* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Web Services and Silverlight

**Contents**

[Lab 07: Working with WCF Services in Silverlight Applications 3](#_Toc276335815)

[You Will Benefit from this Lab if: 4](#_Toc276335816)

[You Will Learn: 4](#_Toc276335817)

[Business Requirements for the Silverlight application include: 5](#_Toc276335818)

[Exercise 1: Creating a Silverlight-Enabled WCF Service 6](#_Toc276335819)

[Create a new Silverlight Navigation Application 6](#_Toc276335820)

[Exercise 2: Calling a WCF Service from a Silverlight Client 11](#_Toc276335821)

[Create a WCF service proxy 11](#_Toc276335822)

[Exercise 3: Implementing IEditableObject to Support Cancellation 21](#_Toc276335823)

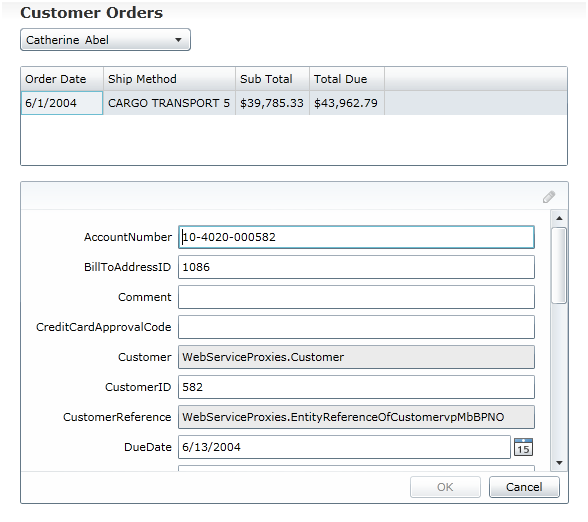
[Implementing IEditableObject on a Partial Class 21](#_Toc276335824)

[Exercise 4 (Optional): Debugging WCF Service Calls 24](#_Toc276335825)

[Debugging with Fiddler 24](#_Toc276335826)

[Summary 27](#_Toc276335827)

Lab 07: Working with WCF Services in Silverlight Applications

* 1. Windows Communication Foundation (WCF) provides an excellent framework for exchanging data across network boundaries. It can be used to exchange data using standards-based Web Service specifications such as Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and WS-I\* standards. Any development framework capable of supporting Web Service standards can be used to call a WCF service including non-.NET frameworks.
  2. To create a WCF service you define the ABCs for the service including the Address, Binding and Contract. The Address defines the URI of the service that a client uses to talk with it. You can think of it as being similar to a phone number or street address. The Binding defines the protocol to use while talking with the service. WCF supports several different protocols such as HTTP, TCP, named pipes and more. Finally, the contract defines the service's Application Programming Interface (API) including the data it can send and receive. The contract includes service operation names, data types used, as well as additional details.
  3. WCF is a key player in Silverlight applications that need to access and manipulate data. Although standard WCF service projects can be created a used with Silverlight, The Silverlight Tools for Visual Studio 2010 provides a *Silverlight-Enabled WCF Service* project template that can be used to create service classes. The template configures services to use binary message encoding combined with the HTTP protocol which provides excellent performance. A WCF service can be called directly from a Silverlight application using a proxy object that is typically created directly in Visual Studio.
  4. In this lab you'll learn how to create a Silverlight-Enabled WCF Service and define operations. You'll also examine the default configuration for Silverlight-enabled services and create a proxy object that can be used to communicate with a service from a Silverlight client. A bonus exercise is also included that demonstrates how to debug WCF service calls using a tool called Fiddler. The user interface that you'll build throughout the lab is shown next:
  5. 
  6. Figure
  7. User Interface Example

You Will Benefit from this Lab if:

* + You need to integrate distributed data into a Silverlight application
  + You're interested in learning more about Windows Communication Foundation fundamentals
  + You're interested in learning how to extend proxy objects using partial classes

You Will Learn:

* + How to create a Silverlight-Enabled WCF Service
  + How to define service operations
  + Best practices for adding code into service operations
  + How to create a proxy object that can be used to call a WCF service
  + How to make asynchronous calls from a Silverlight client to a WCF service
  + How and why to use the IEditableObject interface

Business Requirements for the Silverlight application include:

* + Create a new Silverlight Navigation Application
  + Create an Entity Framework 4 model
  + Create a Silverlight-Enabled WCF Service
  + Customize service methods
  + Create a Silverlight proxy object in Visual Studio
  + Build a user interface
  + Use a proxy object to call a WCF Service
  + Add IEditableObject functionality to a proxy generated class
  1. **Estimated Time: 60 minutes**

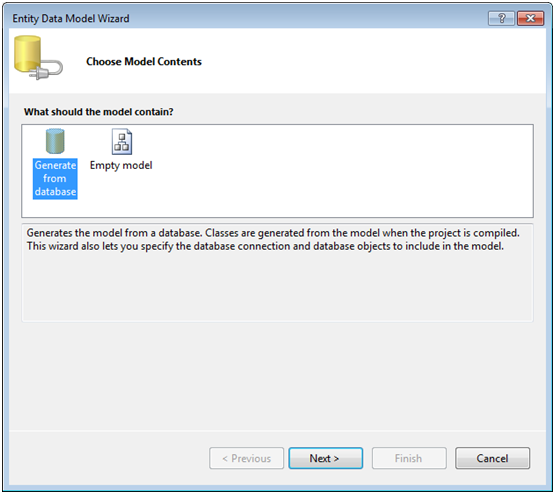
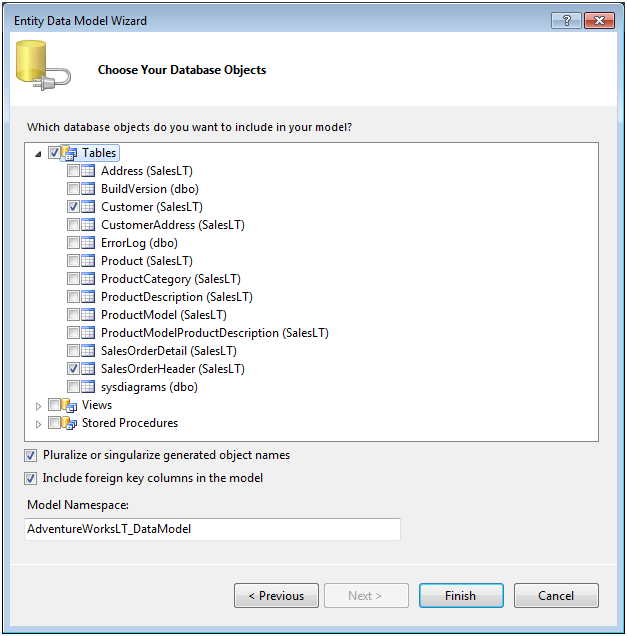
Exercise 1: Creating a Silverlight-Enabled WCF Service

* 1. In this exercise you'll create a new Silverlight Navigation Application and add a Silverlight-Enabled WCF Service to the Web project. You'll then add code into the service's operations to make calls to repository objects responsible for communicating with a database using Entity Framework 4.

Create a new Silverlight Navigation Application

* 1. Create a new **Silverlight Navigation Application** in Visual Studio 2010 named **UsingWCFServices**.
  2. Right-click on the **UsingWCFService.Web** project and select **Add 🡪 Add ASP.NET Folder 🡪 App\_Data**.
  3. Right-click on the **App\_Data** folder and select **Add 🡪 Existing Item** from the menu. Add the following file:

WCFServices/Starting Point/AdventureWorksLT\_Data.mdf

* 1. Add two new folders into **UsingWCFServices.Web** named **Models** and **Services**.
  2. Add a new **ADO.NET Entity Data Model** into the **Models** folder named **AdventureWorksLT.edmx**.
     1. **Note:** The ADO.NET Entity Data Model template can be found in the Data section of the Add New Item dialog. Alternatively, you can use Visual Studio 2010's *Search Installed Templates* feature in the upper-right corner of the dialog window to search for the template as well.
  3. Select **Generate from database** from the options and click the **Next** button.
     1. 
     2. Figure
     3. Entity Framework Wizard
  4. Ensure that **AdventureWorksLT\_Data.mdf** is selected in the drop-down list and click **Next.**
  5. Expand the **Tables** node and select the **Customer** and **SalesOrderHeader** tables and click **Finish.**
     1. 
     2. Figure
     3. Select tables
  6. Right-click on the **Services** folder and add a new **Silverlight-enabled WCF Service** into it. Name the service **CustomersService.svc**.
     1. **Note:** The *Silverlight-enabled WCF Service* template is located in the Silverlight templates section that's available when adding new items into a project.
  7. Take a moment to examine the existing **DoWork** method and notice the **OperationContract** attribute above it. This attribute is used to mark the method as a service operation that can be called from distributed applications. Delete the **DoWork** method as well as its **OperationContract** attribute from the class.

**Note:** Although you can add code logic directly into WCF service methods (often referred to as "operations"), it's recommended that you rely on external classes to handle business rules, interact with data access frameworks, etc. Rather than adding data access code into the service operations you'll rely on a set of data classes named CustomerRepository and SalesOrderHeaderRepository to perform the work in this lab.

* 1. Open **web.config** and locate the **system.serviceModel** element. Notice that a custom binding has been added (locate the **customBinding** element) that uses HTTP and binary message encoding. This combination provides excellent performance when exchanging data between a client and a service.
  2. Right-click on the **UsingWCFServices.Web**/**Models** folder and select **Add 🡪 Existing Item**. Add all of the code files found in the folder shown next into the **Models** folder:

|  |  |
| --- | --- |
| Language | File Location |
| C# | WCFServices/Starting Point/C# |
| VB | WCFServices/Starting Point/VB |

* 1. Open the **CustomerRepository** and **SalesOrderHeaderRepository** classes in the **Models** folder and take a moment to look through their code
     1. **Note:** These classes derive from a custom RepositoryBase class and contain functionality to perform different database operations.
  2. Open the **OperationStatus** class and note that it's used to return status information about different operations that occur in the repository classes.
  3. Add the following code into the **CustomersService** class and resolve any missing namespaces:
     1. C#
     2. ICustomerRepository \_CustomerRepository = new CustomerRepository();  
        ISalesOrderHeaderRepository \_OrderRepository =   
         new SalesOrderHeaderRepository();
     3. Visual Basic
     4. Dim \_CustomerRepository As ICustomerRepository = New CustomerRepository()  
        Dim \_OrderRepository As ISalesOrderHeaderRepository = \_  
         New SalesOrderHeaderRepository()
     5. **Note:** Although this code defines the repository class type to use in the CustomersService class, because each repository class implements an interface the type could be injected into the service at runtime. This is useful in situations where more loosely coupled code is needed.
  4. Add the following public methods and associated parameters into the **CustomersService** class and resolve any missing namespaces:

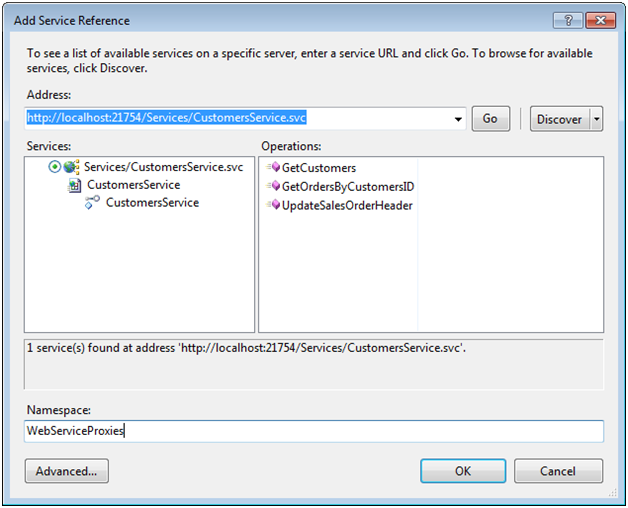
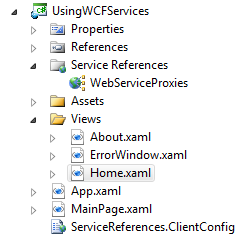
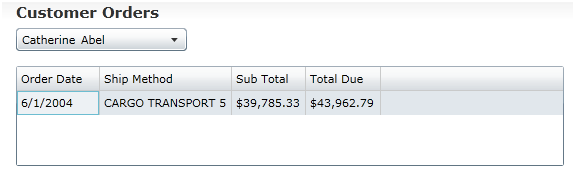
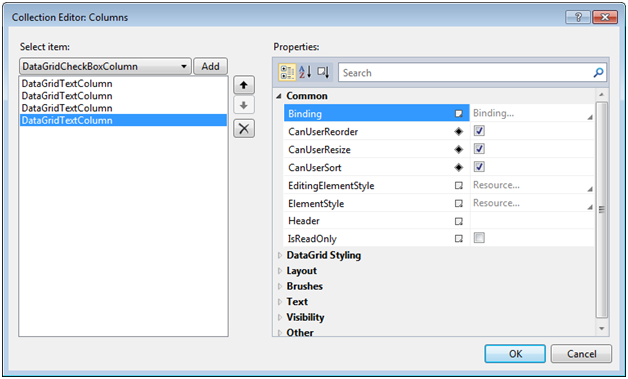
|  |  |  |
| --- | --- | --- |
| Method | Return Type | Parameters |
| GetCustomers | List of Customer | None |
| GetOrdersByCustomerID | List of SalesOrderHeader | Integer named customerID |
| UpdateSalesOrderHeader | OperationStatus | SalesOrderHeader named order |

* 1. Add the **OperationContract** attribute above each of the methods.
  2. Add code into **GetCustomers** to call the **\_CustomerRepository.GetCustomers** method:
     1. C#
     2. return \_CustomerRepository.GetCustomers();
     3. Visual Basic
     4. return \_CustomerRepository.GetCustomers()
  3. Add code into **GetOrdersByCustomerID** to call the \_**OrderRepository**.**GetOrdersByCustomerID** method and return a collection. Pass the service operation's **customerID** parameter to the repository object's method.
  4. Add code into **UpdateSalesOrderHeader** to call the \_**OrderRepository**.**UpdateSalesOrderHeader** method and return an **OperationStatus** object. Pass the service operation's **order** parameter to the repository object's method.
  5. Build the solution and resolve any compilation issues before continuing.
  6. Right-click the **CustomersService.svc** file in the Solution Explorer and select **View in Browser**. You should see a service test page appear.

Exercise 2: Calling a WCF Service from a Silverlight Client

* 1. In this exercise you'll create a WCF service proxy, add controls to a Silverlight user interface and add code to call the WCF service created in the previous exercise. Throughout the exercise you'll see how data from a Web Service can be accessed and bound to controls asynchronously. You'll also push changes made in the Silverlight client back to the WCF service so that data is updated in the database properly.
  2. **Note:** This exercise uses the DataForm control available in the Silverlight Toolkit. If you don't currently have the Silverlight Toolkit installed download and install it from http://silverlight.codeplex.com before continuing.

Create a WCF service proxy

* 1. Right-click on the **UsingWCFServices** project (the Silverlight project) and select **Add Service Reference**. Once the wizard loads click the **Discover** button to locate the service created in the previous exercise.
  2. Drill-down into **CustomersService** to see its service operations and change the value in the **Namespace** text box at the bottom of the wizard to **WebServiceProxies**. Click **OK** to create the proxy object.
     1. 
     2. Figure
     3. Add Service Reference Dialog
  3. You will see a proxy object added into the **UsingWCFServices** project as well as a new file named **ServiceReferences.ClientConfig**.
     1. 
     2. Figure
     3. ClientConfig file
  4. Open **ServiceReferences.ClientConfig** in the code editor and locate the **endpoint** **element's** **address** attribute. The service proxy class reads this value to know how to communicate with the service. An example endpoint is shown next (note that the address attribute's port may be different in your file):
     1. XAML
     2. <endpoint address="http://localhost:37156/Services/CustomersService.svc"
     3. binding="customBinding"   
         bindingConfiguration="CustomBinding\_CustomersService"
     4. contract="CustomersService.Proxies.CustomersService"   
         name="CustomBinding\_CustomersService" />
     5. **Note:** When you move a Silverlight project XAP file between development, test and production servers you'll need to update the address attribute in ServiceReferences.ClientConfig before compiling the project so that it points to the correct WCF service URI. Alternatively, a proxy object can programmatically be assigned the URI to use which can be useful in more dynamic environments.
  5. Open **Home.xaml** located in the **Views** folder of the **UsingWCFServices** project in the Visual Studio 2010 designer window and perform the following steps. The layout of the controls that you'll add on the user interface is shown next:
     1. 
     2. Figure
     3. Controls added to Home.xaml
     4. Change the **TextBlock** from **Home** to **Customer Orders** (you can do this through the Properties window by resetting the existing Text property value or by typing it directly in the XAML)
     5. Delete the **TextBlock** with a value of **Home page content**
     6. Add a **ComboBox** control under the **TextBlock** and give it a name of **CustomersComboBox**
     7. Drag a **DataGrid** control from the ToolBox and place it under the **ComboBox**. Give it a name of **OrdersDataGrid**
     8. Ensure that the **DataGrid** control's **AutoGenerateColumns** property is set to **False**
     9. Set the **DataGrid** control's **IsReadOnly** property to **True**
  6. Select the **DataGrid** in the designer and locate its **Columns** property in the Properties window. Click the ellipsis button to the right of the **Columns** property to open up the collection editor dialog.
  7. From dialog window's **Select item** drop-down list select **DataGridTextColumn** and click **Add**. Add a total of 4 columns as shown next:
     1. 
     2. Figure
     3. The Collection Editor Dialog
  8. Select each **DataGridTextColumn** and change its **Header** property in the Properties section of the dialog window to one of the following values (Order Date would be assigned to the first column and so on):

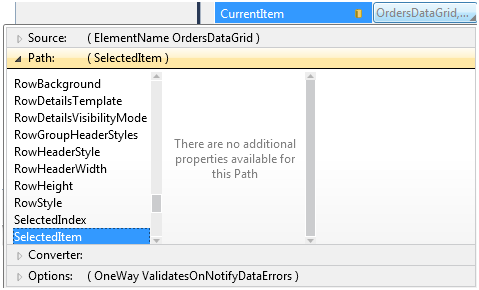
|  |
| --- |
| Header Property |
| Order Date |
| Ship Method |
| Sub Total |
| Total Due |

* 1. Switch to the XAML editor and locate all of the **DataGridTextColumns** elements that have been added.
  2. Add the appropriate **Binding** attribute shown below to each **DataGridTextColumn** element based upon the column's header text:

|  |  |
| --- | --- |
| Header Value | Binding to Add |
| Order Date | Binding="{Binding OrderDate,StringFormat=d}" |
| Ship Method | Binding="{Binding ShipMethod}" |
| Sub Total | Binding="{Binding SubTotal,StringFormat=C}" |
| Total Due | Binding="{Binding TotalDue,StringFormat=C}" |

* 1. Add the following template into the **ComboBox** control using the XAML editor:
     1. XAML
     2. <ComboBox.ItemTemplate>
     3. <DataTemplate>
     4. <StackPanel Orientation="Horizontal">
     5. <TextBlock Text="{Binding FirstName}" />
     6. <TextBlock Text="{Binding LastName}" Margin="5,0,0,0" />
     7. </StackPanel>
     8. </DataTemplate>
     9. </ComboBox.ItemTemplate>
  2. Handle the **ComboBox** control's **SelectionChanged** event by double-clicking on the control in the designer.
  3. Add the following code above the **Home** class's constructor to create an instance of the **CustomersServiceClient** proxy class that will be used to call the WCF service. You'll need to resolve the appropriate namespace.
     1. C#
     2. CustomersServiceClient \_Proxy = new CustomersServiceClient();
     3. Visual Basic
     4. Dim \_Proxy as New CustomersServiceClient()
  4. Locate the **OnNavigatedTo** event handler and add the following code to wire asynchronous calls to the WCF service to callback methods and call the **GetCustomersAync** method:
     1. C#
     2. \_Proxy.GetCustomersCompleted +=   
         (s, args) => CustomersComboBox.ItemsSource = args.Result;
     3. \_Proxy.GetOrdersByCustomerIDCompleted +=   
         (s, args) => OrdersDataGrid.ItemsSource = args.Result;
     4. \_Proxy.UpdateSalesOrderHeaderCompleted += (s,args) =>
     5. {
     6. //Check returned OperationStatus object for status
     7. string errorMsg = (args.Result.Status) ? "succeeded" : "failed";
     8. MessageBox.Show("Update " + errorMsg);
     9. };
     10. \_Proxy.GetCustomersAsync();
     11. Visual Basic
     12. AddHandler \_Proxy.GetCustomersCompleted, \_  
          Sub(s, args) CustomersComboBox.ItemsSource = args.Result
     13. AddHandler \_Proxy.GetOrdersByCustomerIDCompleted, \_  
          Sub(s, args) OrdersDataGrid.ItemsSource = args.Result
     14. AddHandler \_Proxy.UpdateSalesOrderHeaderCompleted, Sub(s,args)
     15. 'Check returned OperationStatus object for status
     16. Dim errorMsg As String = If(args.Result.Status, "succeeded", "failed")
     17. MessageBox.Show("Update " & errorMsg)
     18. End Sub
     19. \_Proxy.GetCustomersAsync()
     20. **Note:** This code hooks proxy object events to event handlers to handle asynchronous calls to the WCF service. It then calls the GetCustomersAsync method to initiate the call to the WCF service.
  5. Add code in the **ComboBox** control's **SelectionChanged** event handler to perform the following steps:
     1. Get the selected **CustomerID** from the **ComboBox**
     2. Call the WCF service's **GetOrdersByCustomerID** method using the **\_Proxy** object.
     3. C#
     4. int custID = ((Customer)CustomersComboBox.SelectedItem).CustomerID;
     5. \_Proxy.GetOrdersByCustomerIDAsync(custID);
     6. Visual Basic
     7. Dim custID As Integer = (CType(CustomersComboBox.SelectedItem, \_  
         Customer)).CustomerID
     8. \_Proxy.GetOrdersByCustomerIDAsync(custID)
  6. Run the project and test the Silverlight application in the browser. As a customer is selected one or more orders will show in the **DataGrid** control.
  7. Switch back to **Home.xaml** and drag a **DataForm** control from the ToolBox and place it under the **DataGrid**.
     1. **Note:** As mentioned at the beginning of this exercise, the DataForm control is part of the Silverlight Toolkit which will need to be installed to complete this exercise. The Silverlight Toolkit is available at http://silverlight.codeplex.com.
  8. Add the following attributes on the **DataForm** element by editing the XAML (or by using the Properties window).

|  |  |
| --- | --- |
| Attribute/Property | Value |
| Name | OrderDataForm |
| CurrentItem | {Binding Path=SelectedItem,ElementName=OrdersDataGrid} |
| AutoEdit | False |
| AutoCommit | False |
| CommandButtonsVisibility | Edit, Commit, Cancel |

* + 1. **Note:** An example of using the Properties window to modify the CurrentItem property is shown next (you can certainly type in the XAML as opposed to doing this visually). Binding a XAML element such as CurrentItem to another XAML element's property is referred to as *element to element binding*.
    2. 
    3. Figure
    4. Binding to SelectedItem
    5. **Note:** In this exercise the DataForm will be used to show all properties of the SalesOrderHeader object. In a real-life application you'd want to constrain the data shown by the DataForm control and eliminate any unnecessary fields that the user won't use.
  1. Add the appropriate XAML and code to handle the **DataForm's** **EditEnded** event.
     1. **Note:** If you need help with this step refer to the code in this lab's Completed folder. To handle the event you can type it into the XAML on the DataForm element and then navigate to the event handler or highlight the control in the designer, view the Properties window, click the lightning bolt icon at the top of the window and then double-click the event.
  2. Within the **EditEnded** event handler add the following code to push any changes made back to the WCF service:
     1. C#
     2. if (e.EditAction == DataFormEditAction.Commit)
     3. {
     4. var order = OrderDataForm.CurrentItem as SalesOrderHeader;
     5. \_Proxy.UpdateSalesOrderHeaderAsync(order);
     6. }
     7. Visual Basic
     8. If e.EditAction = DataFormEditAction.Commit Then
     9. Dim order = CType(OrderDataForm.CurrentItem, SalesOrderHeader)
     10. \_Proxy.UpdateSalesOrderHeaderAsync(order)
     11. End If
     12. **Note:** This code will be called when the user clicks the OK or Cancel buttons on the DataForm control while editing a SalesOrderHeader object.
  3. Run the project and perform the following tasks in the Silverlight application:
     1. Select a customer and then click on an order in the **DataGrid**
     2. Select the edit icon in the **DataForm** (click the pencil icon in the upper-right hand corner of the control) to switch to edit mode
     3. Change the value for the **Comment** field in the **DataForm** to any value you'd like and click the **OK** button. You will see a success message.
     4. Refresh the browser and navigate to the same customer order. Ensure that the comment you modified appears.
     5. **Note:** The code only allows the current item being edited in the DataForm control to be saved and doesn't allow multiple items to be saved as a batch. The next lab covering WCF RIA Services will show a built-in way to track object changes and submit a batch of changed objects back to the server.

Exercise 3: Implementing IEditableObject to Support Cancellation

* 1. While performing the previous tasks you may have noticed that the Cancel button in the DataForm was disabled. In this exercise you'll implement the IEditableObject interface on a partial SalesOrderHeader class to enable cancel functionality within the DataForm control. This partial class will add additional functionality to the existing SalesOrderHeader class created by the proxy generation wizard used earlier in the lab.

Implementing IEditableObject on a Partial Class

* 1. Add a new class into the **UsingWCFServices** project named **SalesOrderHeader** and add the **partial** keyword to the class's definition.
  2. Wrap the class in the following namespace.
     1. **Note:** The namespace matches the namespace of the **SalesOrderHeader** class created by the proxy generation wizard.
     2. C#
     3. UsingWCFServices.WebServiceProxies
     4. Visual Basic
     5. WebServiceProxies
  3. Implement the **IEditableObject** interface on the partial class and resolve the appropriate namespace.
  4. Add the following code into **SalesOrderHeader** to satisfy the interface:
     1. **Note:** The code that follows can be found in this lab's Completed folder if you'd prefer to cut and paste it into the class.
     2. C#
     3. SalesOrderHeader \_OriginalObject;
     4. bool \_Editing;
     5. public void BeginEdit()
     6. {
     7. if (!\_Editing)
     8. {
     9. \_Editing = true;
     10. \_OriginalObject = this.MemberwiseClone() as SalesOrderHeader;
     11. }
     12. }
     13. public void CancelEdit()
     14. {
     15. if (\_Editing)
     16. {
     17. Comment = \_OriginalObject.Comment;
     18. ShipDate = \_OriginalObject.ShipDate;
     19. \_Editing = false;
     20. }
     21. }
     22. public void EndEdit()
     23. {
     24. if (\_Editing)
     25. {
     26. \_Editing = false;
     27. \_OriginalObject = null;
     28. }
     29. }
     30. Visual Basic
     31. Private \_OriginalObject As SalesOrderHeader
     32. Private \_Editing As Boolean
     33. Public Sub BeginEdit() Implements IEditableObject.BeginEdit
     34. If Not \_Editing Then
     35. \_Editing = True
     36. \_OriginalObject = TryCast(Me.MemberwiseClone(), SalesOrderHeader)
     37. End If
     38. End Sub
     39. Public Sub CancelEdit() Implements IEditableObject.CancelEdit
     40. If \_Editing Then
     41. Comment = \_OriginalObject.Comment
     42. ShipDate = \_OriginalObject.ShipDate
     43. \_Editing = False
     44. End If
     45. End Sub
     46. Public Sub EndEdit() Implements IEditableObject.EndEdit
     47. If \_Editing Then
     48. \_Editing = False
     49. \_OriginalObject = Nothing
     50. End If
     51. End Sub
     52. **Note:** The code added into the CancelEdit method only resets the Comment and ShipDate properties if a user presses the cancel button in order to keep the code as short as possible. In a real-world application you would reset all of the properties on the object that the user can edit through the DataForm.
  5. Run the application, select a customer and change the comment value. The cancel button is now enabled and works properly along with the OK button.

Exercise 4 (Optional): Debugging WCF Service Calls

* 1. WCF services provide an excellent way to exchange data between a server and a Silverlight client. However, when using a Silverlight-Enabled WCF Service data is exchanged over the wire in a binary format making it difficult to debug when there's a problem with the messages being sent or received. In addition to issues associated with viewing binary data, how do you view messages sent over the wire in the first place to ensure they're being sent/received correctly? In this exercise you'll learn how to use a tool named Fiddler in conjunction with a WCF binary-encoded message inspector to debug Silverlight service calls.
  2. **Note:** WCF also provides built-in tracing capabilities that allow messages going into and out of a service to be logged and traced. Additional details about configuring WCF tracing can be found at http://msdn.microsoft.com/en-us/library/ms733025.aspx.

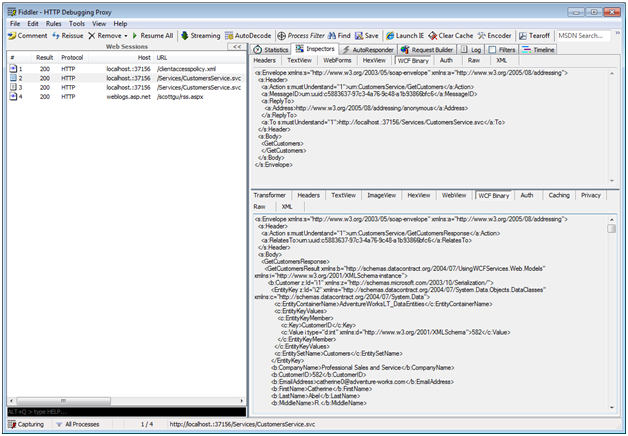
Debugging with Fiddler

* 1. Download and install **Fiddler** from **http://www.fiddler2.com/Fiddler2/version.asp**. Click on the **Install Fiddler2** link to download the executable.
     1. **Note:** Fiddler is a free HTTP proxy tool that can be used to view request and response messages.
  2. Once Fiddler is installed, download the **WCF Binary-encoded Message Inspector for Fiddler** located at the following URL:

http://code.msdn.microsoft.com/wcfbinaryinspector/Release/ProjectReleases.aspx?ReleaseId=4252.

* + 1. **Note:** The WCF Binary-encoded Message Inspector is an extension to Fiddler that automatically deserializes binary-encoded messages into SOAP so that you can view them while using Fiddler.
  1. Open the .zip file you downloaded and extract the **BinaryMessageFiddlerExtension.dll** file to the **[Program Files]\Fiddler2\Inspectors** folder to make it available in Fiddler.
  2. Perform the following tasks:
     1. Start Fiddler
     2. Run the **UsingWCFServices** solution in Visual Studio (press F5)
     3. Select a customer from the ComboBox control once the browser appears.
     4. Switch back to Fiddler and notice that no request or response messages appear. Fiddler doesn't detect request or response messages sent through localhost by default.
  3. Open the **ServiceReferences.ClientConfig** file within the **UsingWCFServices** project in the code editor.
  4. Add a **period** immediately after **localhost** in the **address** attribute to allow Fiddler to view localhost traffic. An example of adding a period immediately after localhost is shown next (note that your port will probably be different): http://localhost**.**:37156/Services/CustomersService.svc
  5. Save the **ServiceReferences.ClientConfig** file.
  6. Right-click on the **UsingWCFServices.Web** project, select **Add 🡪 Existing Item** and add the following file:

WCFServices/Starter Code/clientaccesspolicy.xml

* + 1. **Note:** Ensure the file is added into the root of the UsingWCFServices.Web project. Because the Silverlight application is served from localhost and you're now trying to call the WCF service using "localhost." (note the period after localhost), a cross-domain policy file is required for the service calls to work properly.
  1. Run the application again and then select a customer followed by an order.
  2. Perform the following tasks:
     1. Switch back to Fiddler and notice that requests to the WCF service now show
     2. Select the WCF Binary tab in the right-side of the Fiddler tool (it appears in the upper-right of the request message section and in the lower-right of the response message section) and notice that the request and response binary-encoded messages are converted to text and viewable:
     3. 
     4. Figure 9
     5. Fiddler tool
  3. By using Fiddler and the WCF binary-encoded message inspector you can view data sent between a Silverlight client and a server and more quickly debug problems.

Summary

* 1. In this lab you created an Entity Framework 4 model and a WCF service containing multiple service operations. You also called the service using a proxy object generated using Visual Studio and added custom code to handle the asynchronous callbacks and bind data to controls. Finally, you implemented the IEditableObject interface on a partial class to edit cancel support to the DataForm control. The specific tasks completed are shown next:
  + Create a new Silverlight Navigation Application
  + Create an Entity Framework 4 model
  + Create a Silverlight-Enabled WCF Service
  + Customize service methods
  + Create a Silverlight proxy object in Visual Studio
  + Build a user interface
  + Use a proxy object to call a WCF Service
  + Add IEditableObject functionality to a proxy generated class