* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Using the MVVM Pattern in Silverlight Applications

**Contents**

[Lab 10: Using the MVVM Pattern in Silverlight Applications 3](#_Toc276337280)

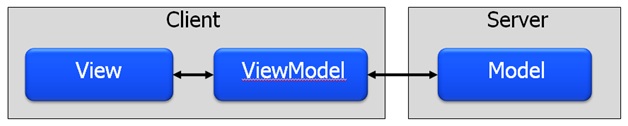
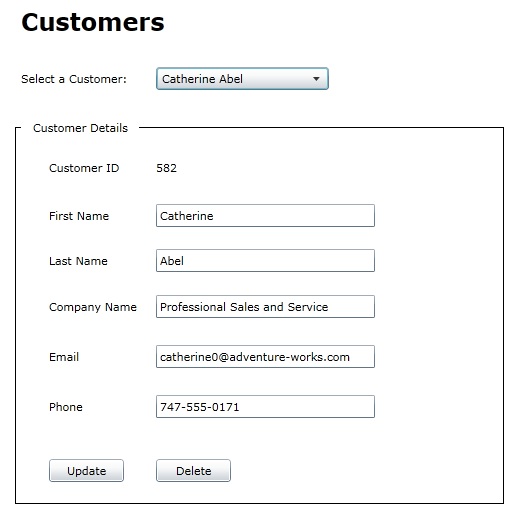
[Exercise 1: Creating a Service Agent Class 6](#_Toc276337281)

[Exercise 2: Creating a ViewModel Class 10](#_Toc276337282)

[Exercise 3: Handling Commands and Binding to a ViewModel 15](#_Toc276337283)

[Summary 19](#_Toc276337284)

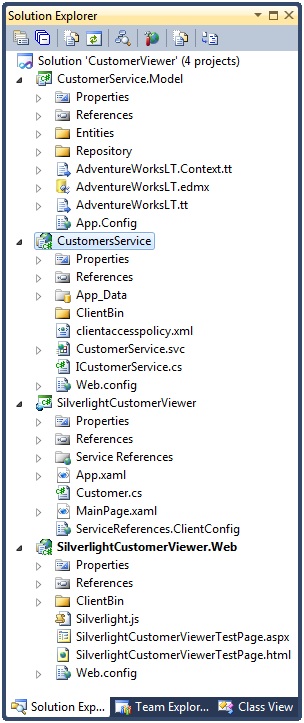
Lab 10: Using the MVVM Pattern in Silverlight Applications

* 1. The Model-View-ViewModel (MVVM) pattern provides a flexible way to build Silverlight applications that promotes code re-use, simplifies maintenance and supports testing. The pattern consists of three different components including the Model (entities and code on the server), the View (the Silverlight screens) and the ViewModel (the glue between the Model and the View). An example of the how the Model, View and ViewModel components relate to each other is shown next:
  2. 
  3. Figure
  4. The MVVM Pattern
  5. In this lab exercise you'll learn how to migrate an existing Silverlight application that uses code-behind files for all of the C# or VB code into a more structured architecture that follows the MVVM pattern. Topics covered include creating a service agent class to handle calls to a WCF service, creating a ViewModelBase class, creating a custom ViewModel class and binding a ViewModel to a View declaratively in XAML. You'll also see how commanding can be used to wire-up events in a View to methods in a ViewModel. The application that you'll work with in the lab exercises is shown next:
  6. 
  7. Figure
  8. Customer Details User Interface
  9. **Note:** It's recommended that you complete the data binding and WCF labs before starting this lab. Some of the steps in the lab will provide the code to use while others will explain the task to perform and let you figure out the code that should be added. If you need help with any of the steps refer to the lab's Completed folder which contains the finished code.
  10. **You Will Benefit from this Lab if**
  + You're interested in building Silverlight applications that take advantage of code re-use, promote testability and reduce maintenance costs
  + You'd like to separate your Silverlight application into distinct modules and provide a pattern that can be followed by a team of developers
  + You're interested in learning the MVVM pattern and the benefits it offers
  1. **You Will Learn:**
  + How to convert an existing Silverlight application to use the MVVM pattern
  + The role of the Model, ViewModel and View in the MVVM pattern
  + How to create a service agent class and the role it can play in an application to promote code re-use
  + How to create a ViewModel class and handle property change notifications
  + How to bind a ViewModel class to a View
  + How to communicate between a View and ViewModel using commanding
  1. **Business Requirements for the Silverlight application include:**
  + Remove existing code from a Silverlight code-beside file
  + Create a service agent interface and class to handle calls to a WCF service
  + Create a ViewModelBase class to handle INotifyPropertyChanged notifications
  + Create a ViewModel class that derives from ViewModelBase and handles data interactions
  + Add support for commanding into a ViewModel class
  + Bind a ViewModel class to a View in XAML
  + Bind Silverlight controls to ViewModel properties
  1. **Estimated Time: 45 minutes**

Exercise 1: Creating a Service Agent Class

* 1. In this exercise you'll remove existing code in a code-beside file and create a service agent class. The service agent class will be used to make calls to a WCF service from within a ViewModel class created later in the lab.
  2. Open the following Visual Studio 2010 solution file based upon your chosen language.

|  |  |
| --- | --- |
| Language | Lab Files Location |
| C# | /MVVMPattern/Starting Point/C#/CustomerViewer.sln |
| Visual Basic | /MVVMPattern/Starting Point/VB/CustomerViewer.sln |

* 1. The following projects are available in the solution:
  + **CustomerService.Model** – Contains entities and data repository classes used to access an AdventureWorks LT database.
  + **CustomersService** – A WCF service application that exposes entities to various applications.
  + **SilverlightCustomerViewer** – A Silverlight project that consumes data from a WCF service.
  + **SilverlightCustomerViewer.Web** – A website project used to host the SilverlightCustomerViewer application.  
    1. 
    2. Figure
    3. Projects in the Solution
  1. Right-click **CustomerService.svc** in the **CustomersService** project and select **View in Browser** from the menu. This will start a local WCF server and show details about the service.
  2. Back in Visual Studio, right-click **SilverlightCustomerViewerTestPage.html** in the **SilverlightCustomerViewer.Web** project and select **View in Browser** from the menu. Select customers from the ComboBox control (note that you may have to wait a moment for the customers to load the first time the application is run) and notice that the data for each customer displays in the details area.
  3. Back in Visual Studio, open **MainPage.xaml.cs** located in the **SilverlightCustomerViewer** project in the code editor and remove all of the existing code within the **MainPage** class except the constructor and the call to **InitializeComponent**.
     1. **Note:** All of the functionality in the code-beside file will be moved into a ViewModel class that you'll create later in this lab.
  4. Open **MainPage.xaml** and remove the **Click** event handlers defined on the two **Button** controls.
  5. Right-click on the **SilverlightCustomerViewer** project and select **Add 🡪 New Folder**. Give the folder a name of **ServiceAgents**.
  6. Add a new class into the **ServiceAgents** folder named **CustomersServiceAgent**.
  7. Add the following interface immediately above the class and resolve any missing namespaces:
     1. **Note:** Resolve any missing namespaces by right-clicking on an unknown type and then selecting Resolve in the menu.
     2. C#
     3. public interface ICustomersServiceAgent
     4. {
     5. void GetCustomers(EventHandler<GetCustomersCompletedEventArgs> callback);
     6. void SaveCustomer(Customer cust,
     7. EventHandler<SaveCustomerCompletedEventArgs> callback);
     8. }
  8. Visual Basic
     1. Public Interface ICustomersServiceAgent  
         Sub GetCustomers(ByVal callback As \_  
         EventHandler(Of GetCustomersCompletedEventArgs))  
         Sub SaveCustomer(ByVal cust As Customer, \_  
         ByVal callback As EventHandler(Of SaveCustomerCompletedEventArgs))  
        End Interface
  9. Implement the **ICustomerService** interface on the **CustomersServiceAgent** class.
  10. Add the following field into the **CustomersServiceAgent** class:
      1. C#
      2. CustomerServiceClient \_Proxy = new CustomerServiceClient();
      3. Visual Basic
      4. Dim \_Proxy as New CustomerServiceClient()
      5. **Note**: The CustomerServiceClient class is a WCF service proxy class that will be used to communicate with the remote service to retrieve customer data and perform update and delete operations.
  11. Add the following code in the **GetCustomers** method to handle calling the WCF service operation:
      1. C#
      2. \_Proxy.GetCustomersCompleted += callback;
      3. \_Proxy.GetCustomersAsync();
      4. Visual Basic
      5. AddHandler \_Proxy.GetCustomersCompleted, callback  
         \_Proxy.GetCustomersAsync()
  12. Add the following code in the **SaveCustomer** method to handle communicating delete and update data operations to the WCF service:
      1. C#
      2. \_Proxy.SaveCustomerCompleted += callback;
      3. \_Proxy.SaveCustomerAsync(cust);
      4. Visual Basic
      5. AddHandler \_Proxy.SaveCustomerCompleted, callback
      6. \_Proxy.SaveCustomerAsync(cust)

Exercise 2: Creating a ViewModel Class

* 1. In this exercise you'll create a ViewModelBase class that provides core functionality that can be used by one or more ViewModel classes. You'll then derive from ViewModelBase and create a ViewModel class that will be used to retrieve and manipulate customer data used in the MainPage.xaml View.
  2. Locate the file named **mvvmInpcPropertyCS.snippet** (for C#) or **mvvmInpcPropertyVB.snippet** (for VB) in the **SilverlightCustomerViewer/CodeSnippets** folder.
  3. Copy the file for your chosen language to the following location (or use **Tools 🡪 Code Snippets Manager** in Visual Studio to import it):

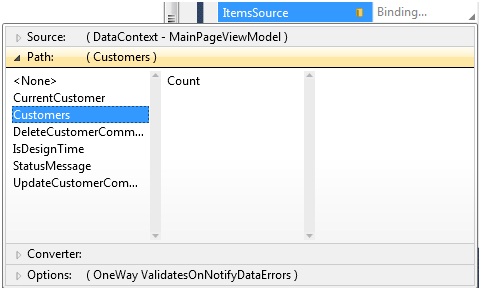
|  |  |
| --- | --- |
| Language | Lab Files Location |
| C# | Documents\Visual Studio 2010\Code Snippets\Visual C#\My Code Snippets |
| Visual Basic | Documents\Visual Studio 2010\Code Snippets\Visual Basic\My Code Snippets |

* + 1. **Note:** The mvvmInpc snippet will be used to create properties within a ViewModel class later in this lab. It's one of several snippets available in MVVM Light (http://mvvmlight.codeplex.com) and has been modified for use in this lab exercise.
  1. Right-click on the **SilverlightCustomerViewer** project and select **Add 🡪 New Folder**. Give the folder a name of **ViewModels**.
  2. Add a new class into the **ViewModels** folder named **ViewModelBase** and mark it as **abstract** (C#) or **MustInherit** (VB).
  3. Implement the **INotifyPropertyChanged** interface on the class and resolve the namespace.
     1. **Note:** INotifyPropertyChanged is an important interface in Silverlight used by the data binding engine to notify controls and other objects when a bound property value changes. By implementing INotifyPropertyChanged on the ViewModelBase class you can write the code once and re-use it across multiple ViewModel classes. The interface is located in the System.ComponentModel namespace.
  4. Once the interface has been implemented on the **ViewModelBase** class, add a method named **OnPropertyChanged** into **ViewModelBase** to handle raising the **PropertyChanged** event from the **INotifyPropertyChanged** interface.
     1. **Note:** If you need help with this step please refer to the ViewModelBase class in the lab's Completed folder. Creating an OnPropertyChanged method was covered in the data binding lab.
  5. Add a new **public Boolean** property into the **ViewModelBase** class named **IsDesignTime**.
  6. Add a **get** block (but no set block) into the property that returns **DesignerProperties.IsInDesignTool** as the value. Resolve any namespaces as needed.
     1. **Note:** When using the MVVM pattern and ViewModel classes it's important to know when code is executing in a design-time tool such as Visual Studio or Expression Blend and when it's executing at runtime. When code runs in a design tool network calls will not execute properly and can error out the designer. The IsDesignTime property will be used to detect where code is running to ensure that ViewModel classes execute properly at design-time.
  7. Add a new class into the **ViewModels** folder named **MainPageViewModel** that derives from **ViewModelBase** .
  8. Add the following properties into **MainPageViewModel** using the mvvmInpc code snippet and resolve any missing namespaces as necessary (type mvvmInpc + tab + tab to use the snippet in C# or mvvminpc + tab to use the snippet in VB):

|  |  |
| --- | --- |
| Property Name | Property Type |
| Customers | ObservableCollection of Customer |
| CurrentCustomer | Customer |
| StatusMessage | string |

* + 1. **Note:** If you need help with creating the properties in this step refer to the MainPageViewModel class in the lab's Completed folder.
  1. Within the **CurrentCustomer** property's **set** block add code to set the **StatusMessage** property to empty strings (add the code within the existing "if" statement).
  2. Add another property named **ServiceAgent** of type **ICustomersServiceAgent** and resolve the namespace (create it as a standard .NET property).
  3. Add the following methods into **MainPageViewModel** to handle retrieving, updating and deleting customer objects:
     1. **Note:** The ObjectState parameter used in the following code is automatically created by the WCF proxy generator when using self-tracking entities in Entity Framework 4. By changing the object state we can easily track whether delete or update operations should occur for a given Customer object.
     2. C#
     3. private void GetCustomers()
     4. {
     5. ServiceAgent.GetCustomers((s, e) => Customers = e.Result);
     6. }
     7. public void UpdateCustomer()
     8. {
     9. SaveCustomer(ObjectState.Modified);
     10. }
     11. public void DeleteCustomer()
     12. {
     13. SaveCustomer(ObjectState.Deleted);
     14. Customers.Remove(CurrentCustomer);
     15. CurrentCustomer = null;
     16. }
     17. private void SaveCustomer(ObjectState state)
     18. {
     19. CurrentCustomer.ChangeTracker.State = state;
     20. ServiceAgent.SaveCustomer(CurrentCustomer, (s, e) =>
     21. {
     22. StatusMessage = (e.Result.Status) ? "Success!" :   
          "Unable to complete operation";
     23. });
     24. }
     25. Visual Basic
     26. Private Sub GetCustomers()  
          ServiceAgent.GetCustomers(Sub(s, e) Customers = e.Result)  
         End Sub  
            
         Public Sub UpdateCustomer()  
          SaveCustomer(ObjectState.Modified)  
         End Sub  
            
         Public Sub DeleteCustomer()  
          SaveCustomer(ObjectState.Deleted)  
          Customers.Remove(CurrentCustomer)  
          CurrentCustomer = Nothing  
         End Sub  
            
         Private Sub SaveCustomer(ByVal state As ObjectState)  
          CurrentCustomer.ChangeTracker.State = state  
          ServiceAgent.SaveCustomer(CurrentCustomer, \_  
          Sub(s, e) StatusMessage = \_  
              If(e.Result.Status, "Success!", "Unable to complete operation"))  
         End Sub
  4. Add the following constructors into **MainPageViewModel**:
     1. C#
     2. public MainPageViewModel() : this(new CustomersServiceAgent())  
        {  
        }  
           
        public MainPageViewModel(ICustomersServiceAgent serviceAgent)  
        {  
            if (!IsDesignTime)  
            {  
                if (serviceAgent != null) ServiceAgent = serviceAgent;  
                GetCustomers();  
          
            }  
        }
     3. Visual Basic
     4. Public Sub New()  
         Me.New(New CustomersServiceAgent())  
        End Sub  
           
        Public Sub New(ByVal serviceAgent As ICustomersServiceAgent)  
         If Not IsDesignTime Then  
         If serviceAgent IsNot Nothing Then  
         Me.ServiceAgent = serviceAgent  
         End If  
         GetCustomers()  
          
         End If  
        End Sub
     5. Note: The first constructor will be called if the ViewModel is instantiated without any parameters. It passes a new instance of the CustomersServiceAgent object to the second constructor which assigns it to the ServiceAgent property when not in design mode (note that instead of hard coding the service agent type it could be injected using a dependency injection framework). The second constructor accepts any object that implements the ICustomersServiceAgent interface allowing mock objects to be passed in for the service agent when testing the ViewModel class. Following this pattern provides a flexible way to work with different types of service agent objects when the application is running as well as when tests need to be executed.
  5. Ensure that the **SilverlightCustomerViewer** project compiles before proceeding to the next exercise.

Exercise 3: Handling Commands and Binding to a ViewModel

* 1. In this exercise you'll add commanding support into the SilverlightCustomerViewer project. Commanding allows events triggered in a View such as a Button's Click event to be routed directly to a ViewModel instance without having to add code in a code-beside file. You'll be introduced to the ICommand interface as well as a RelayCommand class that will be used within MainPageViewModel to handle commands. Finally, you'll bind MainPageViewModel to a View using a declarative binding syntax.
  2. Open the **Commanding/RelayCommand.cs** class in the **SilverlightCustomerViewer** project and note that it implements **ICommand**. This interface is required in order for a Silverlight application to support commanding.
     1. **Note:** Commanding is the process of forwarding events that occur in the user interface to a ViewModel object for processing at runtime. The RelayCommand class will be used to wire properties in a ViewModel to methods that are invoked when any control derived from ButtonBase is clicked in a View. It satisfies the ICommand interface allowing commanding to be used in Silverlight MVVM applications.
  3. Add the following code into **MainPageViewModel** to create properties capable of binding to controls that expose a Command property. Resolve the required namespace.
     1. C#
     2. public RelayCommand UpdateCustomerCommand  
        {  
            get;  
            private set;  
        }  
           
        public RelayCommand DeleteCustomerCommand  
        {  
            get;  
            private set;  
        }
     3. Visual Basic
     4. Private \_UpdateCustomerCommand As RelayCommand
     5. Public Property UpdateCustomerCommand() As RelayCommand
     6. Get
     7. Return \_UpdateCustomerCommand
     8. End Get
     9. Private Set(ByVal value As RelayCommand)
     10. \_UpdateCustomerCommand = value
     11. End Set
     12. End Property
     13. Private \_DeleteCustomerCommand As RelayCommand
     14. Public Property DeleteCustomerCommand() As RelayCommand
     15. Get
     16. Return \_DeleteCustomerCommand
     17. End Get
     18. Private Set(ByVal value As RelayCommand)
     19. \_DeleteCustomerCommand = value
     20. End Set
     21. End Property
  4. Add the following code into **MainPageViewModel** to handle wiring up the properties defined in the previous step to methods that will be invoked once a command is executed:
     1. C#
     2. private void WireCommands()  
        {  
            UpdateCustomerCommand = new RelayCommand(UpdateCustomer);  
            DeleteCustomerCommand = new RelayCommand(DeleteCustomer);  
        }
     3. Visual Basic
     4. Private Sub WireCommands()
     5. UpdateCustomerCommand = New RelayCommand(AddressOf UpdateCustomer)
     6. DeleteCustomerCommand = New RelayCommand(AddressOf DeleteCustomer)  
        End Sub
     7. **Note:** WireCommands handles associating two RelayCommand properties with methods that will be invoked as a command is executed in a View. For example, any Button with a Command property bound to UpdateCustomerCommand will cause the UpdateCustomer method to be invoked.
  5. Add a call to the **WireCommands** method within **MainPageViewModel's** second constructor (add the code immediately under the GetCustomers method call within the constructor).
  6. Add code within the **CurrentCustomer** property's **set** block to assign the **IsEnabled** property of **UpdateCustomerCommand** and **DeleteCustomerCommand** to a value of **true** (add the code within the existing "if" statement).
  7. Open **MainPage.xaml** and add the following xml namespace definition on the **UserControl** element to reference the **ViewModels** namespace:
     1. XAML
     2. xmlns:viewModels="clr-namespace:SilverlightCustomerViewer.ViewModels"
  8. Create a **UserControl.Resources** element within the XAML immediately below the beginning **UserControl** element and add the following code into it:
     1. XAML
     2. <viewModels:MainPageViewModel x:Key="ViewModel" />
  9. Bind the **ViewModel** key created in the previous step to the **LayoutRoot** element's **DataContext** property using a **StaticResource** binding. This will bind a MainPageViewModel object instance to the DataContext.
     1. XAML
     2. DataContext="{Binding Source={StaticResource ViewModel}}"
  10. Build the solution. If you encounter any compilation errors resolve them before continuing.
  11. Right-click on the **ComboBox** control in the designer and select **Properties** from the menu.
  12. Locate the **ItemsSource** property and click on its **Binding** value to view the data binding window. Select **Customers** from the available **Path** values as shown next:
      1. 
      2. Figure
      3. **Binding Data to the ItemsSource Property**
  13. Follow the same procedure shown in the previous step to bind the **ComboBox** control's **SelectedItem** property to **CurrentCustomer**.
      1. **Note:** You'll need to click on the icon to the right of SelectedItem and then select Apply Data Binding to access the data binding window.
  14. Bind the **Text** property of each **TextBox** control to the associated property in **CurrentCustomer**.
      1. **Note:** To do this, first clear any existing bindings by selecting the data icon to the right of the Text property and select Reset Value. You'll need to bind the Text property of each TextBox to CurrentCustomer and then select the appropriate property from CurrentCustomer (FirstName, LastName, etc.) in the data binding window.
  15. Bind the **Command** property of each **Button** control to the appropriate command property in the ViewModel (UpdateCustomerCommand or DeleteCustomerCommand) using the same technique shown in the previous steps.
  16. Drag a **TextBlock** control from the Toolbox and drop it directly to the right of the existing **Delete** button. Change the **Width** of the **TextBlock** to **200**.
  17. Bind the newly added **TextBlock** control's **Text** property to the **StatusMessage** property of the ViewModel using the data binding window.
  18. Run the application and test the functionality. As you update or delete a Customer object a success message should appear to the right of the Button controls. Select a different customer and the success message should disappear.

Summary

* 1. In this exercise you converted the SilverlightCustomerViewer application to use the Model-View-ViewModel pattern. Tasks completed include creating a service agent class for communication with a WCF service as well as a ViewModel class that is responsible for retrieving data and storing it in properties. You also added support for commanding and bound a ViewModel class to a XAML view. Requirements satisfied in this lab include:
  + Remove code located in a Silverlight code-beside file
  + Create a service agent interface and class to handle calls to a WCF service
  + Create a ViewModelBase class to handle INotifyPropertyChanged notifications
  + Create a ViewModel class that derives from ViewModelBase and handles data interactions
  + Add support for commanding into a ViewModel class
  + Bind a ViewModel class to a View in XAML
  + Bind Silverlight controls to ViewModel properties