How to use MVVM Light Toolkit for Windows Phone

This article shows how to get started with using the MVVM Light Toolkit.

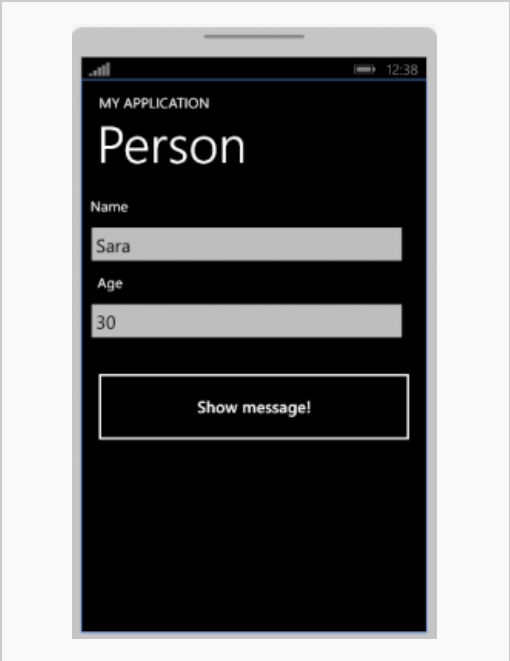
Under Construction: This article is under construction and it may have outstanding issues. If you have any comments please use the comments tab.

Introduction

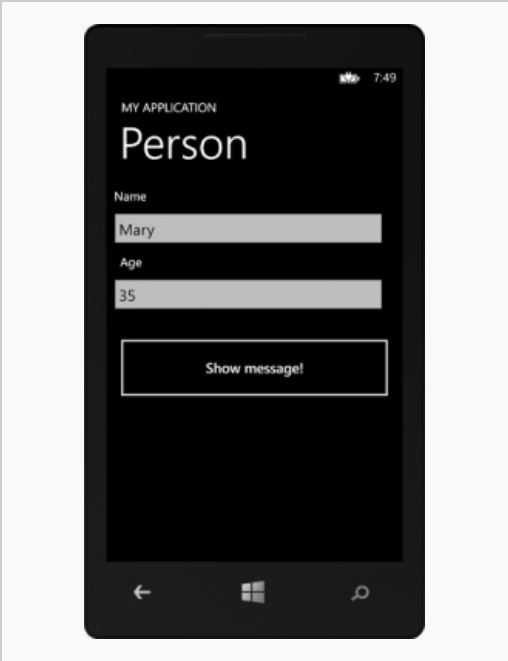
Model-View-ViewModel (MVVM) is an architectural pattern commonly recommended for structuring Windows Phone applications. The pattern separates an application into **Views** (user interface), **Model** ("business logic") and **ViewModel** (mediates the connection between Views and Models and provide "display logic"). The clear separation of the *View* and *Model* allows these to be separately developed by design and developer teams. In addition, with much of the UI display logic moved from the *View* code-behind into the *ViewModel*, a lot more of the UI behaviour can be unit tested. As MVVM is a pattern there are many possible implementations and toolkits.

The lightweight and popular MVVM Light Toolkit is one such toolkit: it delivers a set of libraries that take care of the plumbing to set up an MVVM structure in an app, and provides extra helper classes to make writing MVVM apps easier. The toolkit is characterised by the fact that it is lightweight/minimalist, it makes testing easy, and has been developed to make it easy to work in *Expression Blend*.

This article provides an overview of what the MVVM Light Toolkit offers. Its associated code example shows how MVVM can be used to connect the UI and business logic for entering details about a "Person".



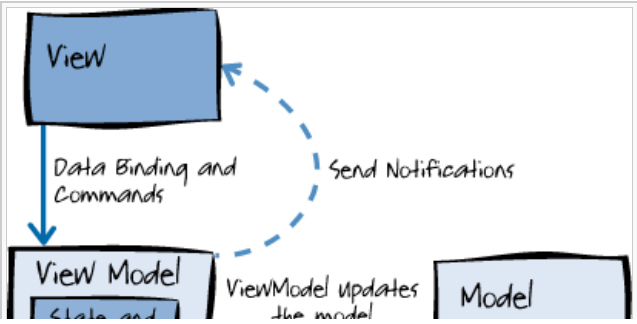
Design Mode



App running

MVVM Light Overview

The image to the right shows the structure of an MVVM app. The *MVVM Light Toolkit* delivers a set of libraries that take care of the plumbing to set up this structure, and provides extra helper classes to make writing MVVM apps easier.



When you add MVVM to a project with Nuget, most of the work of creating the MVVM structure is done for you. Nuget creates a skeleton `ViewModel` and `ViewModeLocator`. The `ViewModeLocator` is not part of the MVVM pattern itself, but is used by the MVVM Light Toolkit to help bind XAML pages with their view model, and to manage all view models and their dependencies.

In order to create an app all you need to do is:

- Extend the skeleton `ViewModel` (possibly using the helper

(C) Copyright Nokia 2014. All rights reserved.

classes discussed below)

- Create a separate *Model* if needed. Note that a separate model is needed to conform to the pattern, but is not mandated by the *MVVM Light Toolkit*.
- Create the *View* (`MainPage`) and bind it to the `ViewModel`
- If you want to have additional *ViewModels* or use services these need to be registered in the `ViewModeLocator`

Diagram with an overview of the MVVM pattern

The main helper classes are listed below:

- `ViewModelBase` class to be used as the base class for `ViewModels`.
- `Messenger` class (and diverse message types) to communicate within the application, and particularly between `ViewModels`.
- `RelayCommand` classes to simplify passing commands from `View` to `ViewModel`.

The *GalaSoft.MvvmLight.Extras* library has optional classes:

- `EventToCommand` behaviour allows developer to bind any event of any UI element to an `ICommand`, for example on the `ViewModel`, directly in XAML.
- `DispatcherHelper` class, a lightweight class helping you to create multi-threaded applications.

Lastly, the Nuget installer adds Visual Studio Intellisense code snippets to speed up the addition of new properties. These include snippets to add new bindable properties to `ViewModel` (mvvminpc), add a new `ViewModel` to a `ViewModelLocator` (mvvmlocatorproperty), add a new attached property to a `DependencyObject` (mvvmslpropa) and add a new dependency property to a `DependencyObject` (mvvmslpropdp).

Installing MVVM Light

Installation instructions are provided in How to install MVVM Light Toolkit for Windows Phone.

Creating and binding to ViewModels (ViewModelLocator)

After installing the tookit using Nuget the project has a new folder *ViewModel* containing files **MainViewModel.cs** and **ViewModelLocator.cs**, which contain skeleton `ViewModel` and `ViewModeLocator` classes, respectively.

To create the `MainViewModel` (or fetch an existing instance) and bind it to a page, all that needs to be done is to add the following line to the page's XAML.

```
DataContext="{Binding MainViewModel, Source={StaticResource Locator}}"
```

The rest of the "plumbing" for binding to a single model has been set up by the Nuget installer, the heart of which is the `ViewModeLocator`. `ViewModelLocator` is a configuration class for an Inversion of control (IoC) Container used by MVVM Light to manage the creation of all view models and their dependencies, and to help connect the *Views* and *ViewModels* owned by the app.

After installation the project **App.xaml** contains an additional line which creates a static (singleton) instance of a `ViewModelLocator` and sets it as the default data source for the app. Because this is set as a datasource for the app, the locator can be used to create and return models referenced in XAML.

```
   <vm:ViewModelLocator x:Key="Locator" d:IsDataSource="True" xmlns:vm="clr-
namespace:MVVMLightSample.ViewModel" />
```

The default skeleton class contains everything needed by the locator for a single view app, except the implementation of the

(C) Copyright Nokia 2014. All rights reserved.

`Cleanup()` method (highlighted), which is discussed in a following section.

ViewModelLocator.cs

```
/// <summary>
/// This class contains static references to all the view models in the
/// application and provides an entry point for the bindings.
/// </summary>
public class ViewModelLocator
{
    /// <summary>
    /// Initializes a new instance of the ViewModelLocator class.
    /// </summary>
    public ViewModelLocator()
    {
        ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
        SimpleIoc.Default.Register<MainViewModel>();
    }

    /// <summary>
    /// Gets the main view model.
```


(C) Copyright Nokia 2014. All rights reserved.

```
        /// </summary>

        /// <value>

        /// The main view model.

        /// </value>

        public MainViewModel MainViewModel

        {

            get

            {

                return ServiceLocator.Current.GetInstance<MainViewModel>();

            }

        }


        /// <summary>

        /// Cleanups this instance.

        /// </summary>

        public static void Cleanup()

        {

            // TODO Clear the ViewModels
```


(C) Copyright Nokia 2014. All rights reserved.

```
            var viewModelLocator = (ViewModelLocator)Application.Current.Resources["Locator"];
```

```
            viewModelLocator.Main.Cleanup();
```

```

```

```
            Messenger.Reset();
```

```
        }
```

```
    }
```

The constructor specifies the specific location provider used. It then registers `MainViewModel` as a type of object that it can create and defines a property which returns an instance of `MainViewModel` from the locator. For more information see IOC Containers and MVVM (MSDN Magazine February 2013 Issue).

The flow of operation is therefore that when the **App.xaml** is run a singleton of `ViewModelLocator` is created and set as a data context. The *View* XAML can bind to the `ViewModel` property it needs using the locator, which will either return the existing instance of the model or create a new one.

Note: For clarity we've removed the following commented code from the `ViewModelLocator` default constructor. If uncommented this code registers a different `IDataService` for use at runtime and design time. This example does provide different runtime/design time data, but does so in the `ViewModel`.

```
////if (ViewModelBase.IsInDesignModeStatic)
////{
////    // Create design time view services and models
////    SimpleIoc.Default.Register<IDataService, DesignDataService>();
////}
////else
////{
////    // Create run time view services and models
////    SimpleIoc.Default.Register<IDataService, DataService>();
////}
```

Creating and binding additional ViewModels

The code example app delivered by this article only has one `ViewModel`.

To add additional *ViewModels* just follow the same approach: each new *ViewModel* needs to derive from `ViewModelBase`, update the `ViewModelLocator` to register each *ViewModel*, and create an appropriate property to get an instance of the view model from the `ServiceLocator`.

```
        public ViewModelLocator()
```

```
        {
```


(C) Copyright Nokia 2014. All rights reserved.

```
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

            SimpleIoc.Default.Register<MainViewModel>();

            SimpleIoc.Default.Register<AnotherViewModel>();

        }

        ...

        public AnotherViewModel Another

        {

            get

            {

                return ServiceLocator.Current.GetInstance<AnotherViewModel >();

            }

        }
```

Then bind to the new `Another` property from the *View* XAML the same way:

```
DataContext="{Binding Another, Source={StaticResource Locator}}"
```

Creating the ViewModel (MainViewModel)

MainViewModel.cs contains the `MainViewModel` class that represents the view model for main page and this class inherits from `ViewModelBase` which represent the base class for the `ViewModel` classes in MVVM pattern. For this sample, the `MainViewModel` will contain one property called `Person` which raise a notification when the `Person` value is changed.

(C) Copyright Nokia 2014. All rights reserved.

Note: because we use the `Set()` method the notification will be raised only when it is needed (ie when the value changes).

The `MainViewModel` will be similar to the code snippet below:

```
public class MainViewModel : ViewModelBase
 {
     /// <summary>
     /// The person
     /// </summary>
     private Person _person;

     /// <summary>
     /// Initializes a new instance of the MainViewModel class.
     /// </summary>
     public MainViewModel()
     {
         if (IsInDesignMode)
         {
             // Code runs in Blend --> create design time data.
             _person = new Person { Name = "Sara", Age = 30 };
         }
         else
         {
             // Code runs "for real"
             _person = new Person { Name = "Mary", Age = 35 };
         }

         PropertyChanged += MainViewModel_PropertyChanged;
         ShowMessageCommand = new RelayCommand(ShowMessage);
     }

     /// <summary>
     /// Gets or sets the person.
     /// </summary>
     /// <value>
     /// The person.
     /// </value>
     public Person Person
     {
         get
         {
             return _person;
         }
         set
         {
             Set("Person", ref _person, value);
         }
     }

     /// <summary>
     /// Gets the show message command.
     /// </summary>
     /// <value>
     /// The show message command.
     /// </value>
     public ICommand ShowMessageCommand { get; private set; }
```


(C) Copyright Nokia 2014. All rights reserved.

```
        /// <summary>
        /// Unregisters this instance from the Messenger class.
        /// <para>To cleanup additional resources, override this method, clean
        /// up and then call base.Cleanup().</para>
        /// </summary>
        public override void Cleanup()
        {
            base.Cleanup();
            PropertyChanged -= MainViewModel_PropertyChanged;
        }

        /// <summary>
        /// Handles the PropertyChanged event of the MainViewModel control.
        /// </summary>
        /// <param name="sender">The source of the event.</param>
        /// <param name="e">The <see
cref="System.ComponentModel.PropertyChangedEventArgs"/> instance containing the event
data.</param>
        private void MainViewModel_PropertyChanged(object sender,
System.ComponentModel.PropertyChangedEventArgs e)
        {
            if (e.PropertyName == "Person")
            {
                //Do something if necessary
            }
        }

        /// <summary>
        /// Shows the message.
        /// </summary>
        private void ShowMessage()
        {
            Messenger.Default.Send("This is a message.");
        }
    }
```

Note: A `Cleanup` method is useful to un-register the view model instance from the `Messenger` class and to cleanup additional resources. Override this method, clean up and then call `base.Cleanup()`.

Creating the Model (ObservableObject)

The class `Person` is represented by:

```
    public class Person : ObservableObject
     {
        public string _name;
        public int _age;

        public string Name
        {
            get
            {
                return _name;
```


(C) Copyright Nokia 2014. All rights reserved.

```
            }
            set
            {
                Set("Name",ref _name, value);
            }
        }

        public int Age
        {
            get
            {
                return _age;
            }
            set
            {
                Set("Age",ref _age, value);
            }
        }

    }
```

`Person` is an `ObservableObject` that represents a base class for objects of which the properties must be observable and it inherits from `INotifyPropertyChanged` and `INotifyPropertyChanging`.

Creating the view (MainPage)

In **MainPage.xaml.cs** we have:

```
public partial class MainPage
   {
       /// <summary>
       /// Initializes a new instance of the <see cref="MainPage"/> class.
       /// </summary>
       public MainPage()
       {
           InitializeComponent();

           Messenger.Default.Register<string>(this, ShowMessage);
       }

       /// <summary>
       /// Shows the message.
       /// </summary>
       /// <param name="message">The message.</param>
       private void ShowMessage(string message)
       {
           MessageBox.Show(message);
       }
   }
```

Note: The `Messenger` is a class allowing objects to exchange messages, and is possible to send any kind of object. This useful for communicating between view models when a page has more than one view model binding.

In **MainPage.xaml** we have

(C) Copyright Nokia 2014. All rights reserved.

```
<phone:PhoneApplicationPage x:Class="MVVMLightSample.MainPage"

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"
                            xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
                            xmlns:shell="clr-
namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
                            DataContext="{Binding Main, Source={StaticResource
Locator}}"
                            FontFamily="{StaticResource PhoneFontFamilyNormal}"
                            FontSize="{StaticResource PhoneFontSizeNormal}"
                            Foreground="{StaticResource PhoneForegroundBrush}"
                            Orientation="Portrait"
                            SupportedOrientations="Portrait"
                            shell:SystemTray.IsVisible="True"
                            mc:Ignorable="d">

    <!--  LayoutRoot is the root grid where all page content is placed  -->
    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

       <!--  TitlePanel contains the name of the application and page title  -->
        <StackPanel x:Name="TitlePanel"
                    Grid.Row="0"
                    Margin="12,17,0,28">
            <TextBlock Margin="12,0"
                       Style="{StaticResource PhoneTextNormalStyle}"
                       Text="MY APPLICATION" />
            <TextBlock Margin="9,-7,0,0"
                       Style="{StaticResource PhoneTextTitle1Style}"
                       Text="Person" />
        </StackPanel>

        <!--  ContentPanel - place additional content here  -->
        <Grid x:Name="ContentPanel"
              Grid.Row="1"
              Margin="12,0,12,0">
            <TextBlock HorizontalAlignment="Left"
                       VerticalAlignment="Top"
                       Text="Name"
                       TextWrapping="Wrap" />
            <TextBox Height="70"
                     Margin="-10,32,10,505"
                     Text="{Binding Person.Name}"
                     TextWrapping="Wrap" />
            <TextBlock Margin="10,107,0,0"
                       HorizontalAlignment="Left"
                       VerticalAlignment="Top"
                       Text="Age"
                       TextWrapping="Wrap" />
```


(C) Copyright Nokia 2014. All rights reserved.

```
            <TextBox Height="70"
                     Margin="-10,139,10,398"
                     Text="{Binding Person.Age}"
                     TextWrapping="Wrap" />

            <Button Height="115"
                    Margin="0,0,0,256"
                    VerticalAlignment="Bottom"
                    Command="{Binding ShowMessageCommand}"
                    Content="Show message!" />
        </Grid>

        <!--
            Uncomment to see an alignment grid to help ensure your controls are
            aligned on common boundaries.  The image has a top margin of -32px to
            account for the System Tray. Set this to 0 (or remove the margin altogether)
            if the System Tray is hidden.

            Before shipping remove this XAML and the image itself.
        -->
        <!--  <Image Source="/Assets/AlignmentGrid.png" VerticalAlignment="Top"
Height="800" Width="480" Margin="0,-32,0,0" Grid.Row="0" Grid.RowSpan="2"
IsHitTestVisible="False" />  -->
    </Grid>

</phone:PhoneApplicationPage>
```

How to use services

This code example doesn't use any standard services, but you can see several examples in other articles on the wiki (for example How to add marketplace review using the Cimalino Windows Phone Toolkit). Most services are registered in the same way, and used similarly.

As discussed previously, MVVM Light uses an IoC Container (in this case "SimpleIoc") to manage dependencies and create components and services. When a component or service is required the container will return an existing instance or create a new one if needed. Typically the component or service is created on demand, but you can also specify a parameter when registering to create the object immediately.

The container also supports a very useful feature called *constructor dependency injection*. When *SimpleIoc* creates an instance of a class that has a constructor parameter, it checks if an object of the parameter's type has been registered, and if so it passes and object of the parameter's type to the new instance.

Therefore to use a service we first register it, and then we ensure that the ViewModel (that is to use the service) takes the service's interface as a parameter. When the ViewModel is created "SimpleIoc" also creates and passes it the service object which it can then use to "invoke" the service.

Register the service

Register the service in the `ViewModelLocator` constructor as shown below using the service interface and implementation classes (ViewModelLocator.cs).

```
32.         public ViewModelLocator()
```

```
33.         {
```

```
34.             ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
```


(C) Copyright Nokia 2014. All rights reserved.

```

            if (!SimpleIoc.Default.IsRegistered<IMarketplaceReviewService>())
            {
                SimpleIoc.Default.Register<IMarketplaceReviewService, MarketplaceReviewService>();
            }

            SimpleIoc.Default.Register<MainViewModel>();
        }
```

Define an appropriate ViewModel

Then define the MainViewModel constructor with a parameter that is of the same type of our service interface, and assign this object to a private member. MVVM Light automatically creates this service along with the `MainViewModel`.

```
    public class MainViewModel : ViewModelBase
    {
        // The marketplace review service.
        private readonly IMarketplaceReviewService _marketplaceReviewService;

        // Initializes a new instance of the MainViewModel class.
        // Parameter of type IMarketplaceReviewService
        public MainViewModel(IMarketplaceReviewService marketplaceReviewService)
```


(C) Copyright Nokia 2014. All rights reserved.

```
16.         {
17.             _marketplaceReviewService = marketplaceReviewService;
18.         ...
```

The rest of the process is service-specific and "plumbing" - connecting the ViewModel to the View, and perhaps sending a command to "invoke" the service:

```
_marketplaceReviewService.Show();
```

For more information on IOC Containers, see:

- IOC Containers and MVVM (MSDN Magazine February 2013 Issue)
- 10 things you might have missed about MVVM Light (Nico Vermeir blog)
- Inversion of Control (Wikipedia)
- How to add marketplace review using the Cimalino Windows Phone Toolkit

Source code

The source code can be found here: http://github.com/saramgsilva/MVVMLightSamples

Related Sample

- Advanced About Page for Windows Phone apps

References

- MVVM Light Toolkit
- Articles and tutorials about the MVVM "light" toolkit
- MVVM Light Toolkit - 1 Introducing MVVM Light Toolkit
- Using MVVM Light to drive a Windows Phone 7 / Silverlight 4 map viewer
- Silverlight TV 13: MVVM Light Toolkit
- Deep Dive MVVM
- Windows Phone 7 – MVVM Light and Unit Testing Example
- Windows Phone 7 – MVVM Light Messaging and Dynamic Application Bar
- IOC Containers and MVVM (MSDN Magazine February 2013 Issue)
- 10 things you might have missed about MVVM Light (Nico Vermeir blog)

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.