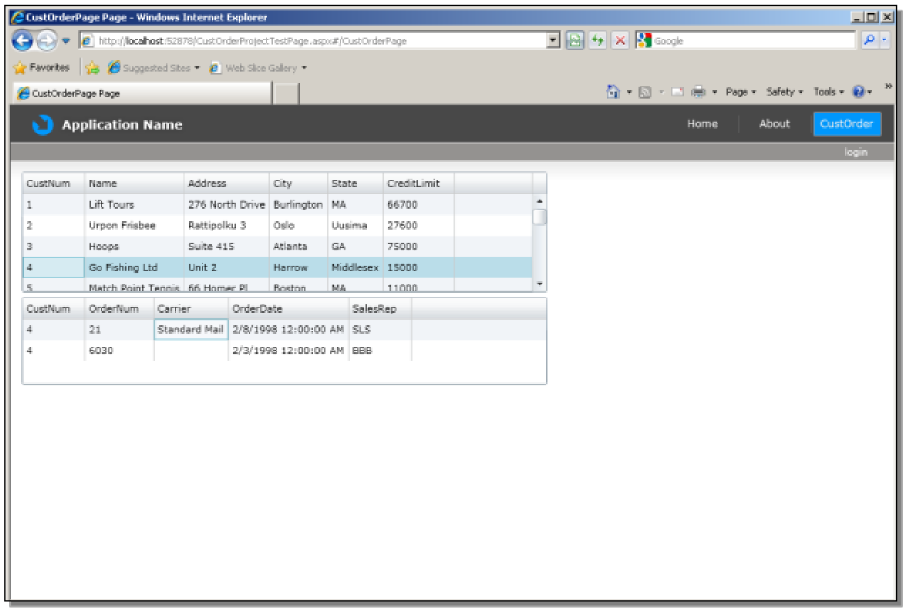
```
        d:DesignWidth="640" d:DesignHeight="480"
        Title="CustOrderPage Page" xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xa
    <Grid x:Name="LayoutRoot">
        <sdk:DataGrid AutoGenerateColumns="False" Height="100" HorizontalAlignment="Left" M
    </Grid>
</navigation:Page>
```

I show you how to define the code that loads data from OpenEdge into the client, and how to set the data source for controls like the grid to display. And I show you how to create an event handler to provide support for events that are triggered in the user interface, such as selecting a customer in its grid:



```
                        break;
                    case "Carrier":
                        e.Column.DisplayIndex = 3;
                        break;
                    case "SalesRep":
                        e.Column.DisplayIndex = 4;
                        break;

                }
            }

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

    }
}
```

And when I run what I wind up with, you'll see a very simple user interface retrieving and displaying data from OpenEdge, which you can then extend to take advantage of all the advanced features that Silverlight provides:



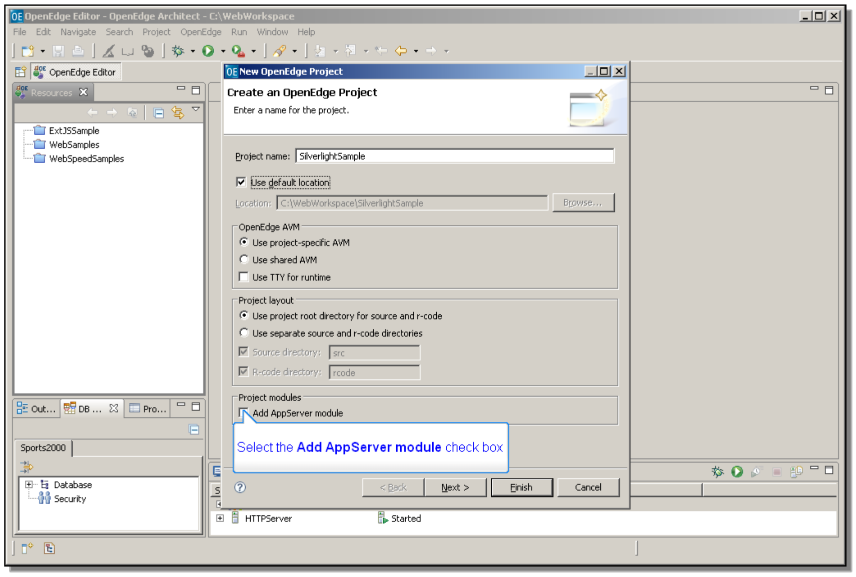
CustNum	Name	Address	City	State	CreditLimit
1	Lift Tours	276 North Drive	Burlington	MA	66700
2	Urpon Frisbee	Rattipolku 3	Oslo	Uusima	27600
3	Hoops	Suite 415	Atlanta	GA	75000
4	Go Fishing Ltd	Unit 2	Harrow	Middlesex	15000
5	Match Point Tennis	66 Homer Pl	Boston	MA	11000

CustNum	OrderNum	Carrier	OrderDate	SalesRep
4	21	Standard Mail	2/8/1998 12:00:00 AM	SLS
4	6030		2/3/1998 12:00:00 AM	BBB

So there's a lot to it, but I do try to explain every step as I go along. The next section of this document goes through all the pieces of the process that use the tools and technologies of OpenEdge.

THE APPSERVER AND THE .NET PROXY

This section corresponds to the second in the series of video presentations introducing WCF RIA Services and Silverlight. In this section I briefly review the support for OpenEdge AppServer in Architect, so that I can define an ABL procedure to run on an AppServer, and then generate a .NET proxy from ProxyGen that will serve as the intermediary between the Silverlight Domain Service and OpenEdge, returning an ABL ProDataSet to display in the Silverlight UI. To manage my ABL procedure from OpenEdge Architect, I need to create a new OpenEdge project, where I will put the ABL sample proceduress that are to be invoked from Silverlight. The key step in making the project AppServer-aware is to add the **AppServer Module** to the project, so that changes I make in Architect will be published to a directory in the AppServer's ProPath:


New OpenEdge Project

Create an OpenEdge Project

Enter a name for the project.

Project name: SilverlightSample

[✓] Use default location

Location: C:\WebWorkspace\SilverlightSample

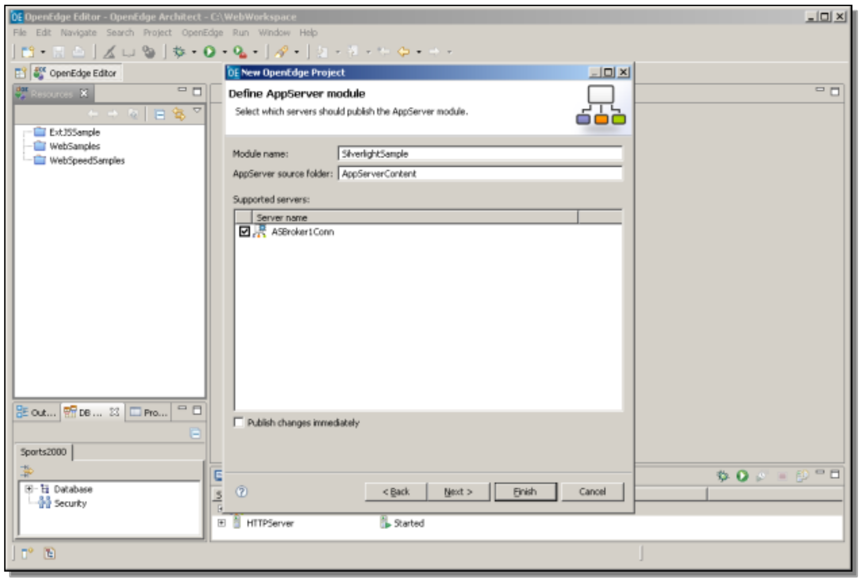
OpenEdge AVM
- (●) Use project-specific AVM
- ( ) Use shared AVM
- [ ] Use TTY for runtime

Project layout
- (●) Use project root directory for source and r-code
- ( ) Use separate source and r-code directories
- [✓] Source directory: src
- [✓] R-code directory: rcode

Project modules
- [ ] Add AppServer module

Select the **Add AppServer module** check box

Continuing through the New Project wizard, I can select the connection object for AppServer **asbroker1** that I have defined in another session in the RIA series:


New OpenEdge Project

Define AppServer module

Select which servers should publish the AppServer module.

Module name: SilverlightSample

AppServer source folder: AppServerContent

Supported servers:

Server name
[✓] ASBroker1Conn

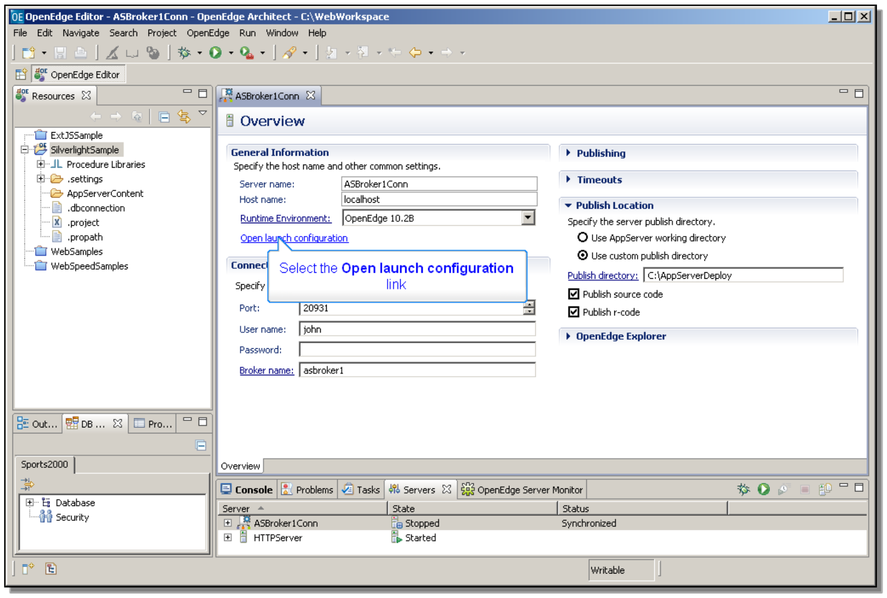
[ ] Publish changes immediately

OpenEdge Editor - OpenEdge Architect - C:\WebWorkspace
File Edit Navigate Search Project OpenEdge Run Window Help
Resources
ExtJSSample
WebSamples
WebSpeedSamples
Sports2000
Database
Security
HTTPServer
Started
< Back
Next >
Finish
Cancel

If there is already ABL code in the source directory named AppServerContent, I can publish that to the AppServer immediately. In other pages of the wizard, I can confirm the ProPath the project uses for development, which includes the **AppServerContent** directory, the source folder for procedures to publish to the

AppServer. The sample procedures use the sports2000 database, so that must be specified as a database connection.

Once the project definition is complete, if you double-click on the AppServer connection in the **Servers View**, Architect brings up an **Overview** page where you can review and modify many aspects of how the connection is configured. Under **Publish Location**, for instance, I can confirm that I have set up this connection to publish AppServer content from the source directory defined for the project the folder named AppServerContent) to a folder called **AppServerDeploy**. This is in the AppServer's ProPath and so will be found when OpenEdge runs the procedures on asbroker1. I can also review settings for the AppServer that are part of its launch configuration, by selecting that from the same Overview page:



General Information
Specify the host name and other common settings.

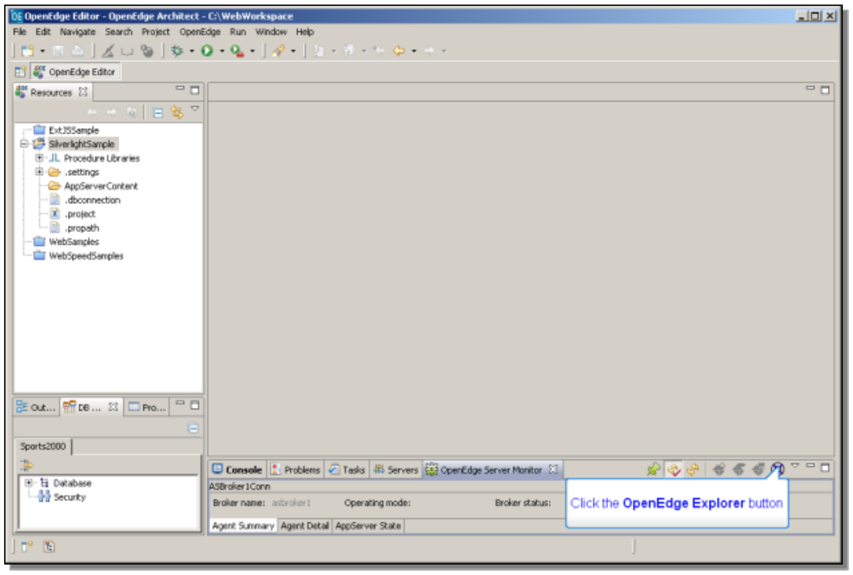
Server name: ASBroker1Conn
Host name: localhost
Runtime Environment: OpenEdge 10.2B

Connect
Specify
Port: 20931
User name: john
Password:
Broker name: asbroker1

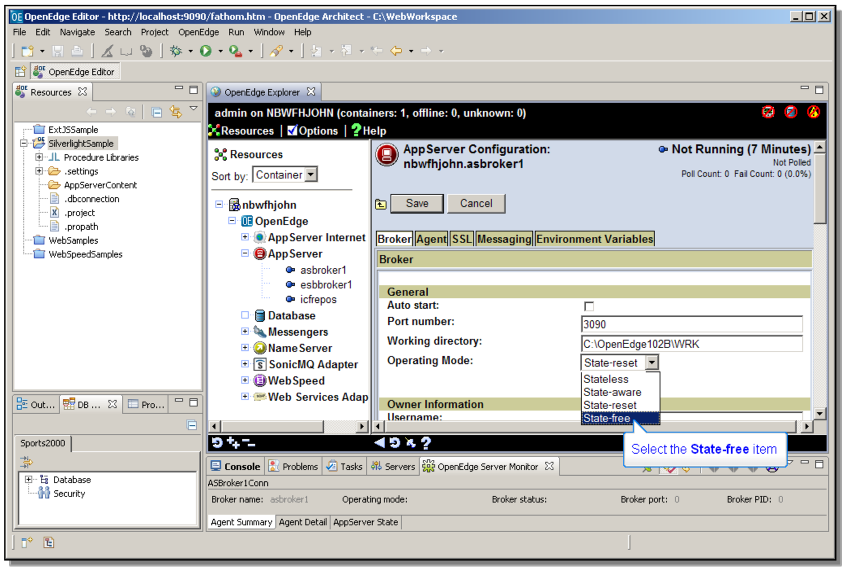
Publish Location
Specify the server publish directory.
Use AppServer working directory [✓ Use custom publish directory]
Publish directory: C:\AppServerDeploy
☑ Publish source code
☑ Publish r-code

Server	State	Status
ASBroker1Conn	Stopped	Synchronized
HTTPServer	Started	

Here I can review the various settings that apply when I run a procedure from the project, including its ProPath, where I can see the AppServerDeploy folder, as well as other settings. In the **Server Monitor View**, which along with the Servers View is a part of Architect's AppServer perspective, I can start OpenEdge Explorer:


OpenEdge Editor - OpenEdge Architect - C:\WebWorkspace
Click the OpenEdge Explorer button

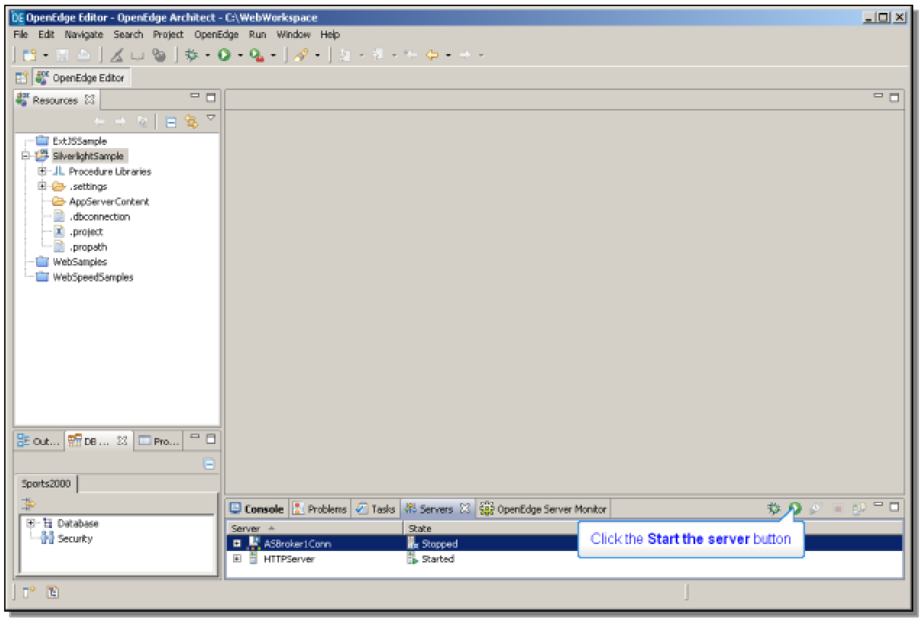
If I then open the properties of the default AppServer broker, then among other things, I can verify that the operating mode of the AppServer is set properly. In this case the default Operating Mode is still set, which is **state-reset**. This would mean that a connection to the AppServer will bind the selected AppServer agent to the client making the call, and reset the AppServer context on each connection. It’s possible to work this way using Silverlight, but it’s almost always preferable to configure your application so that you don’t bind an AppServer session. So I change the operating mode to state-free:


OpenEdge Editor - http://localhost:9090/fathom.htm - OpenEdge Architect - C:\WebWorkspace
AppServer Configuration: nbwfhjohn.asbroker1
Not Running (7 Minutes)
Save
Cancel
Broker
Agent
SSL
Messaging
Environment Variables
Stateless
State-aware
State-reset
State-free
Select the State-free item

Port number: 3090
Working directory: C:\OpenEdge102B\WRK
Operating Mode: State-reset

This means that I can only run non-persistent procedures from the client, via the proxies that I generate. But a non-persistent procedure can turn around and run anything it needs to in the OpenEdge session on the back end, including persistent procedures that it starts, or already running instances that it uses, all without binding the AppServer agent’s session to the client. Observing this basic principle will also simplifies defining the .NET proxies, as well as the Silverlight Domain Service classes that the client uses to access the proxies. So it’s an especially good principle to follow in this case, since in OpenEdge 10.2B it's necessary to code the Domain Service classes by hand.

I can check the Agent properties as well, and confirm the database connection as a part of the startup parameter list. This serves as a reminder that there are several ways to set properties for AppServers: here in OpenEdge Explorer, in the Servers View and the Server Monitor View in Architect, and in the launch configuration settings from the Server View's Overview page. To show another example of that, rather than starting the AppServer from here in Explorer, I can do it from the Server View in Architect, by pressing the start button in that view:



Once the AppServer is initialized, I can check its properties in the Server Monitor View. If I pull up the tabs to show more information, I can see the AppServer status, the number of agents, and more.

Now I'm ready to define the first procedure I want to run on the AppServer. I've got a starting point for that procedure defined, which is called **GetCustOrders.p**. Again, this is a non-persistent procedure with just a single entry point. At the top are definitions for a customer table and a related table with orders, and a ProDataSet that combines them. Below that is code that defines a simple query – no input parameters yet – and fills the DataSet with data from the corresponding tables in the database.

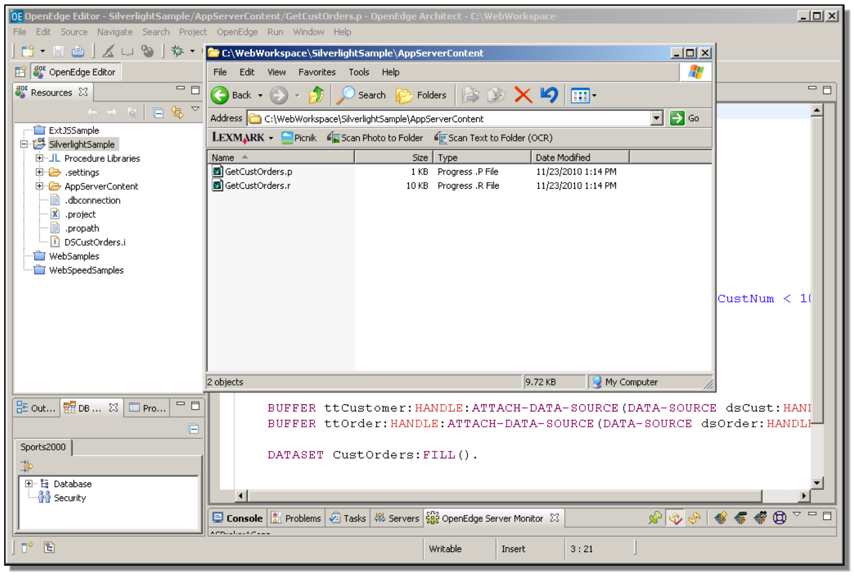
I first make a simple change to the procedure to test out the code publishing mechanism of the AppServer-enabled project. I remove the data definitions from the procedure itself, and paste them into a separate ABL include file, called **DSCustOrders.i**:

```
/*------------------------------------------------------------------------
    File        : DSCustOrders.i
    Purpose     :

    Syntax      :

    Description :

    Author(s)   : john
    Created     : Tue Nov 23 13:13:12 EST 2010
    Notes       :
  ----------------------------------------------------------------------*/

/* ***************************  Definitions  ************************** */


DEFINE TEMP-TABLE ttCustomer BEFORE-TABLE ttCustomerBefore
    FIELD CustNum AS INT
    FIELD NAME AS CHAR
    FIELD Address AS CHAR
    FIELD City AS CHAR
    FIELD State AS CHAR
    FIELD CreditLimit AS DECIMAL
    INDEX CustNum IS UNIQUE PRIMARY CustNum.


DEFINE TEMP-TABLE ttOrder BEFORE-TABLE ttOrderBefore
    FIELD OrderNum AS INT
    FIELD CustNum AS INT
    FIELD OrderDate AS DATE
    FIELD Carrier AS CHAR
    FIELD SalesRep AS CHAR
    INDEX OrderNum IS UNIQUE PRIMARY OrderNum.


DEFINE DATASET CustOrders FOR ttCustomer, ttOrder
        DATA-RELATION CustOrd FOR ttCustomer, ttOrder
          RELATION-FIELDS (CustNum, CustNum).
```

Back in the procedure, I type in the name of the new include file, and save the modified procedure in the AppServerContent directory:

```
/* Procedure GetCustOrders.p */

    {DSCustOrders.i}

    DEFINE OUTPUT PARAMETER DATASET FOR CustOrders.

    DEF VAR hCustOrders AS HANDLE.
    DEF VAR hqCust      AS HANDLE.

    DEFINE QUERY qCust FOR Customer.

    hqCust = QUERY qCust:HANDLE.
    hqCust:QUERY-PREPARE("FOR EACH Customer WHERE Customer.CustNum < 100").

    DEFINE DATA-SOURCE dsCust   FOR QUERY qCust.
    DEFINE DATA-SOURCE dsOrder  FOR Order.

    DATASET CustOrders:EMPTY-DATASET().

    BUFFER ttCustomer:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsCust:HANDLE).
    BUFFER ttOrder:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsOrder:HANDLE).

    DATASET CustOrders:FILL().
```

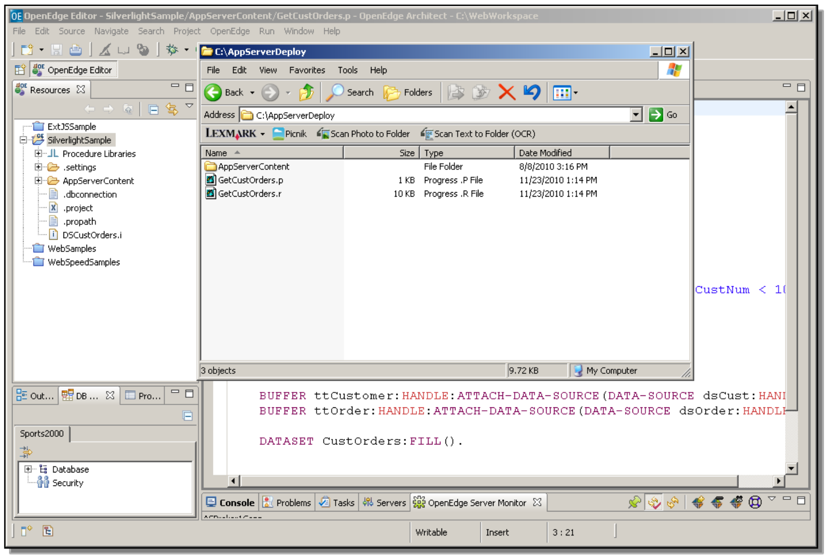
I can confirm that the .p and .r files are in AppServerContent:



Name	Size	Type	Date Modified
GetCustOrders.p	1 KB	Progress .P File	11/23/2010 1:14 PM
GetCustOrders.r	10 KB	Progress .R File	11/23/2010 1:14 PM

```
CustNum < 1(

BUFFER ttCustomer:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsCust:HANI
BUFFER ttOrder:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsOrder:HANDLE

DATASET CustOrders:FILL().
```

And I can quickly confirm that they were published to the AppServerDeploy folder as well, so when the client runs this procedure on asbroker1, the agent's propath will find the procedure here:

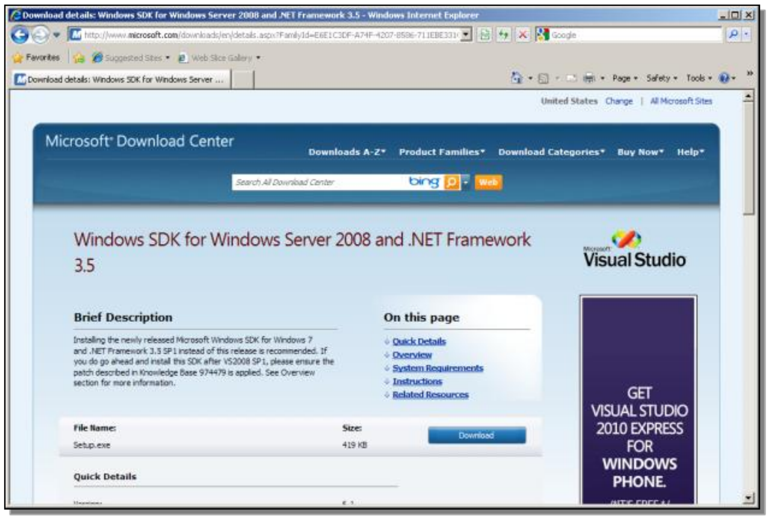

OpenEdge Editor - SilverlightSample/AppServerContent/GetCustOrders.p - OpenEdge Architect - C:\WebWorkspace
C:\AppServerDeploy
3 objects
9.72 KB
My Computer

Name	Size	Type	Date Modified
AppServerContent		File Folder	8/8/2010 3:16 PM
GetCustOrders.p	1 KB	Progress .P File	11/23/2010 1:14 PM
GetCustOrders.r	10 KB	Progress .R File	11/23/2010 1:14 PM

```
CustNum < 1(

BUFFER ttCustomer:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsCust:HANI
BUFFER ttOrder:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsOrder:HANDLE

DATASET CustOrders:FILL().
```

Now I have an AppServer configured and started where all my ABL procedures can be run. The next step in the process is to use the **OpenEdge Proxy Generator** to generate a .NET proxy for the procedure, in the form of a DLL. ProxyGen uses the c-sharp compiler installed under the Windows folder, as well as the xsd executable – for generating service definitions – from the .NET SDK, to generate the proxy.

If you don't already have the **.NET Framework SDK** installed, you'll have to locate the appropriate version at the Microsoft Download Center and install it.



Microsoft Download Center
Downloads A-Z
Product Families
Download Categories
Buy Now
Help
Visual Studio
GET VISUAL STUDIO 2010 EXPRESS FOR WINDOWS PHONE.

Windows SDK for Windows Server 2008 and .NET Framework 3.5

Brief Description

Installing the newly released Microsoft Windows SDK for Windows 7 and .NET Framework 3.5 SP1 instead of this release is recommended. If you do go ahead and install this SDK after VS2008 SP1, please ensure the patch described in Knowledge Base 974479 is applied. See Overview section for more information.

On this page

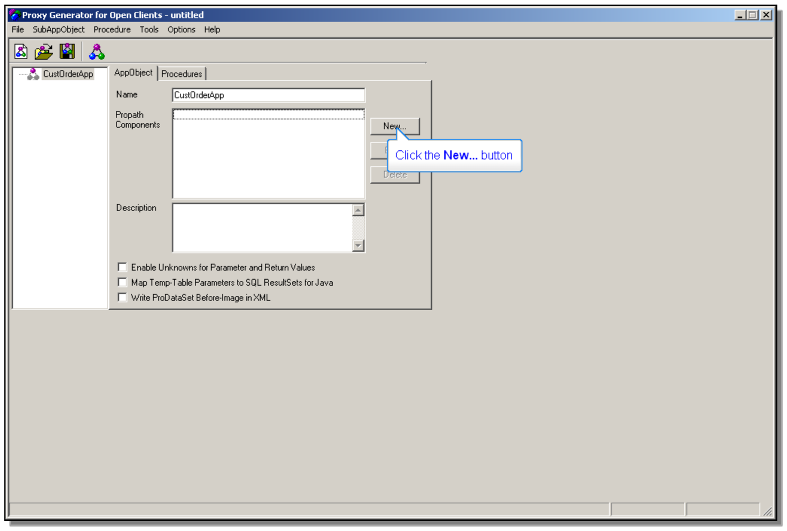
- Quick Details
- Overview
- System Requirements
- Instructions
- Related Resources

File Name:	Size:
Setup.exe	419 KB

Quick Details

OpenEdge 10.2B supports versions 2.0, 3.0, and 3.5, so you can install the latest of those versions, 3.5, for use with OpenEdge 10.2B. By default, ProxyGen expects version 2.0, so if you have that installed, there's no need to upgrade. In any case, SDK version 4.0 will not be supported until OpenEdge 11.

Once you've got the .NET SDK installed, you can generate your proxy. In ProxyGen, I create a new proxy for the first sample procedure, and name the **AppObject** for the proxy **CustOrderApp**. The AppObject can be used as a container for a number of related procedure calls. In this case, as I explained, I'm restricting the calls to be non-persistent, single-entry-point .p's, and I add the first of those here. First I add the SilverlightSample directory to the ProPath for the proxy:



Proxy Generator for Open Clients - untitled
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject
Procedures
New...
Delete
Click the New... button

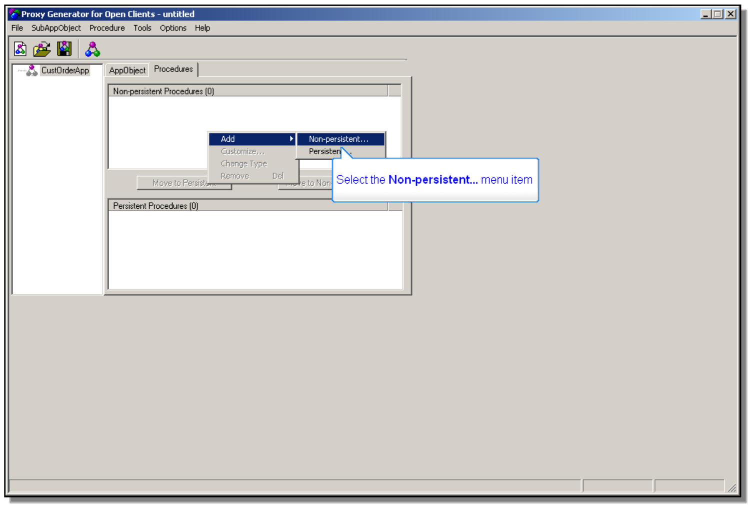
Name: CustOrderApp

Propath Components

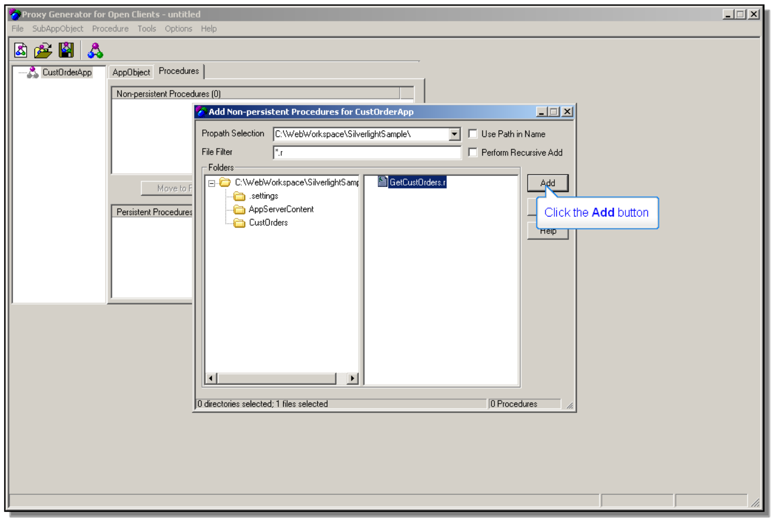
Description

- [ ] Enable Unknowns for Parameter and Return Values
- [ ] Map Temp-Table Parameters to SQL ResultSets for Java
- [ ] Write ProDataSet Before-Image in XML

That directory is under the folder where all the projects for the Architect sample workspace called WebWorkspace are stored. Next I select the **Procedures** tab to define the **ProcObject** for my first procedure, by right-clicking in that area and selecting **Add ->Non-persistent procedure**:

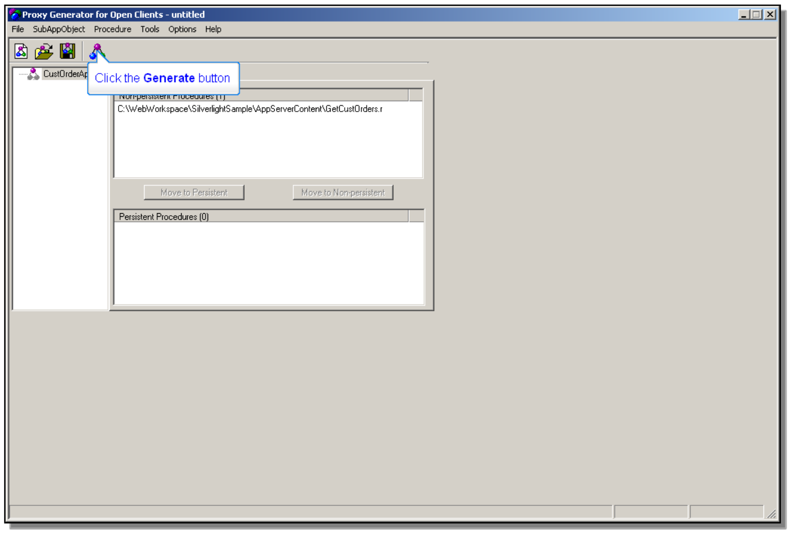

Proxy Generator for Open Clients - untitled
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject Procedures
Non-persistent Procedures (0)
Add
Customize...
Change Type
Remove Del
Non-persistent...
Persistent
Select the Non-persistent... menu item
Move to Persistent
Move to Non
Persistent Procedures (0)

In the AppServerContent directory, I select the r-code file **GetCustOrders.r**, and add that to the proxy:


Proxy Generator for Open Clients - untitled
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject Procedures
Non-persistent Procedures (0)
Add Non-persistent Procedures for CustOrderApp
Use Path in Name
Perform Recursive Add
Folders
C:\WebWorkspace\SilverlightSamp
.settings
AppServerContent
CustOrders
GetCustOrders.r
Add
Click the Add button
Help
Move to P
Persistent Procedures

Propath Selection: C:\WebWorkspace\SilverlightSample\
File Filter: *.r
0 directories selected; 1 files selected
0 Procedures

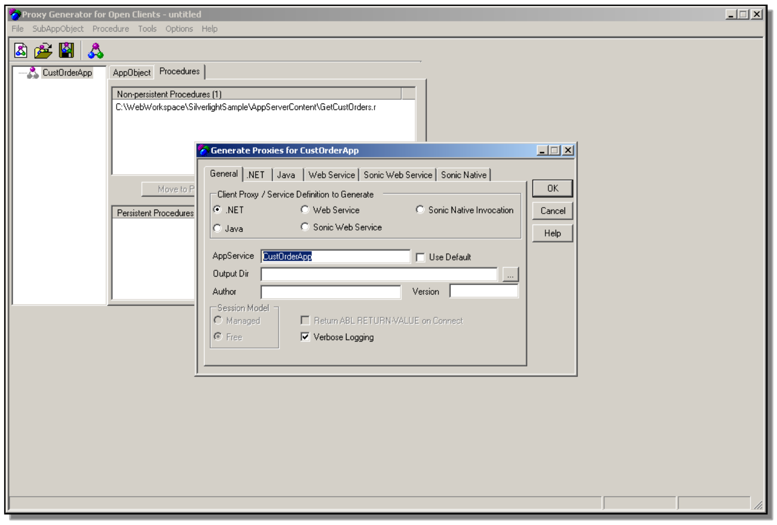
Then I click **Close** to finish, and click the **Generate** button here:


Proxy Generator for Open Clients - untitled
File SubAppObject Procedure Tools Options Help
CustOrderAp
Click the **Generate** button
Move to Persistent
Move to Non-persistent

Non-persistent Procedures (1)
C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.r

Persistent Procedures (0)

There are a number of choices for what type of proxy or service definition to generate. I leave that choice set to .NET and fill in some additional information on the **General** tab. The AppServer name isn't the same as the AppObject name, so I have to change that, by unchecking the default AppServer checkbox and changing it to **asbroker1**:


Proxy Generator for Open Clients - untitled
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject
Procedures
Move to P
Generate Proxies for CustOrderApp
General
.NET
Java
Web Service
Sonic Web Service
Sonic Native
OK
Cancel
Help

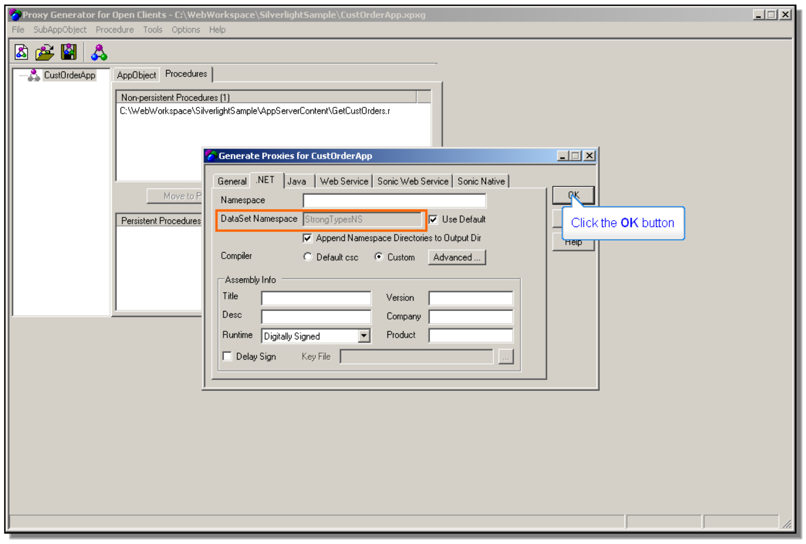
Non-persistent Procedures (1)
C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.r

Client Proxy / Service Definition to Generate: [✓ .NET] Java Web Service Sonic Web Service Sonic Native Invocation

AppService: CustOrderApp ☐ Use Default
Output Dir:
Author: Version:
Session Model: Managed [✓ Free]
☐ Return ABL RETURN-VALUE on Connect
☑ Verbose Logging

Next I have to identify the output directory where the proxy will be saved. I locate the SilverlightSample folder under my workspace, and make that the **Output Directory**. Then I select the .NET tab to set properties specific to the .NET proxy.

First I give the proxy a namespace, just a way to make sure references are unique relative to other sets of proxies. I name it **CustOrdersNS**, as a grouping for all calls related to this data object. This also gives me a default DataSet namespace of **CustOrdersNS.StrongTypesNS**, which I will reference later in the c-sharp code I write as part of the Silverlight support code.


Proxy Generator for Open Clients - C:\WebWorkspace\SilverlightSample\CustOrderApp.xpxg
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject Procedures
Move to P
Generate Proxies for CustOrderApp
General .NET Java Web Service Sonic Web Service Sonic Native
OK
Help
Advanced ...
Click the **OK** button

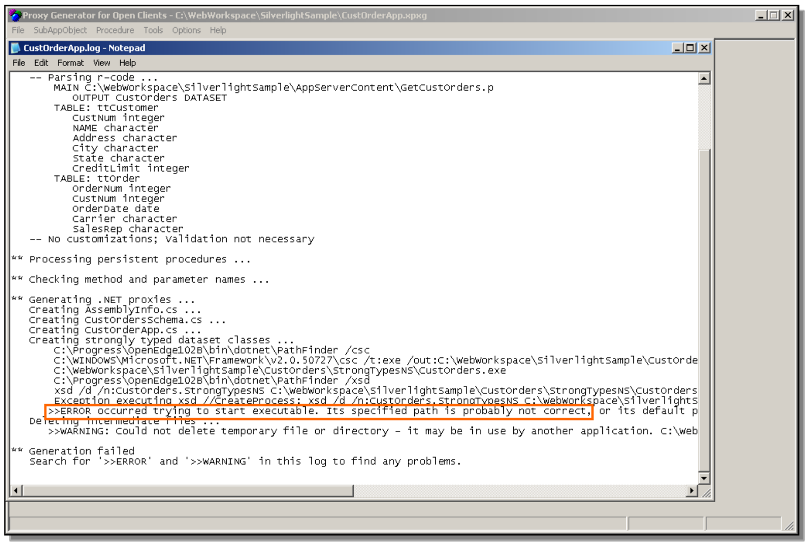
Non-persistent Procedures (1)
C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.r

Persistent Procedures

Namespace
DataSet Namespace StrongTypesNS ✓ Use Default
✓ Append Namespace Directories to Output Dir
Compiler Default csc [✓ Custom]

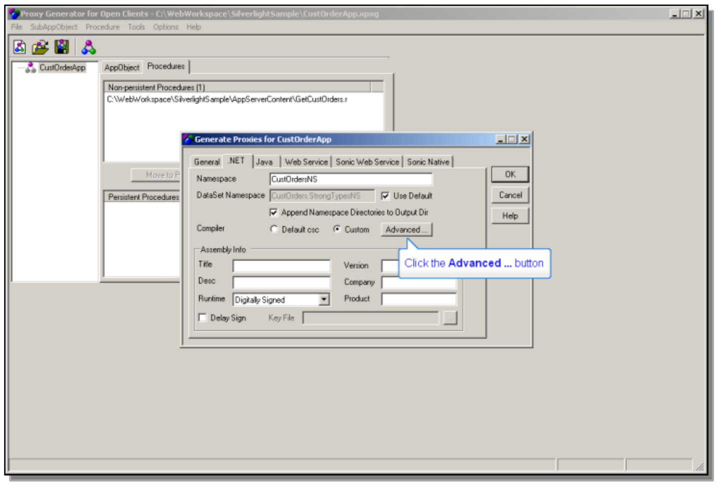
Assembly Info
Title
Version
Desc
Company
Runtime Digitally Signed
Product
☐ Delay Sign
Key File

When I click **OK**, ProxyGen asks me to confirm the name of the proxy and where I'm saving it. Here I need to explain an error that you may need to deal with when you first generate a .NET proxy, which relates to the choice of which release of the .NET Framework SDK you have installed on your machine. In OpenEdge 10.2B there is an issue with ProxyGen using a utility called Pathfinder that expects to locate files under the directory for SDK release 2.0. If you get a generation error, and look at the ProxyGen log file, you may see the reference to Version 2.0, and an error message that says that it couldn't locate the **xsd** executable to run, such as this:


Proxy Generator for Open Clients - C:\WebWorkspace\SilverlightSample\CustOrderApp.xpxg
File SubAppObject Procedure Tools Options Help
CustOrderApp.log - Notepad
File Edit Format View Help

```
   -- Parsing r-code ...
      MAIN C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.p
         OUTPUT CustOrders DATASET
      TABLE: ttCustomer
         CustNum integer
         NAME character
         Address character
         City character
         State character
         CreditLimit integer
      TABLE: ttOrder
         OrderNum integer
         CustNum integer
         OrderDate date
         Carrier character
         SalesRep character
   -- No customizations; Validation not necessary

** Processing persistent procedures ...

** Checking method and parameter names ...

** Generating .NET proxies ...
   Creating AssemblyInfo.cs ...
   Creating CustOrdersSchema.cs ...
   Creating CustOrderApp.cs ...
   Creating strongly typed dataset classes ...
      C:\Progress\OpenEdge102B\bin\dotnet\PathFinder /csc
      C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc /t:exe /out:C:\WebWorkspace\SilverlightSample\CustOrde
      C:\WebWorkspace\SilverlightSample\CustOrders\StrongTypesNS\CustOrders.exe
      C:\Progress\OpenEdge102B\bin\dotnet\PathFinder /xsd
      xsd /d /n:CustOrders.StrongTypesNS C:\WebWorkspace\SilverlightSample\CustOrders\StrongTypesNS\CustOrders
      Exception executing xsd //CreateProcess: xsd /d /n:CustOrders.StrongTypesNS C:\WebWorkspace\SilverlightS
     >>ERROR occurred trying to start executable. Its specified path is probably not correct, or its default p
   Deleting intermediate files ...
     >>WARNING: Could not delete temporary file or directory - it may be in use by another application. C:\Web

** Generation failed
   Search for '>>ERROR' and '>>WARNING' in this log to find any problems.
```

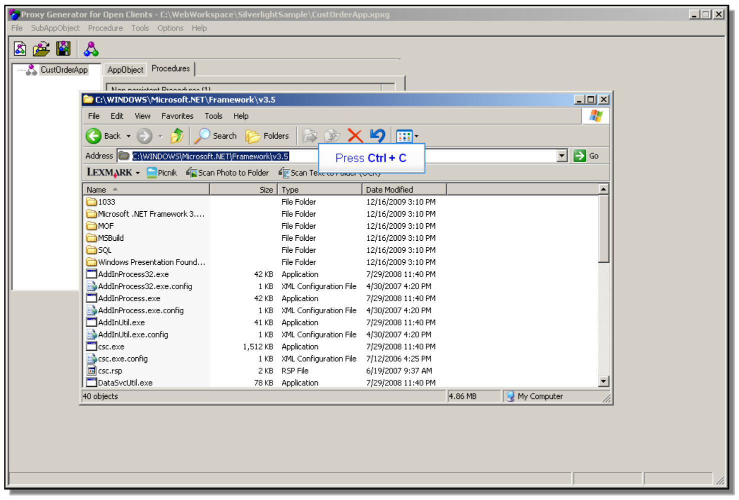
To avoid this problem, you can provide the pathnames to the compiler and xsd generator to ProxuGen. To illustrate this, I go back to pressing the Generate button to get the process started, and again select the .NET tab. But this time I don't use the default for the c-sharp compiler and xsd generator pathnames. I select **Custom** and then click **Advanced**:


Proxy Generator for Open Clients - C:\WebWorkspace\SilverlightSample\CustOrderApp.xpxg
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject
Procedures
Move to P
Generate Proxies for CustOrderApp
General
.NET
Java
Web Service
Sonic Web Service
Sonic Native
OK
Cancel
Help
Advanced ...
Click the **Advanced ...** button

Non-persistent Procedures (1)
C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.r
Persistent Procedures

Namespace: CustOrdersNS
DataSet Namespace: CustOrders.StrongTypesNS [✓ Use Default]
[✓ Append Namespace Directories to Output Dir]
Compiler: Default csc [✓ Custom]
Assembly Info
Title:
Version:
Desc:
Company:
Runtime: Digitally Signed
Product:
Delay Sign: ☐
Key File:

In front of the default executable names I need to place the correct pathnames where they can be found. Here for instance I've located my system's **csc.exe** file, in the 3.5 directory under WINDOWS:

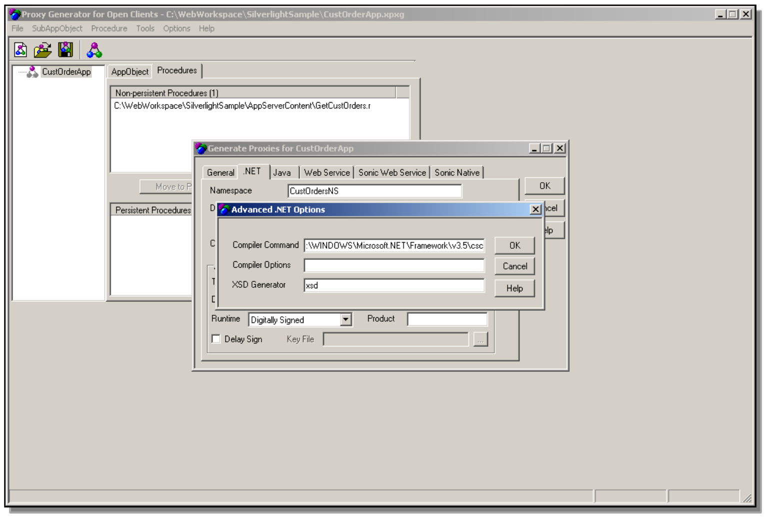

Proxy Generator for Open Clients - C:\WebWorkspace\SilverlightSample\CustOrderApp.xpxg
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject
Procedures
C:\WINDOWS\Microsoft.NET\Framework\v3.5
File Edit View Favorites Tools Help
Back
Search
Folders
Go
LEXMARK
Picnik
Scan Photo to Folder
Scan Te
Press **Ctrl + C**
My Computer

Address: C:\WINDOWS\Microsoft.NET\Framework\v3.5

Name	Size	Type	Date Modified
1033		File Folder	12/16/2009 3:10 PM
Microsoft .NET Framework 3....		File Folder	12/16/2009 3:10 PM
MOF		File Folder	12/16/2009 3:10 PM
MSBuild		File Folder	12/16/2009 3:10 PM
SQL		File Folder	12/16/2009 3:10 PM
Windows Presentation Found...		File Folder	12/16/2009 3:10 PM
AddInProcess32.exe	42 KB	Application	7/29/2008 11:40 PM
AddInProcess32.exe.config	1 KB	XML Configuration File	4/30/2007 4:20 PM
AddInProcess.exe	42 KB	Application	7/29/2008 11:40 PM
AddInProcess.exe.config	1 KB	XML Configuration File	4/30/2007 4:20 PM
AddInUtil.exe	41 KB	Application	7/29/2008 11:40 PM
AddInUtil.exe.config	1 KB	XML Configuration File	4/30/2007 4:20 PM
csc.exe	1,512 KB	Application	7/29/2008 11:40 PM
csc.exe.config	1 KB	XML Configuration File	7/12/2006 4:25 PM
csc.rsp	2 KB	RSP File	6/19/2007 9:37 AM
DataSvcUtil.exe	78 KB	Application	7/29/2008 11:40 PM

40 objects 4.86 MB

I copy that pathname, and paste it in front of **csc** in the ProxyGen **Advanced .NET Options** dialog:



Proxy Generator for Open Clients - C:\WebWorkspace\SilverlightSample\CustOrderApp.xpxg

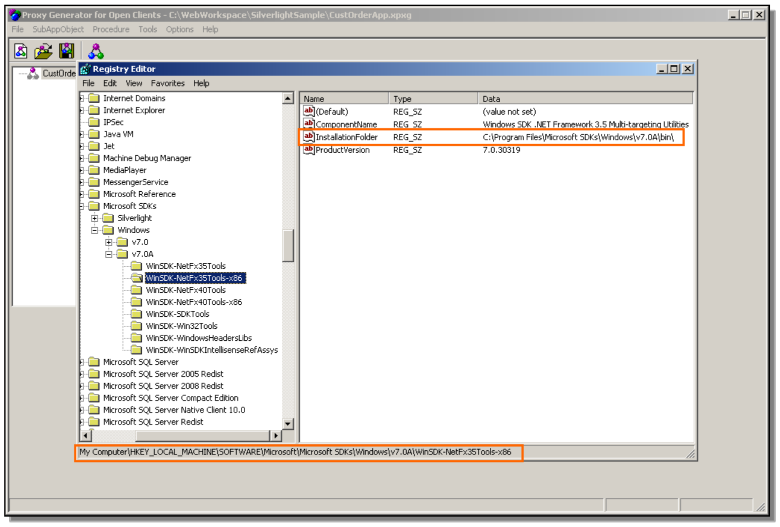
Non-persistent Procedures (1)
C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.r

Generate Proxies for CustOrderApp — .NET
Namespace: CustOrdersNS

Advanced .NET Options
Compiler Command: :\WINDOWS\Microsoft.NET\Framework\v3.5\csc
Compiler Options:
XSD Generator: xsd

Runtime: Digitally Signed
Product:
Delay Sign
Key File:

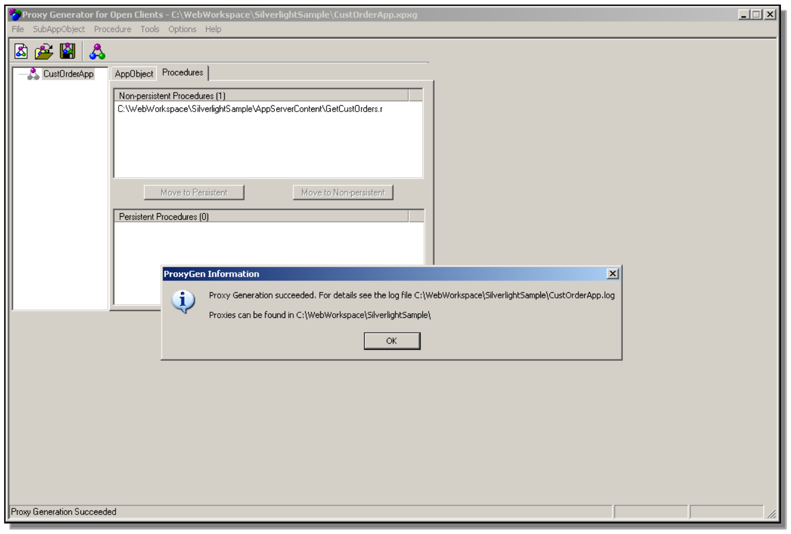
Next you need to identify where the xsd executable is located. If you don't what its location is you can search for its path in the registry, by running regedit, and following the path indicated at the bottom of this screenshot to find the value of the **Installation Folder**. This tells you what the pathname is:



Name	Type	Data
(Default)	REG_SZ	(value not set)
ComponentName	REG_SZ	Windows SDK .NET Framework 3.5 Multi-targeting Utilities
InstallationFolder	REG_SZ	C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\
ProductVersion	REG_SZ	7.0.30319

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SDKs\Windows\v7.0A\WinSDK-NetFx35Tools-x86

If you then enter that pathname back in the Advanced .NET Options dialog, everything should be defined correctly, and when you press OK the proxy will be generated successfully.


Proxy Generator for Open Clients - C:\WebWorkspace\SilverlightSample\CustOrderApp.xpxg
File SubAppObject Procedure Tools Options Help
CustOrderApp
AppObject
Procedures
Move to Persistent
Move to Non-persistent
ProxyGen Information
OK

Non-persistent Procedures (1)
C:\WebWorkspace\SilverlightSample\AppServerContent\GetCustOrders.r

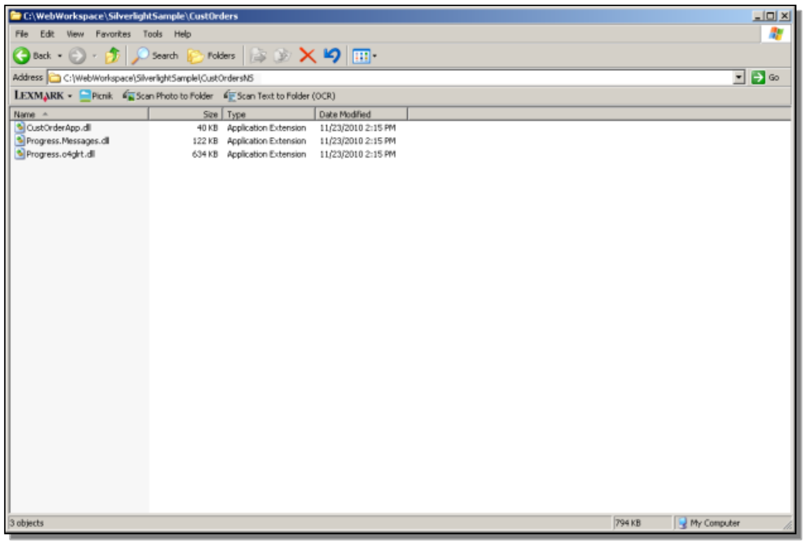
Persistent Procedures (0)

Proxy Generation succeeded. For details see the log file C:\WebWorkspace\SilverlightSample\CustOrderApp.log

Proxies can be found in C:\WebWorkspace\SilverlightSample\

Proxy Generation Succeeded

For my example, if I look in the SilverlightSample folder, I see a directory called **CustOrdersNS**, which is the name of my namespace, and in that is the **CustOrderApp.dll** file that is my .NET proxy, along with two other Progress dll's that I will need to reference from the c-sharp code that I'll be working on next:


C:\WebWorkspace\SilverlightSample\CustOrders
File Edit View Favorites Tools Help
Back
Search
Folders
Address C:\WebWorkspace\SilverlightSample\CustOrdersNS
Go
LEXMARK
Picnik
Scan Photo to Folder
Scan Text to Folder (OCR)

Name	Size	Type	Date Modified
CustOrderApp.dll	40 KB	Application Extension	11/23/2010 2:15 PM
Progress.Messages.dll	122 KB	Application Extension	11/23/2010 2:15 PM
Progress.o4glrt.dll	634 KB	Application Extension	11/23/2010 2:15 PM

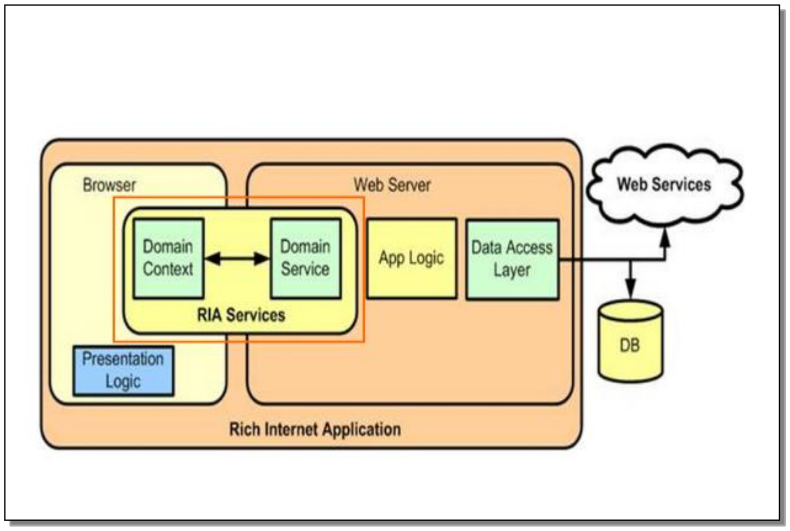
3 objects 794 KB My Computer

At this point, I've got the AppServer project defined in Architect, so that I can develop ABL procedures to run from Silverlight. I've configured and started that AppServer broker, and I've generated the .NET proxy for my first Silverlight call. The next step is to code a c-sharp class that will define my ProDataSet to the DomainService that will run in the Web server on behalf of the Silverlight client.

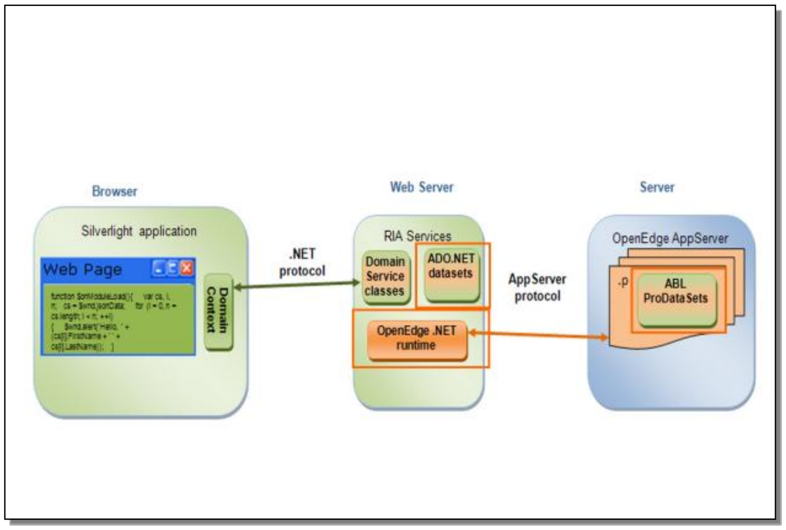
CREATING THE ENTITY CLASS

In this section of the paper and the video it accompanies, the third in the video series, I show how to create the C# **Entity Class** that represents the ABL ProDataSet data on the .NET side of the project. The diagram below shows an overview of the pieces that I need to assemble in order to load the Silverlight presentation with data from an ABL procedure on the OpenEdge Appserver. The key parts where RIA

Services is concerned are the **DomainService** that runs in the Web server, and its proxy on the client side, called the **DomainContext**.



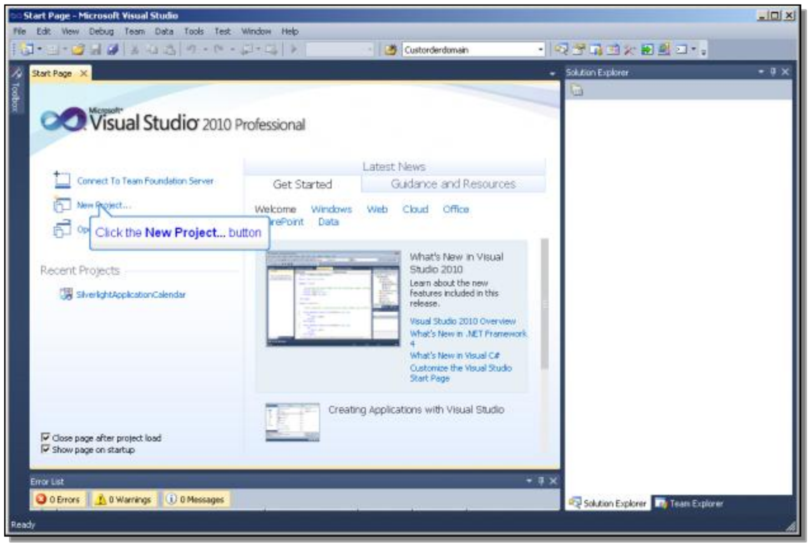
The OpenEdge .NET runtime in the next diagram represents the connection to the the .NET proxy I generated--which runs in the Web server--and which routes client requests back to ABL procedures running in the OpenEdge AppServer.



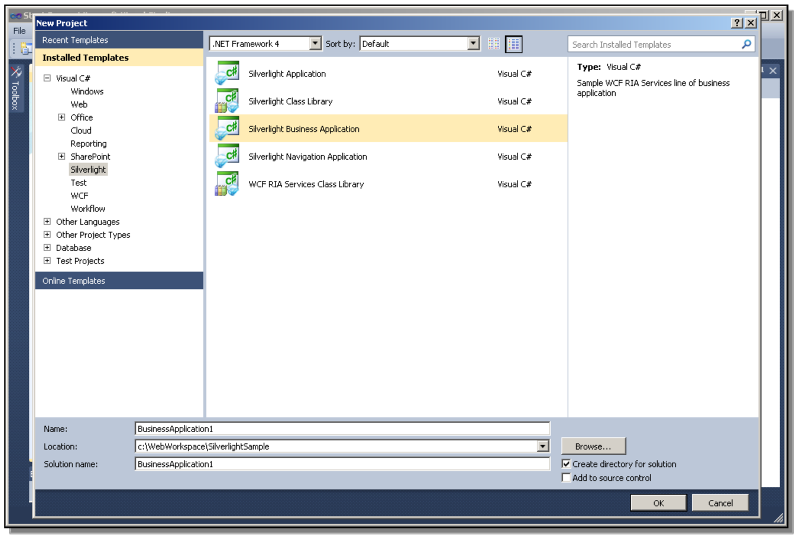
The .NET proxy accepts the parameters from a client-side request and passes them to the right ABL procedure. One of those parameters, in this first sample the only one, corresponds to a ProDataSet in my ABL procedure that holds data from two related temp-tables. The .NET proxy represents that ProDataSet internally as an ADO.NET DataSet.

I need a C-sharp class that represents that data to Silverlight as an object, defined in the entity class. Using OpenEdge 10.2B, you have to create the class file by hand, though once you see how a sample looks, it can act as a template for any number of different ProDataSets. This is part of what it is expected that ProxyGen will generate starting in OpenEdge 11. The entity class is an object-oriented representation of the data defined in the ProDataSet include file.

To get started, I create a new project in Visual Studio 2010 to hold all the files on the .NET side of the project.

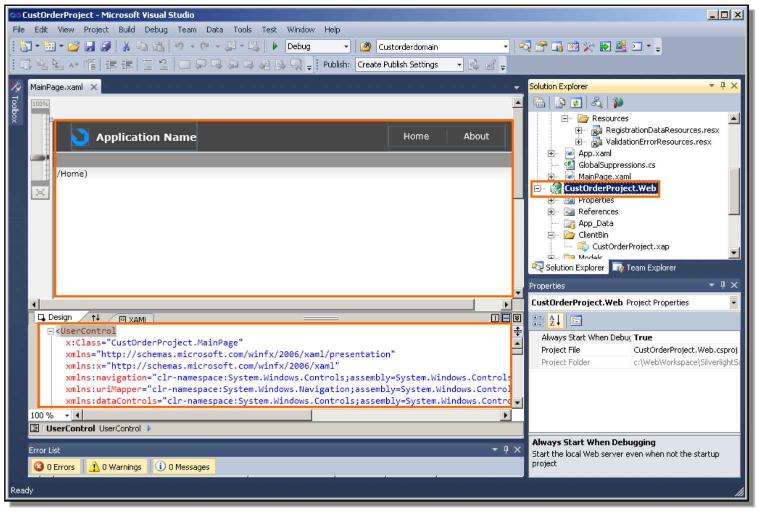

Start Page - Microsoft Visual Studio
Visual Studio 2010 Professional
Connect To Team Foundation Server
New Project...
Click the New Project... button
Recent Projects
SilverlightApplicationCalendar
Latest News
Get Started
Guidance and Resources
Welcome Windows Web Cloud Office SharePoint Data
What's New in Visual Studio 2010
Learn about the new features included in this release.
Visual Studio 2010 Overview
What's New in .NET Framework 4
What's New in Visual C#
Customize the Visual Studio Start Page
Creating Applications with Visual Studio
Close page after project load
Show page on startup
Solution Explorer
Error List
0 Errors 0 Warnings 0 Messages
Solution Explorer Team Explorer
Ready

Instead of selecting just the basic Silverlight Application, I'm going to select a different template called **Silverlight Business Application**. This does a lot more to get you started in building the client side and web server support for a typical business application:


New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Templates
Silverlight Application Visual C#
Silverlight Class Library Visual C#
Silverlight Business Application Visual C#
Silverlight Navigation Application Visual C#
WCF RIA Services Class Library Visual C#
Type: Visual C#
Sample WCF RIA Services line of business application
Browse...
OK
Cancel

Name: BusinessApplication1
Location: c:\WebWorkspace\SilverlightSample
Solution name: BusinessApplication1
[✓] Create directory for solution
[ ] Add to source control

I replace the default name with a name of my own, **CustOrderProject**, and click **OK** to get the beginnings of the application generated. In the screenshot below you can see some of what got generated as the starting point for a Silverlight Business Application. There's a **MainPage** that has already got Home and About buttons on it, and you can see a fragment of the XAML that defines the UI in the code pane. You can also see that Visual Studio automatically created a Web project to go along with the client project.

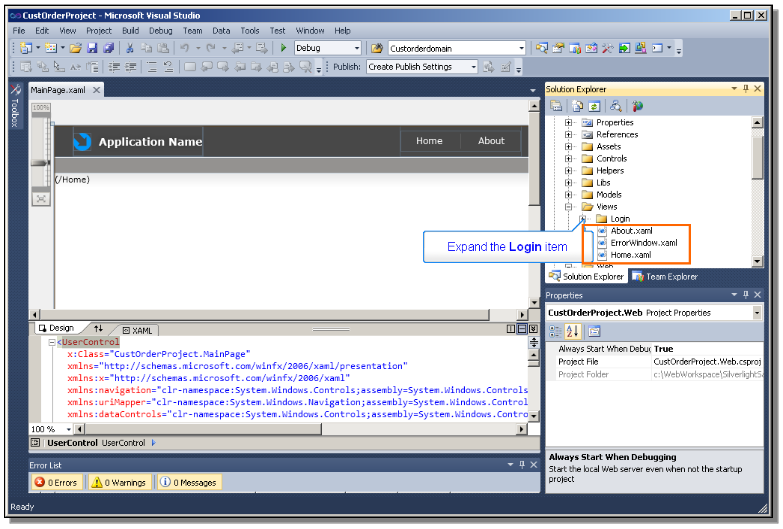


```
<UserControl
  x:Class="CustOrderProject.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls
  xmlns:uriMapper="clr-namespace:System.Windows.Navigation;assembly=System.Windows.Control
  xmlns:dataControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Contro
```

Always Start When Debug	True
Project File	CustOrderProject.Web.csproj
Project Folder	c:\WebWorkspace\SilverlightS

Always Start When Debugging
Start the local Web server even when not the startup project

Drilling down in the client project, I can look at what's been generated in the Views, and see these XAML files for the pages that the default buttons on the MainPage link to:

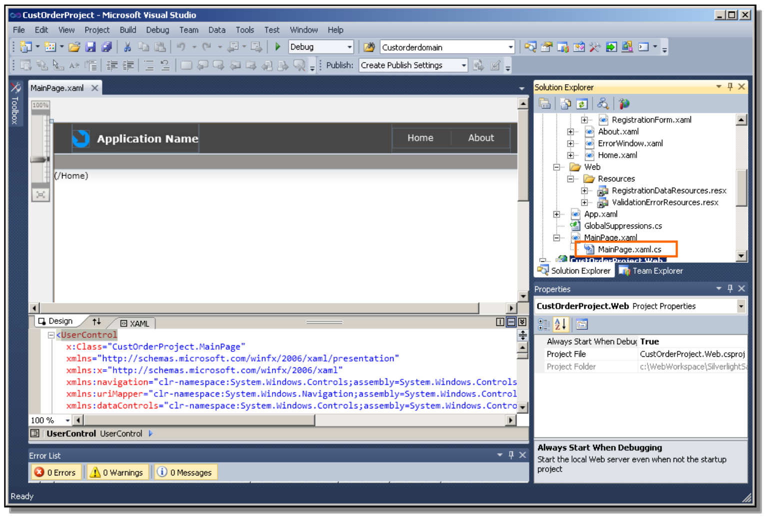


```
<UserControl
  x:Class="CustOrderProject.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls
  xmlns:uriMapper="clr-namespace:System.Windows.Navigation;assembly=System.Windows.Control
  xmlns:dataControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Contro
```

Always Start When Debug	True
Project File	CustOrderProject.Web.csproj
Project Folder	c:\WebWorkspace\SilverlightS

Always Start When Debugging
Start the local Web server even when not the startup project

Under the Login group, there are additional starting XAML files for login and registration pages. So you see that the business template gives you a quick starting point that you can expand to get the beginnings of a real working application. Expanding the project entry for the XAML file for the MainPage, you can see the c-sharp *code-behind* file for the MainPage, containing the code that brings up the sample pages the buttons link to.

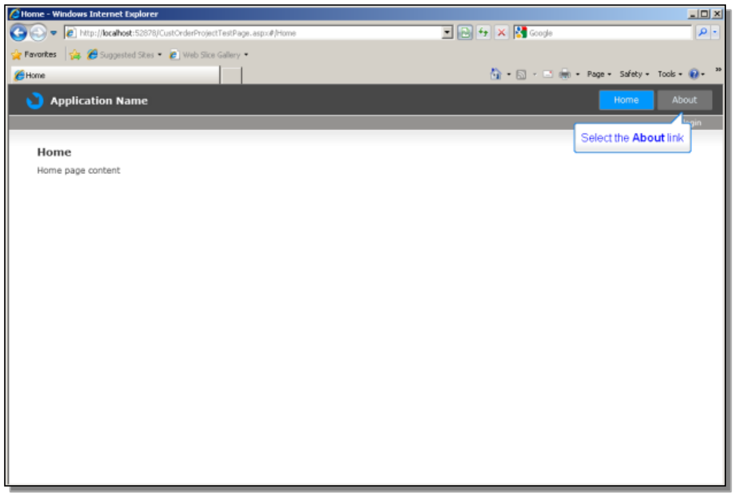


```
<UserControl
  x:Class="CustOrderProject.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls
  xmlns:uriMapper="clr-namespace:System.Windows.Navigation;assembly=System.Windows.Control
  xmlns:dataControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Contro
```

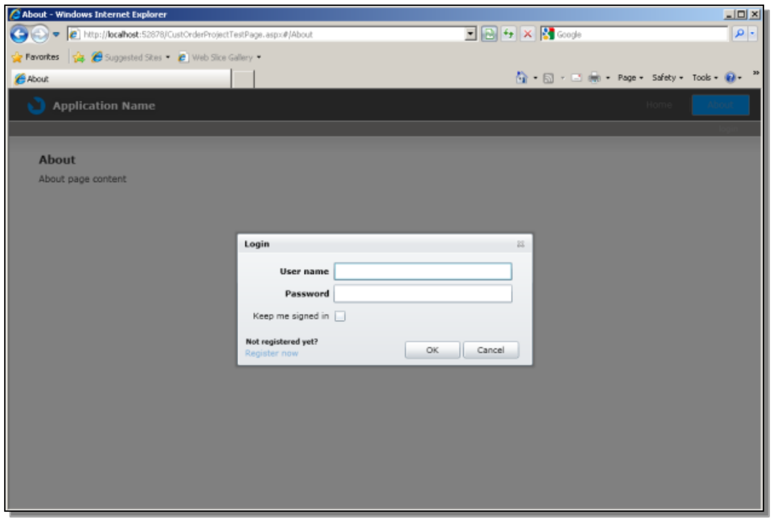
Always Start When Debu	True
Project File	CustOrderProject.Web.csproj
Project Folder	c:\WebWorkspace\SilverlightS

Always Start When Debugging
Start the local Web server even when not the startup project

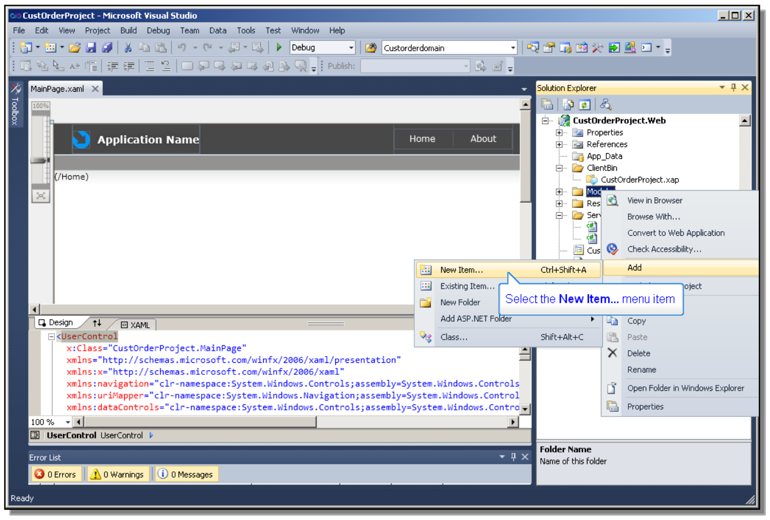
If I just run what I've been given to start with, you can see the Home and About pages that are defined in the Silverlight Business Application template:



It isn’t much, but it’s instructive to look at and to then expand on in your own application. Clicking **Login**, I get the skeleton of a login dialog:

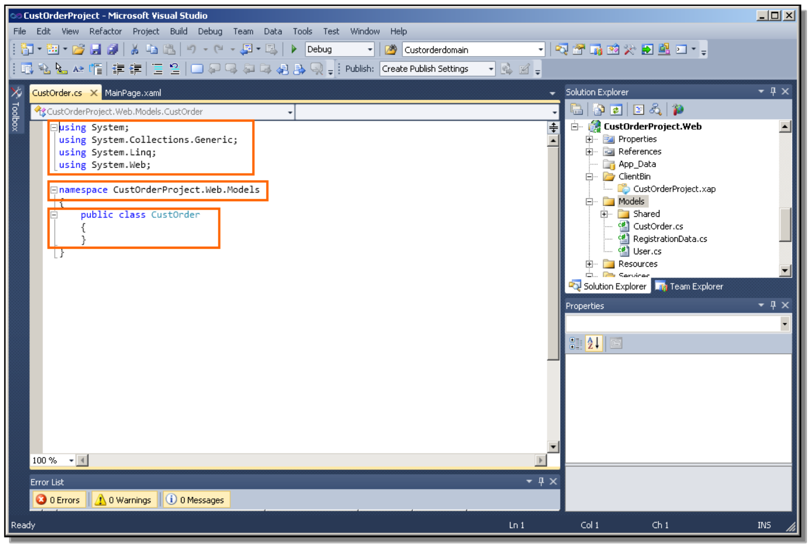


I start coding what I need to add to that to get a working application to retrieve and display customers and orders by right-click the **Models** group in the Web project, and selecting **Add**, and **New Item**:



```
<UserControl
    x:Class="CustOrderProject.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls
    xmlns:uriMapper="clr-namespace:System.Windows.Navigation;assembly=System.Windows.Control
    xmlns:dataControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Contro
```

I select the template just named **Class** from all the alternatives I can add to the project. This is going to become my entity class, containing the object-oriented definition of my ProDataSet. I call the class **CustOrder.cs**. I'm given several general **`using`** statements to start with, a **`namespace`** that places my new class in the **`CustOrderProject.Web`** project, and the beginnings of a class definition:



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

namespace CustOrderProject.Web.Models
{
    public class CustOrder
    {
    }
}
```

I need a couple of additional **`using`** statements: as shown in the code fragment below:

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

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

The first is **`System.ComponentModel.DataAnnotations`**. The data annotations in C# are enclosed in square brackets, and they provide special information to the compiler, such as which fields in the data represent unique key values. The second one, **`System.ServiceModel.DomainServices.Server`**, provides access to additional annotations such as **`[Include]`**, which links the Customer data to the Order data for each Customer, and **`[Association]`**, which defines the relationship between the two tables.

Within the class definition itself, I need to name every field in the **`ttCustomer`** temp-table that I want exposed to the user interface. The first, the **`CustNum`**, is the key field, so I identify it using that **DataAnnotation**. And then I define the field itself, as a property.

```
namespace CustOrderProject.Web.Models
{
    public class CustOrder
    {
        [Key]
        public int CustNum { get; set; }
```

Data binding in .NET is basically binding against public properties of an object, so each field is defined as a property, with an optional getter and setter. If there's no code to execute when the field is read or updated, then you can just use this simple form, much as you can in object-oriented ABL. Otherwise you could define a block of code to execute when the field is read or updated, along with a backing variable to hold the value that's returned or set.

I need to do the same for each of the other fields in the ttCustomer temp-table:

```
        public string Name { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public decimal CreditLimit { get; set; }
```

The basic data type mappings are straightforward. You can learn all the details of data types and everything else you need to know about mapping ABL to .NET in the OpenEdge product documentation.

Next I need to define the **Order** object and **ttOrder** fields as elements that are contained in and associated with each Customer. This is where defining your data definitions on the OpenEdge side is important, as you design Business Entities and other parts of the application on the server side. In a real world application, it might make more sense to consider a **Customer** and an **Order** to be separate entities, each perhaps with its own complex data definition, Customers with multiple addresses, say, and each Order with its OrderLines. But in this example I'm presuming that you retrieve customers and their orders together, which is why there's one ProDataSet definition on the backend and one entity class definition here for CustOrder as a single entity. I start defining the connection between them using the **Include** annotation. Then I use another annotation, **Association**, to define the relationship between the two tables. I name the association **Cust_Order**, and name the key fields from each table that are joined to connect them, just as I did in the Data-Relation in the ProDataSet definition:

```
        [Include]
        [Association("Cust_Order", "CustNum", "CustNum")]
```

Now I define what it is I'm associating with Customers, an enumeration of Order objects that I call **Orders**. This is also a property of CustOrder with its own getter and setter.

```
        public IEnumerable<Order> Orders { get; set; }
```

Finally, I need to define a constructor for the CustOrder class that's executed each time an instance of the class is created. It populates the Orders by creating, in turn, a **List** object, which is a new instance of the Order objects that have a CustNum that matches the CustNum of the customer. And that's the end of the CustOrder entity class:

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

Now I need a class to define the Order object. Just as for the first class, it requires a property definition for each field to be exposed, with **Key** annotations for key fields.

```
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; }
    }
```

Now I can save and compile the entire entity class file, and the first part of my C-Sharp coding is done. The entire entity class source file code for **CustOrder.cs** is repeated 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; }
        public string Name { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        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; }
    }
}
```

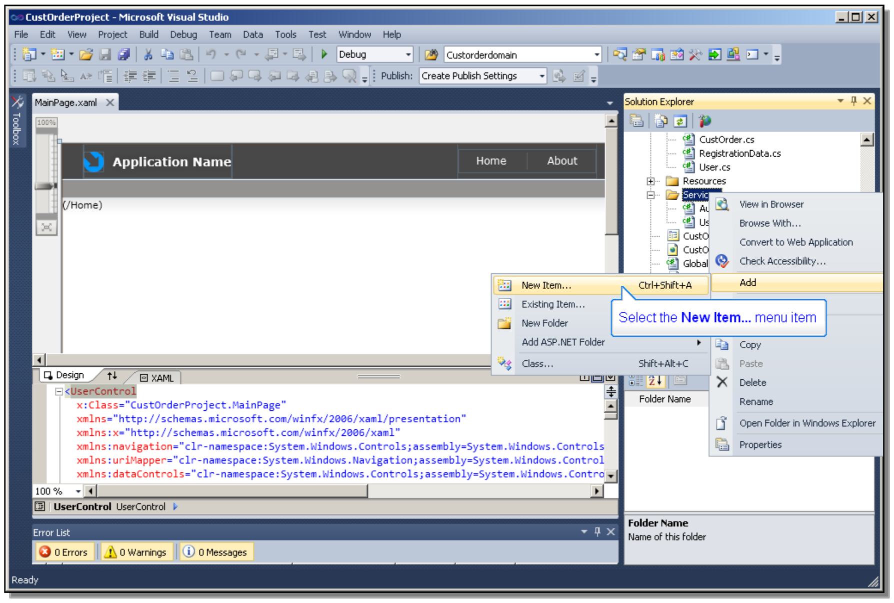
This may have looked like a lot, going through it for the first time, but if you use an example like this one as a starting point, you can code or even generate yourself the entity classes for a number of different data entities. In the next section I tackle the DomainService class, which provides Silverlight with access to the data I've defined here and in my .NET proxy.

BUILDING A DOMAINSERVICE CLASS

This section of the document corresponds to parts four and five of the RIA Services video series.

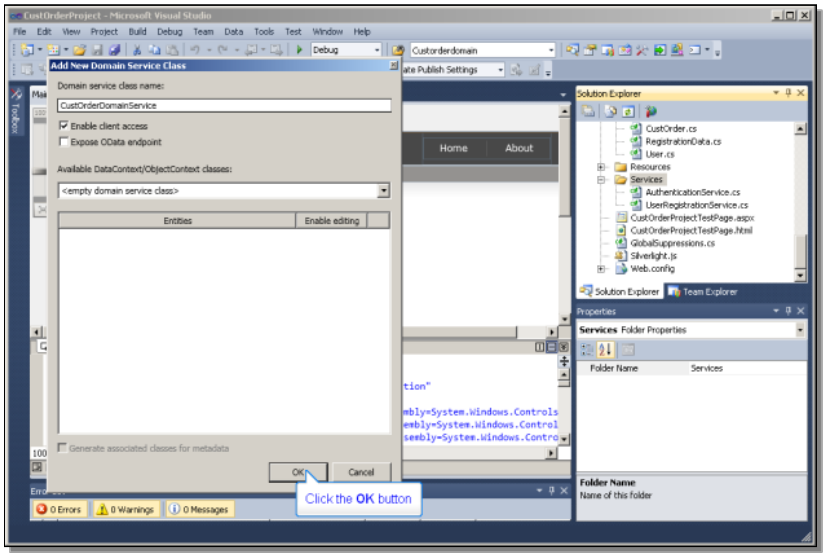
Now that I've created the entity class that defines my data as an object, it's time to create the DomainService class. Remember that the ProDataSet from the ABL procedure is transformed into an ADO.NET DataSet by the .NET proxy generated earlier. That data needs to be moved in turn to an **entity**, an instance of the class created in the previous section, for use by Silverlight. The DomainService class runs in the Web server. It has access to the OpenEdge .NET proxy – I show you how that connection is made in a moment – and also the EntityClass, so that's how it runs in tandem with the OpenEdge components on the backend. On the other side, when I'm done coding the DomainService and I build the solution, Visual Studio will generate the client part of accessing the data domain, called the **DomainContext** class. And that in turn will be the connection to the Silverlight user interface.

Looking at the **Services** folder in my Web project, you see the **Authentication** and **User Registration** services for which the Silverlight Business Application template generates a default user interface and skeleton support code. The DomainService class must be added to this group. I select **Add**, And **New Item…**



```
<UserControl
  x:Class="CustOrderProject.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls
  xmlns:uriMapper="clr-namespace:System.Windows.Navigation;assembly=System.Windows.Control
  xmlns:dataControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Contro
```

from all the different class templates I can choose from, I select **Domain Service Class**. I don't want the default name of **`DomainService1`**, partly because it's important to end the name with the word **`Service`**: when I do the build, the code generator will locate every reference ending in **`Service`**, such as **`CustOrderDomainService`**, and generate an appropriate reference ending in **`Context`**, such as **`CustOrderDomainContext`**. So I name it **`CustOrderDomainService`**, and add it to my project. There's a checkbox in the dialog that appears labeled "**Enable client access**", and I need to make sure to leave that checked on, because that's what causes the DomainContext class to be generated, since that is what enables client access to my data domain. I don't need to add or change anything else here, so I just click **OK**.


Add New Domain Service Class

Domain service class name: CustOrderDomainService

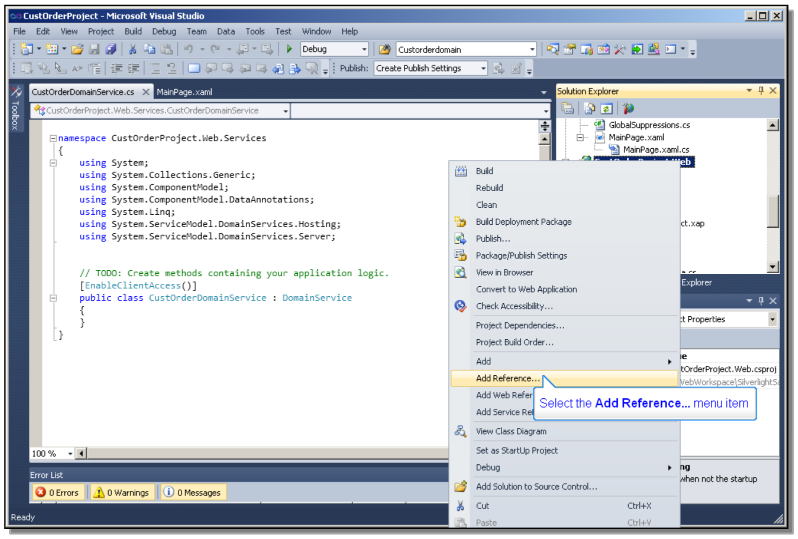
[✓] Enable client access
[ ] Expose OData endpoint

Available DataContext/ObjectContext classes: <empty domain service class>

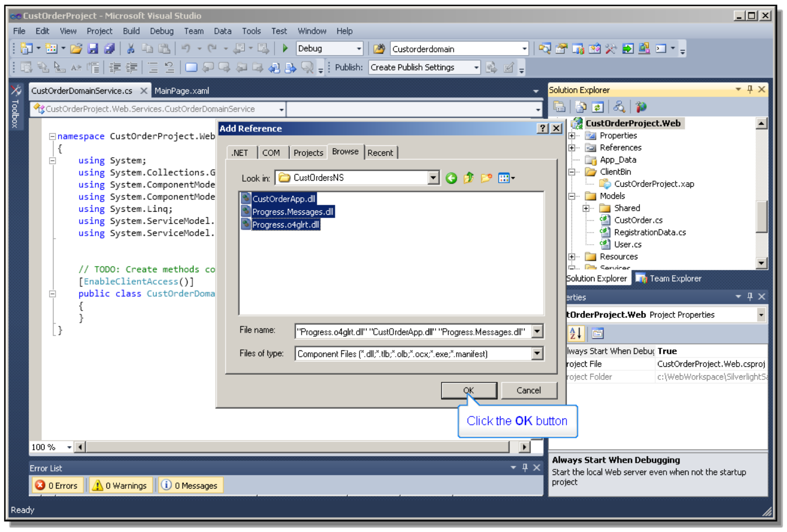
Entities	Enable editing

[ ] Generate associated classes for metadata

Back in the Solution Explorer for the Web project, I need to establish the connection between the DomainService class and the OpenEdge proxy. I right-click, and select **Add Reference**. I have to add References to the .NET proxy as well as a couple of common Progress DLL's that the Silverlight project needs.


```
namespace CustOrderProject.Web.Services
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.ComponentModel.DataAnnotations;
    using System.Linq;
    using System.ServiceModel.DomainServices.Hosting;
    using System.ServiceModel.DomainServices.Server;


    // TODO: Create methods containing your application logic.
    [EnableClientAccess()]
    public class CustOrderDomainService : DomainService
    {
    }
}
```

I find the proxy and other DLLs in the **`CustOrdersNS`** folder of my SilverlightSample project. Remember that CustOrdersNS is the namespace that I designated in ProxyGen when I created the proxy, and this folder got created to hold the proxy DLL and other related DLLs because of that. In this folder, you can see the DLL which is the .NET proxy for my CustOrderApp Object, as well as two standard DLLs I need to reference, one for **`Progress.Messages`**, and the other for references in the code that will start **`Progress.Open4GL`**.


CustOrderProject - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Custorderdomain
Publish: Create Publish Settings
CustOrderDomainService.cs
MainPage.xaml
CustOrderProject.Web.Services.CustOrderDomainService
Add Reference
.NET COM Projects Browse Recent
Look in: CustOrdersNS
OK
Cancel
Click the OK button
Solution Explorer
Team Explorer
100 %
Error List
0 Errors
0 Warnings
0 Messages
Ready

```
namespace CustOrderProject.Web
{
    using System;
    using System.Collections.G
    using System.ComponentMode
    using System.ComponentMode
    using System.Linq;
    using System.ServiceModel.
    using System.ServiceModel.


    // TODO: Create methods co
    [EnableClientAccess()]
    public class CustOrderDoma
    {
    }
}
```

CustOrderApp.dll
Progress.Messages.dll
Progress.o4glrt.dll

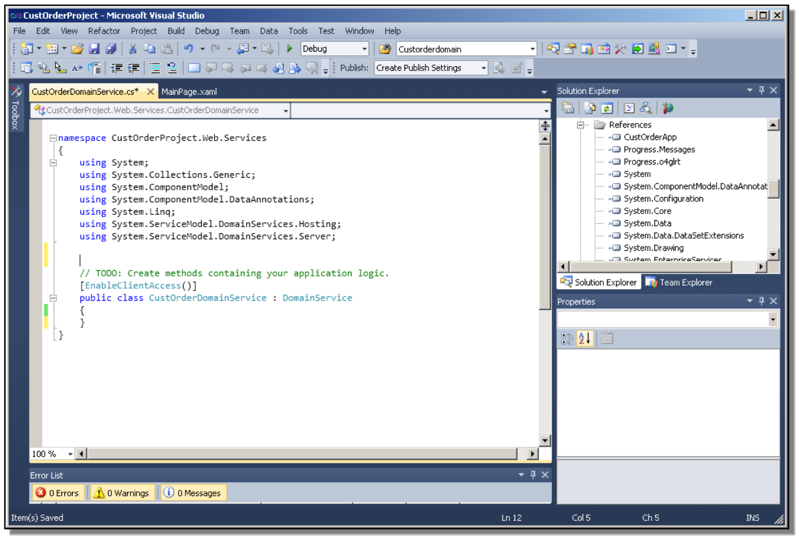
File name: "Progress.o4glrt.dll" "CustOrderApp.dll" "Progress.Messages.dll"
Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

CustOrderProject.Web
- Properties
- References
- App_Data
- ClientBin
  - CustOrderProject.xap
- Models
  - Shared
  - CustOrder.cs
  - RegistrationData.cs
  - User.cs
- Resources
- Services

lways Start When Debug	True
roject File	CustOrderProject.Web.csproj
roject Folder	c:\WebWorkspace\SilverlightS

Always Start When Debugging
Start the local Web server even when not the startup project

Back in Visual Studio, the Solution Explorer has opened up the References folder in the Web project, and the three new references are there -- the one for the AppObject proxy itself, and the two standard DLLs. The code shown is the starting point that's provided for me for the DomainService class. Some useful **`using`** statements have been generated, along with an annotation that confirms that I selected **Enable client access**, so the DomainContext class will be generated:


CustOrderProject - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Team Data Tools Test Window Help
Debug
Custorderdomain
Publish: Create Publish Settings
CustOrderDomainService.cs*
MainPage.xaml
CustOrderProject.Web.Services.CustOrderDomainService
Solution Explorer
Team Explorer
Properties
100 %
Error List
0 Errors
0 Warnings
0 Messages
Item(s) Saved
Ln 12
Col 5
Ch 5
INS

```
namespace CustOrderProject.Web.Services
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.ComponentModel.DataAnnotations;
    using System.Linq;
    using System.ServiceModel.DomainServices.Hosting;
    using System.ServiceModel.DomainServices.Server;

    
    // TODO: Create methods containing your application logic.
    [EnableClientAccess()]
    public class CustOrderDomainService : DomainService
    {
    }
}
```

References
- CustOrderApp
- Progress.Messages
- Progress.o4glrt
- System
- System.ComponentModel.DataAnnotat
- System.Configuration
- System.Core
- System.Data
- System.Data.DataSetExtensions
- System.Drawing
- System.EnterpriseServices

There are a few additional **`using`** statements I need. Remember that the **`using`** statements effectively extend the search path for references in the code, just as you can do in object-oriented ABL.

First I must explain something about how the AppObject instance – a running instance of my proxy – is going to be created. I could simply create an instance of it each time this DomainService is run, but that involves some overhead, of course. Even though the AppObject on the OpenEdge side is non-persistent, so

GetCustOrders.p is run each time the client wants to retrieve Customers and Orders, I can create a single instance of the AppObject in the Web server and leave it running as long as the application domain remains active. This doesn't bind anything on the OpenEdge side, but it makes the DomainService more efficient. I do that by putting the AppObject instance into a cache, and I need the **using** statements **System.Web**, and **System.Web.Caching** for my cache references:

```
using System.Web;
using System.Web.Caching;
```

Next I need **using** statements for the Progress DLLs I added as References:

```
using Progress.Open4GL;
using Progress.Open4GL.Proxy;
```

The next needed statement is for **System.Data**, for references to the **DataTables** of my entity class. I need a **using** statement for the **Models** folder where my entity class file is located. And finally, a **using** statement for my **CustOrdersNS** namespace, and remember that within that namespace, ProxyGen created for me a namespace called **StrongTypesNS** for references to my ProDataSet:

```
using System.Data;
using Models;
using CustOrdersNS;
using CustOrdersNS.StrongTypesNS;
```

These **using** statements will qualify the references to everything I need in the code that follows.

The **class** statement for the DomainService is generated for me. The first executable statement I need to add defines a reference to my proxy AppObject, and it's a reference to the type **CustOrderApp**:

```
    public class CustOrderDomainService : DomainService
    {
        private CustOrdersNS.CustOrderApp _appObj = null;
```

Next I need an instance of the ADO.NET DataSet that will come back from the proxy DLL on the OpenEdge side:

```
        CustOrdersDataSet CustOrderDS = null;
```

And now I start the **CustorderDomainService** constructor that gets run when the class is instantiated. Inside a standard **try** block, I create a reference to the cache I referred to, which is scoped to the **HttpRuntime**:

```
        public CustOrderDomainService()
            : base()
        {
            try
            {
                Cache COCache = HttpRuntime.Cache;
```

The next statement tries to retrieve an instance of my AppObject from the cache using the **Get** method, and a key value that I've assigned. I'm just calling the object I'm storing in the cache **COAppObj**, which is an arbitrary key value:

```
                _appObj = (CustOrderApp)COCache.Get("COAppObj");
```

If it isn't there, then this is the first time the DomainService has been called, so I assign a string to the URL of my OpenEdge AppServer, and I define the **SessionModel** of my OpenEdge runtime properties to state-free, which is represented by the value 1:

```
                if (_appObj == null)
                {
                    string urlString = "AppServer://localhost/asbroker1";
                    RunTimeProperties.SessionModel = 1; // state-free
```

I define a **Connection** object using that string. Then I assign my AppObject reference to a new instance of my CustOrderApp proxy, passing the connection string to get me going. And then I insert that AppObject reference into the cache, using the **COAppObj** key value that I later use to retrieve it with.

```
                    Connection connObj = new Connection(urlString, "", "", "");
                    _appObj = new CustOrderApp(connObj);
                    COCache.Insert("COAppObj", (object)_appObj);
                }
```

So each time after the first, when I run this DomainService, I will find the already running instance of my AppObject in the cache, and be able to reuse it. And again, this happens without binding anything on the OpenEdge AppServer. Once I've either created or located the running AppObject instance, I create a new instance of my DataSet object:

```
                CustOrderDS = new CustOrdersDataSet();
            }
```

Finally, I just put in a default catch block for any errors:

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

And that's the end of the constructor. At this point I've got a reference to my OpenEdge proxy, and I'm ready to call the one entry point it has so far, which corresponds to the GetCustOrders.p procedure on the AppServer.

Ahead of the first method definition following the class constructor I insert a data annotation that tells Visual Studio what type of access my first method is performing. RIA Services supports specific access types for queries, update, insert, and delete operations, and a general purpose type of operation called Invoke. In this first example I'm just creating a query. The method that query designation applies to has a return type of **`IEnumerable`** for my CustOrder entity. **`IEnumerable`** here is the interface that defines a set of data rows that my user interface will be able to interate through.

```
        [Query]
        public IEnumerable<CustOrder> GetCustOrders()
        {
```

`CustOrder`, remember, is the name of the entity class that I created earlier, the object representation of the top-level table in my ABL ProDataSet. I call this method **`GetCustOrders`**, to match the name of my ABL procedure.

Remember that after I create the DomainService class, the Silverlight Business Application support in Visual Studio will generate the client-side class that connects the UI to the DomainService in the Web server. That new generated class is what I've been calling the DomainContext. The **`[Query]`** data annotation that I entered above the method definition means that I will get a method in that DomainContext called **`GetCustOrdersQuery`**, which is what the user interface support code will invoke to get data into the UI. So there are a lot of pieces that get connected to get the data from one end to the other, but I'm showing you all the parts that you're responsible for – the rest gets generated for you.

Within the usual **`try`** block, the new method in turn invokes the **`GetCustOrders`** entry in the AppObject – that's using the .NET proxy to access my ABL procedure on the OpenEdge AppServer:

```
            try
            {
                _appObj.GetCustOrders(out CustOrderDS);
```

The proxy converts the ProDataSet returned return by the ABL procedure to an ADO.NET DataSet, which is represented here in **`CustOrderDS`**, an instance of the **`CustOrdersDataSet`**. Within that DataSet there are two .NET DataTables called **`customers`** and **`orders`**, derived from the ProDataSet's two temp-tables:

```
                DataTable customers = (DataTable)CustOrderDS.Tables["ttCustomer"];
                DataTable orders = (DataTable)CustOrderDS.Tables["ttOrder"];
```

Next I define a query to iterate over the customers. Its type again is **`IEnumerable`**, the type of my **`CustOrder`** entity. I name the query object **`custquery`**. Here's where we get into the rather odd syntax that you have to use to define sets of data in this technology. This is called **Language Integrated Query** (**LINQ**, pronounced "link"), a Microsoft .NET Framework component that adds native data querying capabilities to .NET languages. You can study the Microsoft documentation for details, but an example here will get you started in the right direction. **Custrow** is the name of a **row** object. Once again I designate the **customers** table as being an **Enumerable** using the method **AsEnumerable**:

```
                 IEnumerable<CustOrder> custquery =
                     from custrow in customers.AsEnumerable()
                     select new CustOrder
                     {
```

Within that block of code I name each element and retrieve the field value for the corresponding DataTable field. Note the special statement at the end that populates an **Orders** object with the result returned from another method I'll code next, called **getOrders**, using the current **CustNum** as the key value to filter **Orders** with.

```
                     {
                         CustNum = custrow.Field<int>("CustNum"),
                         Name = custrow.Field<string>("Name"),
                         Address = custrow.Field<string>("Address"),
                         City = custrow.Field<string>("City"),
                         State = custrow.Field<string>("State"),
                         CreditLimit = custrow.Field<decimal>("CreditLimit"),
                         Orders = getOrders(orders, custrow.Field<int>("CustNum")),
                     };
```

Finally, **GetCustOrders** returns the **custquery** I just built, again as an **Enumerable**.

```
                 return custquery.AsEnumerable();
                 }
```

I finish up with a placeholder for a **catch** block that will do any error handling:

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

The **getOrders** method, which builds an **orderquery** with rows from the **Orders** DataTable that match the **custNum** value passed, in looks very similar. For completeness the entire code for the DomainService class is shown here:

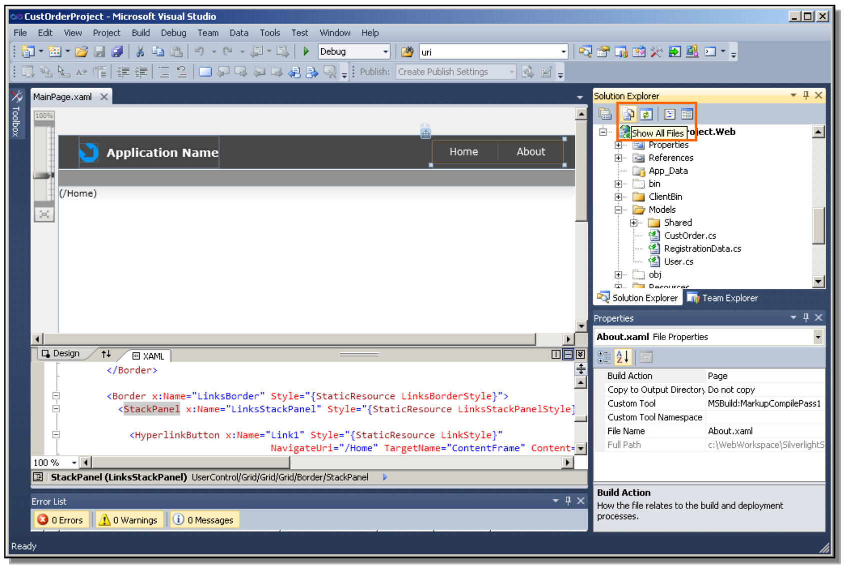
```
namespace CustOrderProject.Web.Services
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.ComponentModel.DataAnnotations;
    using System.Linq;
    using System.ServiceModel.DomainServices.Hosting;
    using System.ServiceModel.DomainServices.Server;

    using System.Web;
    using System.Web.Caching;
    using Progress.Open4GL;
    using Progress.Open4GL.Proxy;
    using System.Data;
    using Models;
    using CustOrdersNS;
    using CustOrdersNS.StrongTypesNS;

    // TODO: Create methods containing your application logic.
    [EnableClientAccess()]
    public class CustOrderDomainService : DomainService
    {
        private CustOrdersNS.CustOrderApp _appObj = null;

        CustOrdersDataSet CustOrderDS = null;

        public CustOrderDomainService()
            : base()
        {
            try
            {
                Cache COCache = HttpRuntime.Cache;
                _appObj = (CustOrderApp)COCache.Get("COAppObj");
                if (_appObj == null)
                {
                    string urlString = "AppServer://localhost/asbroker1";
                    RunTimeProperties.SessionModel = 1; // state-free
                    Connection connObj = new Connection(urlString, "", "", "");
                    _appObj = new CustOrderApp(connObj);
                    COCache.Insert("COAppObj", (object)_appObj);
                }
                CustOrderDS = new CustOrdersDataSet();
            }
            catch (System.Exception e)
            {
                throw new Exception(e.Message);
            }
        }

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

                DataTable customers = (DataTable)CustOrderDS.Tables["ttCustomer"];
                DataTable orders = (DataTable)CustOrderDS.Tables["ttOrder"];

                IEnumerable<CustOrder> custquery =
                    from custrow in customers.AsEnumerable()
                    select new CustOrder
                    {
                        CustNum = custrow.Field<int>("CustNum"),
                        Name = custrow.Field<string>("Name"),
                        Address = custrow.Field<string>("Address"),
                        City = custrow.Field<string>("City"),
                        State = custrow.Field<string>("State"),
                        CreditLimit = custrow.Field<decimal>("CreditLimit"),
                        Orders = getOrders(orders, custrow.Field<int>("CustNum")),
                    };
                return custquery.AsEnumerable();
            }
            catch (System.Exception e)
            {
                throw new Exception(e.Message);
            }
        }

        private IEnumerable<Order> getOrders(DataTable orders, int custNum)
        {
            IEnumerable<Order> orderquery =
                from orderrow in orders.AsEnumerable()
                where orderrow.Field<int>("CustNum") == custNum
```

```
                        select new Order
                        {
                            OrderNum = orderrow.Field<int>("OrderNum"),
                            CustNum = orderrow.Field<int>("CustNum"),
                            OrderDate = orderrow.Field<DateTime>("OrderDate"),
                            Carrier = orderrow.Field<string>("Carrier"),
                            SalesRep = orderrow.Field<string>("SalesRep"),

                        };

                    return orderquery;
                }
            }
        }
```

This one example should provide sufficient detail that you can use it as the basis for examples you can start to build using your own data and your own ABL procedures. In the next section, I finally get to the Silverlight user interface itself, showing you a few key parts of the DomainContext class, and then a very simple page for the Silverlight Business Application that takes the Customer and Order data and displays it using Silverlight datagrid controls.

BUILDING A USER INTERFACE

In this section, corresponding to part six of the video series, I'm finally ready to define a simple user interface in Silverlight that will display the data retrieved through the .NET proxy and the DomainService class. First, having completed the code for the DomainService, I do a **Build** of my CustOrderProject solution. This generates the DomainContext class that is the client counterpart to the DomainService. When the Build completes, I can look at a few key parts of that generated class. I select the button in the Solution Explorer marked **Show All Files**:



```
            </Border>

            <Border x:Name="LinksBorder" Style="{StaticResource LinksBorderStyle}">
              <StackPanel x:Name="LinksStackPanel" Style="{StaticResource LinksStackPanelStyle}

                <HyperlinkButton x:Name="Link1" Style="{StaticResource LinkStyle}"
                                     NavigateUri="/Home" TargetName="ContentFrame" Content=
```

100 %

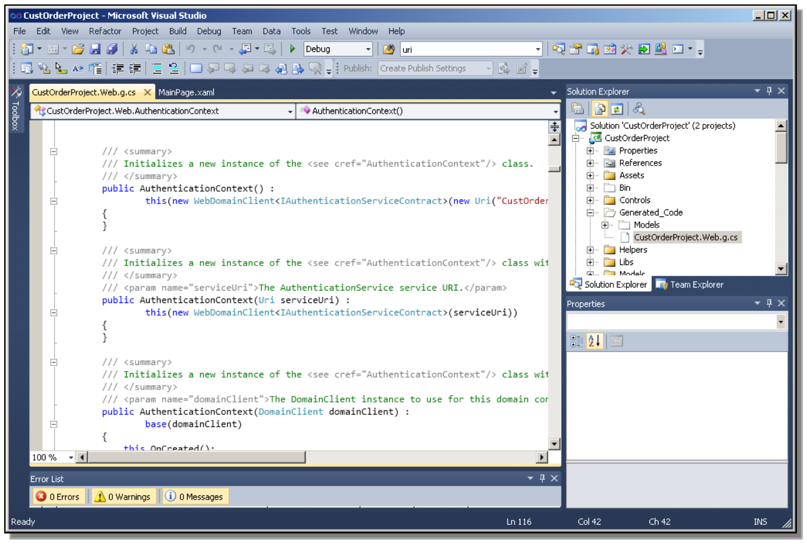
StackPanel (LinksStackPanel) UserControl/Grid/Grid/Grid/Border/StackPanel

About.xaml File Properties

Build Action	Page
Copy to Output Directory	Do not copy
Custom Tool	MSBuild:MarkupCompilePass1
Custom Tool Namespace	
File Name	About.xaml
Full Path	c:\WebWorkspace\SilverlightS

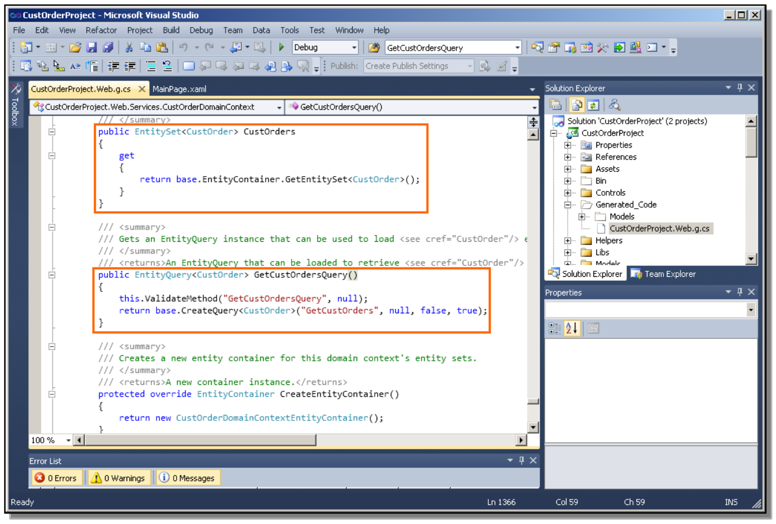
Build Action
How the file relates to the build and deployment processes.

I need to do this to find the DomainContext class, because, being a fully generated class, derived from the DomainService, it's not intended that you would ever edit it, and normally you shouldn't need to look at it. It's in a folder that I can now see named **Generated_Code**. At the top there is a warning that you shouldn't ever edit the file: any changes you made would be lost the next time you did a build. Further down in the body of the file, you can see just a bit of the extensive code that's generated to support the authentication and user registration features of the Silverlight Business Application template:



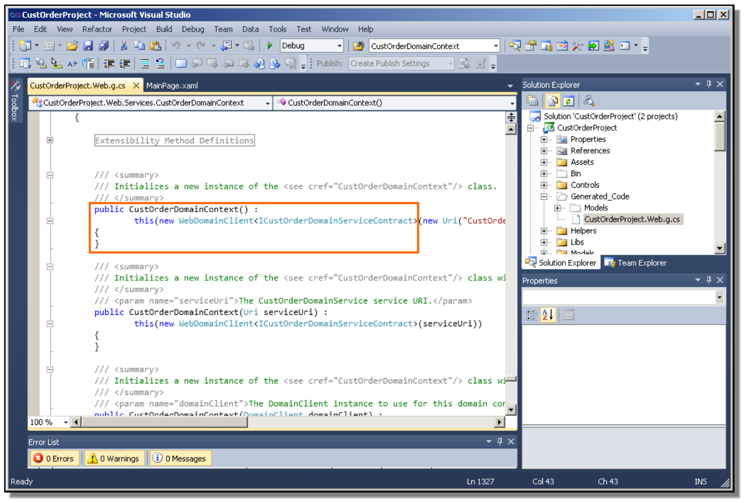
```
/// <summary>
/// Initializes a new instance of the <see cref="AuthenticationContext"/> class.
/// </summary>
public AuthenticationContext() :
        this(new WebDomainClient<IAuthenticationServiceContract>(new Uri("CustOrder
{
}

/// <summary>
/// Initializes a new instance of the <see cref="AuthenticationContext"/> class wit
/// </summary>
/// <param name="serviceUri">The AuthenticationService service URI.</param>
public AuthenticationContext(Uri serviceUri) :
        this(new WebDomainClient<IAuthenticationServiceContract>(serviceUri))
{
}

/// <summary>
/// Initializes a new instance of the <see cref="AuthenticationContext"/> class wit
/// </summary>
/// <param name="domainClient">The DomainClient instance to use for this domain cor
public AuthenticationContext(DomainClient domainClient) :
        base(domainClient)
{
    this.OnCreated();
```

At the top of the next screenshot you can **GetCustOrdersQuery** see a property named **CustOrders** that's defined as an **EntitySet** of the object **CustOrder**. I'll be referencing CustOrders as a set of CustOrder objects later in this document. Remember that using the **Query** data annotation in brackets in the DomainService class ahead of the **GetCustOrders** method causes a method to be generated in the DomainContext called. The next screenshot shows that method. It will be invoked later from the user interface support code:



```
/// </summary>
public EntitySet<CustOrder> CustOrders
{
    get
    {
        return base.EntityContainer.GetEntitySet<CustOrder>();
    }
}

/// <summary>
/// Gets an EntityQuery instance that can be used to load <see cref="CustOrder"/> e
/// </summary>
/// <returns>An EntityQuery that can be loaded to retrieve <see cref="CustOrder"/>
public EntityQuery<CustOrder> GetCustOrdersQuery()
{
    this.ValidateMethod("GetCustOrdersQuery", null);
    return base.CreateQuery<CustOrder>("GetCustOrders", null, false, true);
}

/// <summary>
/// Creates a new entity container for this domain context's entity sets.
/// </summary>
/// <returns>A new container instance.</returns>
protected override EntityContainer CreateEntityContainer()
{
    return new CustOrderDomainContextEntityContainer();
}
```

Remember as well that I said it's important to end the DomainService class name with the string **Service**, as that gets automatically converted to **Context** as a naming convention here. Below you can see several overloaded constructors for the **CustOrderDomainContext** class:


CustOrderProject - Microsoft Visual Studio
CustOrderProject.Web.g.cs
MainPage.xaml
CustOrderProject.Web.Services.CustOrderDomainContext
CustOrderDomainContext()
Solution Explorer
Error List
0 Errors
0 Warnings
0 Messages
Ready
Ln 1327

```
{

    Extensibility Method Definitions

    /// <summary>
    /// Initializes a new instance of the <see cref="CustOrderDomainContext"/> class.
    /// </summary>
    public CustOrderDomainContext() :
            this(new WebDomainClient<ICustOrderDomainServiceContract>(new Uri("CustOrde
    {
    }

    /// <summary>
    /// Initializes a new instance of the <see cref="CustOrderDomainContext"/> class wi
    /// </summary>
    /// <param name="serviceUri">The CustOrderDomainService service URI.</param>
    public CustOrderDomainContext(Uri serviceUri) :
            this(new WebDomainClient<ICustOrderDomainServiceContract>(serviceUri))
    {
    }

    /// <summary>
    /// Initializes a new instance of the <see cref="CustOrderDomainContext"/> class wi
    /// </summary>
    /// <param name="domainClient">The DomainClient instance to use for this domain cor
```

There's also a lot of code to support all of the DataTables and individual fields that the DomainService class references. This is all part of what you get for free using the standard support of RIA Services and the Silverlight Business Application template.

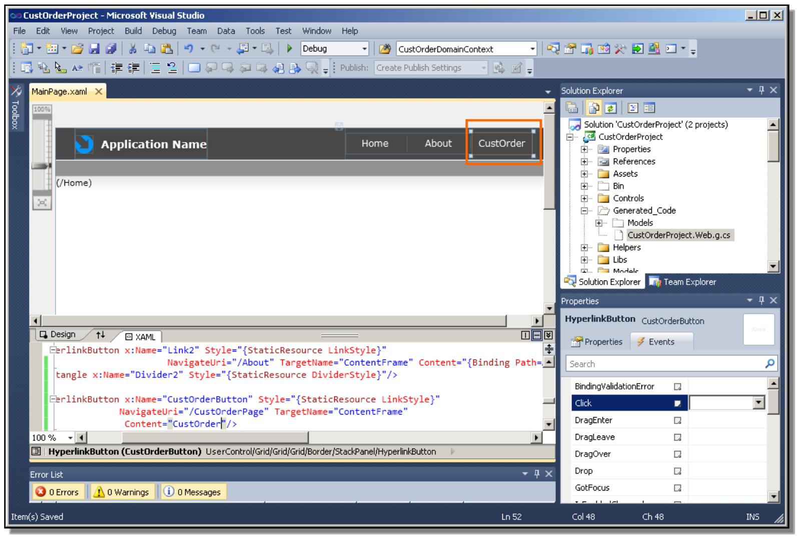
Back in the user interface MainPage that the Business Application template started me with, you can see buttons for the default Home and About pages that you could use to get started. Scrolling over to see the XAML for those pages, you can see the XAML definition for the **Rectangle** controls that outline the two **HyperLinkButton** controls, as they're called, for the **Home** and **About** buttons:

```
<HyperlinkButton x:Name="Link1" Style="{StaticResource LinkStyle}"
    NavigateUri="/Home" TargetName="ContentFrame"
    Content="{Binding Path=ApplicationStrings.HomePageTitle,
    Source={StaticResource ResourceWrapper}}"/>

<Rectangle x:Name="Divider1" Style="{StaticResource DividerStyle}"/>

<HyperlinkButton x:Name="Link2" Style="{StaticResource LinkStyle}"
    NavigateUri="/About" TargetName="ContentFrame”
    Content="{Binding Path=ApplicationStrings.AboutPageTitle,
    Source={StaticResource ResourceWrapper}}"/>
```

You can of course build your user interface by selecting controls from the Toolbox, as I'll do in a moment for the DataGrids that will display Customers and Orders. But I'll do this edit in the XAML itself just to show that you are free to edit the code directly if you want to. Also, I want another button that looks just like the first two, which is easy to do this way. I paste in a copy of the **Rectangle** and **HyperLinkButton** control definitions. In the copy of those definitions, I give the rectangle a new name, **Divider2**, and I name the button **CustOrderButton**. The page to run when it's clicked isn't the About page, but rather a new page that I'll create in a moment called **CustOrderPage**. The **Content** property is the label that's displayed.

```
<Rectangle x:Name="Divider2" Style="{StaticResource DividerStyle}"/>

<HyperlinkButton x:Name="CustOrderButton" Style="{StaticResource LinkStyle}"
       NavigateUri="/CustOrderPage" TargetName="ContentFrame"
       Content="CustOrder"/>
```

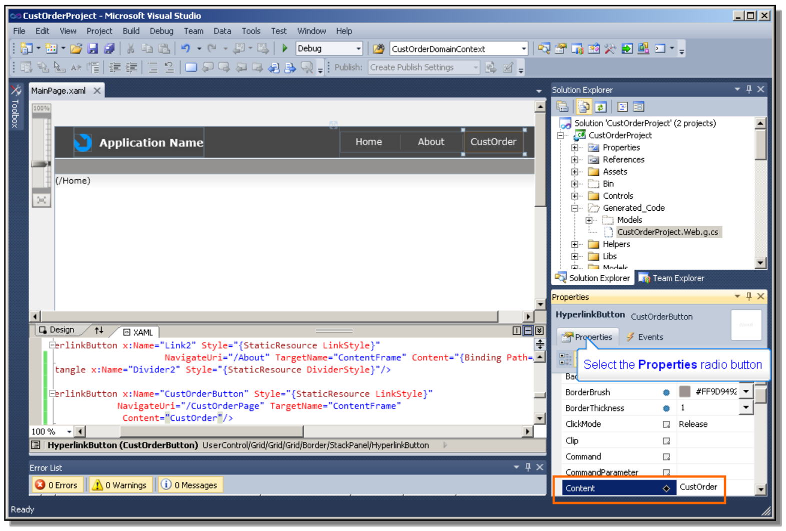
I can see the changes reflected immediately in the design pane:



```
erlinkButton x:Name="Link2" Style="{StaticResource LinkStyle}"
                     NavigateUri="/About" TargetName="ContentFrame" Content="{Binding Path=
tangle x:Name="Divider2" Style="{StaticResource DividerStyle}"/>

erlinkButton x:Name="CustOrderButton" Style="{StaticResource LinkStyle}"
           NavigateUri="/CustOrderPage" TargetName="ContentFrame"
            Content="CustOrder"/>
```

HyperlinkButton (CustOrderButton) UserControl/Grid/Grid/Grid/Border/StackPanel/HyperlinkButton

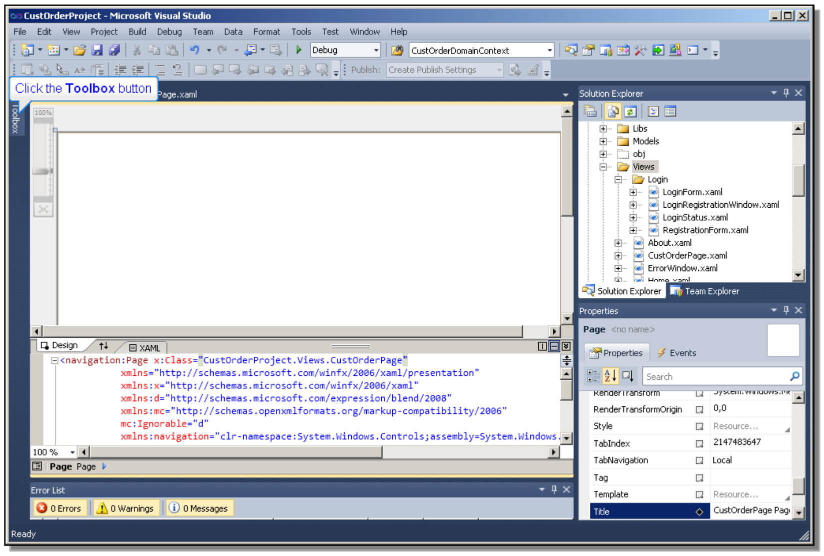
I can also see the properties displayed over in the **Properties** tab:



BorderBrush	#FF9D9492
BorderThickness	1
ClickMode	Release
Clip	
Command	
CommandParameter	
Content	CustOrder

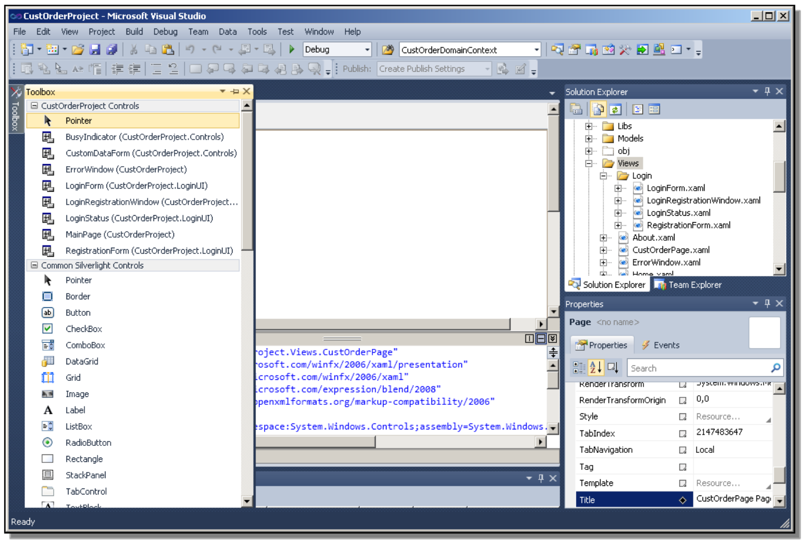
I can make changes in any of those three places.

Now it's time to create the new page of XAML where I'll put the controls to display Customers and Orders. I right-click on the **Views** and select **Add**, and **Class**. From all the templates available, I select **Silverlight Page**, and give it the name I used in the new **`HyperLinkButton`** on the MainPage, namely **`CustOrderPage`**. I then **Add** that to the project. This gives me a blank page to work with:

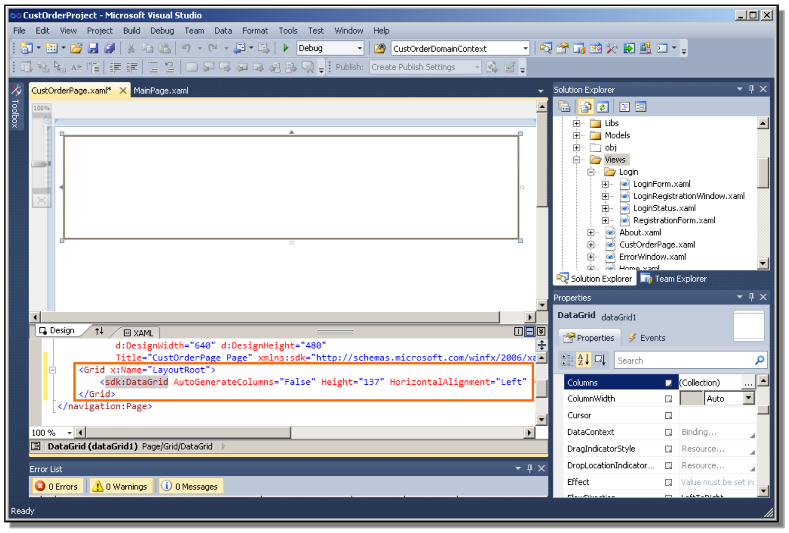


```
<navigation:Page x:Class="CustOrderProject.Views.CustOrderPage"
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
           xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
           mc:Ignorable="d"
           xmlns:navigation="clr-namespace:System.Windows.Controls;assembly=System.Windows.
```

I'm going to drop two grid controls onto the page, one to browse Customers and the second to display Orders for a selected Customer. Opening up the Toolbox, I select the **DataGrid**, and just drag that onto the design canvas, and drop it:

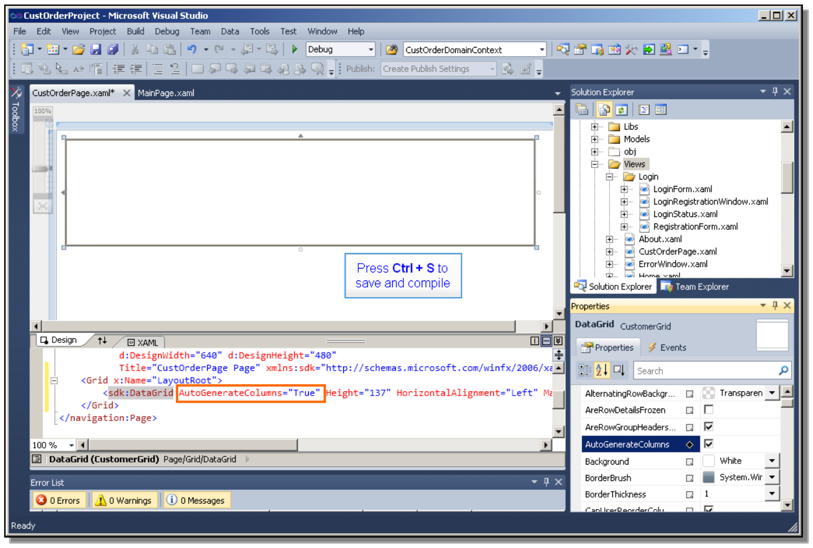

CustOrderProject - Microsoft Visual Studio
Toolbox
CustOrderProject Controls
Pointer
BusyIndicator (CustOrderProject.Controls)
CustomDataForm (CustOrderProject.Controls)
ErrorWindow (CustOrderProject)
LoginForm (CustOrderProject.LoginUI)
LoginRegistrationWindow (CustOrderProject...
LoginStatus (CustOrderProject.LoginUI)
MainPage (CustOrderProject)
RegistrationForm (CustOrderProject.LoginUI)
Common Silverlight Controls
Pointer
Border
Button
CheckBox
ComboBox
DataGrid
Grid
Image
Label
ListBox
RadioButton
Rectangle
StackPanel
TabControl
Solution Explorer
Properties
Ready

Immediately the generated XAML code reflects the initial properties of the new control. If I reposition and resize the control visually, the property values in the code are changed as I go:



```
        d:DesignWidth="640" d:DesignHeight="480"
        Title="CustOrderPage Page" xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xa
    <Grid x:Name="LayoutRoot">
        <sdk:DataGrid AutoGenerateColumns="False" Height="137" HorizontalAlignment="Left"
    </Grid>
</navigation:Page>
```

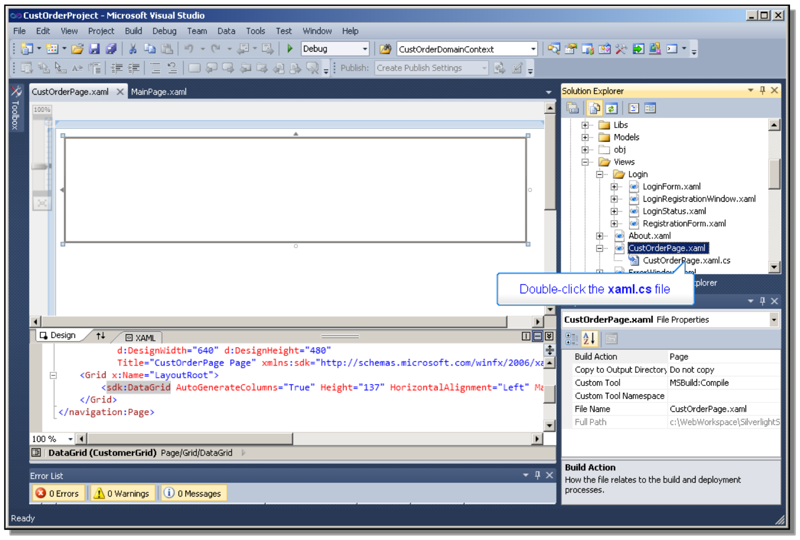
I can set those properties in the **Properties** pane as well, of course. For instance I change the default control name to **`CustomerGrid`**. For simplicity's sake I also set the property called **`AutoGenerateColumns`**, which does just what the name implies. Again I can see that property setting reflected in the XAML for the page:



Press **Ctrl + S** to save and compile

```
        d:DesignWidth="640" d:DesignHeight="480"
        Title="CustOrderPage Page" xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xa
    <Grid x:Name="LayoutRoot">
        <sdk:DataGrid AutoGenerateColumns="True" Height="137" HorizontalAlignment="Left" Ma
    </Grid>
</navigation:Page>
```

In the introductory video on Silverlight I showed you that there is a C# file of executable code associated with each XAML page that's generated. This is where event handlers and other user interface logic will go. To see that file, I expand the XAML file entry in the **Solution Explorer**. The supporting C# file, referred to as the code-behind, has the same name, with .cs appended to it.


CustOrderProject - Microsoft Visual Studio
Solution Explorer
CustOrderPage.xaml
CustOrderPage.xaml.cs
Double-click the xaml.cs file

```
d:DesignWidth="640" d:DesignHeight="480"
Title="CustOrderPage Page" xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xa
<Grid x:Name="LayoutRoot">
    <sdk:DataGrid AutoGenerateColumns="True" Height="137" HorizontalAlignment="Left" Ma
</Grid>
</navigation:Page>
```

DataGrid (CustomerGrid) Page/Grid/DataGrid

CustOrderPage.xaml File Properties

Build Action	Page
Copy to Output Directory	Do not copy
Custom Tool	MSBuild:Compile
Custom Tool Namespace	
File Name	CustOrderPage.xaml
Full Path	c:\WebWorkspace\SilverlightS

Build Action
How the file relates to the build and deployment processes.

A new C# user interface support file is generated automatically as a starting point for any UI logic that's needed.

As with the other C# classes in these examples, I need a few **`using`** statements beyond the ones provided. First there are a couple to support the **`DomainServices Client`**, and then a second for **`ApplicationServices`**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Windows.Navigation;

using System.ServiceModel.DomainServices.Client;
using System.ServiceModel.DomainServices.Client.ApplicationServices;
```

Next I need to add the namespace of my own project's Web project, and within that, a reference to the **`Services`** folder containing the DomainService, and another to the **`Models`** folder, where the CustOrder entity class is located:

```
using CustOrderProject.Web;
using CustOrderProject.Web.Services;
using CustOrderProject.Web.Models;
```

The generated code includes the start of the class that bears the same name as my XAML page:

```
namespace CustOrderProject.Views
{
    public partial class CustOrderPage : Page
    {
```

The first thing I need to add to the class is a reference to an instance of my DomainContext class. I name it **CODomainContext**. Remember that this is the client side of the code that gets data and other parameters back and forth between client and server:

```
        CustOrderDomainContext CODomainContext;
```

Then in the constructor for the class, after the standard InitializeComponent method, I create an instance of the DomainContext class:

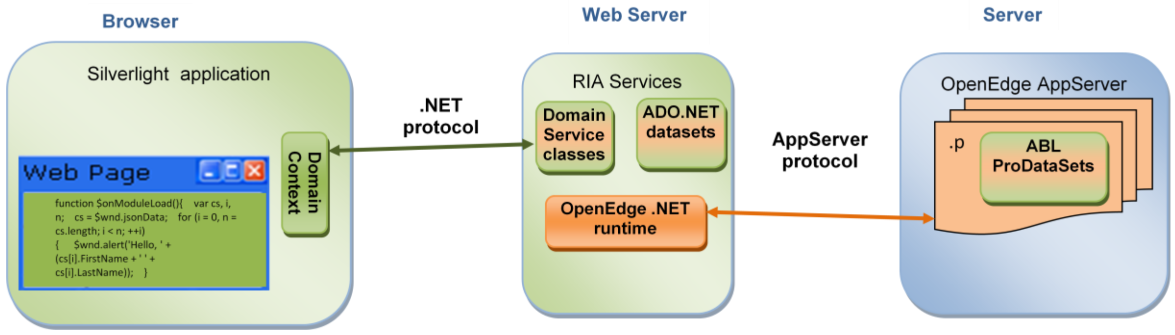
```
        public CustOrderPage()
        {
            InitializeComponent();

            CODomainContext = new CustOrderDomainContext();
```

There's a standard **Load** method that the DomainContext inherits that I can use to create a **LoadOperation** for the CustOrder entity. I call it **loadOp**, and within the DomainContext, I invoke the **Load** method:

```
            LoadOperation<CustOrder> loadOp =
                CODomainContext.Load(CODomainContext.GetCustOrdersQuery(),
                    OnLoadCompleted, null);
```

That **LoadOperation** of **TEntity** is a generic type that is basically a typesafe pointer to the asynchronous **Load** method of the DomainContext. The method that **Load** uses to load data is the **GetCustOrdersQuery** method that I showed you earlier, which was generated in the DomainContext class because I specified in the DomainService that **GetCustOrders** is a query operation.

This is the front end of a series of references that go from the XAML user interface to this C# supporting code, to a query method in the DomainContext class, to the **GetCustOrders** method that runs in the DomainService in the Web server, and then via the .NET proxy dll to the **GetCustOrders.p** ABL procedure running in the AppServer:



```
function $onModuleLoad(){    var cs, i,
n;    cs = $wnd.jsonData;    for (i = 0, n =
cs.length; i < n; ++i)
{      $wnd.alert('Hello, ' +
(cs[i].FirstName + ' ' +
cs[i].LastName));    }
```

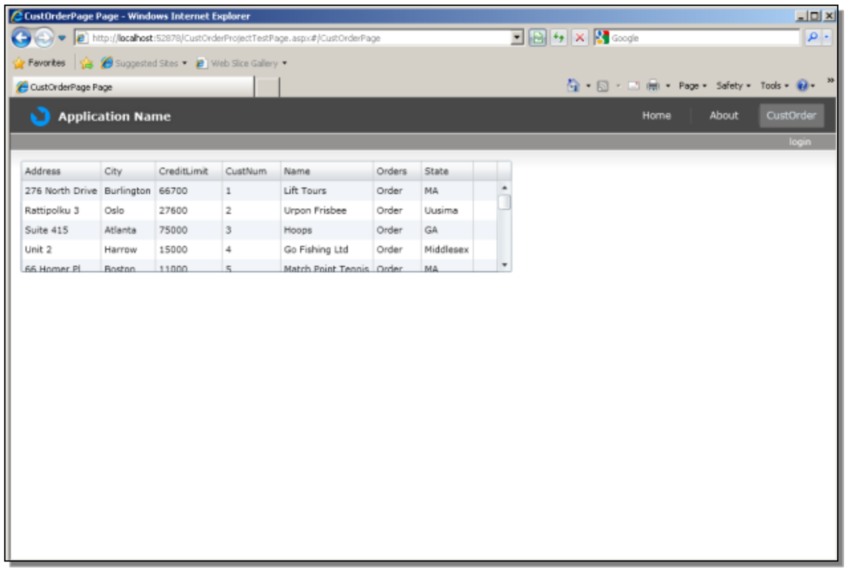
The **Load** operation is asynchronous, so I specify a method to invoke when the **Load** completes, and there's an optional flag to indicate whether I want an exception thrown if there's an unhandled error condition. Instead I define a simple completion method. If there’s an error, it displays a message box with the error text and marks the error as handled:

```
        private void OnLoadCompleted(LoadOperation<CustOrder> lo)
        {
            if (lo.HasError)
            {
                MessageBox.Show(string.Format("Retrieving data failed: {0}",
                    lo.Error.Message));
                lo.MarkErrorAsHandled();
            }
        }
```

Now I have the supporting class with a method to load data by running **GetCustOrders.** The only other thing required is to assign that data to be the data source for the Customer DataGrid. This is called the **ItemsSource** property of the grid control. I set that to the **CustOrders** EntitySet seen earlier in this document, in the generated DomainContext class.

```
            CustomerGrid.ItemsSource = CODomainContext.CustOrders;
```

Now I have everything needed to retrieve the **CustOrders** DataSet and display at least the Customers in the Silverlight UI. Let’s see what happens when I run the project. The first time you do this in a session, a popup appears to show you that an ASP.NET Development Server has been started to host the test web page for the application. Then the the default Home page comes up. The port number of the Development Server in the URL, along with the test page that is part of the generated code in the Web project that supports the application. Clicking the new CustOrder button, I see the **CustOrderPage** the first of the Customers retrieved via **GetCustOrders.p**:



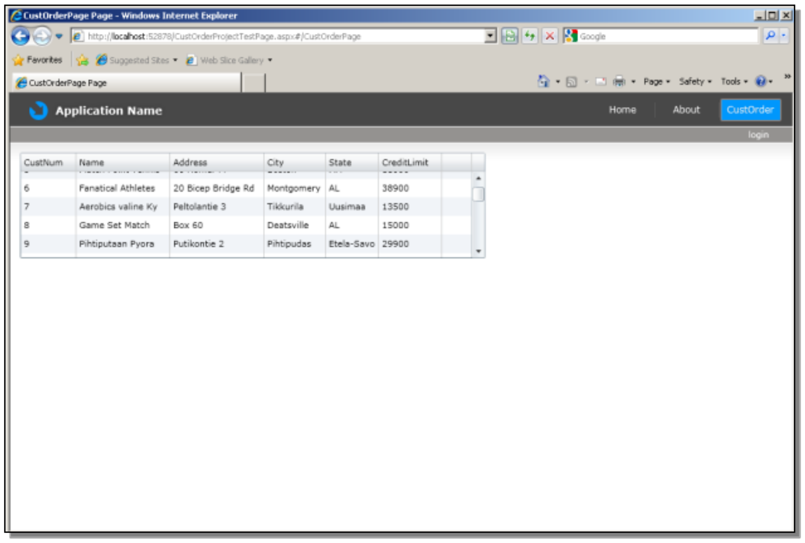
Address	City	CreditLimit	CustNum	Name	Orders	State
276 North Drive	Burlington	66700	1	Lift Tours	Order	MA
Rattipolku 3	Oslo	27600	2	Urpon Frisbee	Order	Uusima
Suite 415	Atlanta	75000	3	Hoops	Order	GA
Unit 2	Harrow	15000	4	Go Fishing Ltd	Order	Middlesex
66 Homer Pl	Boston	11000	5	Match Point Tennis	Order	MA

Because I just set the **AutoGenerateColumns** property for the grid, I get a column for every field, with the field name as a header, but they’re in alphabetical order, which isn’t very useful, and there’s a column representing the Orders for the Customer that doesn't contain any meaningful data. So I next refine the grid definition a bit with a new event handler to address that. Back in the design window, showing the XAML for the **CustOrdersPage**, I select the **Events** tab. The grid control supports a large number of events. I want one that controls what happens when columns are automatically generated, because of the property setting I made. The event I want is called **AutoGeneratingColumn**. If I double-click in the event handler name fill-in, I'm taken back to the C# file and given a starting skeleton for an event handler method.

Just as in the OpenEdge GUI for .NET, if you've used the support for that user interface technology, an event handler takes two parameters. The first is the object that generated the event, the **sender**, and the second is an object simply named **e** that contains the arguments that define the specific data associated with the event. The **case** statement shown below puts each column in the right order in the grid, and also specifies that the **Orders** property within the Customer object shouldn't be displayed in the **CustomerGrid**:

```
        private void CustomerGrid_AutoGeneratingColumn(object sender,
           DataGridAutoGeneratingColumnEventArgs e)
        {
           string headername = e.Column.Header.ToString();

           switch (headername)
           {
               case "CustNum":
                   e.Column.DisplayIndex = 0;
                   break;
               case "Name":
                   e.Column.DisplayIndex = 1;
                   break;
               case "Address":
                   e.Column.DisplayIndex = 2;
                   break;
               case "City":
                   e.Column.DisplayIndex = 3;
                   break;
               case "State":
                   e.Column.DisplayIndex = 4;
                   break;
               case "CreditLimit":
                   e.Column.DisplayIndex = 5;
                   break;
               case "Orders":
                   e.Column.Visibility = Visibility.Collapsed;
                   break;
           }
       }
```

Now I can save and re-run the application. I go back to the **CustOrdersPage**, and now the fields are in the right order:


CustOrderPage Page - Windows Internet Explorer
http://localhost:52878/CustOrderProjectTestPage.aspx#/CustOrderPage
Favorites
Suggested Sites
Web Slice Gallery
CustOrderPage Page
Page
Safety
Tools
Application Name
Home
About
CustOrder
login

CustNum	Name	Address	City	State	CreditLimit
6	Fanatical Athletes	20 Bicep Bridge Rd	Montgomery	AL	38900
7	Aerobics valine Ky	Peltolantie 3	Tikkurila	Uusimaa	13500
8	Game Set Match	Box 60	Deatsville	AL	15000
9	Pihtiputaan Pyora	Putikontie 2	Pihtipudas	Etela-Savo	29900

Now I need another grid control to display Orders in. Back in the Toolbox, I grab another DataGrid, and drop it onto the design canvas. As before I can reposition and resize it visually, and the generated XAML code reflects the changes I make. In this case I name the grid **OrderGrid**. Once again I set the **AutoGenerateColumns** property, and in the events list, I find the same **AutoGeneratingColumn** event defined for the Customer grid. I paste in a similar **case** statement based on the value of the header string for each column, to get the columns into a reasonable display order.

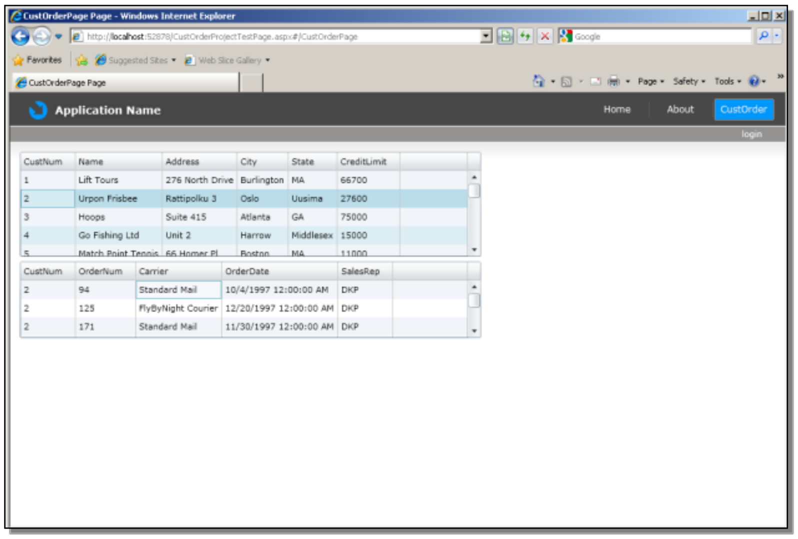
One more thing I need to do is to define an event on the **CustomerGrid** that repopulates the **OrderGrid** with the right Orders when I select a Customer. That's the **SelectionChanged** event. Because it's the default event for a DataGrid, I can just double-click on the control itself and get an event handler for it. I want to identify the current instance of the **CustOrder** entity that the top-level grid is navigating. I call that **selectedCust**. It's an object of type **CustOrder**, so I cast it as such. Within the **CustomerGrid**, it's the object identified by the property **selectedItem**. Next I need to define the data source for the **OrderGrid** just as I did for the **CustomerGrid**. I set the **ItemsSource** property to be the set of Orders for the selected Customer:

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

As a reminder, this is the code in the entity class that defines **Orders** as one of the properties of the top-level **CustOrder** object. That's what I'm using to populate the **OrderGrid**:

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

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

Back in the supporting class for the **CustOrderPage**, I can save these latest changes, do a Build of the solution, and run it again. In the **CustOrdersPage**, I see Customers and an empty **OrderGrid**, because it isn't populated until I select a Customer. When I do that, the **OrderGrid** shows the Orders for the Customer:


CustOrderPage Page - Windows Internet Explorer
http://localhost:52878/CustOrderProjectTestPage.aspx#/CustOrderPage
Favorites
Suggested Sites
Web Slice Gallery
CustOrderPage Page
Page
Safety
Tools
Application Name
Home
About
CustOrder
login

CustNum	Name	Address	City	State	CreditLimit
1	Lift Tours	276 North Drive	Burlington	MA	66700
2	Urpon Frisbee	Rattipolku 3	Oslo	Uusima	27600
3	Hoops	Suite 415	Atlanta	GA	75000
4	Go Fishing Ltd	Unit 2	Harrow	Middlesex	15000
5	Match Point Tennis	66 Homer Pl	Boston	MA	11000

CustNum	OrderNum	Carrier	OrderDate	SalesRep
2	94	Standard Mail	10/4/1997 12:00:00 AM	DKP
2	125	FlyByNight Courier	12/20/1997 12:00:00 AM	DKP
2	171	Standard Mail	11/30/1997 12:00:00 AM	DKP

For reference, the complete **CustOrderPage.xaml.cs** file is reproduced here:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Windows.Navigation;

using System.ServiceModel.DomainServices.Client;
using System.ServiceModel.DomainServices.Client.ApplicationServices;

using CustOrderProject.Web;
using CustOrderProject.Web.Services;
using CustOrderProject.Web.Models;

namespace CustOrderProject.Views
{
    public partial class CustOrderPage : Page
    {
        CustOrderDomainContext CODomainContext;
        public CustOrderPage()
        {
            InitializeComponent();

            CODomainContext = new CustOrderDomainContext();

            LoadOperation<CustOrder> loadOp =
                CODomainContext.Load(CODomainContext.GetCustOrdersQuery(),
                    OnLoadCompleted, null);
            CustomerGrid.ItemsSource = CODomainContext.CustOrders;
        }

        private void OnLoadCompleted(LoadOperation<CustOrder> lo)
        {
            if (lo.HasError)
            {
                MessageBox.Show(string.Format("Retrieving data failed: {0}",
                    lo.Error.Message));
                lo.MarkErrorAsHandled();
            }
        }
        // Executes when the user navigates to this page.
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
        }

        private void CustomerGrid_AutoGeneratingColumn(object sender,
            DataGridAutoGeneratingColumnEventArgs e)
        {
            string headername = e.Column.Header.ToString();

            switch (headername)
            {
                case "CustNum":
                    e.Column.DisplayIndex = 0;
                    break;
                case "Name":
                    e.Column.DisplayIndex = 1;
                    break;
                case "Address":
                    e.Column.DisplayIndex = 2;
                    break;
                case "City":
                    e.Column.DisplayIndex = 3;
                    break;
                case "State":
                    e.Column.DisplayIndex = 4;
                    break;
                case "CreditLimit":
                    e.Column.DisplayIndex = 5;
                    break;
                case "Orders":
                    e.Column.Visibility = Visibility.Collapsed;
                    break;
            }
        }

        private void OrderGrid_AutoGeneratingColumn(object sender,
DataGridAutoGeneratingColumnEventArgs e)
        {
            string headername = e.Column.Header.ToString();

            switch (headername)
            {
                case "CustNum":
                    e.Column.DisplayIndex = 0;
                    break;
                case "OrderNum":
                    e.Column.DisplayIndex = 1;
                    break;
                case "OrderDate":
                    e.Column.DisplayIndex = 2;
                    break;
                case "Carrier":
                    e.Column.DisplayIndex = 3;
                    break;
                case "SalesRep":
                    e.Column.DisplayIndex = 4;
                    break;

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

        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();
            }
        }
    }
}
```

I’ve finally gotten you all the way through the process of retrieving data from an ABL procedure and displaying it in a Silverlight user interface using RIA Services. The purpose here has been to show you concrete examples of the coding you need to do to get the data binding part to work, since that’s the essential element that connects your ABL application to a Silverlight UI. Clearly there’s a lot for you to learn in order to build a UI that reflects all the capabilities of Silverlight, but that part you should be able to learn from the materials on the Silverlight site. This series continues with examples of how to do updates using RIA Services, as well as the Invoke operation that passes non-dataset parameters to and from a request.