## Building a composite UI with WPF and MVVM

What do we do when a logic of a UI screen becomes too complex? Or what do we do when there is a certain part of the screen that we want to reuse on other screens? In both cases there is the same answer - UI composition. Basically composing our screen from smaller sub screens. Each sub screen focuses on its primary goal and can be reused in other places in the application.

This article is the first one in the two articles Blog series. Here we will show a simple composite WPF MVVM application. In the second article we will enhance our app and add Unit Tests.

The full source code for the solution can be found here: <https://github.com/EvgenyTC/UICompositionSample>

In WPF the primary mean for UI composition is a UserControl. It encapsulates the UI and the connected logic and can be placed on other screens like the following:

<UserControl x:Class="UICompositionSample.Views.ChildUserControl"

... >

<TextBlock>Child control</TextBlock>

</UserControl>

*Code Listing 1: ChildUserControl.xaml*

<Window x:Class="UICompositionSample.Views.ParentView"

xmlns:local="clr-namespace:UICompositionSample.Views"

... >

<Grid>

<local:ChildUserControl/>

</Grid>

</Window>

*Code Listing 2: ParentWindow.xaml*

In simple cases the static placing of UserControl into parent Window can be enough. But for more complex scenarios we need something more powerful. Let’s design a hypothetical application. It is a viewer that allows a user to navigate through a list of items and shows a preview for the selected item. Depending on the item content it can be a Text or Image or other kind of a preview.

Text Item 1

Image Item 1

Text Item 2

Image1 Preview

Preview is shown for the selected item inside the Main Window

Main Window

*Image1: The structure of the Viewer app*

Analysis of the main window gives us the following tasks:

* Show the list of items with the selection and preview area
* Generate a suitable preview for the selected item

Consequently, we will split our application like following:

* MainWindow will be responsible for the general layout and items selection
* Each type of preview will be developed in its own UserControl
* In the run time, MainWindow will create a preview UserControl matching the selected item and display it into a designated area

We will develop our application with MVVM so that it can be unit tested

*A small detour into MVVM. Model-View-ViewModel approach was introduced by Microsoft in WPF and features a full separation of UI (View), UI Logic (ViewModel) and Data (Model). A UserControl built with MVVM will have a View and ViewModel. To create a new instance of the UserControl in the run time we will need to create its View, ViewModel and connect between them. To simplify this task as well as other routine tasks in WPF we will use a Caliburn.Micro framework.*

*For UI composition MVVM frameworks employ one of the 2 approaches:*

* *View First – in the run time your code specifies the child View to display in a designated area of the parent View. The framework then creates the child View and ViewModel and connects between them*
* *ViewModel First – in the run time your code creates the child ViewModel and binds it into a designated area of the parent View. The framework then creates the View and connects it with the ViewModel*

*The advantage of the ViewModel First approach is that we create a child ViewModel in our code and thus can pass all the needed parameters into its constructor. We will use this approach below.*

Now let’s see some code. In the MainView we will have a grid with 2 columns. The column 0 will display a list of items and the column 1 will contain a placeholder for a preview:

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="1\*"/>

<ColumnDefinition Width="2\*"/>

</Grid.ColumnDefinitions>

<ListView Grid.Column="0" ItemsSource="{Binding Items}" SelectedItem="{Binding SelectedItem, Mode=TwoWay}">

<ListView.ItemTemplate>

<DataTemplate>

<TextBlock Text="{Binding Name}"/>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

<!--this is the way to tell Caliburn.Micro to inject here a view

that corresponds to the PreviewViewModel-->

<ContentControl Grid.Column="1" cal:View.Model="{Binding PreviewViewModel}"/>

</Grid>

*Code Listing 3: XAML markup from the MainView*

The preview placeholder is bound to PreviewViewModel property of the MainViewModel. Upon selection change the correct ViewModel is created and loaded into PreviewViewModel:

//each item derives from IContentItem

ObservableCollection<IContentItem> \_items;

public ObservableCollection<IContentItem> Items => \_items;

//on selected item change a preview is updated

IContentItem \_selectedItem;

public IContentItem SelectedItem

{

get => \_selectedItem;

set

{

if (\_selectedItem != value)

{

\_selectedItem = value;

UpdatePreview(\_selectedItem);

NotifyOfPropertyChange(nameof(SelectedItem));

}

}

}

//preview ViewModel property for binding into view

object \_previewViewModel;

public object PreviewViewModel

{

get => \_previewViewModel;

set

{

if (\_previewViewModel != value)

{

\_previewViewModel = value;

NotifyOfPropertyChange(nameof(PreviewViewModel));

}

}

}

//correct preview ViewModel is created

//according to the type of the selected item

private void UpdatePreview(IContentItem item)

{

if (item is TextItem)

{

PreviewViewModel = new TextPreviewViewModel(item as TextItem);

}

else if (item is ImageItem)

{

PreviewViewModel = new ImagePreviewViewModel(item as ImageItem);

}

else

{

PreviewViewModel = null;

}

}

*Code Listing 3: Code from MainViewModel*

As you can see the logic of UI composition is very straight forward. Each time when we select an item we create the corresponding ViewModel and place it into PreviewViewModel property. UpdatePreview method servers as a factory here, creating a correct instance of the preview per selected item type.

The PreviewViewModel property is bound to ContentControl in the MainView. Once its updated the MainView is notified. Then Caliburn.Micro finds the View that matches the created ViewModel and injects it into the MainView.

We can easily add new types of items and new types of previews by adding new ifs into the UpdatePreview function. We also have a complete control on the data that is passed to child ViewModels. In our application it is only the selected data item. But in more complex scenarios it can be an arbitrary data and services allowing interactions between various parts of UI.

In the second article of the blog series we will enhance our UI and see how to create Unit tests for a composite screen.