Lab 1: Universal Windows Platform (UWP) App

Throughout the labs, we’ll create the MovieWatchr Universal Windows App: an app to track the movies you want to watch and log what you thought of the ones you have watched. In this first lab exercise, we’ll create a basic version to define its style and interaction. In subsequent labs we’ll connect the app to the TMDb.org API to retrieve relevant movie information and to Azure App Service to make our data available across devices.

At any time during the lab, you can take a look at the solution in the **H01\End** folder to see a sample implementation of everything we’re going to be doing in this lab.

# Preparation

Make sure you have the following installed on your machine:

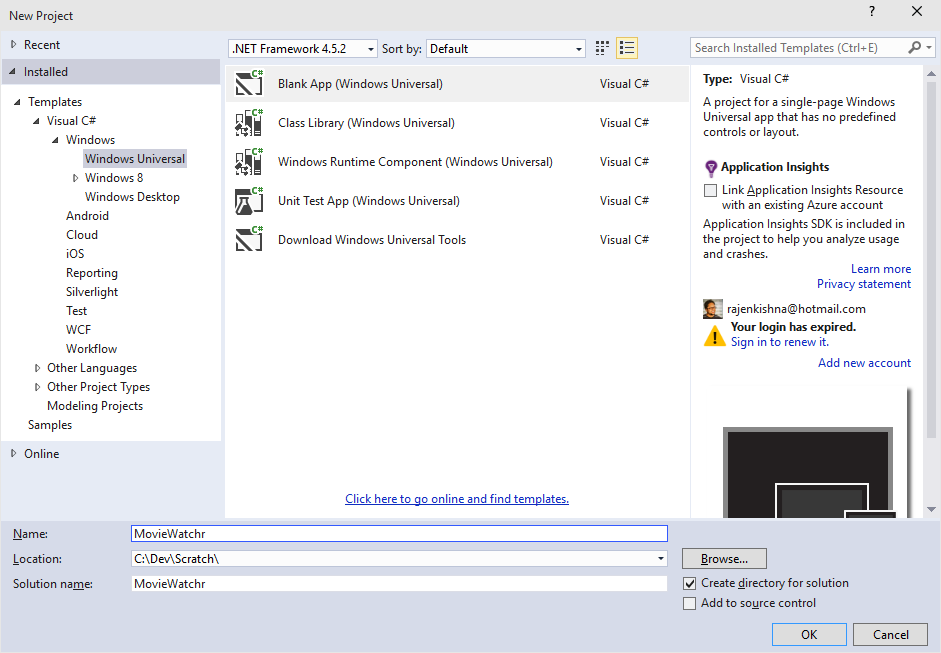
* Windows 10 Technical Preview (build 10074)
  + Note that installing Windows in a virtual machine will prohibit you from using the Emulator and will limit you to testing on actual phone/ARM hardware
* Visual Studio 2015 RC with the Windows 10 Tools

## Starting with the base solution

If you are familiar with the basics of Visual Studio like creating new projects, including files, adding references, etc. you may want to start with the base solution that is available in the **H01\Start** folder. The following paragraph will describe how you can get to that exact same state starting from creating a completely new project.

## Starting from scratch

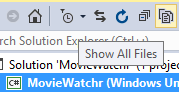
Once you have installed everything on your PC, launch Visual Studio and select **New Project…** on the left hand side on the Start Page. In the New Project dialog that pops up, navigate to the **Templates 🡪 Visual C# 🡪 Windows 🡪 Windows Universal** node on the left hand side and select the **Blank App (Windows Universal) Visual C#** template. As a name, enter **MovieWatchr** and select a location where the solution will be stored. Click on **OK** when ready.



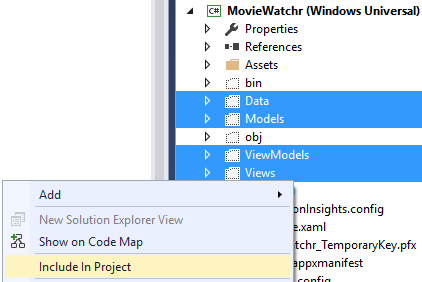
Once the solution and project have been created, go to the **H01\Code** folder of this lab and copy its contents to the folder where the solution was created, overwriting existing files. Make sure you don’t copy the contents into the project folder, but the one above, which should look like this before and after copying the files:

|  |  |
| --- | --- |
| **Before** | **After** |
|  |  |

Back in Visual Studio, select the **MovieWatchr (Windows Universal)** project in Solution Explorer on the right hand side and click on the **Show All Files** button:



Select the **Data, Models, ViewModels** and **Views** folders and right-click to show the context menu. From this menu, select **Include In Project** to add these folders and their contents to the current project:

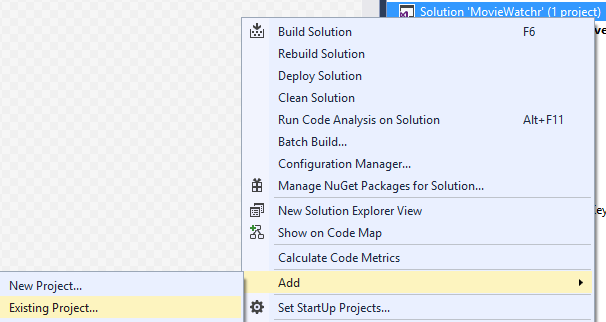


Optionally, click the Show All Files button again to hide the files and folders that are not a part of the project.

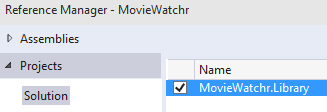
Next, delete the **MainPage.xaml** file that exists in the root of the project and add a new page within the **Views** folder, called **MainPage.xaml** by right-clicking on the folder and selecting **Add 🡪 New Item…** and selecting **Blank Page** from the pop-up dialog.



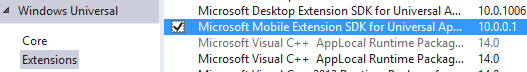
Right-click on the **Solution ‘MovieWatchr’** node in Solution Explorer, select **Add 🡪 Existing Project…** and browse to the **MovieWatchr.Library** folder, selecting the **MovieWatchr.Library.csproj** file.



Right-click on **References** under the **MovieWatchr (Windows Universal)** project, select **Add Reference…** and in the Reference Manager pop-up dialog select the check box next to the **MovieWatchr.Library** project under the **Projects 🡪 Solution** node:



In the same Reference Manager pop-up, go to the **Windows Universal 🡪 Extensions** node and select the check box next to **Microsoft Mobile Extension SDK for Universal App Platform**.



Next, open the **Views 🡪 MainPage.xaml 🡪 MainPage.xaml.cs** file and change the following line:



public sealed partial class MainPage : Page

into

public sealed partial class MainPage : BasePage

and open the **Views 🡪 MainPage.xaml** file and change the topmost XAML node from

<Page

into

<local:BasePage

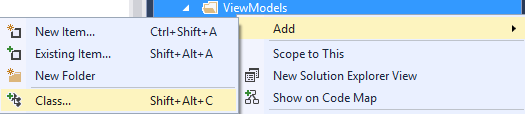
Lastly, open the **App.xaml 🡪** **App.xaml.cs** file and add the following line to the top of the file:

using MovieWatchr.Views;

# Setting up our ViewModel to bind against

To be able to display the data in the Visual Studio designer as well as the running application, we’re going to be using the Model-View-ViewModel (MVVM) pattern. This involves creating a ViewModel for our MainPage to bind against, displaying the list of movies and the movie details. In the previous step you already copied in the necessary models, which you can find in the **Models 🡪 Movie.cs** class file.

Let’s start by creating a new class in the **ViewModels** directory of our **MovieWatchr** project, by right-clicking on that folder and selecting **Add 🡪 Class…**, which we’ll name **MainViewModel.cs**.



Change the following line:

class MainViewModel

into

public class MainViewModel : BindableBase

to utilize the logic we have in the **BindableBase.cs** class in our MovieWatchr.Library project to support data binding. Resolve the **BindableBase** class by pressing Ctrl+., the lightbulb icon next to that line or manually adding the following using statement at the top of the page:

using MovieWatchr.Library.Helpers;

Copy in the following property inside the MainViewModel to allow data binding against the movie we have selected (we’ll set that up in the next step) and make sure you resolve the Movie class to **MovieWatchr.Models**:

private Movie \_selectedMovie;

public Movie SelectedMovie

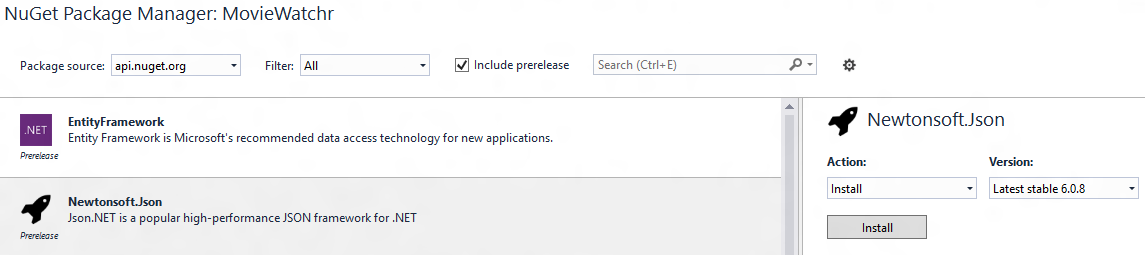
{

get { return \_selectedMovie; }

set { SetProperty(ref \_selectedMovie, value); }

}

Finally, we’ll be using the **Newtonsoft.Json** NuGet package to deserialize our json file under **Data 🡪 StaticData.json**, so right-click the MovieWatch project and select **Manage NuGet Packages…** and search for **Newtonsoft.Json** in the NuGet window, installing the latest stable version:



After these steps, verify the solution builds successfully, by either selecting **Build 🡪 Build Solution** from the top menu in Visual Studio or pressing the F6/Ctrl+Shift+B keyboard shortcut.





# Creating the user interface

Now we’ll start building the user interface of our app. Let’s start out with some basic XAML to save some time. This will lay out the page for us in some convenient sections that will hold the list of movies and provide us with a container to show the movie details. A header is also included to display additional UI and the page/app title. Open the **Views 🡪 MainPage.xaml** file and replace the **Grid** node inside the **local:BasePage** with the following XAML:

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<!-- Header -->

<Grid Grid.Row="0">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="Auto" />

</Grid.ColumnDefinitions>

<TextBlock x:Name="HeaderText" Grid.Column="0" Text="my movies" FontSize="30" FontWeight="ExtraLight" Margin="12,0,0,8" />

</Grid>

<Grid x:Name="ContentRoot" Grid.Row="1">

<Grid.ColumnDefinitions>

<ColumnDefinition x:Name="MasterListViewColumn" Width="\*" />

<ColumnDefinition x:Name="DetailGridColumn" Width="1.5\*" />

</Grid.ColumnDefinitions>

<!-- Master List -->

<Grid x:Name="MasterListGrid" Grid.Column="0">

<Grid.RowDefinitions>

<RowDefinition Height="\*" />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<ListView x:Name="MasterListView" Grid.Row="0">

</ListView>

</Grid>

<!-- Detail view -->

<Grid x:Name="DetailGrid" Grid.Column="1">

<Grid.RowDefinitions>

<RowDefinition Height="\*" />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<ScrollViewer Grid.Row="0" HorizontalScrollMode="Disabled" VerticalScrollBarVisibility="Auto">

<Grid x:Name="MovieDetailsGrid">

</Grid>

</ScrollViewer>

</Grid>

</Grid>

</Grid>

We need the following user interface elements in our app:

* The “Master” ListView (the placeholder is on the page, but it needs to have a DataTemplate to display data)
* The “Detail” view needs to contain elements that display the various properties of the movie (poster image, plot, rating, etc.)
* An “add” button to add new movies to the collection in the app
* Various VisualStates to adapt the user interface to the various screen/window sizes (used in conjuction with AdaptiveTriggers)

## The Master ListView

As you can see, there’s some static data available for the moment in the **Data 🡪 StaticData.json** file. We can use that data in the Visual Studio designer as the DataSource for the ListView by adding a CollectionViewSource to our page’s resources:

<local:BasePage.Resources>

<CollectionViewSource x:Name="MoviesViewSource" Source="{Binding Movies}"

d:Source="{Binding Movies, Source={d:DesignData Source=/Data/StaticData.json, Type=data:StaticDataSource}}"/>

</local:BasePage.Resources>

Don’t forget to add the data namespace to the page to correctly resolve the StaticDataSource class:

xmlns:data="using:MovieWatchr.Data"

The ListView still needs an ItemsSource for the data to show up, as well as a DataTemplate to display the various items. Add those now and confirm the data displays in the Visual Studio designer.

If you take a look at the **StaticData.json** file, you will notice that the poster images are all relative paths. To correctly display the images, we’ll use a **Converter** to add the missing part of the URL. Add a folder called **Converters** to the **MovieWatchr** project and add a new class called **PosterPathToImageUrlConverter.cs**. We’ll be implementing the **IValueConverter** interface, so the code for the class will look like this:

public class PosterPathToImageUrlConverter : IValueConverter

{

private string ImageBaseUrl = "http://image.tmdb.org/t/p/";

public object Convert(object value, Type targetType, object parameter, string language)

{

if (value == null || !(value is string)) return string.Empty;

return string.Format("{0}{1}{2}", ImageBaseUrl, parameter == null ? "w92" : parameter, value);

}

public object ConvertBack(object value, Type targetType, object parameter, string language)

{

throw new NotImplementedException();

}

}

What this converter does, is add the missing part of the URL to the value that’s passed in. All we need now is to add a static resource for our converter, by adding the namespace declaration and resource to our **App.xaml** file:

xmlns:c="using:MovieWatchr.Converters"

<Application.Resources>

<c:PosterPathToImageUrlConverter x:Key="PosterPathToImageUrlConverter" />

</Application.Resources>

If we now use the following binding statement, our converter will be used:

{Binding PosterPath, Converter={StaticResource PosterPathToImageUrlConverter}}

If everything was added correctly, we now see the design time data in the Visual Studio designer.

## The Details view

Similar to the Master ListView, we can use the designer to display our data, while we lay out the various user interface elements on the page. For example, we can set the design time DataContext of the **MovieDetailsGrid** node to the selected item in our MasterListView:

d:DataContext="{Binding SelectedItem, ElementName=MasterListView}"

Now we can start laying out all kinds of elements inside the Grid to display the details of our movie. Take a look at the **Models 🡪 Movie.cs** class to see what the various properties are named.

### Runtime data

We’ve now added the design time data, but when running the app, nothing is displayed. For now, let’s re-use the static data in the app to display data at runtime, by adding a **ViewModel** to our page’s code-behind file:

public BindableBase ViewModel { get { return ViewModelLocator.Main; } }

We also need to hook up the ViewModel to root node of our **MainPage.xaml**:

DataContext="{Binding ViewModel, RelativeSource={RelativeSource Self}}"

Lastly, when the page is loaded, we need to load the data from our **StaticData.json** file. We’ll hook into the MainPage’s Loaded event for this:

Loaded += MainPage\_Loaded;

private void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

ViewModelLocator.Main.LoadMovies();

}

The implementation of the LoadMovies() method inside our MainViewModel.cs will look like this:

public async void LoadMovies()

{

var staticDataFolder = await Package.Current.InstalledLocation.GetFolderAsync("Data");

var staticData = await staticDataFolder.GetFileAsync("StaticData.json");

var staticDataContent = await FileIO.ReadTextAsync(staticData);

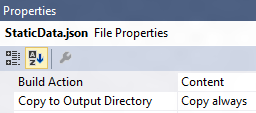
var staticDataSource = JsonConvert.DeserializeObject<StaticDataSource>(staticDataContent);

Movies = new ObservableCollection<Movie>(staticDataSource.Movies);

}

Note that we need an **ObservableCollection<Movie>** property as well, to hold our movie collection which we can bind against.

Make sure the **Build Action** of the **Data 🡪 StaticData.json** file is set to **Content**:



## The “add” button

You may have noticed that the **MasterListGrid** element on the page has two RowDefinitions. You can use the second row to display a **CommandBar**, which holds **PrimaryCommands** and **SecondaryCommands**. Add an **AppBarButton** to the PrimaryCommands to display an “add” button, which can display additional UI to prompt the user for a movie name. An example of doing this can be found in the **H01\End** solution (tip: you can use a Flyout to display “pop-up” UI and add it as a child to the **AppBarButton.Flyout** node.

## Adapting to screen/window size changes

To respond to these changes, we’ll be using the new **AdaptiveTriggers** to listen to specific window sizes. These can hold any value, but for this lab, we’ll be using 0, 1024 and 1366. Add the following XAML to the **MainPage.xaml** as a child of the main Grid node:

<VisualStateManager.VisualStateGroups>

<VisualStateGroup>

<VisualState x:Name="Full">

<VisualState.StateTriggers>

<AdaptiveTrigger MinWindowWidth="1366" />

</VisualState.StateTriggers>

<VisualState.Setters>

</VisualState.Setters>

</VisualState>

<VisualState x:Name="Medium">

<VisualState.StateTriggers>

<AdaptiveTrigger MinWindowWidth="1024" />

</VisualState.StateTriggers>

<VisualState.Setters>

</VisualState.Setters>

</VisualState>

<VisualState x:Name="Narrow">

<VisualState.StateTriggers>

<AdaptiveTrigger MinWindowWidth="0" />

</VisualState.StateTriggers>

<VisualState.Setters>

</VisualState.Setters>

</VisualState>

</VisualStateGroup>

</VisualStateManager.VisualStateGroups>

Inside the **VisualState.Setters** we can reference specific elements and properties and set their value. For example, adding the following as a setter in the MinWindowWidth=”0” node:

<Setter Target="MovieSearchBox.Visibility" Value="Collapsed" />

will result in the wide search box being hidden when the window/screen size is below 1024 in width.

At this point, feel free to expand the app with functionality as you see fit and take a look at the **H01\End** solution to see an example of an implementation that is more fleshed out.