

Refcard #135

The MVVM Design Pattern

A Formula for Elegant, Maintainable Mobile Apps

by Colin Melia

Learn how to use Silverlight to create gorgeous mobile apps for the Windows Phone. Included in this Refcard is everything from an explanation of the MVVM Design Pattern to some examples of MVVM Project Templates.

Free PDF

[↓ DOWNLOAD](#)[☆ SAVE](#)

SECTION 1

Overview

If you are developing a Silverlight application for the Windows Phone, then this Refcard is probably for you. For all but the most trivial of applications, the Model-View-ViewModel (MVVM) pattern provides a solid pattern to follow for building a well structured and maintainable application. Microsoft has taken that message seriously

structured and maintainable application. Microsoft has taken that message seriously, with Windows Phone by basing three of the four Visual Studio application project templates on the MVVM pattern. This card provides an explanation of the MVVM pattern, how it's supported and how to follow it on Windows Phone. As a side benefit, the content is almost entirely applicable to Silverlight desktop, web, slate and Windows Presentation Foundation (WPF) applications as well.

Why MVVM on Windows Phone?

MVVM is an architecture pattern introduced by John Gossman in 2005 specifically for use with WPF as a concrete application of Martin Fowler's broader Presentation Model pattern. Implementation of an application, based on the MVVM patterns, uses various platform capabilities that are available in some form for WPF, Silverlight desktop/web, and on Windows Phone with a little help from other libraries. Many commercial applications, including Microsoft Expression products, were built following MVVM.

The benefits of MVVM are listed as follows and can be largely summed up in the phrase "Separation of Concerns".

- **Modular architecture:** given good inter-layer interface definitions, components can be built and well tested independently.
- **Loose coupling:** with one-way dependencies, changes to one layer don't require the other to be changed, rebuilt or retested.
- **Role separation:** responsibilities and expertise can be focused (e.g., designers can build UI without needing to write code).
- **Tool friendly:** the design of the pattern and the capabilities of the platform mean that different tools best suited to the skills of the user can be used. These tools include Visual Studio for developers and Blend for UI designers.
- **Maintainability:** as with other patterns, a well-structured design makes it easier to make modifications when updates or upgrades are done.
- **Less coding:** the separation also leads to areas of development with less code, which means less room for error; it also means less regression testing when changes are made.
- **Testability:** the pattern enables automated unit testing of code and minimizes the need for UI-based testing.



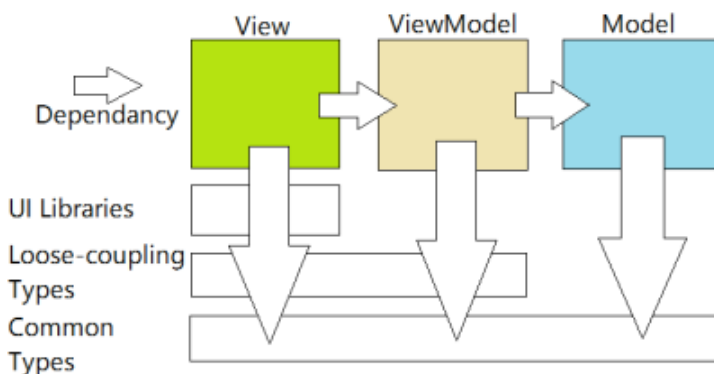
The MVVM pattern is not a set of all-or-nothing rules that one must strictly adhere to. It may not be appropriate for all applications, especially small ones where implementing MVVM may require too much overhead.

SECTION 2

MVVM Explained

Introduction

The MVVM pattern encourages developers to build their application as three layers with the dependencies shown.



Model

The Model is the object model for the application. It can also be the 'layer' for data and/or business logic that are completely devoid of UI features and any dependency on the platform UI libraries or runtime. It can hold state and/or perform processing on the state relevant to the business/problem domain. The data or the operations performed on the data may be dealt with entirely in process memory (e.g., the current value of rolled dice), or they may be retrieved, stored and processed remotely using a data repository/service (e.g., a database or web service) with the model in memory representing a subset of the available data (e.g., specific customer records).



By creating a Model that is not bound to a single platform, it is easier to share the data model across Silverlight, WPF and WP7 platforms with the potential for reuse of model documentation, direct code or even binary libraries. Recommend basing MVVM development on Silverlight 3 to ease code sharing with WP7, Silverlight 3/4 and WPF 4.

View

The View is a 'layer' that represents and handles all the UI elements, including both displayed UI (e.g., classic 'buttons' and more sophisticated displays) and UI-user interactions (e.g., screen touch, button press, etc.).

There can be several Views over the same data with varying detail, depth or representation (e.g., summary + details, order + order line items, charting).

One of the capabilities a platform needs to enable the MVVM pattern well is the ability to create UI declaratively with text instead of code. Those declarations can include parts that 'bind' visual elements to data available in the Model, such that value or list changes stay in sync between the View and Model data. This minimizes or replaces imperative code for settings values between the View elements and data in the Model. Also, since code in the UI can be difficult to test (with automation), this reduces or eliminates the amount of UI-updating-related code to be tested. The concept of binding can also be used to bind a user-driven event to an action to be performed against the data in the Model.

Creation of a View declaratively also opens up the possibility that someone with good design/interactivity sensibilities can design the View (perhaps with a more designer-focused tool) without coding/library expertise and somewhat independently from the software developer.

ViewModel

While a View can be bound to the data in the Model or actions against that data, in practice this is not done directly but via another 'layer' called the ViewModel or VM. It is a "Model of the View" since it is like an abstraction of the View (with UI code) but also a specialization of the Model that the View can bind to. Creation of the ViewModel may be appropriate because:

- It may be that the Model code is not controlled by a developer and also does not expose data in a way that allows a View to be bound to it. In this case, the ViewModel wraps the Model data to expose in a way that allows for declarative binding with update notifications.
- The Model may have data types that may not directly match the types used by the UI components/libraries (e.g., a boolean value on the Model may need to become a visibility enumeration specific to the UI libraries). In this case, the ViewModel wraps the Model and performs the conversion between the View and Model via the `IValueConverter` interface when data binding.
- There are other complex operations beyond conversion to be performed that are not UI related but don't necessarily belong in the Model for the business/problem domain (e.g., some kind of Aggregation or Visualization computation). Having this code in the ViewModel means it's easier to test.
- The state of UI selection needs to be stored and tracked, but this does not belong in the Model.

The ViewModel is also usually responsible for initiating operations to retrieve and store Model data, which allows it to track operation state and, therefore, expose visual feedback or state information to which the View can bind. The actual operation (and obtaining progress) may be delegated to Model-specific classes.

Inter-Layer Dependency

By deliberately having the ViewModel in the middle, the following becomes possible:

- The View only knows about the ViewModel. It does not know about, nor does it have any reference to or dependency on, the Model. The reverse is also true; therefore, the Model and View can be maintained separately using the ViewModel as the 'buffer'.
- The ViewModel only knows about the Model. It does not know about the View. Therefore, new views can be added without affecting the ViewModel. Given the use of binding, it may even be possible to update the ViewModel without breaking the View.

- The Model only knows about itself. It does not know what ViewModels are wrapping it or what Views are created on top of the ViewModels.
- In any case, all layers share a common knowledge of and are tightly coupled to the basic types/objects of the underlying platform.

SECTION 3

MVVM on Windows Phone



For the latest release of the free Windows Phone developer tools, go to <http://create.msdn.com/>. A complete guide to building and deploying a simple MVVM-based WP7 application for the marketplace is available - <http://bit.ly/WP7Die>.

To start creating a Windows Phone application based on the MVVM pattern, select File->New Project in Visual Studio, select “Silverlight for Windows Phone” from the Installed Templates and then select one of the template types. To build from scratch, select “Windows Phone Application” or see the section below about MVVM support in other templates.

Model

The Model is based on a CLR type from a simple value type as a property...

...to a more complex set of nested classes, e.g., Contact...

```
1
2 public enum PhoneType
3 {
4     Work,
5     Home,
6     Mobile
7 }
8 public class PhoneNumber
9 {
10    public String Number { get; set; }
11    public PhoneType PhoneType { get; set; }
12 }
13 public class Contact
14 {
15    public String Name { get; set; }
16    public List<PhoneNumber> PhoneNumbers { get; set; }
17 }
```

A Model may be placed in a separate class file or project/assembly, or (as is often the case for simple cases) it may be incorporated into the ViewModel.

ViewModel

The ViewModel is a CLR class (typically in a separate class file and sometimes in a separate assembly) that encapsulates the Model or incorporates it. If the Model includes nested objects or collections of objects, then a corresponding ViewModel

class hierarchy may be created. The primary goal is to expose the Model data and actions on the model data to the View for binding.

If the View only takes values from the ViewModel once at initialization, there would be nothing more to do. But typically, the UI updates to reflect value and collection changes in the ViewModel data (and possibly vice versa, e.g., for TextBox changes).

The built-in UI controls for single values on Windows Phone (e.g. TextBlock, TextBox, etc.) used in the View are amongst the controls that look for the INotifyPropertyChanged interface on classes to which they are bound; therefore, the interface should be implemented on the ViewModel. It consists of just one event, and it is standard practice to create a helper method to fire the event, which is then called by the setter method of the properties as they are changed.

Using INotifyPropertyChanged in a class requires this statement.

```
1
2 using System.ComponentModel;
```

A ViewModel incorporating a simple Model with non-nested/collection properties may look like this:

```
1
2 {
3     private int Value;
4
5     public int MyProperty
6     {
7         get { return Value;}
8         set
9         {
10            if (Value == value )
11                return;
12            Value = value;
13            OnPropertyChanged("Value");
14        }
15    }
16
17    private void OnPropertyChanged(String PropName)
18    {
19        if(PropertyChanged != null)
20            PropertyChanged(this, new PropertyChangedEventArgs(PropName));
21    }
22
23    public event PropertyChangedEventHandler PropertyChanged;
24 }
```

Remember that the integer named Value is the Model data in this class. When the property setter method for Value is called, it checks to see if there is a change. If so, calls the helper method that will inform the View (if bound to that object and property) that something has changed.

If a Model contains a collection of items that can change and the UI must update to reflect those changes, the collection must implement the INotifyCollectionChanged

interface. Alternatively, use the `Generic ObservableCollection<T>` class to do the work, which requires this include statement:

```
1
2 using System.Collections.ObjectModel;
```

A ViewModel called `MyValuesVM` (note the 's') containing Model data that is a Collection of objects of type `MyValueVM` would then look like this:

```
1
2 public class MyValuesVM : INotifyPropertyChanged
3 {
4
5     private ObservableCollection<MyValueVM> Values;
6
7     public ObservableCollection<MyValueVM> MyProperty
8     {
9
10        get { return Values;}
11        set
12        {
13            if( Values == value )
14                return;
15            Values = value;
16            OnPropertyChanged("Values");
17        }
18    }
19
20
21    private void OnPropertyChanged(String PropName)
22    {
23        if(PropertyChanged != null)
24            PropertyChanged(this, new PropertyChangedEventArgs(Prop
25 Name));
26    }
27
28    public event PropertyChangedEventHandler PropertyChanged;
29 }
```

In this case, the following changes would be notified to the View:

- The whole collection is set
- Membership of the collection changes
- The Value property on individual `MyValueVM` items change



If a Model has nested/collection objects and the UI needs to bind to changes in the properties of those objects and there is access to the Model code, then it be may easier to incorporate the model/classes into a hierarchy of ViewModel classes. Otherwise (if the Model is not editable), then access to data change events from the Model is needed to fire the `PropertyChanged` event. To help catch otherwise silent runtime binding notification errors, the helper function could be augmented with a reflection-based check to ensure the `PropName` passed in matches an actual property.

View - DataContext

The View layer in Silverlight applications can be implemented in code or declaratively using XAML (where, simply put, at initialization, elements become objects and attributes becomes properties). For Windows Phone, the View is a typically a `PhoneApplicationPage` or a `UserControl`.

Any object used in the View that inherits from `FrameworkElement` (i.e., the visual controls) has a `DataContext` property which can be set to a .NET object. Any descendent `FrameworkElement` in the visual tree of the XAML page will also have the same value for its `DataContext` unless explicitly overridden (in which case its nested XAML descendants have that new value, and so on).

The `DataContext` is typically created and set in one of 4 ways:

1. Per View/page creation in the page/UserControl constructor (the xaml.cs file), for example:

```
1
2 public partial class MainPage : PhoneApplicationPage
3 {
4
5     public MainPage()
6     {
7         this.DataContext = new CustomerVM();
8         InitializeComponent();
9     }
10 }
```

Per View/page creation in the page/UserControl XAML, for example with this XML namespace at the top of the View page:

```
1
2 xmlns:myapp="clr-namespace:MyAppNamespace"
```

... and this resource XAML inside the page root element...

```
1
2 <phone:PhoneApplicationPage.Resources>
3     <myapp:CustomerVM x:Key="MyCustomerVM"/>
4 </phone:PhoneApplicationPage.Resources>
```

... and the `DataContext` set on a `FrameworkElement` item.

```
1
2 <Grid x:Name="TopGrid" DataContext="{StaticResource MyCustomerVM}">
3
4 </Grid>
```

2. Per Application instance – in this case, the `ViewModel` (and possibly a hierarchy of nested `ViewModels`) is created in the `App.xaml` XAML as a

resource (as above), and then parts of it are bound as the DataContext at the root of different Views. This makes sense when the Model data must be available throughout the application's lifetime.

- Using a ViewModel locator in the XAML. The DataContext of each view is bound to properties of an application-instance-bound object using the Path syntax. When the property of the root ViewModel is retrieved (following the View-specific Path) for binding, the getter method can dynamically hand out an instance of the appropriate ViewModel. Combine this with dependency injection and mock encapsulated Models (when the application-instance-bound object is created) for a powerful way to dynamically assign real or mock ViewModels and Models, thus providing support for ViewModel testing, web-service-based Model integration testing and design-time visual designer editing.



Keep Models encapsulated rather than incorporated into ViewModels to enable clean dependency injection testing using mock Models.

View - Binding

With the DataContext property correctly set on the FrameworkElement-derived visual element, various properties can be 'bound' to properties (or descendent properties) of the DataContext object. This can be done programmatically or declaratively.

Given a set of ViewModels like this (with notification helpers and calls omitted for brevity)...

```
1
2 public class CustomerVM
3 {
4     public String ID { get; set; }
5     public String Name { get; set; }
6     public ContactPreferencesVM Preferences { get; set; }
7 }
8 public class ContactPreferencesVM
9 {
10    public Boolean CanCall { get; set; }
11    public Boolean CanEmail { get; set; }
12 }
```

The declarative XAML excerpt may look like this...

```
1
2 <StackPanel Orientation="Horizontal" Name="TopPanel">
3     <TextBlock Text="{Binding ID}"/>
4     <StackPanel>
5         <TextBox Text="{Binding Name, Mode=TwoWay}"/>
6         <StackPanel Orientation="Horizontal" DataContext="{Binding
7 Preferences}"/>
8         <CheckBox IsEnabled="{Binding CanEmail}"/>
9         <CheckBox IsEnabled="{Binding CanPhone}"/>
10    </StackPanel>
11 </StackPanel>
12 </StackPanel>
```

... which shows how to use inherited and explicit descendent DataContext and the basic binding syntax.

The example also shows different data binding options (below) with TwoWay being applicable to a TextBox where the updated should be transferred back to the ViewModel to set the Model data.

Mode	Effect in MVVM
OneTime	ViewModel property value copied to View element upon initialization. Seldom used, but could help performance.
OneWay (default)	ViewModel property value copied to View element upon initialization and when PropertyChanged event called.
TwoWay	Same as one way, plus value copied back from View element to ViewModel property – triggering event various by visual element type.

View – Collection Binding

When the ViewModel exposes model data as a collection, use a control in the View that derives from ItemsControl (e.g., ListBox) and set the ItemsSource to the collection. The control creates a sub-visual tree for each object in the collection and automatically sets its DataContext to the item object. ItemsControls have an ItemTemplate property, which can be set in XAML to a collection of XAML visual elements (including the bindings) to represent visually each object in the collection, for example:

```
1
2 <ItemsControl DataContext="{Binding MyContact}" ItemsSource="{Binding
3 PhoneNumbers}">
4 <ItemsControl.ItemTemplate>
5 <DataTemplate>
6 <StackPanel Orientation="Horizontal">
7 <TextBlock Text="{Binding Number}"/>
8 </StackPanel>
9 </DataTemplate>
10 </ItemsControl.ItemTemplate>
11 </ItemsControl>
```



Overall, using data binding and other View model coupling (shown later) reduces or eliminates UI code that can otherwise be hard to test when driven by user actions.

View - Converters

When Model types don't match View types (e.g., Enumeration type to String), the ViewModel can do the conversation or a converter can be used. To create a converter, create a class derived from System.Windows.Data.IValueConverter and implement Convert() and ConvertBack(), declare an instance of the class in the page (or application) resource (as with ViewModel creation case 2 above) and then add the converter to the binding syntax, for example.

```
1
2 <StackPanel Orientation="Horizontal">
3   <TextBlock Text="{Binding Number}"/>
4   <TextBlock Text="{Binding PhoneType, Converter={StaticResource
5 MyPhoneTypeToStringConverter}"/>
6 </StackPanel>
```








Use Converts (vs. conversion in ViewModel) if they are reusable and relate to UI on one end, not for transforming business data.

SECTION 4

MVVM Project Templates

Project templates based on MVVM

Three of the project templates included with the free Windows Phone developer tools, as indicated, are based on MVVM.

 Windows Phone Application	Visual C#
 Windows Phone Databound Application	Visual C#
 Windows Phone Class Library	Visual C#
 Windows Phone Panorama Application	Visual C#
 Windows Phone Pivot Application	Visual C#



Build on these templates and create custom

Visual Studio templates for increased team productivity - <http://msdn.microsoft.com/en-us/library/6db0hwky.aspx>.



Build on these templates and create custom

Visual Studio templates for increased team productivity - <http://msdn.microsoft.com/en-us/library/6db0hwky.aspx>.

Quick Guide to the MVVM Project Templates

The information in the previous sections together with the following guide to MVVM pattern use in the provided templates should help bring clarity to basic use of MVVM on Windows Phone.

- A single instance of the ViewModel called MainViewModel (incorporating Model data) is created using the singleton pattern and exposed as the ViewModel property on the App object in the App.xaml.cs file.
- The ViewModel class is defined under the ViewModels folder. It has a collection of ItemViewModel objects implemented using ObservableCollection.
- MainPage is the primary View and its DataContext is set in code to App.ViewModel in the page's constructor in MainPage.xaml.cs.
- For the Databound Application, the DataContext for the DetailsPage View is set in code to an item in the App.ViewModel.Items collection in the page's constructor in DetailsPage.xaml.cs.



Consider separating Models, Views and ViewModels into separate folders in Visual Studio Solution Explorer.

MVVM Commands

Introduction

In order to take actions (e.g., load/save data), perform operations (e.g., compute some results) or perform some navigation, a method is needed to convert a user interaction event within the View (button press, touch, etc.) into a method call on the ViewModel class. This is done by exposing 'Commands' on the ViewModel that, when executed, either perform the action in the ViewModel or delegate it to the Model or some other library.

ICommand

A command is implemented as an object property on the ViewModel (which can be bound to by the View) that supports the ICommand interface that is defined on Windows Phone under System.Windows.Input.

```
1
2 public interface ICommand
3 {
4     bool CanExecute(Object parameter);
5     void Execute(Object parameter);
6
7     event EventHandler CanExecuteChanged;
8 }
```

The Execute() method is self explanatory. The CanExecute() method is provided so that a visual element bound to the object can query whether execution is currently possible and potentially update itself visually to indicate the state.

The CanExecuteChanged event should be raised by the ViewModel whenever the ability execute or not has changed.

On the interaction side, there needs to be a way to connect user interactions with exposed commands.

Silverlight controls have events to which code-based event handlers can be attached, but this is UI code that is hard to test. In the Silverlight platform for the desktop/web, the controls derived from ButtonBase (e.g., Button) have a Command property which can be bound to the ICommand-based object properties on the ViewModel, and the Button works with the ICommand interface as one would expect. However, these are not implemented in Silverlight 3 and Silverlight for Windows Phone.

MVVM Light

In addition to no direct ICommand support in Windows Phone for controls to bind to commands, exposing potentially many command objects, with each one as a private class nested inside the ViewModel class can be quite time-consuming and verbose.

While it's technically possible to improve these two issues by creating helper classes (to make exposing commands easier) and creating Behaviours (to bind UI events to

(to make exposing commands easier) and creating Behaviours (to bind UI events to commands) that can be used in code, XAML and XAML-based tools like Blend, it makes sense to use available third-party libraries to accelerate this process.

One such library is part of the MVVM Light toolkit available from GalaSoft - <http://www.galasoft.ch/mvvm/installing/manually/> or use NuGet.

To use MVVM Light libraries to easily expose commands in a concise way:

1. Add a reference to Galasoft.MvvmLight.WP7.dll, Galasoft.MvvmLight.Extras.WP7.dll & System.Windows.Interactivity.dll that come in the toolkit
2. In the ViewModel class file, add:

```
1
2 using GalaSoft.MvvmLight.Command;
```

3. Expose a property of type RelayCommand on the ViewModel:

```
1
2 public RelayCommand MyCommand { get; private set; }
```

4. Create a private method that performs the command

```
1
2 private void DoMyCommand()
3 {
4 // Do command
5 }
```

5. In the ViewModel constructor, instantiate the command so that it calls the private method when Execute() is called on ICommand by the UI.

```
1
2 private void DoMyCommand()
3 {
4 // Do command
5 }
```

6. Optionally, add a second lambda expression to the RelayCommand constructor that will be checked when CanExecute() is called on ICommand by the UI.
7. If necessary, call RaiseCanExecuteChanged() on the command object when the ability to execute the command changes, so that bound UI knows to call CanExecute() to update any visual cues.
8. Repeat steps 3 to 7 for each command to be exposed.

To use the libraries to bind control events (e.g., a single touch that is equivalent to a left mouse button down) to exposed commands in XAML (while writing no code):

1. Add the MVVM Light namespace at the others at the top of the XAML

```
1
2 xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.
3 Windows.Interactivity"
4 xmlns:mvmextra="clr-namespace:GalaSoft.MvvmLight.
5 Command;assembly=GalaSoft.MvvmLight.Extras.WP7"
```

2. Add this XAML like this inside the control's element

```
1
2 <i:Interaction.Triggers>
3 <i:EventTrigger EventName="MouseLeftButtonDown">
4 <mvmextra:EventToCommand Command="{Binding MyCommand}"/>
5 </i:EventTrigger>
6 </i:Interaction.Triggers>
```

3. Optionally, if the control has an `IsEnabled` property to be set according to the `CanExecute` property and `CanExecuteChanged` event on the `ICommand`-enabled object, set the `MustToggleIsEnabledValue` to `True`, e.g.
- 4.



```
<mvmextra:EventToCommand Command="{Binding MyCommand}"
MustToggleIs EnabledValue="True"/>
```

5. Repeat 2 to 3 for all controls to be bound to exposed commands.

SECTION 6

Other MVVM Libraries

Using MVVM Libraries

As shown, the MVVM Light toolkit (for WP7, Silverlight and WPF on CodePlex) is easy to start with and also includes other features:

- A `ViewModelBase` class
- Messenger system for inter-`ViewModel` communication
- Visual Studio project and item templates
- Visual Studio code snippets



To ease implementation, consider inheriting `ViewModels` from a library



base class (or create one), including support for the INotifyPropertyChanged interface inheritance/implementation and helper functions.

Caliburn Micro

Caliburn Micro is a small but powerful micro-framework for WP7, Silverlight and WPF on CodePlex that supports MVVM development including these features:

- ActionMessages – for flexible binding of UI actions to ViewModel methods (achieving what Commands do)
- Bootstrapper & ViewLocator – for pattern configuration and handing out ViewModels to Views
- Screens and Conductors – for tracking active screens and selections

Prism

Prism 4 is a free library from Microsoft Patterns & Practices group for WP7, Silverlight and WPF, originally concerned with building composite application. It includes a WP7 library with helpful features (including MVVM support):

- Commands with DelegateCommand – similar to RelayCommand in MVVM Light.
- Pub/sub eventing
- Run-time data template selection
- Application Bar helpers
- UI Interaction Helpers



Read the in-depth Windows Phone 7 Developer Guide from Microsoft Patterns & Practices to see Prism is a use in sample MVVM-based application - <http://msdn.microsoft.com/en-us/library/gg490765.aspx>



Blendability

Making WP7 MVVM applications ‘blendable’

The free tools for Windows Phone include a version of Blend that can be used for design. Blend has a UI optimized for UI design, over coding.

If a developer produces ViewModel and Model classes, a UI designer can then produce interactive UI in Blend and bind to the ViewModel, producing XAML in the project that can then be loaded back into Visual Studio.

Blend (and the Visual Studio XAML designer) actually instantiates the XAML and calls the constructor of the page class (i.e., the View). This may cause the ViewModel to also be instantiated. For Blend (and Visual Studio) to work well with MVVM-based applications for designing there are a few guidelines to follow:

- Don't call web services or database in the View or ViewModel constructor – the designer may not load.
- To see the design of the UI, try to use XAML over code, since controls added programmatically will not show up.
- Instantiate empty collections in constructors so they can be bound to.
- If possible, use the full ViewModelLocator pattern to allow switching at design-time to mock ViewModels with design data or mock Models injected into the ViewModel. If that route doesn't suit, then at least consider using design data (see below).



To detect if the application is in a designer tool, use `System.ComponentModel.DesignerProperties.IsInDesignTool`

Design Data

The project templates in the tools that support MVVM also show examples of design-time data under the SampleData folder. Since XAML can fundamentally be used to declare objects and properties, it can be used to declare sample ViewModels (including nested object/collection properties). By using the `d:DataContext` (with the design-time namespace prefix) in XAML (see `MainPage.xaml` in the Data Bound application), the `DataContext` can be set on the page or sub-element to use the static sample data.



A designer using Blend to build UI on top of ViewModel classes provided by a developer can use a feature in Blend that automatically generates XAML sample data based on the class properties of the ViewModel or Model as well as generate XAML sample data based on XML files.

Persistence

Data persistence

A ViewModel should wrap the action of saving/loading data it represents (which may be delegated to a Model class). This may use a web service (using the WebClient class, the

HttpWebRequest/Response class, or service proxy class inc. OData proxies) or if data is stored locally, this would involve classes under System.IO and System.IO.IsolatedStorage or third party databases (e.g., Sterling DB or Perst). If properties are serializable, it may be convenient to just serialize the ViewModel and Model state using classes under System.Xml.Serialization.

Having an application-instance-based object with one or more ViewModel properties may provide a convenient ViewModel 'hub' for all the pages in your application.

Tombstoning

An application may have volatile session-specific Model data (e.g., data entry in progress and not yet saved) and/or View Model UI state that can be lost if the application is 'tombstones' (de-activated by the OS to preserve foreground application performance or to perform another task and possibly never re-activated).

ViewModels should subscribe under PhoneApplicationService to the Deactivated event and use its Current.State dictionary to save this transient state on Deactivated and load back (if necessary/ available) in both the ViewModel constructor and the Activated event, the transient data along with any non-session-specific data. See more on tombstoning on MSDN:

[http://msdn.microsoft.com/en-us/library/ff817008\(v=VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff817008(v=VS.92).aspx)



When using MVVM with TextBox controls bound to data, use something like the Prism UpdateTextBindingOnPropertyChanged class in XAML to ensure all Text changes are transferred through the binding prior to a tombstone or Application bar event.

Publications

Featured

Latest

Popular

ABOUT US

About DZone
Send feedback
Careers

ADVERTISE

Media Kit
sales@dzone.com
+1 (919) 443-1644

CONTRIBUTE ON DZONE

MVB Program
Zone Leader Program

LEGAL

Terms of Service
Privacy Policy

CONTACT US

150 Preston Executive Drive
Cary, NC 27513
info@dzone.com
+1 (919) 678-0300

LET'S BE FRIENDS

