Basics of MVVM in WPF

In this blog post, you learn how easy it is to use the Model-View-View-Model (MVVM) design pattern in WPF applications. This blog post is a step-by-step illustration of how to build a WPF application to display a list of users. You are going to perform the following steps.

- Create some base classes needed for every WPF application
- Create a local SQL Server express database and add a User table
- Load users directly using code-behind on a WPF user control
- Load users on a WPF user control using a view model class

Create WPF Application

To start, write code to load a list box control in WPF without using the MVVM design pattern. Create a new WPF project named **WPF.Sample**.

Create CommonBase Class

Right mouse-click on your WPF project and select Add | New Folder… Name the folder **Common**.

Add a new class file within this folder called **CommonBase.cs**. Replace the code within this file with the following code.

```
using System.ComponentModel;
using System.Reflection;

namespace Common.Library
{
  /// <summary>
  /// This class implements the INotifyPropertyChanged event
  /// </summary>
  public class CommonBase : INotifyPropertyChanged
  {
    /// <summary>
    /// The PropertyChanged Event to raise to any UI object
    /// </summary>
    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// The PropertyChanged Event to raise to any UI object
    /// The event is only invoked if data binding is used
    /// </summary>
    /// <param name="propertyName">The property name
                that is changing</param>
    protected void RaisePropertyChanged(string propertyName)
    {
      // Grab a handler
      PropertyChangedEventHandler handler = this.PropertyChanged;

      // Only raise event if handler is connected
      if (handler != null) {
        PropertyChangedEventArgs args =
          new PropertyChangedEventArgs(propertyName);

        // Raise the PropertyChanged event.
        handler(this, args);
      }
    }
  }
}
```

Create ViewModelBase Class

Right mouse-click on the **Common** folder again and add another new class file named **ViewModelBase.cs**. Replace all the code within this file with the following code.

```
namespace Common.Library
{
  /// <summary>
  /// Base class for all view models
  /// </summary>
  public class ViewModelBase : CommonBase
  {
  }
}
```

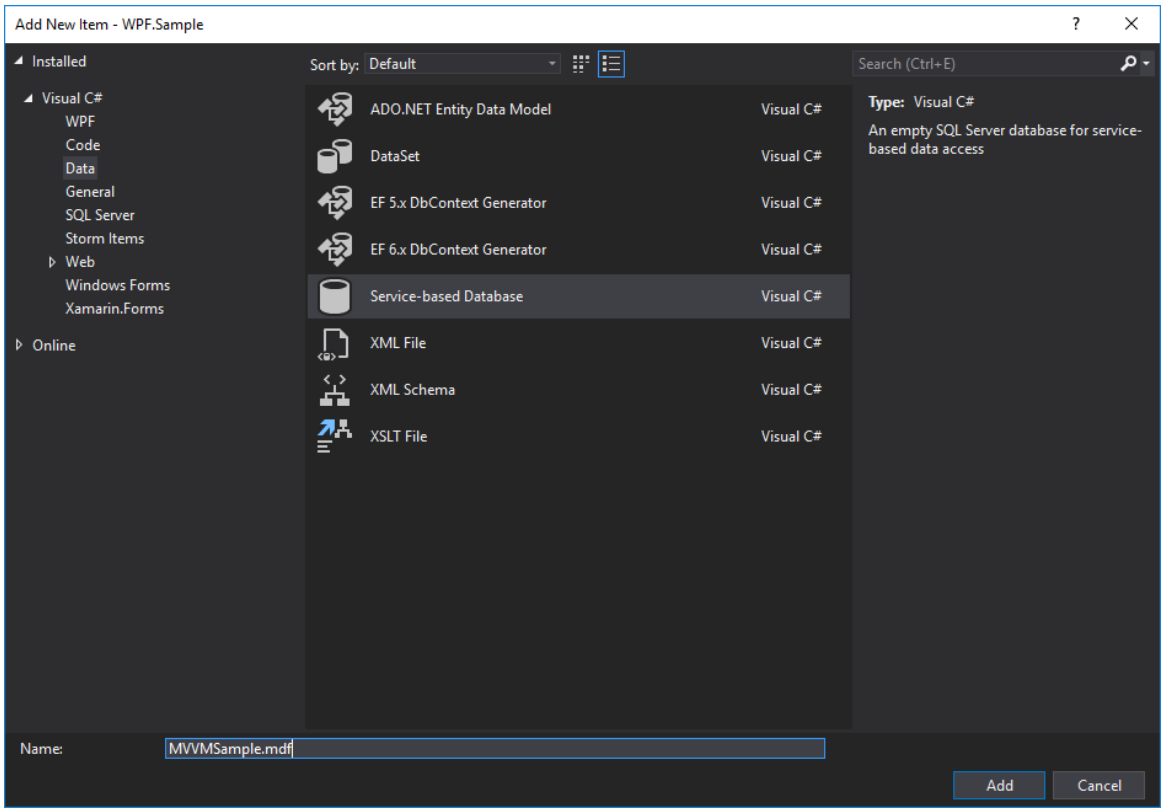
Create Local Database

You can create your own SQL Server database using Visual Studio.

Add a folder to your project named **App_Data**.

Right mouse-click on the App_Data folder and select Add | New Item…

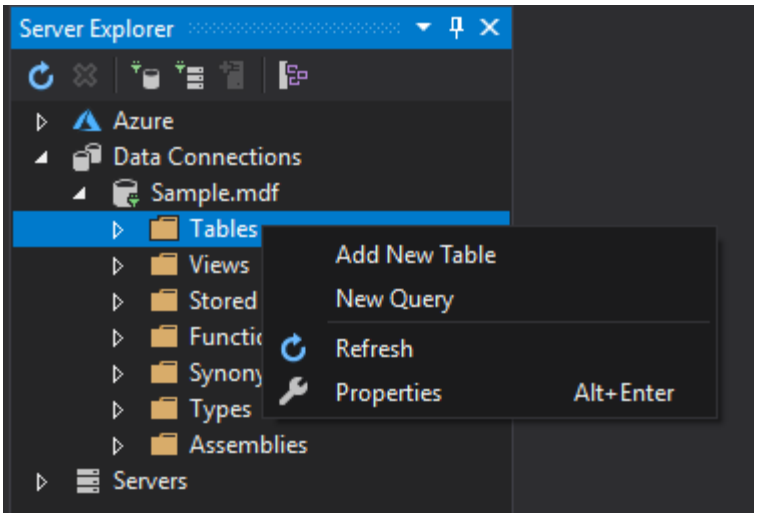
Select **Data | Service-based Database**



Set the name to **MVVMSample.mdf** and click the Add button.

Double-click on the MVVMSample.mdf file that is created and the Server Explorer window will open up.

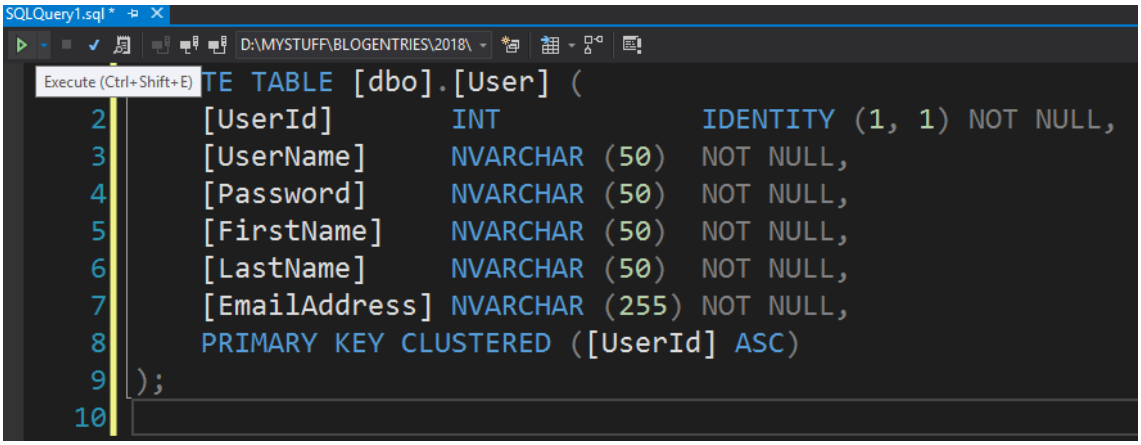
Right mouse-click on the Tables folder and select New Query



Put the following code into the query window.

```
CREATE TABLE [dbo].[User] (
    [UserId]       INT             IDENTITY (1, 1) NOT NULL,
    [UserName]     NVARCHAR (50)   NOT NULL,
    [Password]     NVARCHAR (50)   NOT NULL,
    [FirstName]    NVARCHAR (50)   NOT NULL,
    [LastName]     NVARCHAR (50)   NOT NULL,
    [EmailAddress] NVARCHAR (255) NOT NULL,
    PRIMARY KEY CLUSTERED ([UserId] ASC)
);
```

Click the Execute button to add this table to the database



Expand the Tables folder to view the new table.

Right mouse-click on Tables folder and select Refresh.

Right mouse-click on the table and select Show Table Data from the menu.

Enter the following data into this table.

NOTE: You don't fill in the UserId. That is created automatically for you.

Add User Data

UserId	UserName	Password	FirstName	LastName	EmailAddress
1	PShaffer	P@ssw0rd	Paul	Shaffer	PShaffer@netinc.com
2	JSmith	P@ssw0rd	John	Smith	JSmith@netinc.com
3	BJones	P@ssw0rd	Bruce	Jones	BJones@netinc.com

Add Entity Framework Classes

Right mouse-click on the **WPF.Sample** project and select **Manage NuGet Packages…**

Click the Browse tab and locate the **EntityFramework** by Microsoft.

Install that package into your WPF project.

Create Folders

Add a new folder called **EntityClasses**.

Add a new folder called **Models**.

Create a User Entity Class

Right mouse-click on the **EntityClasses** folder and create a class called **User.cs**.

Add the following code within this file.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Common.Library;

namespace WPF.Sample
{
  [Table("User", Schema = "dbo")]
  public class User : CommonBase
  {
    private int _UserId;
    private string _UserName = string.Empty;
    private string _Password = string.Empty;
    private string _FirstName = string.Empty;
    private string _LastName = string.Empty;
    private string _EmailAddress = string.Empty;
    private bool _IsLoggedIn = false;

    [Required]
    [Key]
    public int UserId
    {
      get { return _UserId; }
      set {
        _UserId = value;
        RaisePropertyChanged("UserId");
      }
    }

    [Required]
    public string UserName
    {
      get { return _UserName; }
      set {
        _UserName = value;
        RaisePropertyChanged("UserName");
      }
    }

    [Required]
    public string Password
    {
      get { return _Password; }
      set {
        _Password = value;
        RaisePropertyChanged("Password");
      }
    }

    [Required]
    public string FirstName
    {
      get { return _FirstName; }
      set {
        _FirstName = value;
        RaisePropertyChanged("FirstName");
      }
```

```
    }

    [Required]
    public string LastName
    {
      get { return _LastName; }
      set {
        _LastName = value;
        RaisePropertyChanged("LastName");
      }
    }

    [Required]
    public string EmailAddress
    {
      get { return _EmailAddress; }
      set {
        _EmailAddress = value;
        RaisePropertyChanged("EmailAddress");
      }
    }

    [NotMapped]
    public bool IsLoggedIn
    {
      get { return _IsLoggedIn; }
      set {
        _IsLoggedIn = value;
        RaisePropertyChanged("IsLoggedIn");
      }
    }
  }
}
```

Create a DbContext Class

Right mouse-click on the WPF.Sample project and add a new folder named **Models**.

Right mouse-click on the **Models** folder and create a new class called **SampleDbContext.cs**.

Add the following code into this file.

```
using System.Data.Entity;

namespace WPF.Sample
{
  public partial class SampleDbContext : DbContext
  {
    public SampleDbContext() : base("name=MVVMSample")
    {
    }

    public virtual DbSet<User> Users { get; set; }
  }
}
```

Add a Connection String

To use the data layer from WPF.Sample project, you need to add a connection string and tell the Entity Framework where to find the database you created.

Open the App.config file in the WPF.Sample project.

Add the following section.

```
<connectionStrings>
  <add name="MVVMSample"
       connectionString="Server=(localdb)\MSSQLLocalDB;
               AttachDbFilename=|DataDirectory|MVVMSample.mdf;
               Database=MVVMSample;Trusted_Connection=Yes;"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

Set DataDirectory

If you are using a local SQL Server express database you created within the App_Data folder of your project, you need to set the "DataDirectory" domain property to the path of where the database is located. You do this one time when your WPF application starts up. It is this domain property that is used to replace the |DataDirectory| placeholder you see in the connection string in the App.config file.

Open the App.xaml.cs file and override the OnStartup event within the App class.

```
protected override void OnStartup(StartupEventArgs e)
{
  base.OnStartup(e);

  // Set the DataDirectory for Entity Framework
  string path = Environment.CurrentDirectory;
  path = path.Replace(@"\bin\Debug", "");
  path += @"\App_Data\";

  AppDomain.CurrentDomain.SetData("DataDirectory", path);
}
```

Add a User List User Control

Create a new folder named **UserControls**.

Right mouse-click on the UserControls folder and add a user control named **UserListControl** to this project. Modify this user control to look like the following:

```
<UserControl x:Class="WPF.Sample.UserControls.UserListControl"
             ...  XML namespaces here
             mc:Ignorable="d"
             d:DesignHeight="450"
             d:DesignWidth="800"
             Loaded="UserControl_Loaded">
  <ListView Name="lstData"
            ItemsSource="{Binding}">
    <ListView.View>
      <GridView>
        <GridViewColumn Header="User ID"
           Width="Auto"
           DisplayMemberBinding="{Binding Path=UserId}" />
        <GridViewColumn Header="User Name"
           Width="Auto"
           DisplayMemberBinding="{Binding Path=UserName}" />
        <GridViewColumn Header="First Name"
           Width="Auto"
           DisplayMemberBinding="{Binding Path=FirstName}" />
        <GridViewColumn Header="Last Name"
           Width="Auto"
           DisplayMemberBinding="{Binding Path=LastName}" />
        <GridViewColumn Header="Email"
           Width="Auto"
           DisplayMemberBinding="{Binding Path=EmailAddress}" />
      </GridView>
    </ListView.View>
  </ListView>
</UserControl>
```

Be sure to create the UserControl_Loaded event.

Add Code to Load Users

In the code behind the UserListControl.xaml.cs file, add a using statement.

```
using System.Linq;
```

Modify the UserControl_Loaded event procedure to call a method named LoadUsers(). Create the LoadUsers() method immediately after the UserControl_Loaded event.

```
private void UserControl_Loaded(object sender,
  System.Windows.RoutedEventArgs e)
{
  LoadUsers();
}

private void LoadUsers()
{
  SampleDbContext db = null;

  try {
    db = new SampleDbContext();

    lstData.DataContext = db.Users.ToList();
  }
  catch (Exception ex) {
    System.Diagnostics.Debug.WriteLine(ex.ToString());
  }
}
```

The code in the LoadUsers() method uses the Entity Framework to retrieve user data from the User table. Apply the ToList() method in order to retrieve a local copy of the data that is bindable to any WPF control. Assign that list of users to the DataContext property of the ListView control you named lstData.

On the ListView control, you set the ItemsSource property to {Binding}. This tells the list box that you will be binding the data at runtime by setting the DataContext property. Once you set the DataContext property, WPF binds each row of data to the template you created in the ListView.

Modify the Main Window

On the MainWindow.xaml, add the following within the <Grid> element.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Menu Grid.Row="0"
        IsMainMenu="True">
    <MenuItem Header="_File">
      <MenuItem Header="E_xit"
                Click="MenuExit_Click" />
    </MenuItem>
    <MenuItem Header="Users"
              Click="MenuUsers_Click" />
  </Menu>
  <!-- Content Area -->
  <Grid Grid.Row="1"
        Margin="10"
        HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Name="contentArea" />
</Grid>
```

Open the **MainWindow.xaml.cs** file.

Add a using statement.

```
using WPF.Sample.UserControls;
```

Create the MenuUserList_Click event procedure. Add the following code:

```
private void MenuExit_Click(object sender, RoutedEventArgs e)
{
  this.Close();
}

private void MenuUsers_Click(object sender, RoutedEventArgs e)
{
  contentArea.Children.Add(new UserListControl());
}
```

Try it Out

Run the application and view the users.

The Problem

That is all there is to writing the code behind and having it load data from a User table. The problem with the above code is that to test this code, someone must run the program and verify that this code works as it is supposed to. If you make a change to the database or to the UI, you will then need to have someone run the application again to ensure it still works. You must repeat this test each time a change is made. This becomes very tedious and time consuming for the developer and the tester.

Use MVVM to Load Users

Now that you have users loading into a list view control using code behind, let's now move this code into a view model class and use data binding. Right mouse-click on the project and add a new folder named **ViewModels**. Right mouse-click on the ViewModels folder and add a new class named **UserListViewModel.cs**. You are going to add a property to this view model class that is an ObservableCollection of User objects. Add the code below to this new file you created.

```
using System;
using System.Collections.ObjectModel;
using Common.Library;

namespace WPF.Sample.ViewModels
{
  public class UserListViewModel : ViewModelBase
  {
    public UserListViewModel() : base()
    {
      LoadUsers();
    }

    private ObservableCollection<User> _Users =
       new ObservableCollection<User>();

    public ObservableCollection<User> Users
    {
      get { return _Users; }
      set {
        _Users = value;
        RaisePropertyChanged("Users");
      }
    }

    Public virtual void LoadUsers()
    {
      SampleDbContext db = null;

      try {
        db = new SampleDbContext();

        Users = new ObservableCollection<User>(db.Users);
      }
      catch (Exception ex) {
        System.Diagnostics.Debug.WriteLine(ex.ToString());
      }
    }
  }
}
```

The code in the UserListViewModel class is not that much more code than you wrote in the code behind.

Bind UserListViewModel Class to XAML

Once you have the UserListViewModel class created, bind this class to your user control. Any WPF user control (or Window) may create an instance of any class in your application within XAML. To do this, you need to do two things.

1. Create an XML namespace to reference a namespace in your C# code
2. Create an instance of the class within that namespace and assign a key value to it

Open the UserListControl.xaml file and add an XML namespace (See Figure 1) as shown in the code in bold to reference the namespace that the UserListViewModel is created within.

```
<UserControl x:Class="WPF.Sample.UserControls.UserListControl"
       xmlns="http://schemas.microsoft.com/..."
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:mc="http://schemas.openxmlformats.org/..."
       xmlns:d="http://schemas.microsoft.com/..."
       xmlns:vm="clr-namespace:WPF.Sample.ViewModels"
     ...
```

The XML namespace defined here is the equivalent of aliasing a .NET namespace in C#.

```
using vm = WPF.Sample.ViewModels;
```

Once you have a .NET namespace aliased, you can now create any instance of a class in that name using code like the following:

```
vm.UserListViewModel viewModel = new vm.UserListViewModel();
```

This type of aliasing is not done too often in .NET because you generally just provide a using statement and you can create an instance of your class. However, if you have two classes with the same name in different namespaces, you would have to fully qualify one or the other to use them in the same block of code. Instead of typing out the complete namespace for one of them, using an alias can save some keystrokes.

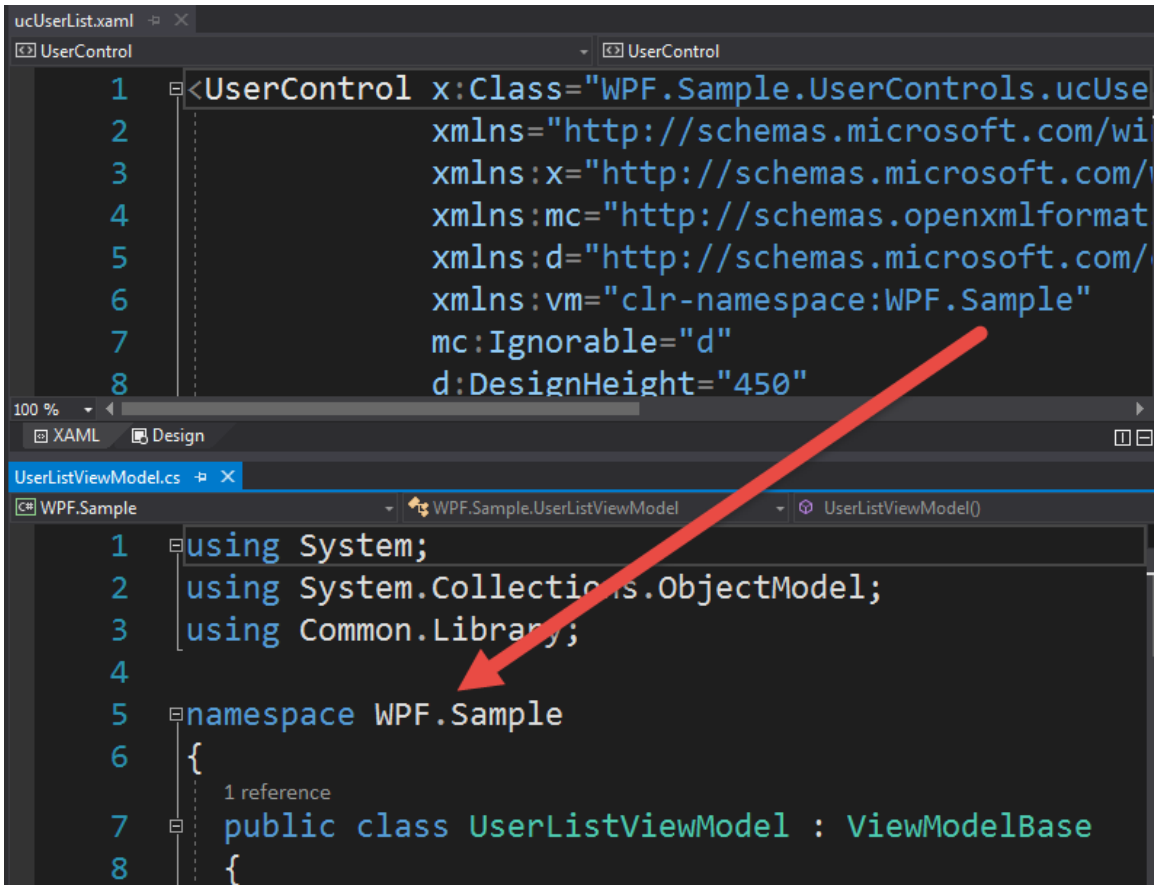


Figure 1: Create an XML namespace to reference view models within a namespace in your project.

Once you have defined this XML namespace, create an instance of the UserListViewModel class. Add a <UserControl.Resources> element just before the <Grid> element. Add an element that begins with the aliased namespace "vm", followed by a colon, then the class name you wish to create an instance of. As soon as you type the colon, Visual Studio will provide you with an IntelliSense list of classes within that namespace (see Figure 2).

```
<UserControl.Resources>
  <vm:UserListViewModel x:Key="viewModel" />
</UserControl.Resources>
```

To bind to this XAML instantiated class, you must provide a key name so you can reference it. In this case, you gave the key name of "viewModel".

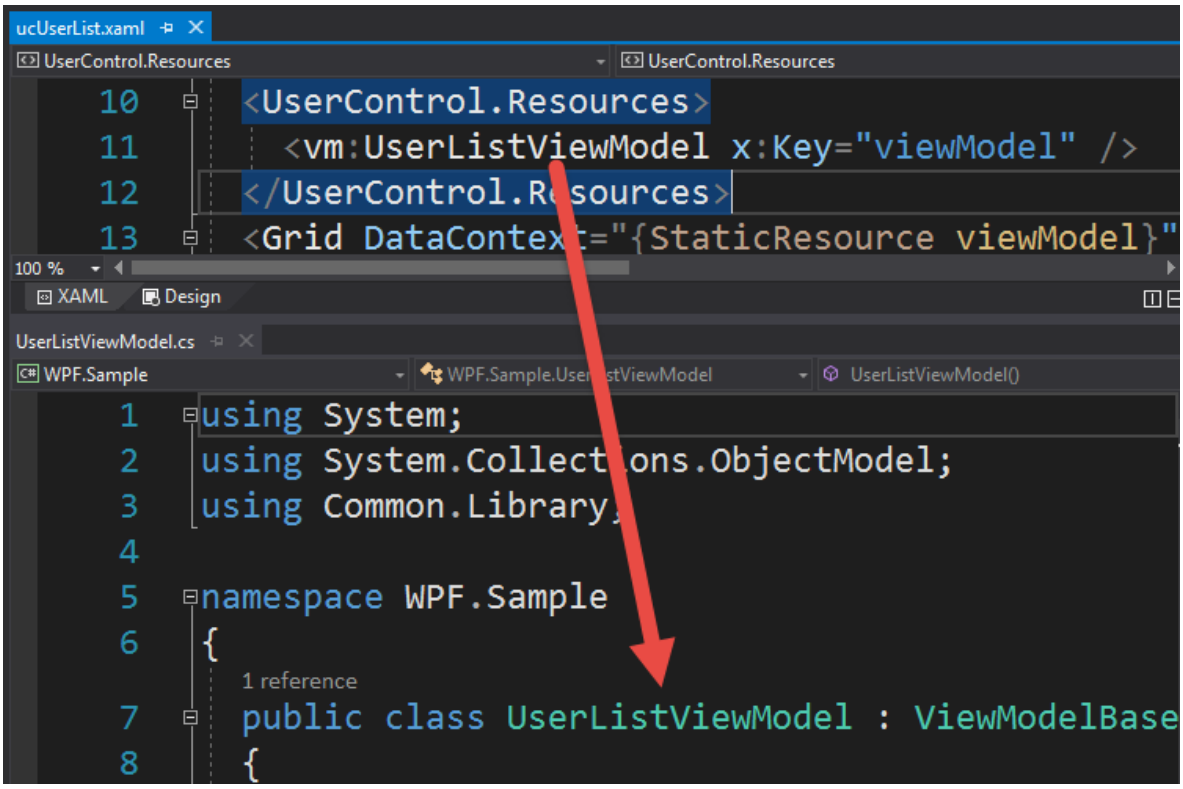


Figure 2: Create an instance of a view model in XAML using the alias of the namespace and assigning a key name.

Fix Up User Control

As you are now going to be using all of the code you created in the User view model class, you may now remove code from your UserListControl control. Open the **UserListControl.xaml** file and remove the following attribute from the <UserControl> element.

```
Loaded="UserControl_Loaded"
```

Remove the Name="lstData" from the ListView control.

Open the **UserListControl.xaml.cs** file and remove the UserControl_Loaded() event and the LoadUsers() method you created earlier in this blog post.

Bind View Model to List View

You can take advantage of the parent-child hierarchy of XAML for data-binding. Normally, you create a single view model to bind to a single user control (or window) and bind controls to properties in the view model. Open the **UserListControl.xaml** file and locate the <Grid> element. Modify the <Grid> element to bind to the view model. Now you see why the key name "viewModel" you

created earlier is important. It provides us with a way to bind the DataContext property of the <Grid> element to the user view model.

```
<Grid DataContext="{StaticResource viewModel}">
```

All controls contained within the <Grid> may now bind to any property within the view model class. Modify the ListView control to bind to the Users property (see Figure 3) you created in the UserListViewModel class.

```
<ListView ItemsSource="{Binding Path=Users}">
```

In Figure 3, you see that the DataContext of the <Grid> is bound to the static resource you created named "viewModel". This static resource is an instance of the UserListViewModel class created by XAML when the user control is instantiated. Finally, the ListView control's ItemsSource property is bound to the Users property within the UserListViewModel class. Since the ListView control is a child of the <Grid> element, it has complete access to any of the properties of the class it is bound to.

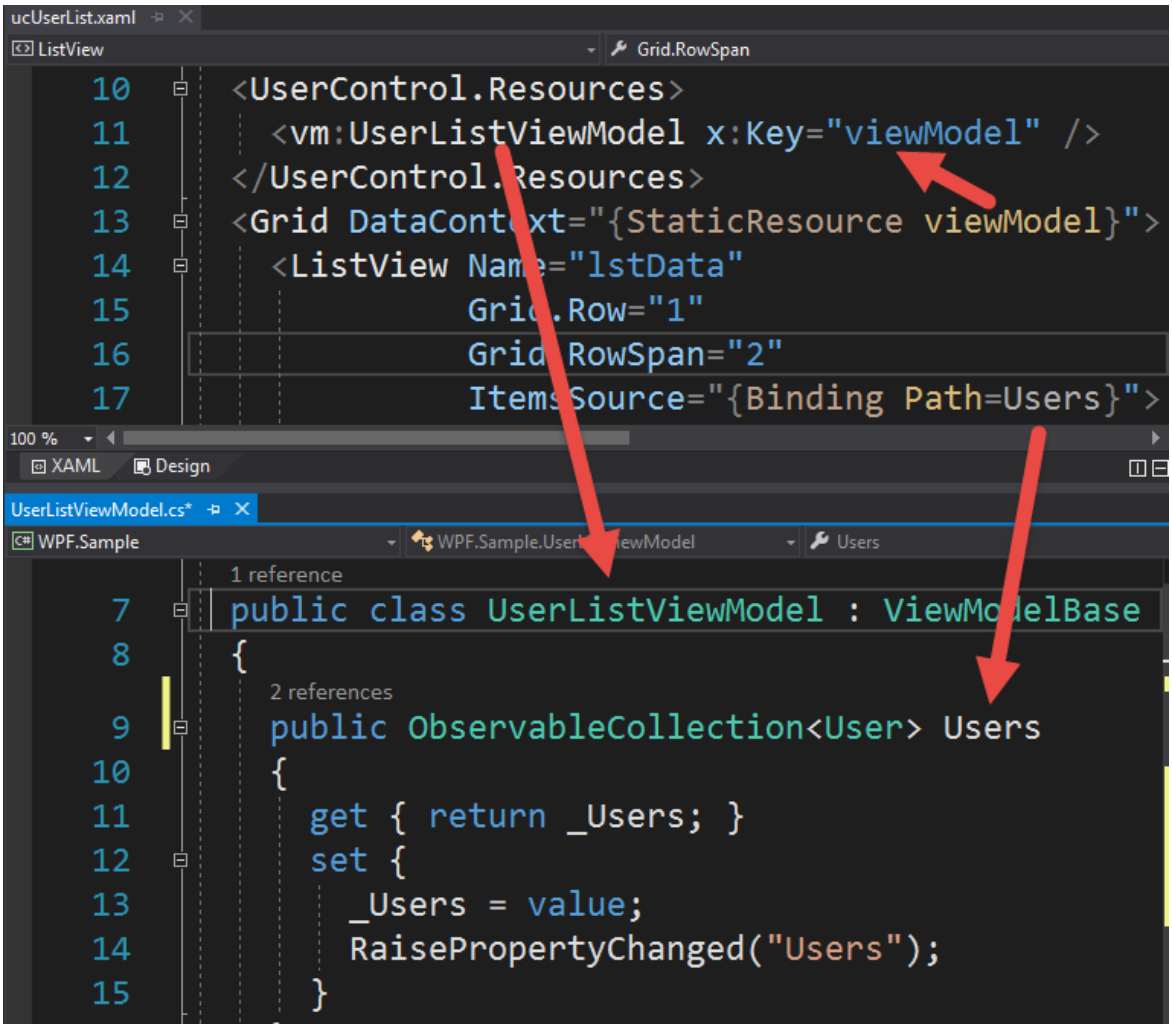


Figure 3: Bind to a property in the view model to provide data to your UI controls.

Summary

Hopefully, this post helped you see how easy it is to move code from the code behind your user interface and put it into a class. That is the whole key to MVVM; simply moving code into a class. Do not worry about being 100% “code behind free.” That is an almost impossible goal and most often requires you to write more code. If your event procedures in your UI are simply doing UI code or making a single call to a method in your View Model, then you have accomplished the goals of MVVM; namely reusability, maintainability, and testability. You also get one more benefit: having event procedures in your UI makes it easier to follow the flow of the application.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Fairway/PDSA Blog,” then select “Basics of MVVM in WPF” from the dropdown list.