Stateless View Models

In this lab, we will code in bursts. Get the starting source code and follow along. I will commit changes after every burst, so you can get caught up if you get lost.

## Gitting the Source Code

If you have a git client installed, then clone the repository. Create a working branch. For example:

cd c:\projects

git clone git://github.com/dallasxaml/StatelessViewModels.git

cd StatelessViewModels

git checkout –b take1

After each successful burst, commit your changes. If you ever get lost, commit that branch, go back to master, and create a new one:

git add –A

git commit –m "I missed that."

git checkout master

git pull

git checkout –b take2

## Downloading the Source Code

If you don’t have git installed, then go to the following URL and click on the “ZIP” button:

https://github.com/dallasxaml/StatelessViewModels

Unzip to your project folder. If you ever get lost, go back to the web page and download the zip again.

# Update Controls

This workshop will be based on the Update Controls library, which is available in NuGet. Create a new WPF Application project and add the following package:

UpdateControls.App

This creates a ViewModels folder with the following items:

* ViewModelLocator
* MainViewModel
* ItemHeader
* ItemViewModel

It also creates a Models folder with:

* Document
* Item
* Selection

# ViewModelLocator

The ViewModelLocator class creates all of the view models. Create an instance of the view model locator in App.xaml.

<Application x:Class="StatelessViewModels.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:vm="clr-namespace:StatelessViewModels.ViewModels"

StartupUri="MainWindow.xaml">

<Application.Resources>

<vm:ViewModelLocator

x:Key="Locator" />

</Application.Resources>

</Application>

Now you can access the view model locator in any of the views.

# Main View

Create a folder called Views. Create a User Control in this folder called MainView. Set the DataContext to the Main property of the ViewModelLocator.

DataContext="{Binding Main, Source={StaticResource Locator}}"

Add the MainView to the MainWindow.xaml.

<Window x:Class="StatelessViewModels.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:v="clr-namespace:StatelessViewModels.Views"

Title="MainWindow" Height="350" Width="525">

<Grid>

<v:MainView />

</Grid>

</Window>

Define a new property on the MainViewModel called Name. This should access the Name property of the Document.

public string Name

{

get { return \_document.Name; }

set { \_document.Name = value; }

}

Data bind this property to a text box and a text block on the view

<StackPanel>

<TextBox

Text="{Binding Name,

Mode=TwoWay,

UpdateSourceTrigger=PropertyChanged}" />

<TextBlock

Text="{Binding Name}" />

</StackPanel>

Run the program and see the two controls working together. Notice that you did not implement INotifyPropertyChanged. You didn’t even tell the base class (what base class?) to fire it.

# Collections

The main view model has a property called Items that returns a collection of ItemHeaders. It uses linq to project the set of items in the document into ItemHeader objects. Create a ListBox that data binds to this collection.

<ListBox

ItemsSource="{Binding Items}">

<ListBox.ItemTemplate>

<DataTemplate>

<TextBlock

Text="{Binding Name}" />

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

Take a look at the document. It uses an IndependentList<Item> to store the items. This is like ObservableCollection<Item>, but it supports linq. That’s why we can project the items to ItemHeaders.

The main view model has an AddItem property that adds an item to the collection. Data bind this property to a button.

<Button

Command="{Binding AddItem}"

Content="Add Item" />

Run the app and click the button. The new item appears because it was added to an IndependentList. It has the name “<New Item>” because that’s the behavior added by ItemHeader.

# Selection

Data bind the selected item to the SelectedItem property of the view model.

<ListBox

ItemsSource="{Binding Items}"

SelectedItem="{Binding SelectedItem, Mode=TwoWay}">

...

</ListBox>

Now the main view model will set the SelectedItem property of the selection model. Other view models can depend upon this. For example, define an ItemDetailViewModel.

public class ItemDetailViewModel

{

private readonly Selection \_selection;

public ItemDetailViewModel(Selection selection)

{

\_selection = selection;

}

public string Name

{

get

{

if (\_selection.SelectedItem == null)

return "<No item is selected>";

else

return \_selection.SelectedItem.Name;

}

set

{

if (\_selection.SelectedItem != null)

\_selection.SelectedItem.Name = value;

}

}

}

Add it to the view model locator.

public object ItemDetail

{

get

{

return ViewModel(() => new ItemDetailViewModel(\_selection));

}

}

Create an ItemDetailView data bound to this view model.

DataContext="{Binding ItemDetail, Source={StaticResource Locator}}"

<TextBox Text="{Binding Name, Mode=TwoWay}" />