# WINDOWS 10 DEVELOPMENT

falafel.com

# Windows 10 Development: Table of Contents

In this book we'll dive into some of the basics you'll need to build real-world applications, such as newly updated Falafel 2 Go for Windows 10 application. We'll look at the following topics:

## About the Author

**Josh Morales** loves all things Microsoft and Windows, and develops solutions for Web, Desktop and Mobile using the .NET Framework and everything in the Microsoft Stack. He has several apps in the Windows and Windows Phone stores, and also works with the Apps for Office API to create custom solutions for Microsoft Office and Office 365. His other passion is music, and in his spare time Josh spins and produces electronic music under the name DJ SelArom.

Blog: http://blog.falafel.com/author/josh-morales/

Twitter: **@SELAROMDOTNET**

**Credits:**

Editor: Noel Rice

Designer: Matt Kurvin

# About Falafel Software

Falafel Software Inc., an 8-time Microsoft Gold Certified partner, has been helping companies and individuals around the world deliver more than expected with the Microsoft Platform and Technologies over the past 13 years.

Organizations looking for help with the Microsoft Platform need to look no further than Falafel Software. Having developed multiple apps using the Microsoft Platform, our name is synonymous with Micorsoft excellence and we currently have multiple Microsoft MVP's and certified Microsoft Developers.

Contact us to discuss your specific

## Microsoft Mentoring

When training isn't enough. Falafel Software's mentoring packages are 20 or 40 hour blocks of flexible consulting hours you can use as you wish. Spend time with a Xamarin expert in an online conference learning the best practices for your specific project or have your mentor help build your solution. Learn more about Microsoft Mentoring.

## Other Resources

Falafel Software's team of tech experts are always exploring new technologies and pushing the limits of Xamarin. Visit http://blog.falafel.com/ for in-depth technical blogs sharing our experiences.

# Introduction

The long-awaited Windows 10 release is finally here, bringing a whole new level of interactivity, security, productivity and entertainment. And with it comes a whole a whole new App Development model designed to unify and enhance the developer (and ultimately the user) experience.

As an avid fan of everything Microsoft, I'm excited to start this new journey, and in addition to sharing my enthusiasm for the platform, I thought I would go a step further and share my developer experience.

We begin by looking at the tools needed to get started, some helpful resources you'll need along the way, and create the obligatory "Hello, World" project to kick things off. From there we'll review many of the different building blocks you need to build the infrastructure for your app, including handling navigation and managing state.

We'll also look at a few of the new controls available to Windows 10 developers and how we used them in our demo showcase app "Falafel2Go".

Finally, we'll look at some of the strategies for adding responsive UI to your apps, including building custom containers and triggers to automatically adapt the UI to match the device and screen size.

## Source Code

The full source code of the sample projects created in this book are available on the Falafel Software Github repository: https://github.com/FalafelSoftwareInc/Windows10Development

This repo is updated as new samples are created for new blog posts and chapters, so be sure to watch this repo for updates!

# Hello, Windows 10: Getting Started

As we learned (with much excitement) at Build 2015, there are actually many different strategies to building apps for Windows 10, including JavaScript via PhoneGap and WinJS. There is also Project Westminster which serves as a bridge between Web Apps and Windows 10. You will soon even have the ability to compile your iOS and Android apps to run on Windows 10.

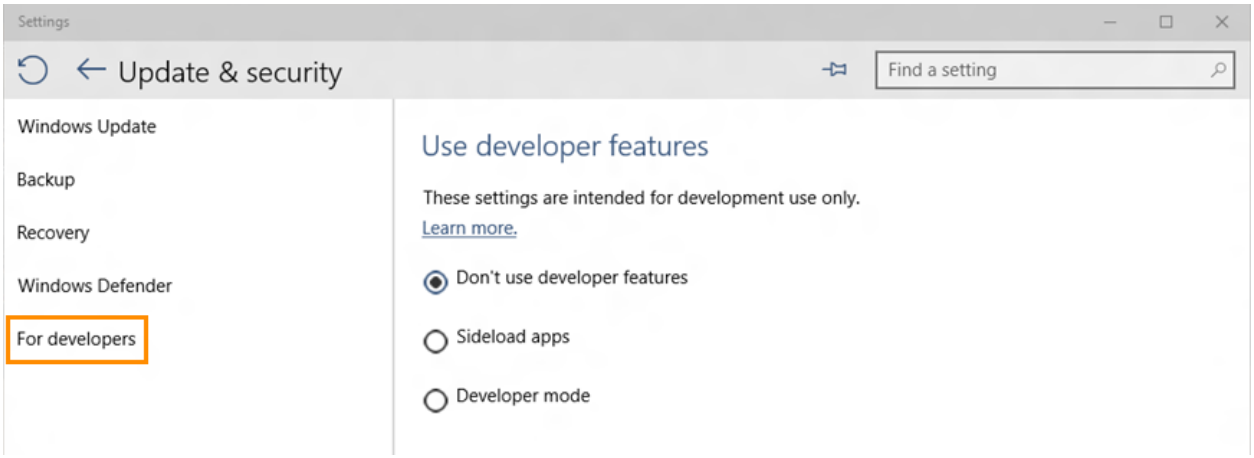
## XAML / C#

Although I do hope to explore and write about all of these, my focus will be exclusively on the XAML/C# story. But of course, stay tuned and if there are any topics or questions you'd like me to explore, be sure to tweet me @SelAromDotNet or get in touch with us.

## Getting Started

To build apps for Windows 10 you'll need at the very least the following:

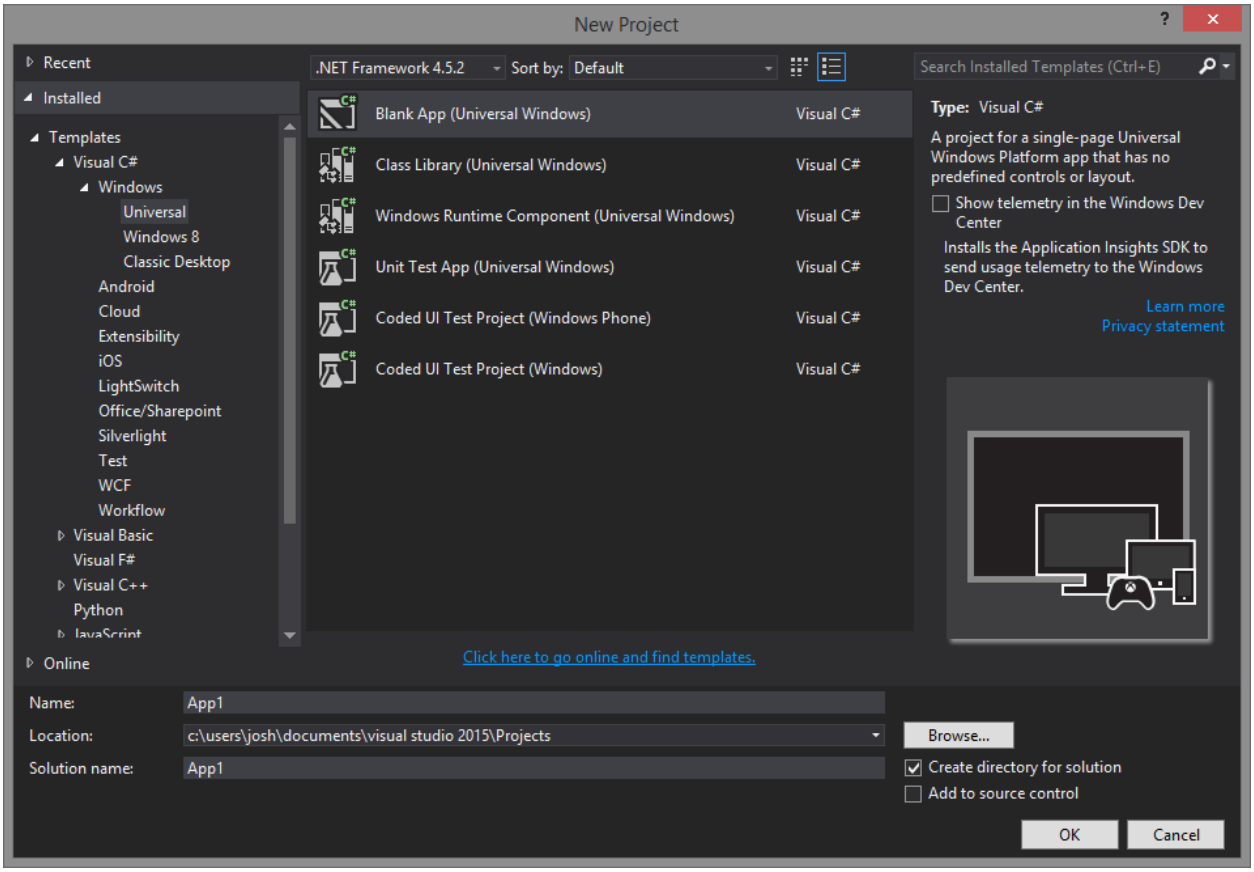
- Windows – Apps for Windows 10 can be built with Windows version as far back as 7 via remote debugging and emulators; however for the very best developer experience, it is recommended to be running Windows 10 Pro
- Visual Studio 2015 – During installation it's important to select the option to include "Universal Windows App Development Tools" so the appropriate tools are installed. Otherwise you can install them separately for Windows 10 here: Windows Software Development Kit (SDK) for Windows 10
- You also want to make sure your PC is enabled for development, which is in the Settings section under Update & Security:




More details on developer mode are here: Enable your device for development

## Hello, Windows 10

One Visual Studio is installed with the appropriate tools, you can create a new Windows Universal App from the C# > Windows Templates:



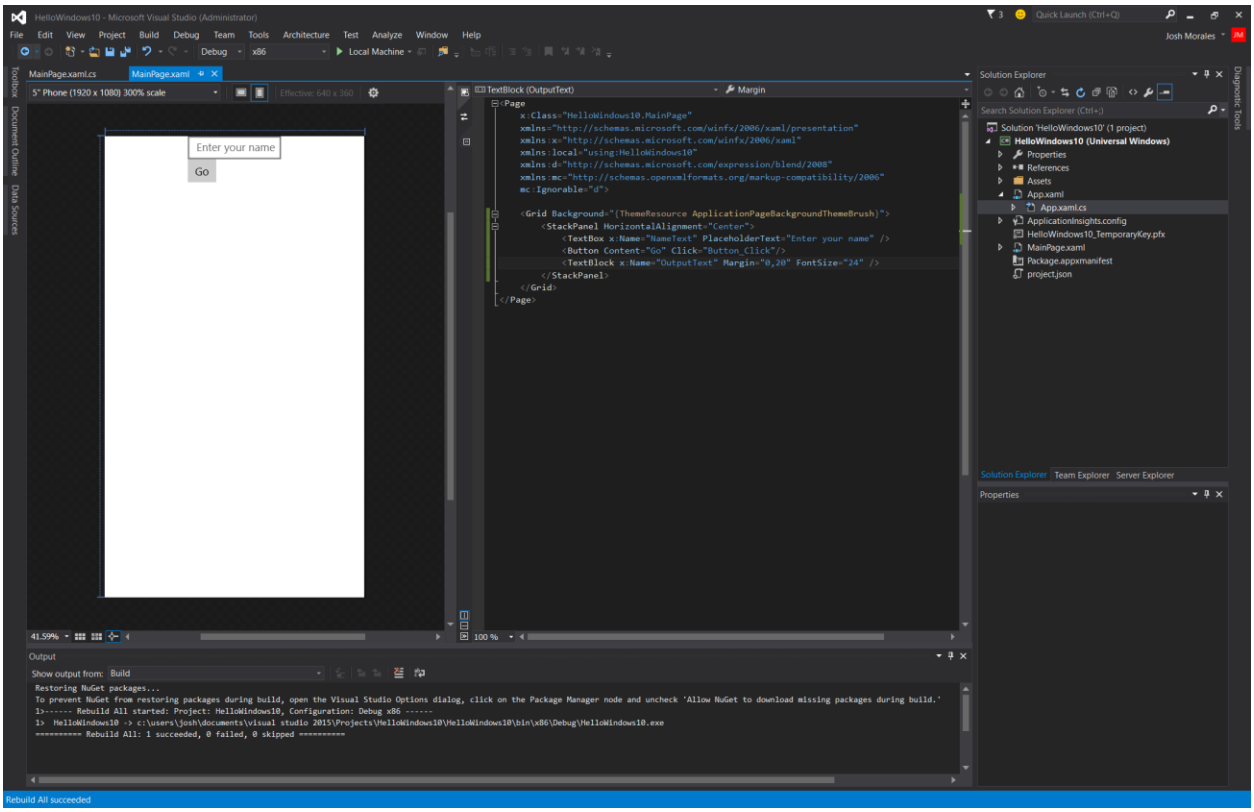

While previous versions of SDKs for Windows/Phone development included many different start templates, Windows 10 currently only includes the Blank App template. This means in addition to your app design, you are responsible for ensuring that you take the appropriate steps to handle navigation and state.

We'll explore some strategies to achieve this in a future chapter, but for now let's just use the Blank App for C# and create the project.

Visual Studio generates a simple project with a single page, which we can update with a bit of XAML to create the "Hello, World" experience:

```
<Page
    x:Class="HelloWindows10.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:HelloWindows10"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <StackPanel HorizontalAlignment="Center">
            <TextBox x:Name="NameText" PlaceholderText="Enter your name" />
            <Button Content="Go" Click="Button_Click" />
            <TextBlock x:Name="OutputText" Margin="0,20" FontSize="24" />
        </StackPanel>
    </Grid>
</Page>
```

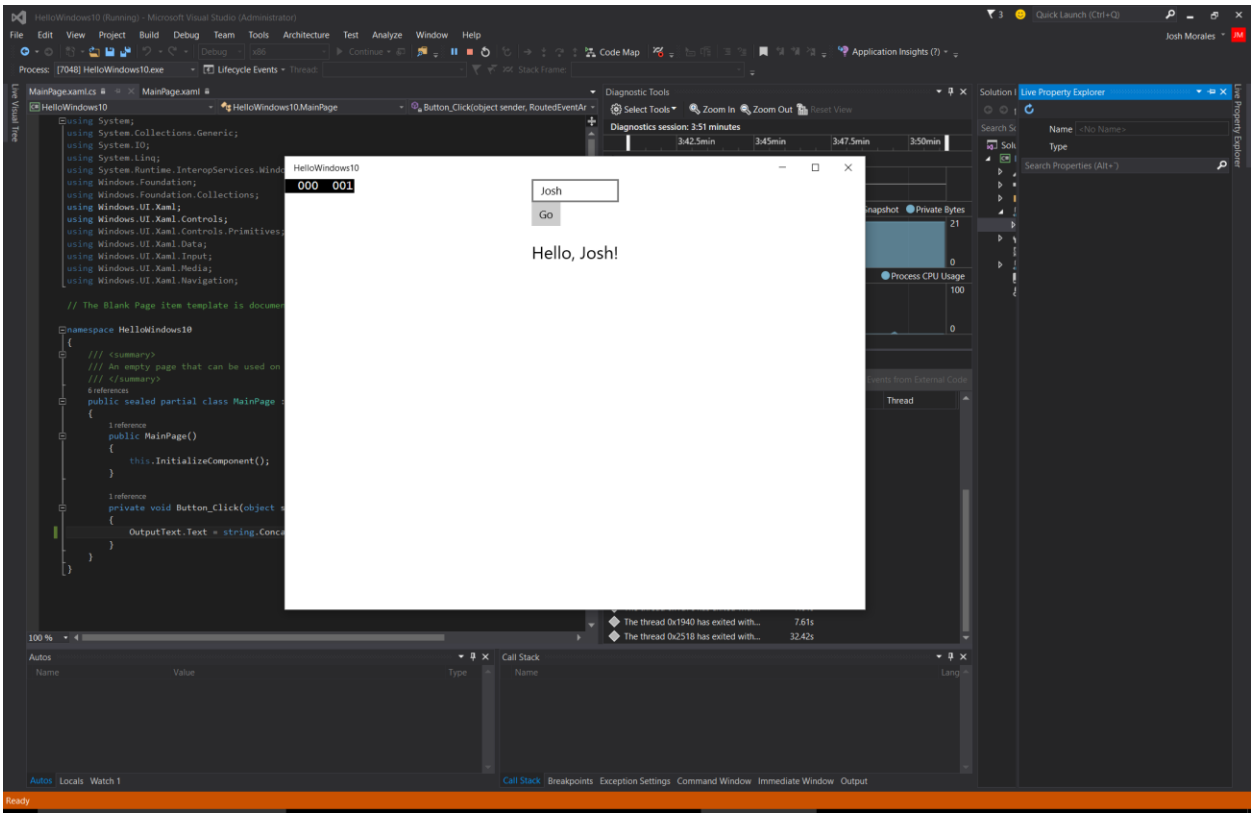
If you're running Visual Studio 2015 in Windows 10 you'll see your changes in the designer immediately:



Switch to the code-behind to complete the event handler for the Button click, which displays the appropriate message to the user:

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
       OutputText.Text = string.Concat("Hello, ", NameText.Text, "!");
    }
}
```

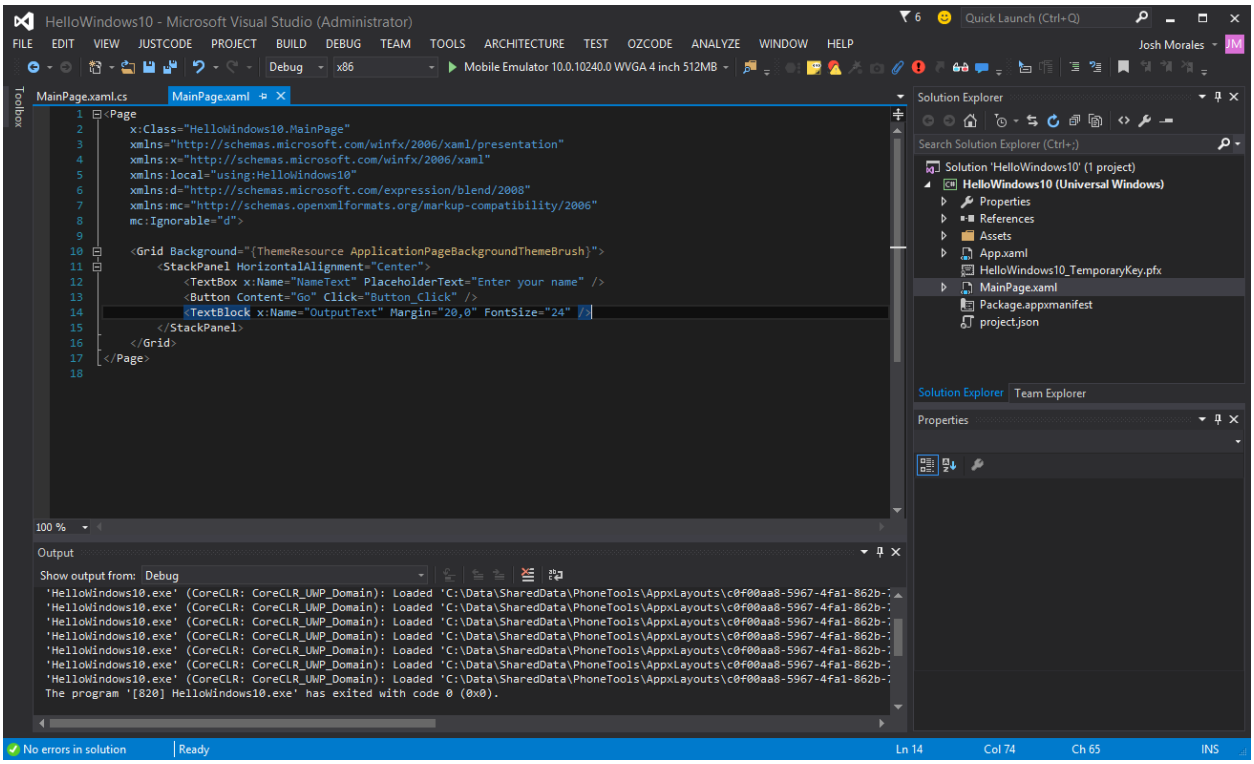
If you are already running on Windows 10, you can simply hit F5 to run the project on the local machine, where you should get the following screen and can enter your name to see the welcome message.




Of course, since this is a Windows Universal App, the exact same code can now run on the Mobile device family (Windows Phone). You can verify this by switching the device target from Local Machine to one of the installed emulators (or attached device).

Running the project deploys the app to the emulator or device revealing the same interactive message that behaves identically to the desktop:


Note that although you can develop Windows 10 apps on previous versions of Windows, you don't get any designer support and can only edit the XAML directly.



Since obviously Windows 10 apps won't run on previous versions of Windows, it's also impossible to launch the apps directly on your machine. However, you can use remote debugging, the emulator, or an attached device by selecting the (more limited) options in the device target switcher:


Finally keep in mind that you must be running at least Windows 8.1 Professional (not Home) on a PC that supports Client Hyper-V and Second Level Address Translation (SLAT) in order for the emulators to run. If you do not have such a setup, you can use remote debugging or attach a physical device to the PC.

## Recommended Resources

There are a lot of great resources available for learning more about Windows 10 development. Here are some of my favorites:

- MSDN Documentation – Obligatory, but the How To section is especially helpful in finding quick answers to common tasks
- Build 2015 – Recordings on Channel 9 from the premiere event
- Microsoft Virtual Academy – Jerry Nixon and Andy Wigley hosted a three day event all about developing for Windows 10. Catch the must-see recordings here
- Windows 10 Code Samples – Github repo full of sample code and projects demonstrating a wide range of functionality.
- Blend Jump Start – Another MVA Course; if you are entirely new to Windows Development you definitely want to take time to learn what Blend can do for you. Though this course targets VS2013, many if not all of the concepts will certainly transfer to VS2015.

Finally, of course, remember that Falafel is always here to assist, and through our consulting, mentoring, and training services we can help take your Windows 10 project to the next level

## Wrapping Up and Next Steps

Now that we've got the obligatory "Hello, World" project under our belt, we can begin to look into what's involved in building a real world app. In the following chapters, we'll explore more useful topics like State Management, Navigation, MVVMLight and more.

# Upgrading from Preview SDK to Release

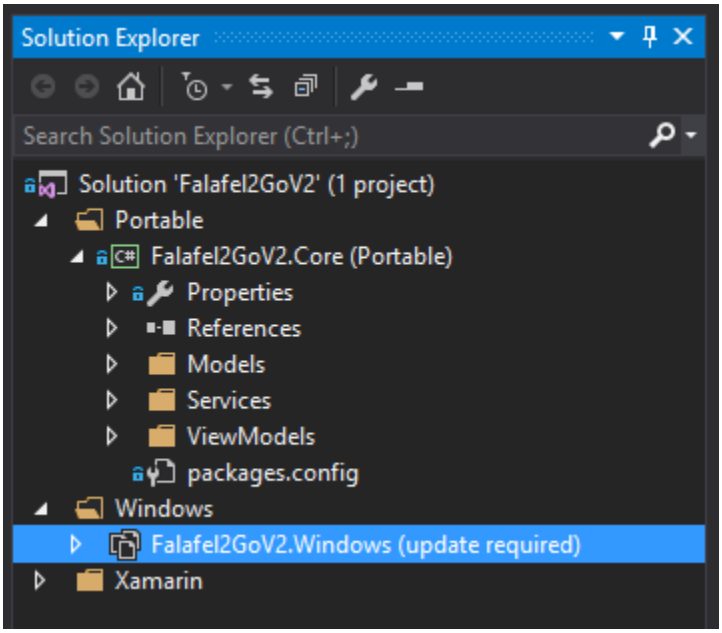
Before diving into the main parts of Windows 10 Development, there may be some of you (like me!) that were already building apps with the preview SDK for Windows 10. If you attempt to open a preview solution with the final version of the SDK installed, you will very likely have issues (as I did!).

In this quick bonus chapter, I'll cover just a few of the issues I had to fix to get my project updated and running with the latest releases of both Windows 10 and Visual Studio 2015.

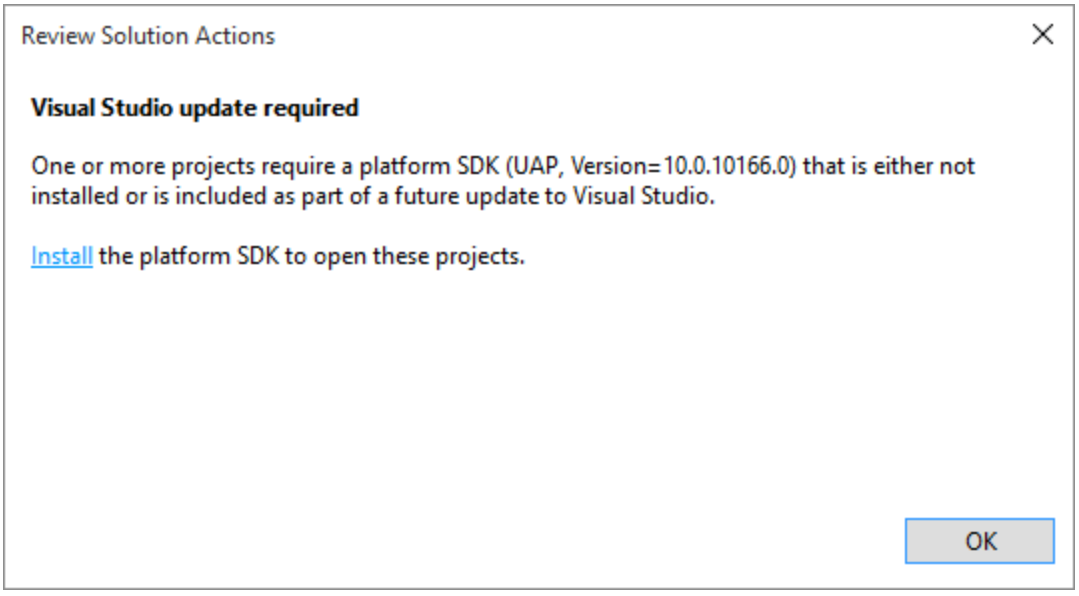
Note that your mileage may vary, as every project is different, but this is what worked for me, and hopefully it can help others facing similar issues.

## Project Error: Update Required

If you did what I did and uninstalled the previous preview SDK before installing the final version, the first issue you're likely to face is the Solution Explorer unloading your project with the message: update required.

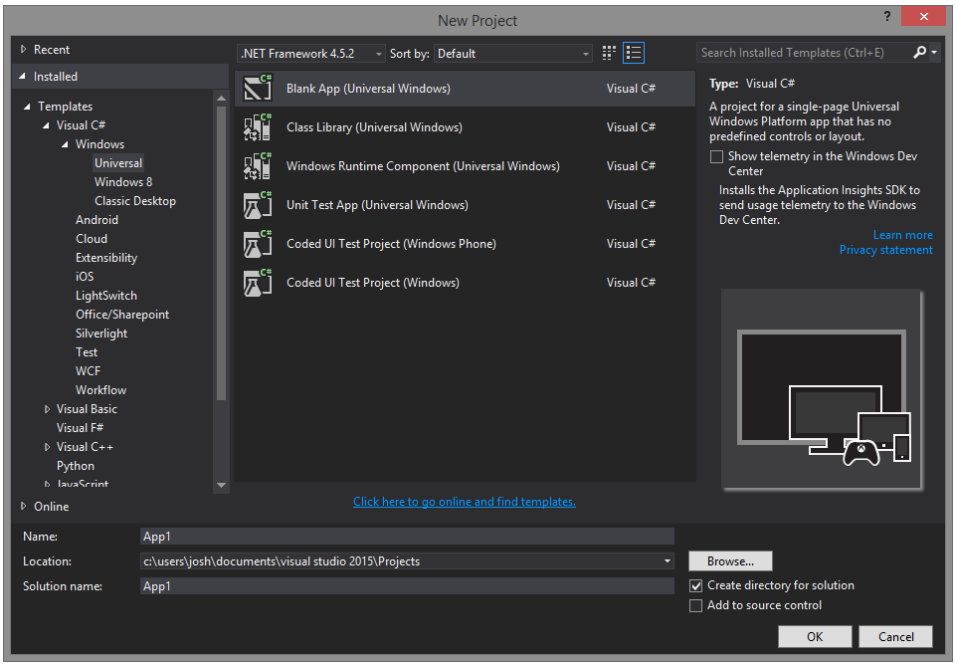



If you attempt to reload the project in the solution explorer, you'll get a dialog message confirming that you no longer have the version of the SDK required to open the project:



Now, even if I wanted to install the previous SDK again (which I don't!) the link in the dialog no longer appears to be valid as it takes you instead to the Microsoft home page. This isn't unexpected as the preview SDKs are clearly obsolete.

Another option would be to simply create a new project in Visual Studio, which would of course use the final installed SDK to generate the project, ensuring it will run on the release version of Windows.




However, the Falafel2Go project I'm working on has gone through a lot of changes, and is already checked into our Git repo, and I didn't want to go through the entire process of rebuilding the project from scratch.

Instead, I decided to create a new blank project, and compare its contents to my solution, updating mine to match the final version as much as possible. Here were the changes required to get my project running.

## Update the Windows Project File (csproj)

Comparing my C# project file (csproj) with the one in the blank project revealed several differences, the key of which were two properties for TargetPlatformVersion and TargetPlatformMinVersion.

While my project file had the preview SDK version:

```
    <TargetPlatformVersion>10.0.10166.0</TargetPlatformVersion>
    <TargetPlatformMinVersion>10.0.10166.0</TargetPlatformMinVersion>
```

The Blank app referenced the release version of the SDK:

```
    <TargetPlatformVersion>10.0.10240.0</TargetPlatformVersion>
    <TargetPlatformMinVersion>10.0.10240.0</TargetPlatformMinVersion>
```

Saving this change will let Visual Studio know you're using the latest installed SDK, and at this point you would be able to reload the project in the Solution Explorer, but WAIT!

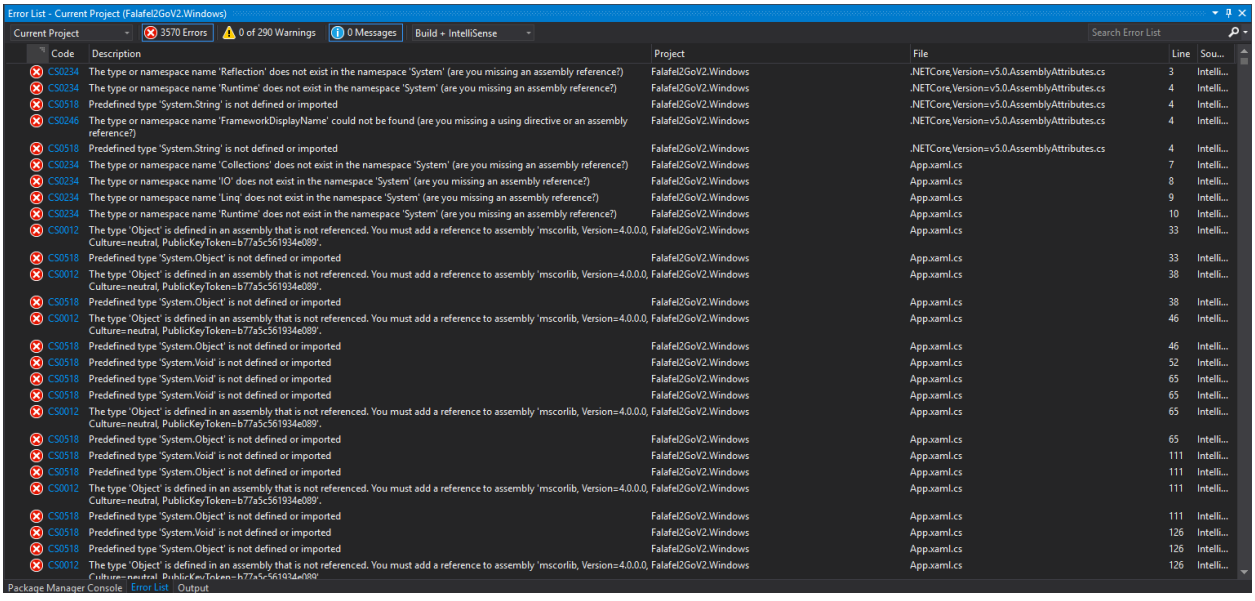
Another discrepancy I only painfully later discovered is the fact that the generated app has a reference to the project.json file and associated project.lock.json file. These are special files that list all the dependencies and frameworks that a project references:

```
  <ItemGroup>
    <!-- A reference to the entire .Net Framework and Windows SDK are automatically
included -->
    <Content Include="ApplicationInsights.config">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
    <None Include="project.json" />
  </ItemGroup>
```

Do not miss this one! Copy it to the project folder and add it as a reference to the project file.

Although I now slightly recall hearing about this file before, at the time I assumed it was a reference sample data, and thus did not include it when updating my.csproj file. This resulted in a *massive* list of framework errors, such as:

> Predefined type 'System.Object' is not defined or imported
>
> Predefined type 'System.Void' is not defined or imported
>
> The type 'Object' is defined in an assembly that is not referenced. You must add a reference to assembly 'mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'
>
> The type or namespace name 'Runtime' does not exist in the namespace 'System' (are you missing an assembly reference?)




I was clearly referencing the correct framework in the project, and any error list that extensive has to be a configuration issue (I would hope!), and after painstakingly inspecting every file in the project I finally realized that I'd skipped a critical section.

So again, don't skip this the reference to project.json and the project.lock.json files when updating your .csproj file!

Finally, you also want to make sure you update the definitions for the Debug Constants. While the preview used “WINDOWS_UAP” the release SDK appears to expect “WINDOWS_UWP”:

```
<DefineConstants>DEBUG;TRACE;NETFX_CORE;WINDOWS_UWP</DefineConstants>
```

At last I was able to correctly open the project in Visual Studio, but even after updating the reference to project.json and project.lock.json you might still have the crazy list of errors I saw. This is because the project is still using the preview bits for all of the references.

But before we can update them, we have to also update the manifest to reference the release SDK as well.

## Update the App Manifest File (Package.appxmanifest)

Sure enough, while my app manifest file had a reference to a specific SDK version:

```
<Dependencies>
    <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.10069.0"
MaxVersionTested="10.0.10069.0" />
</Dependencies>
```

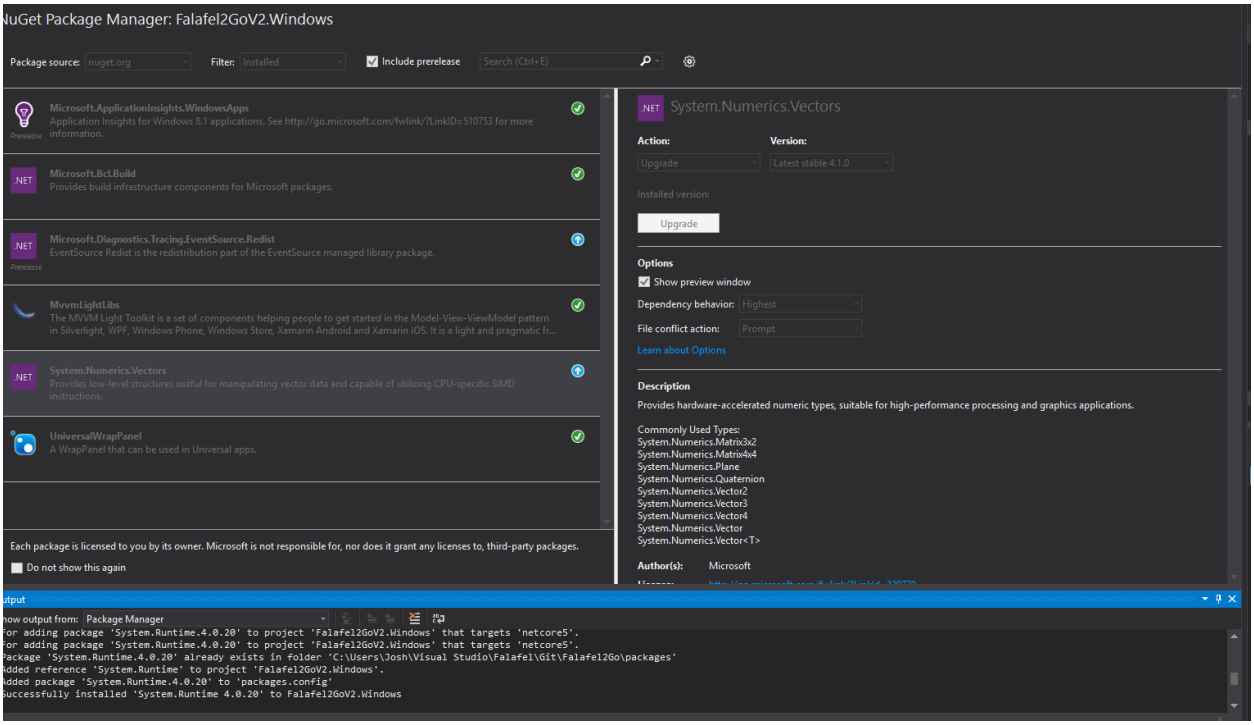
The Blank App simply referenced Windows 10:

```
<Dependencies>
    <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.0.0"
MaxVersionTested="10.0.0.0" />
</Dependencies>
```

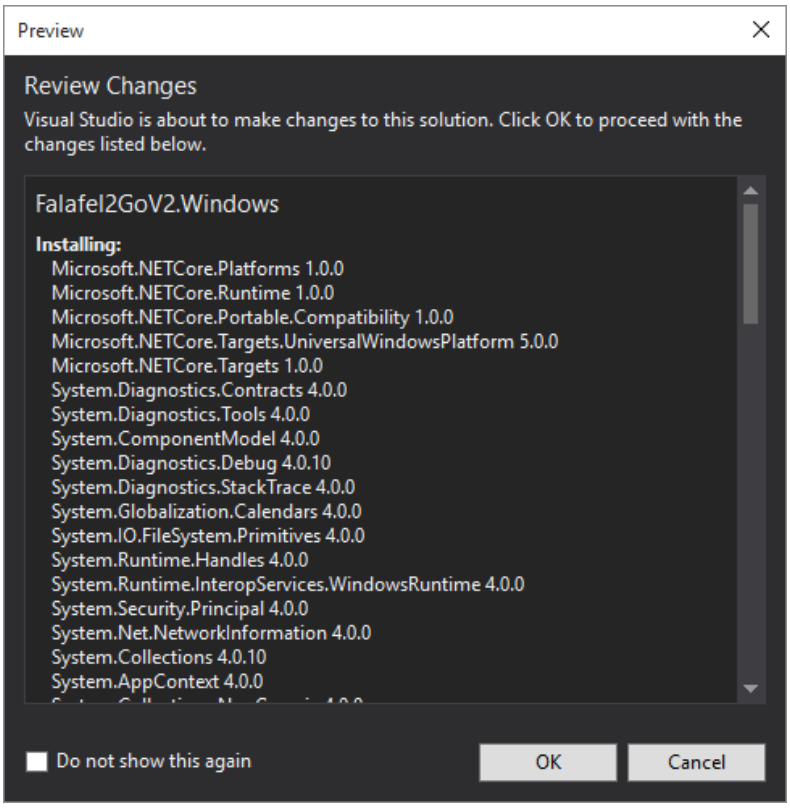
I updated it to match and proceed to update the NuGet Packages.

## Update NuGet Packages, Remove Preview References

Next I proceeded to upgrade all the NugetPackages in my project to the latest versions with the nifty new Nuget Explorer:




Although my project did have the correct reference to the Windows platform, I noticed that the Blank App had an additional Nuget Package [Microsoft.NETCore.UniversalWindowsPlatform] installed, and apparently includes ALL the goodies and dependencies of the platform, so I added it to my project as well.



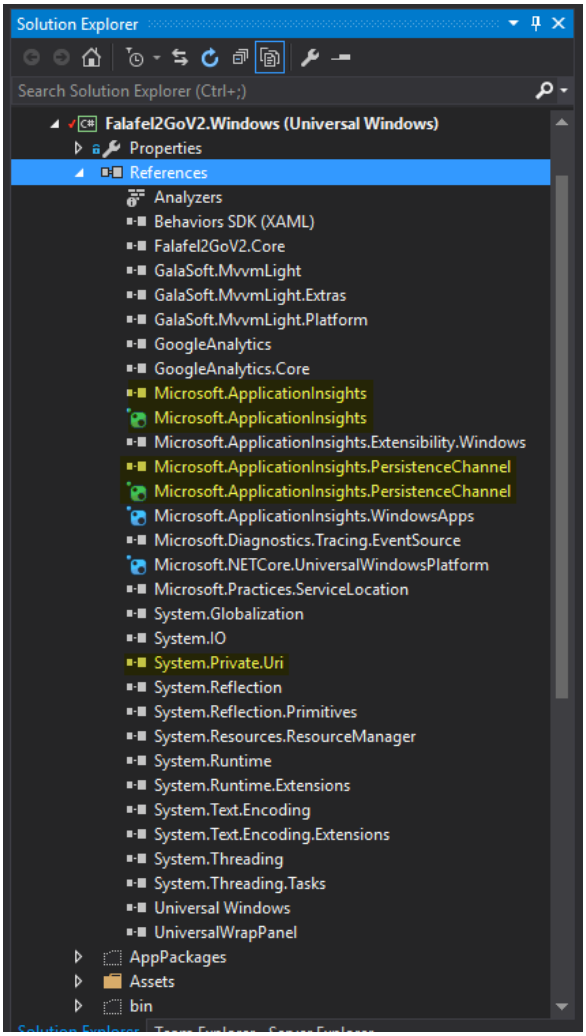

**Note:** if you didn't add the project.json reference, this step will probably NOT work for you as it certainly didn't for me, give you the error message:

> Could not install package 'Microsoft.NETCore.Platforms 1.0.0'. You are trying to install this package into a project that targets '.NETCore,Version=v5.0', but the package does not contain any assembly references or content files that are compatible with that framework. For more information, contact the package author.

This is especially frustrating since the project was clearly configured for .NET 5 and resulted in a lot of wasted time… So just to drive the point home one last time: don't forget to add the project.json and project.lock.json files!

If all else fails, remove EVERYTHING via NuGet, and start with the UniversalWindowsPlatform package, adding your packages one by one until you're back where you need to be.

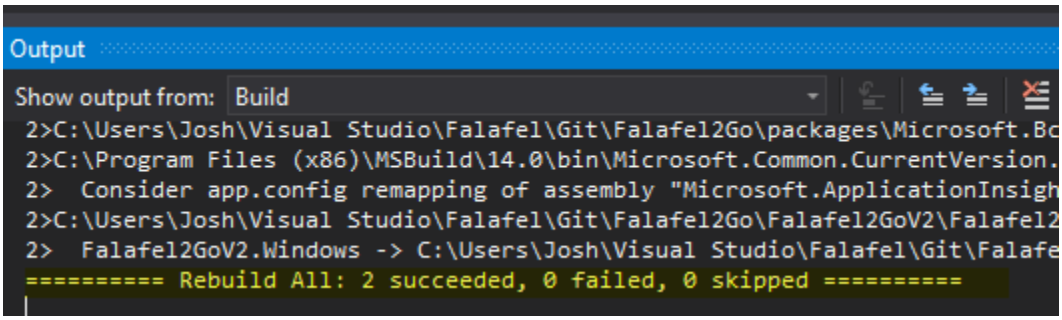
Finally, when you're done, you may end up with duplicate references to some libraries from the older SDK and NuGet packages, as well as references to libraries that have been rolled into the UniversalPlatformLibrary:




This will result in errors like:

> The type 'Uri' exists in both 'System.Private.Uri, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a' and 'System.Runtime, Version=4.0.20.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a'

Simply remove the extra references in the Solution Explorer, and at long last you should be good to go!



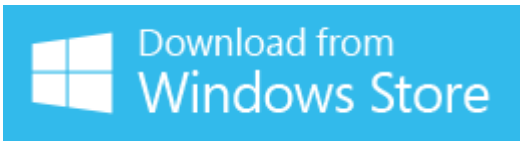

## Wrapping Up and Next Steps

Depending on your project, the steps outlined here may not get you 100% updated to the release SDK, but hopefully sharing my experience will help others who have invested considerable development time in the Preview SDK to avoid having to start fresh with the release.

Now that we've got our demo project updated and ready to go we can dive into the good stuff about Windows 10 development.

# Introducing Falafel2Go for Windows 10

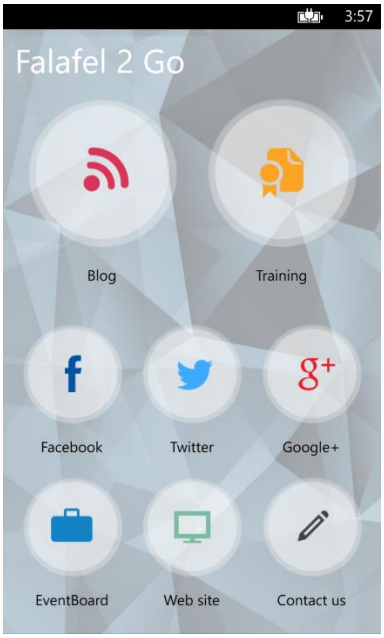
Now that Windows 10 is live, it made sense for Falafel to dive right in, updating our showcase demo app Falafel2Go to take advantage of the new features of the platform. Our previous update was built using Xamarin, and with just a few changes, we were able to update it to run on Windows 10.




We'll be using this app as a guide and demo to explore some of the cross-platform strategies in a future publication, but you can download it today and try it out!

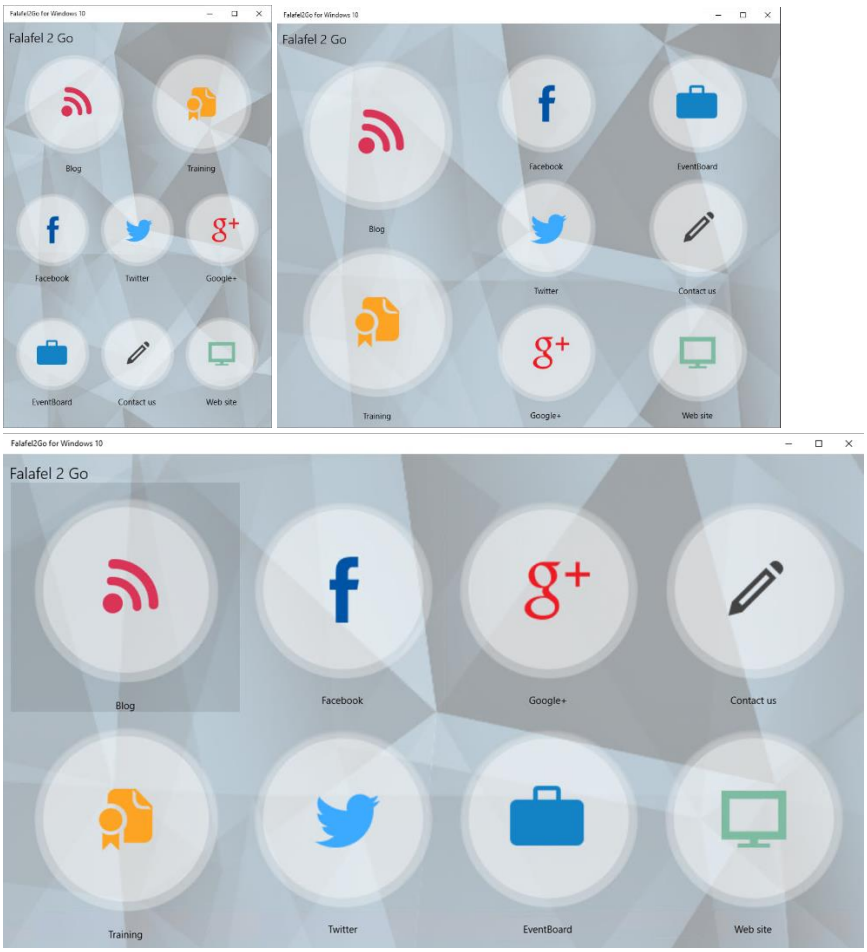
## Responsive UI

One of the key updates to the application is how it takes advantage of the new responsive controls available in the platform. The previous version of the app lays out the startup tiles in a fixed grid, which doesn't support a landscape orientation and certainly doesn't support other device types and sizes like a desktop monitor.





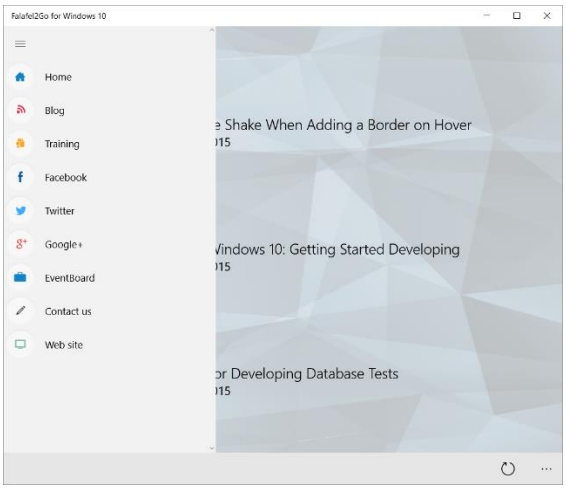
The new Windows 10 RelativePanel control now allow us to easily create a fluid, responsive layout that automatically adjusts to fill the screen with a layout that makes sense, re-flowing the content to accommodate a wide assortment of screen sizes, shapes, and orientations.




And even though Windows 10 Mobile is a few months away, it looks great in the emulator, sporting a snazzy new landscape view:

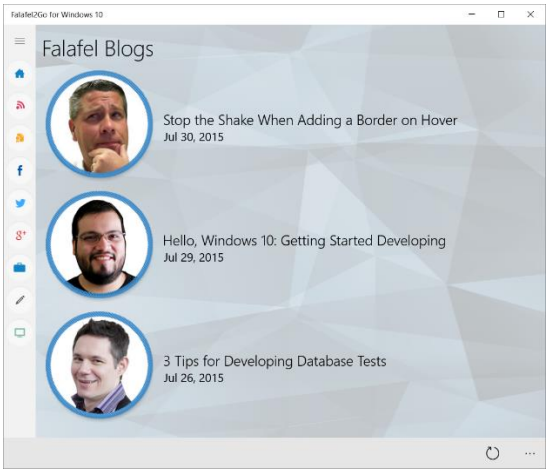

It even looks and runs great on my Windows 10 Mobile preview device (Lumia 925):


A new responsive navigation control is also made possible by the SplitView control. This new command menu allows you to jump to the different sections directly, instead of having to navigate back to the home screen.



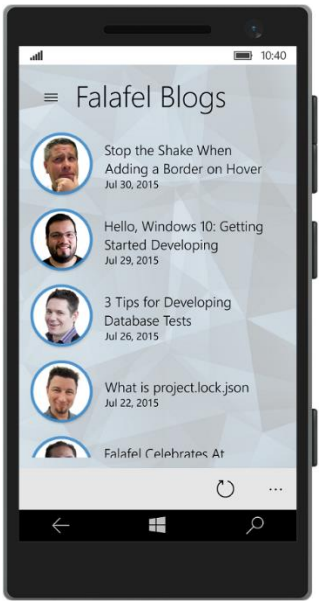

Since the control is natively responsive, we also can take advantage of the VisualStateManager and its support for custom StateTriggers to transform the control to accommodate different screen sizes.

For example, on the desktop, in a medium size we see the menu:

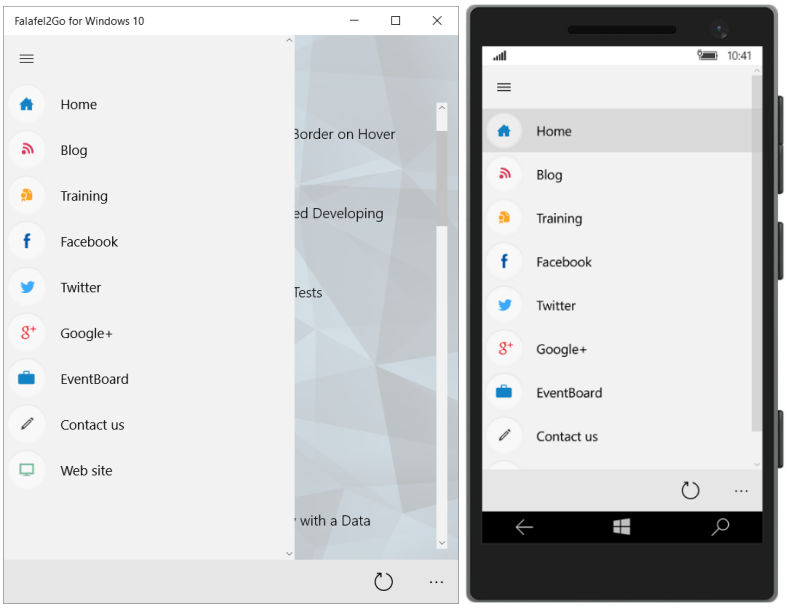



If we reduce the size of the app, the navigation menu collapses to an icon:

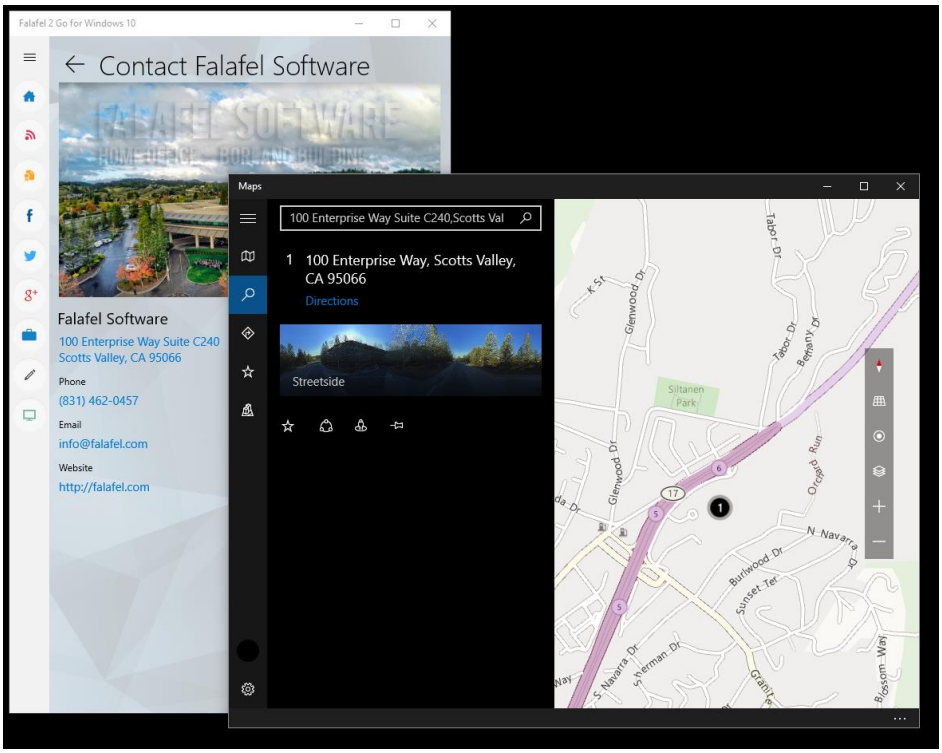

And on the phone, which also has a more limited real estate, we see a similar view:




Clicking the menu button reveals the menu, which renders appropriately to match the platform on which it is running:




Finally, the contact screen takes advantage of the Uri schemes so that regardless of the device, you can launch the appropriate app to get in touch or see where we are located. For example, here's the maps app launched on Windows 10 Desktop after clicking our address:




## Wrapping Up and Next Steps

As part of our journey through Windows 10 Development, we'll see how we were able to utilize the new controls, as well as create new ones to achieve the final layout of the new version of Falafel2Go. We'll see how VisualStates and custom triggers can help make your app responsive, and tailor the experience to any device and screen size.

In the meantime, head over to the Windows Store and install Falafel2Go for Windows 10, be sure and rate/review the app to share your feedback!

# Getting Started with the MVVM Light Toolkit

In the next few chapters we'll dive into the basics of using the MVVM Light Toolkit with Windows 10 to build a simple project two-page with the MVVM design pattern. In this chapter we'll show how to setup the toolkit in a Windows 10 project, and some of the basic components you'll need to define to follow the MVVM pattern.

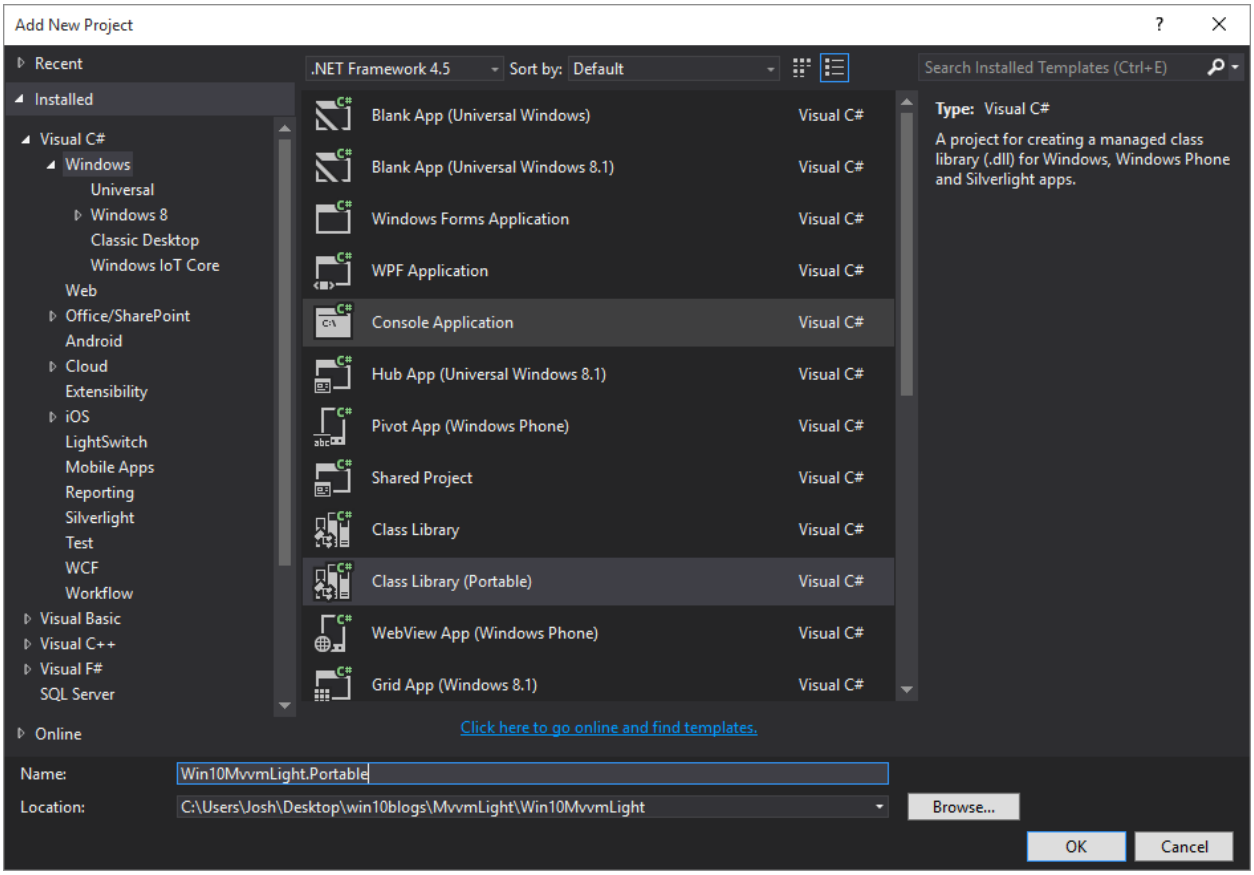
## MVVM

A complete discussion of the MVVM pattern is outside the scope of this document, so if you are new to the concept and want to learn more I suggest taking a look at articles like this one on MSDN: The MVVM Pattern and this blog on using MVVM with WPF: WPF Apps With The Model-View-ViewModel Design Pattern, the concepts of which are certainly still applicable to Windows Store Apps.

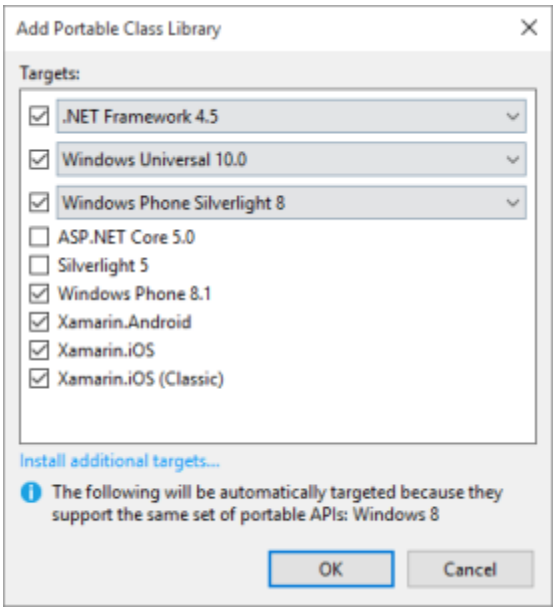
Many implementations of the MVVM pattern are available, such as Prism, and some developers even choose to roll their own framework. However, for this and all future chapters, we'll be using the MVVM Light Toolkit.

## The MVVM Light Toolkit

The MVVM Light Toolkit is an open-source framework for building applications using the MVVM Pattern. Originally available for Silverlight, Windows Store and Windows Phone, it has since been expanded to cover cross-platform devices including iOS and Android via Xamarin. Although we'll dive deeper into supporting other platforms in a future publication, we'll get an early start by leveraging support of MVVM Light for Portable projects, and add a seperate library to contain the Models, ViewModels, and other code we will eventually share with other platforms.




Note that we've selected the ***Class Library (Portable)*** template, which prompts us to select the appropriate targets.

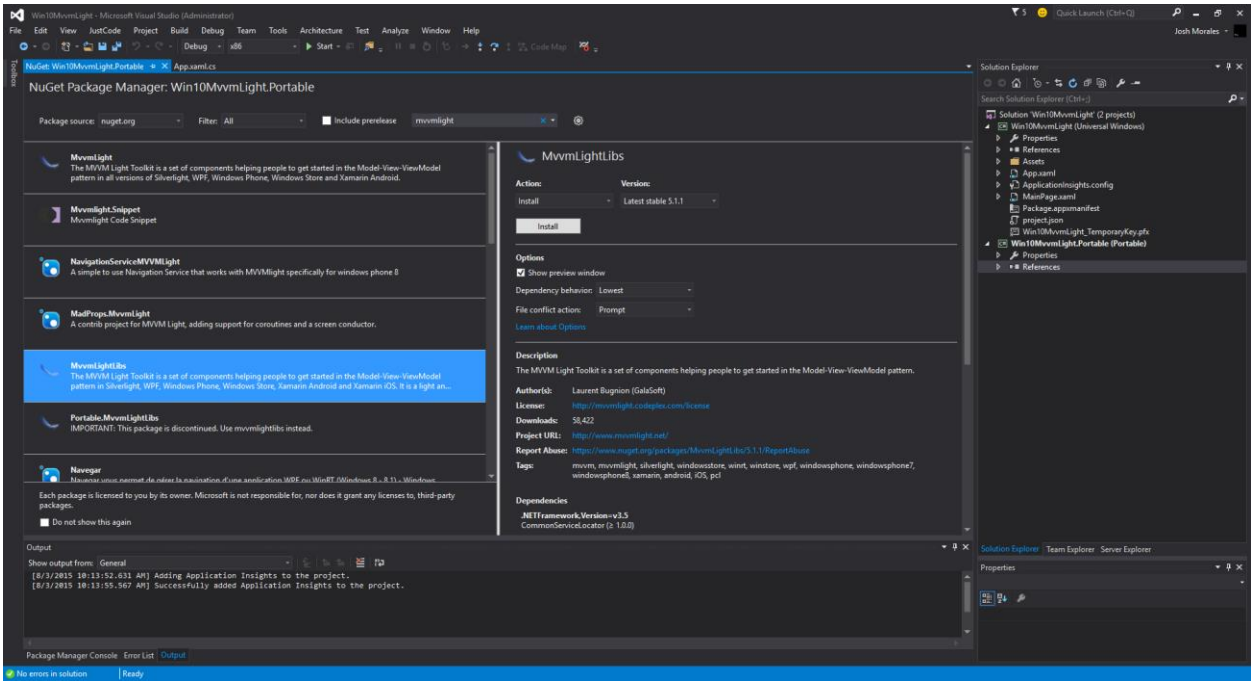



Since the portable project condenses all the APIs to the minimum supported version, selecting Windows 10 as a target actually also allows you to target Windows 8 (and of course, 8.1), meaning that if you wanted to cover these platforms while the Windows 10 roll-out continues, you are certainly able to do so!

The same goes for Windows Phone, and as you can see in the screenshot above, our project can also support running on Android and iOS via the Xamarin Platform.

## Installing MVVM Light via NuGet

Installing MVVM Light is actually quite simple thanks to NuGet. Simply open the NuGet Explorer and search for "mvvmlight", installing the package named "MvvmLightLibs" to both the main app and the portable project.




This registers the portable portions of the project (such as ObservableObject and ViewModelBase) into all projects as well as the platform-specific implementations (such as the NavigationService we'll be looking at in a later chapter) into the Windows project. Any time you want to target an additional platform, be sure to install this package to that project as well to enjoy all the benefits of MVVM Light on that platform!

Now that we have the required libraries in place, we can proceed to build the infrastructure for the app by adding a simple Model, ViewModels, and Views.

## Create a Sample Model

The first thing we need is a sample data model to represent a simple item to display on the page. In a real-world project this might be a blog post served from an RSS feed or a To-do item from an external API. For now, we'll just declare a simple item to display a few properties by adding the following model class to our portable project:

```
public class TestItem
{
        public int Id { get; set; }
        public string Title { get; set; }
        public string Subtitle { get; set; }
        public string HexColor { get; set; }
}
```

## Create Pages (Views)

Next we need to actually display this information in the app, which we'll do by adding some pages in the Windows project. We already have the blank MainPage that was added by default (which we've moved into a Views folder), so we'll simply add a second page with some placeholder controls to display the model.

```
<Page
    x:Class="Win10MvvmLight.Views.SecondPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Win10MvvmLight.Views"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel Background="{Binding SelectedItem.HexColor}" Padding="20">
        <TextBlock Text="Second Page" Style="{ThemeResource TitleTextBlockStyle}" />
        <TextBlock Style="{ThemeResource HeaderTextBlockStyle}" Foreground="White" />
        <TextBlock Style="{ThemeResource SubheaderTextBlockStyle}" Foreground="White" />

    </StackPanel>
</Page>
```

We'll look at the XAML we'll need for the MainPage in the next chapter, as well as how to wire the pages up to the properties of the model, but now that we've got both the model and views defined, let's fill in the last part of the pattern by creating some ViewModels to link it all together.

## Create the ViewModels

Generally, you want to make sure each View (Page) is associated with a specific ViewModel, so we'll create one for each of the pages in our app. We also want to make sure that we include common required behavior like the implementation of ***INotifyPropertyChanged***.

Fortunately, MVVM Light helps by including the ***ViewModelBase*** from which we can inherit all our ViewModels. In our case, we'll go one step further by putting a***BaseViewModel*** in between, so we can share additional global requirements like a helpful ***IsLoading*** property for showing or hiding a loading animation (which we'll demonstrate in a later chapter).

```
public class BaseViewModel : ViewModelBase
{
        private bool isLoading;
        public virtual bool IsLoading
        {
            get { return isLoading; }
            set
            {
                isLoading = value;
                RaisePropertyChanged();
            }
        }
}
```

Now we can proceed to create the MainPageViewModel, which presents a simple list of TestItem objects (which we'll populate in the next chapter). Note that we use the ***ObservableCollection*** as this allows the view to automatically be notified when the collection changes.

```
public class MainPageViewModel : BaseViewModel
{
        private ObservableCollection<TestItem> testItems = new
ObservableCollection<TestItem>();
        public ObservableCollection<TestItem> TestItems
        {
            get { return testItems; }
            set
            {
                testItems = value;
                RaisePropertyChanged();
            }
        }
}
```

We also need a SecondPageViewModel to show the selected item, exposed as a simple property:

```
public class SecondPageViewModel : BaseViewModel
{
        private TestItem selectedItem;

        public TestItem SelectedItem
        {
            get { return selectedItem; }
            set
            {
                selectedItem = value;
                RaisePropertyChanged();
            }
        }
}
```

Now that we have defined all of the Model, Views, and ViewModels required by our app, we need a way to wire them all together, so that each page is aware of its assigned ViewModel from which to retrieve its data.

There are several possible strategies, such as loading and assigning each ViewModel in the code-behind of each page. However, we will explore a strategy that not only makes this association more declarative, but also makes it easier to add design time data we can use while in Visual Studio and Blend (which we'll look at in the next chapter).

## Create a ViewModelLocator

A ***ViewModelLocator*** is a class that centralizes the definitions of all the ViewModels in an app so that they can be cached and retrieved on demand, usually via Dependency Injection. A full discussion of these topics also outside the

scope of this document, but this helpful answer on StackOverflow describes the pattern much better than I ever could, so definitely take a look if you're new to MVVM.

Long story short, the ViewModelLocator needs to setup the IOC provider, register each individual ViewModel, and finally expose those registered ViewModels for use by the rest of the application. This class, added to the portable project, gets the job done for our sample project:

```
public class BaseViewModelLocator
{
        public BaseViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

            SimpleIoc.Default.Register<MainPageViewModel>();
            SimpleIoc.Default.Register<SecondPageViewModel>();
        }

        public MainPageViewModel MainPage
        {
            get { return ServiceLocator.Current.GetInstance<MainPageViewModel>(); }
        }

        public SecondPageViewModel SecondPage
        {
             get { return ServiceLocator.Current.GetInstance<SecondPageViewModel>(); }
        }
}
```

The ***ServiceLocator*** is responsible for retrieving the ViewModel instances, using the ***SimpleIoc.Default*** implementation provided by MVVM Light. By registering them via the SimpleIoc.Default instance in the constructor, we can retrieve those instances from the Views via the public properties defined in the locator class.

We can then register this locator as a global static resource in App.xaml so it is available for all pages to use. These resources are automatically instantiated, both at runtime and designtime, ensuring that both our sample and live data are ready.

```
<Application
    x:Class="Win10MvvmLight.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Win10MvvmLight"
    xmlns:vm="using:Win10MvvmLight.Portable.ViewModels"
    RequestedTheme="Light">
    <Application.Resources>
        <ResourceDictionary>
            <vm:BaseViewModelLocator x:Key="ViewModelLocator" />
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

At last, we can finally write each ViewModel to its appropriate page via the DataContext. Adding it to the MainPageView XAML:

```
<Page
    x:Class="Win10MvvmLight.MainPage"
    ...
    DataContext="{Binding Path=MainPage, Source={StaticResource ViewModelLocator}}">
```

and the SecondPageView:

```
<Page
   x:Class="Win10MvvmLight.Views.SecondPage"
   ...
   DataContext="{Binding Path=SecondPage, Source={StaticResource ViewModelLocator}}">
```

Each page (View) is now automatically wired up to its associated ViewModels. By following this pattern for this and any future views, we get a declarative association without writing a single line of code. We'll see just how useful that is in the next chapter.

## Wrapping Up and Next Steps

Although it doesn't appear like we have accomplished much, what we have done is setup a framework for leveraging the MVVM design pattern in our app using MVVM Light.

We created a simple Model to represent the data we wish to display, the Views to display the actual UI to the user, and the ViewModels to contain the data to be presented.

In addition, since we defined the ViewModels and Model in a portable class; only the View is platform-specific, and would be the only piece we'd need to create new if we want to run the app on a new platform (assuming, of course, that platform supports MVVM Light).

Finally, by adding the ViewModelLocator, we were able to declaratively wire up all of the MVVM components without writing a single line of code or touching the code-behind of our pages. Best of all, as we'll see in the next chapter, that association will facilitate both runtime AND design-time data for our application, making it easier and faster to build the UI.

In the next chapter we'll see how to combine our setup with Blend for Visual Studio to quickly build the UI, using design-time data as a visual aid when creating the XAML layout.

# Adding Design-Time Data with Blend

In the previous chapter, we installed the MVVM Light Toolkit and defined the basic framework for a simple two-page app with the MVVM pattern. With the ViewModels we created, we now have the containers for the data, but it would be even more helpful to pre-populate them with some sample data to aid in the designing of the app.

## Design-Time Data Support

Fortunately, MVVM Light exposes a static property allowing the ViewModel code to detect whether or not it is running inside a visual designer such as Visual Studio or Blend. By leveraging this we can create a simple pattern for exposing design data through each ViewModel.

Also, since we likely are going to want design data on every page, we can add this to our ***BaseViewModel*** as a virtual method, running it via the constructor only when the designer is detected:

```
public BaseViewModel()
{
    if (this.IsInDesignMode)
    {
        LoadDesignTimeData();
    }
}

protected virtual void LoadDesignTimeData() { }
```

Now for every model that requires sample data at design-time, we can simply override this method, and populate the necessary properties with fake data. Since this only runs when detected by the static designer property, the code will safely be ignored at runtime.

Here's what it looks like for the MainPage ViewModel, instantiating a list of sample TestItem objects:

```
protected override void LoadDesignTimeData()
{
     base.LoadDesignTimeData();

     for (var i = 1; i < 10; i++)
     {
         var color = string.Join("", Enumerable.Repeat(i.ToString(), 6));
         var testItem = new TestItem() { Id = i, Title = "Test Item " + i, Subtitle =
"Subtitle " + i, HexColor = string.Concat("#", color) };
         TestItems.Add(testItem);
     }
}
```

and again for the SecondPageViewModel, creating a single TestItem with more sample data.

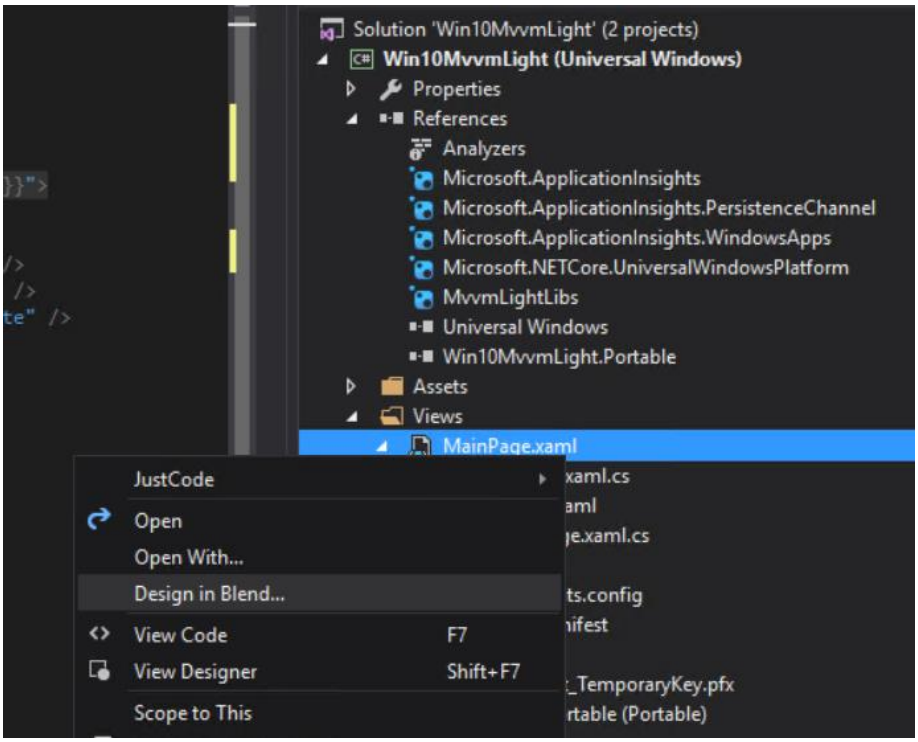
```
protected override void LoadDesignTimeData()
{
     base.LoadDesignTimeData();

     SelectedItem = new TestItem() { Title = "Design Time Selected Item", Subtitle =
"Design subtitle", HexColor = "#333333" };
}
```

Now that we have sample data available at design time, we want to use it as a visual aid to update our views to properly layout and format the data. We can do this with the powerful companion tool Blend for Visual Studio.

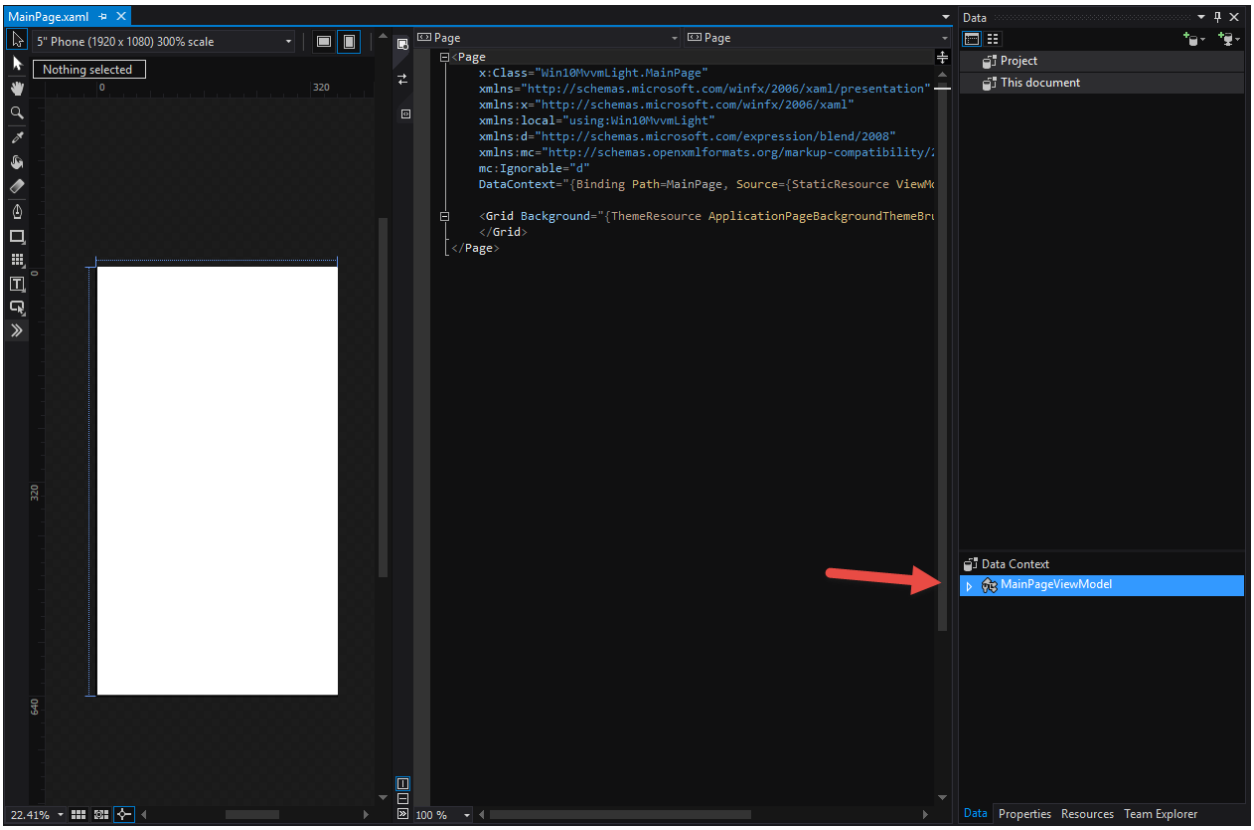
## Blend for Visual Studio

Visual Studio includes and automatically installs with Blend, a companion application that provides a designer-focused view of your Visual Studio projects. A full tour of Blend is outside the scope of this book, but I highly recommend that if you are new to Blend, you take the time to check out this course on Microsoft Virtual Academy: Designing Your XAML UI with Blend Jump Start. Although the course is specific to Blend 2013, virtually all of the content translates directly to VS2015, so it is well worth your time.

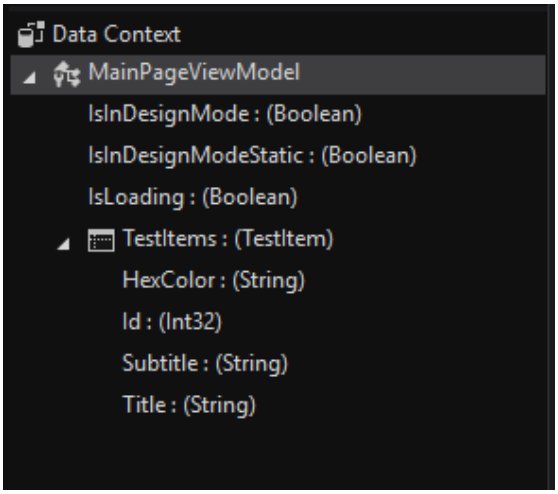
For now, we can jump directly into Blend from Visual Studio by right-clicking the MainPage.xaml in the Solution Explorer and selecting "Design in Blend":



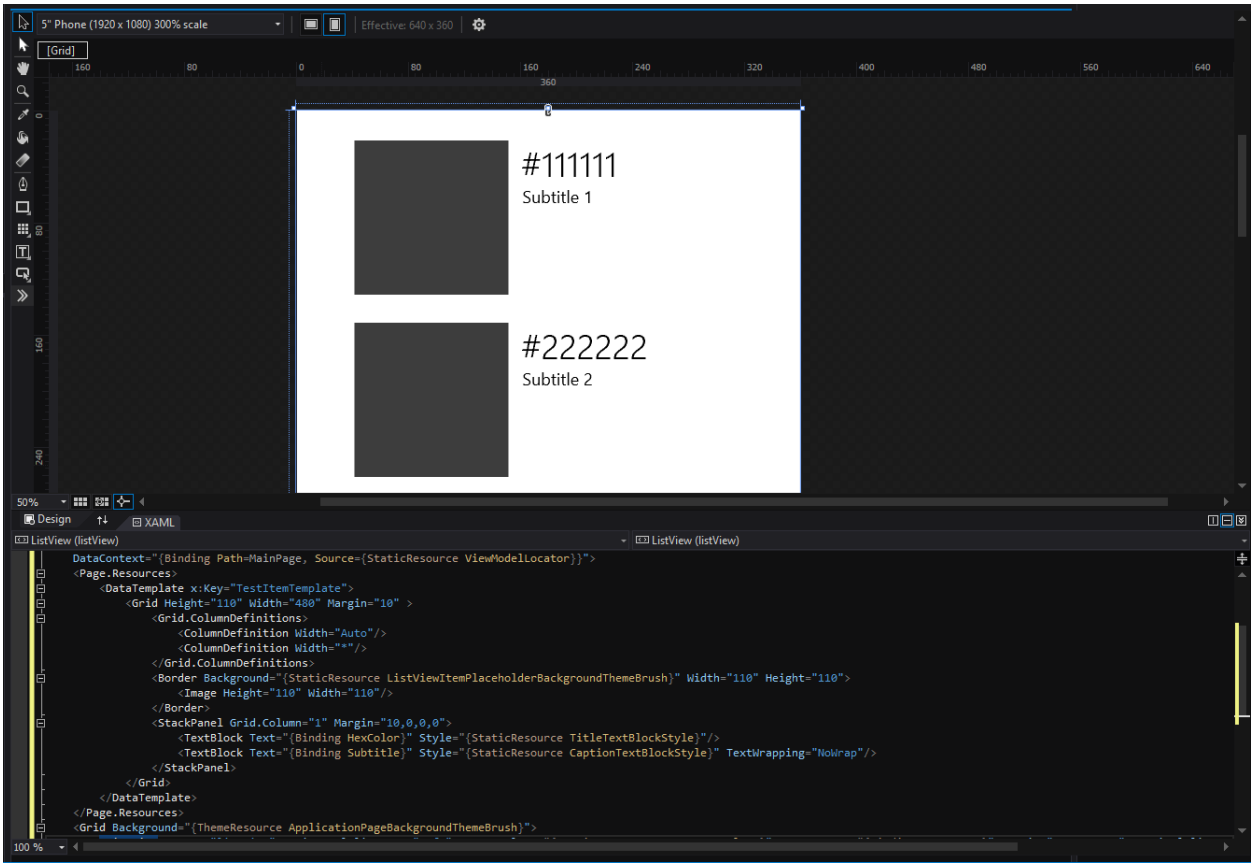

This reveals the same blank view of our MainPage but also gives access to the wide variety of design tools offered by Blend. Specifically, we want to focus on the ***Data*** window for MainPage, revealing the DataContext of the page, strongly-typed to the associated MainPageViewModel.




In fact, you'll notice that not only do we see the strongly-typed ViewModel, but all of its properties as well.




Among the many helpful features of Blend is the ability to drag a property from the Data view directly onto the canvas to quickly generate a simple layout and template. In this case, for the MainPage we want to show the list of TestItems, so dragging that property directly onto the grid reveals a generated ListView and associated ItemTemplate:


5" Phone (1920 x 1080) 300% scale
Effective: 640 x 360
[Grid]
#111111
Subtitle 1
#222222
Subtitle 2
Design
XAML
ListView (listView)
ListView (listView)
DataContext="{Binding Path=MainPage, Source={StaticResource ViewModelLocator}}">
<Page.Resources>
<DataTemplate x:Key="TestItemTemplate">
<Grid Height="110" Width="480" Margin="10" >
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Border Background="{StaticResource ListViewItemPlaceholderBackgroundThemeBrush}" Width="110" Height="110">
<Image Height="110" Width="110"/>
</Border>
<StackPanel Grid.Column="1" Margin="10,0,0,0">
<TextBlock Text="{Binding HexColor}" Style="{StaticResource TitleTextBlockStyle}"/>
<TextBlock Text="{Binding Subtitle}" Style="{StaticResource CaptionTextBlockStyle}" TextWrapping="NoWrap"/>
</StackPanel>
</Grid>
</DataTemplate>
</Page.Resources>
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
100 %

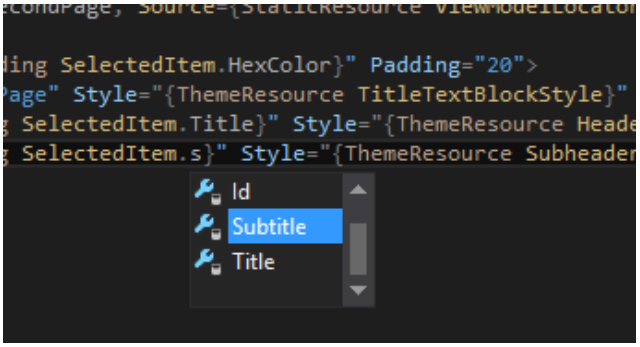
Obviously from the screenshot it can't anticipate all the properties and controls you'd want to use nor their correct placement, but with just a few minor changes to the generated template, we end up with the completed XAML for the MainPageView:

```
<Page
    x:Class="Win10MvvmLight.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Win10MvvmLight"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    xmlns:model="using:Win10MvvmLight.Portable.Model"
    DataContext="{Binding Path=MainPage, Source={StaticResource ViewModelLocator}}">
    <Page.Resources>
        <DataTemplate x:Key="TestItemTemplate">
            <Grid Height="110" Width="480" Margin="10" >
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>
                <Border Background="{Binding HexColor}" Width="110" Height="110">
                    <Image Height="110" Width="110"/>
                </Border>
                <StackPanel Grid.Column="1" Margin="10,0,0,0">
                    <TextBlock Text="{Binding Title}" Style="{StaticResource
TitleTextBlockStyle}"/>
                    <TextBlock Text="{Binding Subtitle}" Style="{StaticResource
CaptionTextBlockStyle}" TextWrapping="NoWrap"/>
                </StackPanel>
            </Grid>
        </DataTemplate>
    </Page.Resources>

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <ListView x:Name="listView"
                  ItemTemplate="{StaticResource TestItemTemplate}"
                  ItemsSource="{Binding TestItems}"
                  Margin="19,12,19,0"
                  ItemClick="listView_ItemClick"
                  IsItemClickEnabled="True"/>

    </Grid>
</Page>
```

We can follow the same steps for the SecondPage by dragging the individual properties from the Data view onto the controls to which you want to bind. However, there is another way to achieve the data binding for the page.

## DataBinding Intellisense

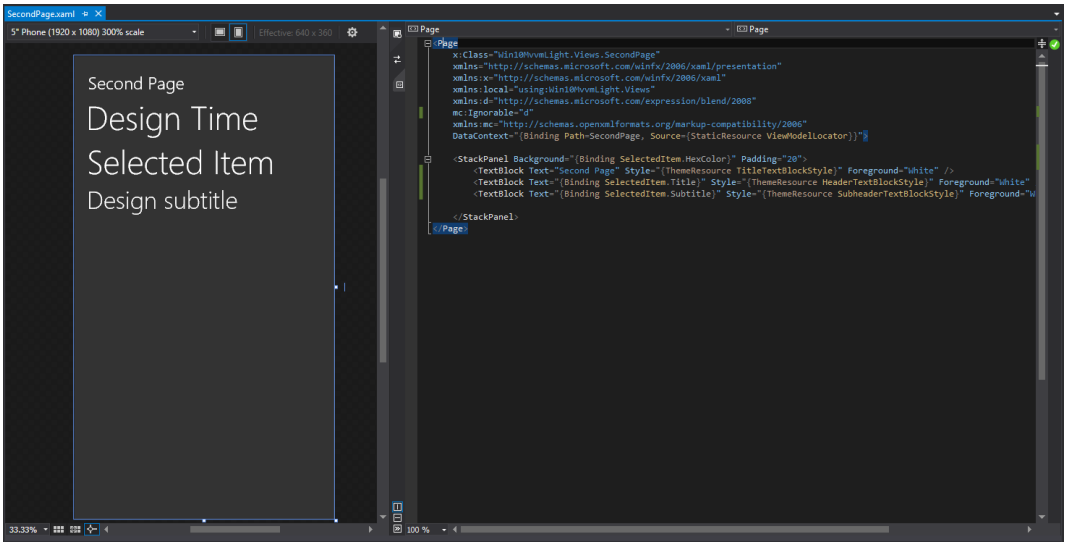
Since we bound the ViewModel to the page declaratively (using the ViewModelLocator from the last chapter), the page is contextually aware of it, and Intellisense is smart enough to help us by exposing strongly-typed data-binding, as seen here updating the SecondPageView controls:




We can use this feature to bind the remaining properties on the page; here's the updated SecondPage XAML with all the databindings:

```
<Page
    x:Class="Win10MvvmLight.Views.SecondPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Win10MvvmLight.Views"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        mc:Ignorable="d"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    DataContext="{Binding Path=SecondPage, Source={StaticResource ViewModelLocator}}">

    <StackPanel Background="{Binding SelectedItem.HexColor}" Padding="20">
        <TextBlock Text="Second Page" Style="{ThemeResource TitleTextBlockStyle}"
Foreground="White" />
        <TextBlock Text="{Binding SelectedItem.Title}" Style="{ThemeResource
HeaderTextBlockStyle}" Foreground="White" />
        <TextBlock Text="{Binding SelectedItem.Subtitle}" Style="{ThemeResource
SubheaderTextBlockStyle}" Foreground="White" />

    </StackPanel>
</Page>
```

And a screenshot of the designer showing the bound properties:




We now have a helpful design-view of all of the pages in our app. This is very helpful if we want to make additional visual changes, enabling immediate feedback in the designer without having to launch the app with live data to see the result.

## Wrapping up and Next Steps

Although we still don't yet have a useable, running app, we've demonstrated how you can use both the MVVM Light Toolkit and the design features of Blend and Visual Studio to quickly layout the pages of an application. By following this pattern, you can quickly define all the screens of your app without every having to run it, which is especially helpful when your app is still in the design phase.

Of course, launching the app still reveals a blank page, since we have only created design-time data.

In the next chapter, we'll look at a simple (yet naïve) strategy for both runtime data and navigation to get the app to a usable state. Later, we'll reveal how to refactor that code into a more elegant and portable solution with more help from the MVVM Light Toolkit.

# Adding Simple Navigation

So far we've setup a few pages with some design-time data to help us layout the app, but running it still yields a blank screen with no interaction possible. We'll remedy this by loading the sample data at runtime and adding a simple navigation implementation to allow us to go back and forth between the pages.

To keep things simple, we'll briefly break from the MVVM pattern, and navigate directly of the page frame via the code behind, and in future chapters see how we can roll this back in. For now, the first thing we need is to have something on the screen to trigger the navigation, so we need to begin by populating the page with some data when the screen loads.

## OnNavigatedTo

Everytime a XAML page loads, a method called OnNavigatedTo executes, and by overriding this method in the code-behind for the page, we can execute custom code to prepare the UI.

In our case, we want to add items to the ViewModel associated with the page. Remember that this ViewModel is already wired up to the ***DataContext*** property of the page, so we can simply cast it to the appropriate type and update it when the page loads.

For simplicity, we'll just load it with a list of fake items, as shown here in the code-behind for the MainPage:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

    var vm = DataContext as MainPageViewModel;
    if (!vm.TestItems.Any())
    {
                vm.TestItems = new
System.Collections.ObjectModel.ObservableCollection<TestItem>(GetFakeRuntimeItems());
    }
}

private List<TestItem> GetFakeRuntimeItems()
{
        var items = new List<TestItem>();
        for (var i = 1; i <= 5; i++)
        {
             var color = string.Join("", Enumerable.Repeat(i.ToString(), 6));
             var testItem = new TestItem() { Id = i, Title = "Runtime Item " + i,
Subtitle = "Subtitle " + i, HexColor = string.Concat("#", color) };
             items.Add(testItem);
        }

        return items;
}
```

Running the app now reveals that same list of sample items:


However, clicking or tapping the items doesn't cause any change in behavior. We need to actually trigger the navigation, which we want to happen whenever an item in the list is clicked.

## Enabling ItemClick on ListView

Fortunately, the ListView already has the event we need, and we need only subscribe to it. However, before we can use it, we have to enable it. You see, by default, the IsItemClickEnabled property of the ListView (and all other ListViewBase controls) is set to false.

So in addition to creating and setting the ItemClick handler you also want to set the IsItemClickEnabled property to true:

```
<ListView x:Name="listView"
      HorizontalAlignment="Left"
      ItemTemplate="{StaticResource TestItemTemplate}"
      ItemsSource="{Binding TestItems}"
      Margin="19,12,19,0"
      VerticalAlignment="Top"
      IsItemClickEnabled="True" ItemClick="listView_ItemClick" />
```

We can now use the handler to perform the actual navigation to the SecondPage, which we do by calling ***Navigate*** on the page's frame:

```
private void listView_ItemClick(object sender, ItemClickEventArgs e)
{
    var selectedItem = e.ClickedItem as TestItem;
    if (selectedItem == null) return;

    this.Frame.Navigate(typeof(SecondPage), selectedItem.Id);
}
```

However, in doing so we probably want to be able to pass the context of the clicked item, so the desired page knows what it should be showing. We do this by using a navigation parameter.

### Navigation Parameters

When an item is clicked, the clicked item is passed to the eventhandler argument as an object, so we want to make sure that we cast it so we can extract the appropriate information needed for navigation, in this case the ID. We can use this value as a parameter to tell the SecondPage which item we actually want to see.

You might be wondering why we passed the ID of the item instead of the complete object itself. It is actually possible to send the complete object, and doing so would certainly simplify loading the SecondPage by using the passed object directly. In fact, doing this in an app will not cause any immediate exceptions…

However, the reason this isn't suggested or recommended is that during suspension of your app (as we'll see in the next chapter on managing application state), the entire state of the application needs to be preserved so that it can be restored upon resume. This includes the navigation history and all of its parameters, so that the back stack can be restored as well, returning the user to the exact same point they last left the application.

Navigation state is preserved by the app frame by calling GetNavigationState(), however, as you can see in this quote from the documentation (emphasis mine):

> The serialization format used by these methods is for internal use only. Your app should not form any dependencies on it. Additionally, ***this format supports serialization only for basic types like string, char, numeric and GUID types.***

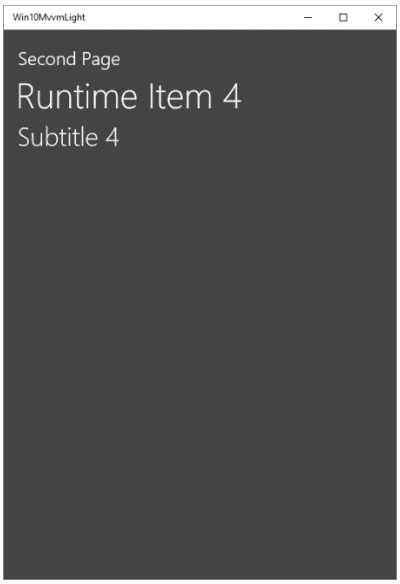
Unless you intend roll your own system for storing and restoring state, you need to make sure the parameters used in navigation are one of the supported simple types, which is why we've used the ID to indicate the desired item.

On the SecondPage, we want to retrieve the item that matches the ID. Again for simplicity, we'll simply instantiate a duplicate list of the same items from the MainPage to ensure that the possible items match, and retrieve the desired item by it's ID value. Just like with the MainPage, we'll fire this in the ***OnNavigatedTo***method:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

        var vm = DataContext as SecondPageViewModel;
        vm.SelectedItem = GetFakeRuntimeItems().FirstOrDefault(i => i.Id ==
(int)e.Parameter);
}

private List<TestItem> GetFakeRuntimeItems()
{
        var items = new List<TestItem>();
        for (var i = 1; i <= 5; i++)
        {
            var color = string.Join("", Enumerable.Repeat(i.ToString(), 6));
            var testItem = new TestItem() { Id = i, Title = "Runtime Item " + i, Subtitle
= "Subtitle " + i, HexColor = string.Concat("#", color) };
            items.Add(testItem);
        }

        return items;
}
```

Obviously a real-world app would not duplicate the data but probably pull it from some locally-cached source, but this simple example should help you get the idea of how the navigation parameter drives the navigation.

Running the app now allows us to click the desired item in the list and see its details on the second page:

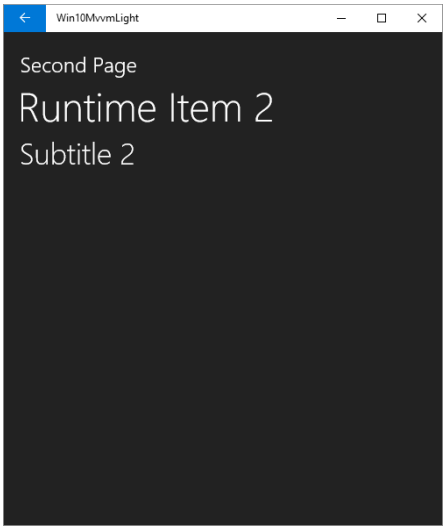



Of course, once we get there we're stuck! We need a way to go back!

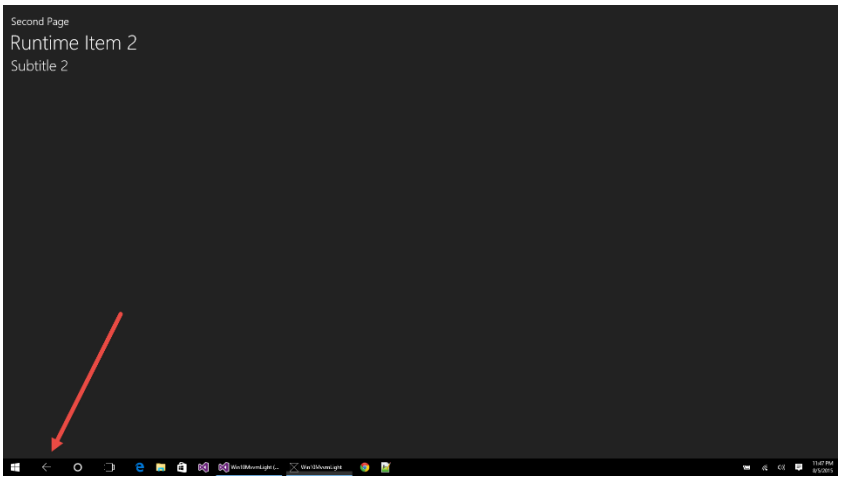
## Back Button on Windows 10

If we were on a device like a phone, we could certainly leverage the existing back button required by the hardware. However, most desktop devices do not feature such a requirement. Fortunately you can use the SystemNavigationManager in Windows 10 to enable a global, app-specific back button to facilitate such navigation, ensuring a consistent experience regardless of the device.

Here's what it looks like in our sample app, shown on the SecondPage app bar, allowing us to go back to the previous page.




The button is inherently responsive, so if we activate tablet mode, it will switch to the bottom of the page, consistent with the OS behavior.




In addition to the AppViewBackButtonVisibility property, the SystemNavigationManager also exposes a BackRequested event that fires when the user actually requests to go back. The best part of this event is that it is setup so that it fires regardless of whether it is triggered by the button on the app bar or a hardware back button such as on a phone. This means the same code can handle either of those events (or any other ways a user might request to go back in the future!), all you have to do is handle the event.

## Responsive Code

Since devices like phones do have the necessary hardware to allow a user to go back, we want to make sure to only enable the app bar back button on devices that don't have such a feature. This is where we can leverage the ability of Windows 10 to detect such features at runtime via the ApiInformation class. We can use the IsTypePresent method to determine whether or not the device running your application supports, defines, and implements a specified type, in this case the HardwareButtons which contain the BackPressed event.

If the device does support and implement this type, we can hide the app bar back button, otherwise we obviously want to show it. Here is the complete code for the SecondPage, again fired OnNavigatedTo that gives the back navigation support we want:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

        var vm = DataContext as SecondPageViewModel;
        vm.SelectedItem = GetFakeRuntimeItems().FirstOrDefault(i => i.Id ==
(int)e.Parameter);

        var currentView = SystemNavigationManager.GetForCurrentView();

    if (!ApiInformation.IsTypePresent("Windows.Phone.UI.Input.HardwareButtons"))
    {
        currentView.AppViewBackButtonVisibility = this.Frame.CanGoBack ?
AppViewBackButtonVisibility.Visible : AppViewBackButtonVisibility.Collapsed;
    }

    currentView.BackRequested += SystemNavigationManager_BackRequested;
}

private void SystemNavigationManager_BackRequested(object sender, BackRequestedEventArgs
e)
{
    if (this.Frame.CanGoBack)
    {
        this.Frame.GoBack();
        e.Handled = true;
    }
}
```

Note that we first have to get the reference to the SystemNaviagtionManager for the current page by calling ***GetForCurrentView***.

The app now shows the expected back button on the screen for the second page. Also as expected, it is hidden from the SecondPage on devices with a dedicated back button, such as the phone:


We could use the same strategy to show an on-canvas back button (like we used to do for Windows 8.1), but we'll leave that as an exercise for the reader.

## Wrapping Up and Next Steps

At last, we have a simple but functional app, complete with both sample and runtime data, that allows us to navigate between the defined pages. However, something interesting happens if, while we are on the SecondPage, we suspend and shutdown the app.

Recall that an app is expected to store the state and resume the user experience where it left off when it was suspended. Our app, however, kicks you back to the beginning on launch, discarding where we were and which item we selected!

In the next chapter we'll see how we can leverage the SuspensionManager class from Windows 8.1 and modify it to work with Windows 10 to preserve and restore state automatically. We'll also see how we can refactor both the navigation and page-load code so that it is fully driven by and runs in our ViewModels, putting us back on the MVVM pattern.

# Maintaining Application State

We have so far created a simple app with two pages, that uses simple Frame navigation to go back and forth. However, the app is still missing one crucial feature: state management. When an app is suspended for whatever reason (such as being minimized on the desktop or navigated away from on the phone), it is up to the developer to maintain the current state so that it can be fully restored where the user left off.

In this chapter we'll look at a simple way we can achieve this by leveraging helper classes from the Windows 8.1 project templates.

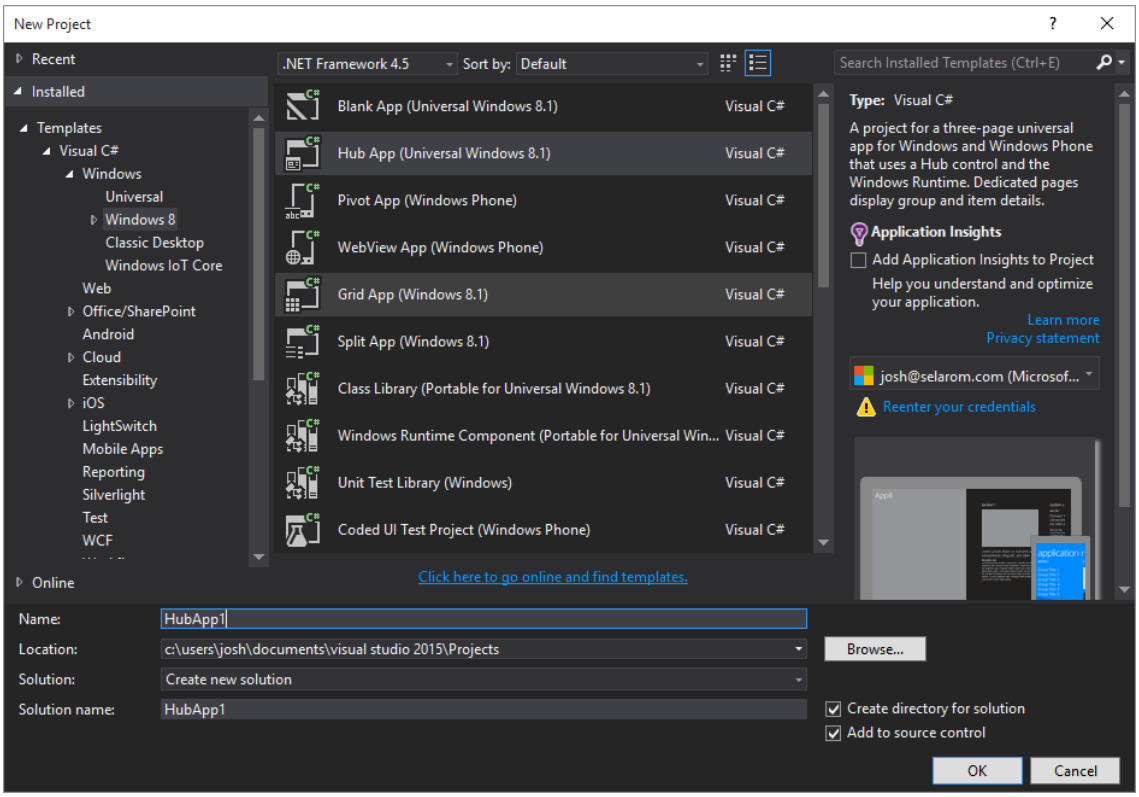
## Template 10

It's worth mentioning that there is a project called Template10 from the Windows XAML team that also addresses these issues, and is discussed in more detail in the MVA course A Developer's Guide to Windows 10. I encourage you to explore that project as an alternative if you are looking for a more full-featured framework for Windows 10.

For now, this chapter provides a quick-and-simple way to add state management to your app by recycling a few helper classes from Windows 8.1.

## Windows 8.1 Universal App Hub Template

One of the more helpful components of the Windows 8.1 Universal SDK was the Hub App Template. It included several pages, controls and layouts demonstrating a functioning app complete with navigation. However, it also included a few base components to facilitate things like maintaining the navigation and application state.




We'll be leveraging the MvvmLight framework for navigation in a future chapter, but we can still leverage the SuspensionManager class from the template to make it easier to manage the state of a Windows 10 app.

### Suspension Manager (Windows 8.1)

The SuspensionManager class from the 8.1 project can be used as-is, straight from the 8.1 template. Here's a copy of the class that you can add to any project.

Note: the full source for this is available on the GitHub repo for this book.

```
    /// <summary>
    /// SuspensionManager captures global session state to simplify process lifetime
management
    /// for an application.  Note that session state will be automatically cleared under
a variety
    /// of conditions and should only be used to store information that would be
convenient to
    /// carry across sessions, but that should be discarded when an application crashes
or is
    /// upgraded.
    /// </summary>
    internal sealed class SuspensionManager
    {
        private static Dictionary<string, object> _sessionState = new Dictionary<string,
object>();
        private static List<Type> _knownTypes = new List<Type>();
        private const string sessionStateFilename = "_sessionState.xml";

        /// <summary>
        /// Provides access to global session state for the current session.  This state
is
        /// serialized by <see cref="SaveAsync"/> and restored by
        /// <see cref="RestoreAsync"/>, so values must be serializable by
        /// <see cref="DataContractSerializer"/> and should be as compact as
possible.  Strings
        /// and other self-contained data types are strongly recommended.
        /// </summary>
        public static Dictionary<string, object> SessionState
        {
            get { return _sessionState; }
        }

        /// <summary>
        /// List of custom types provided to the <see cref="DataContractSerializer"/>
when
        /// reading and writing session state.  Initially empty, additional types may be
        /// added to customize the serialization process.
        /// </summary>
        public static List<Type> KnownTypes
        {
            get { return _knownTypes; }
        }

        /// <summary>
        /// Save the current <see cref="SessionState"/>.  Any <see cref="Frame"/>
instances
        /// registered with <see cref="RegisterFrame"/> will also preserve their current
        /// navigation stack, which in turn gives their active <see cref="Page"/> an
opportunity
        /// to save its state.
        /// </summary>
        /// <returns>An asynchronous task that reflects when session state has been
saved.</returns>
        public static async Task SaveAsync()
        {
            try
            {
                // Save the navigation state for all registered frames
                foreach (var weakFrameReference in _registeredFrames)
                {
                    Frame frame;
                    if (weakFrameReference.TryGetTarget(out frame))
                    {
```

```
                        SaveFrameNavigationState(frame);
                    }
                }

                // Serialize the session state synchronously to avoid asynchronous access
to shared
                // state
                MemoryStream sessionData = new MemoryStream();
                DataContractSerializer serializer = new
DataContractSerializer(typeof(Dictionary<string, object>), _knownTypes);
                serializer.WriteObject(sessionData, _sessionState);

                // Get an output stream for the SessionState file and write the state
asynchronously
                StorageFile file = await
ApplicationData.Current.LocalFolder.CreateFileAsync(sessionStateFilename,
CreationCollisionOption.ReplaceExisting);
                using (Stream fileStream = await file.OpenStreamForWriteAsync())
                {
                    sessionData.Seek(0, SeekOrigin.Begin);
                    await sessionData.CopyToAsync(fileStream);
                }
            }
            catch (Exception e)
            {
                throw new SuspensionManagerException(e);
            }
        }

        /// <summary>
        /// Restores previously saved <see cref="SessionState"/>.  Any <see
cref="Frame"/> instances
        /// registered with <see cref="RegisterFrame"/> will also restore their prior
navigation
        /// state, which in turn gives their active <see cref="Page"/> an opportunity
restore its
        /// state.
        /// </summary>
        /// <param name="sessionBaseKey">An optional key that identifies the type of
session.
        /// This can be used to distinguish between multiple application launch
scenarios.</param>
        /// <returns>An asynchronous task that reflects when session state has been
read.  The
        /// content of <see cref="SessionState"/> should not be relied upon until this
task
        /// completes.</returns>
        public static async Task RestoreAsync(String sessionBaseKey = null)
        {
            _sessionState = new Dictionary<String, Object>();

            try
            {
                // Get the input stream for the SessionState file
                StorageFile file = await
ApplicationData.Current.LocalFolder.GetFileAsync(sessionStateFilename);
                using (IInputStream inStream = await file.OpenSequentialReadAsync())
                {
                    // Deserialize the Session State
                    DataContractSerializer serializer = new
DataContractSerializer(typeof(Dictionary<string, object>), _knownTypes);
                    _sessionState = (Dictionary<string,
object>)serializer.ReadObject(inStream.AsStreamForRead());
```

```
                }

                // Restore any registered frames to their saved state
                foreach (var weakFrameReference in _registeredFrames)
                {
                    Frame frame;
                    if (weakFrameReference.TryGetTarget(out frame) &&
(string)frame.GetValue(FrameSessionBaseKeyProperty) == sessionBaseKey)
                    {
                        frame.ClearValue(FrameSessionStateProperty);
                        RestoreFrameNavigationState(frame);
                    }
                }
            }
            catch (Exception e)
            {
                throw new SuspensionManagerException(e);
            }
        }

        private static DependencyProperty FrameSessionStateKeyProperty =
            DependencyProperty.RegisterAttached("_FrameSessionStateKey", typeof(String),
typeof(SuspensionManager), null);
        private static DependencyProperty FrameSessionBaseKeyProperty =
            DependencyProperty.RegisterAttached("_FrameSessionBaseKeyParams",
typeof(String), typeof(SuspensionManager), null);
        private static DependencyProperty FrameSessionStateProperty =
            DependencyProperty.RegisterAttached("_FrameSessionState",
typeof(Dictionary<String, Object>), typeof(SuspensionManager), null);
        private static List<WeakReference<Frame>> _registeredFrames = new
List<WeakReference<Frame>>();

        /// <summary>
        /// Registers a <see cref="Frame"/> instance to allow its navigation history to
be saved to
        /// and restored from <see cref="SessionState"/>.  Frames should be registered
once
        /// immediately after creation if they will participate in session state
management.  Upon
        /// registration if state has already been restored for the specified key
        /// the navigation history will immediately be restored.  Subsequent invocations
of
        /// <see cref="RestoreAsync"/> will also restore navigation history.
        /// </summary>
        /// <param name="frame">An instance whose navigation history should be managed by
        /// <see cref="SuspensionManager"/></param>
        /// <param name="sessionStateKey">A unique key into <see cref="SessionState"/>
used to
        /// store navigation-related information.</param>
        /// <param name="sessionBaseKey">An optional key that identifies the type of
session.
        /// This can be used to distinguish between multiple application launch
scenarios.</param>
        public static void RegisterFrame(Frame frame, String sessionStateKey, String
sessionBaseKey = null)
        {
            if (frame.GetValue(FrameSessionStateKeyProperty) != null)
            {
                throw new InvalidOperationException("Frames can only be registered to one
session state key");
            }

            if (frame.GetValue(FrameSessionStateProperty) != null)
```

```
            {
                throw new InvalidOperationException("Frames must be either be registered
before accessing frame session state, or not registered at all");
            }

            if (!string.IsNullOrEmpty(sessionBaseKey))
            {
                frame.SetValue(FrameSessionBaseKeyProperty, sessionBaseKey);
                sessionStateKey = sessionBaseKey + "_" + sessionStateKey;
            }

            // Use a dependency property to associate the session key with a frame, and
keep a list of frames whose
            // navigation state should be managed
            frame.SetValue(FrameSessionStateKeyProperty, sessionStateKey);
            _registeredFrames.Add(new WeakReference<Frame>(frame));

            // Check to see if navigation state can be restored
            RestoreFrameNavigationState(frame);
        }

        /// <summary>
        /// Disassociates a <see cref="Frame"/> previously registered by <see
cref="RegisterFrame"/>
        /// from <see cref="SessionState"/>.  Any navigation state previously captured
will be
        /// removed.
        /// </summary>
        /// <param name="frame">An instance whose navigation history should no longer be
        /// managed.</param>
        public static void UnregisterFrame(Frame frame)
        {
            // Remove session state and remove the frame from the list of frames whose
navigation
            // state will be saved (along with any weak references that are no longer
reachable)
            SessionState.Remove((String)frame.GetValue(FrameSessionStateKeyProperty));
            _registeredFrames.RemoveAll((weakFrameReference) =>
            {
                Frame testFrame;
                return !weakFrameReference.TryGetTarget(out testFrame) || testFrame ==
frame;
            });
        }

        /// <summary>
        /// Provides storage for session state associated with the specified <see
cref="Frame"/>.
        /// Frames that have been previously registered with <see cref="RegisterFrame"/>
have
        /// their session state saved and restored automatically as a part of the global
        /// <see cref="SessionState"/>.  Frames that are not registered have transient
state
        /// that can still be useful when restoring pages that have been discarded from
the
        /// navigation cache.
        /// </summary>
        /// <remarks>Apps may choose to rely on <see cref="NavigationHelper"/> to manage
        /// page-specific state instead of working with frame session state
directly.</remarks>
        /// <param name="frame">The instance for which session state is desired.</param>
        /// <returns>A collection of state subject to the same serialization mechanism as
        /// <see cref="SessionState"/>.</returns>
```

```
        public static Dictionary<String, Object> SessionStateForFrame(Frame frame)
        {
            var frameState = (Dictionary<String,
Object>)frame.GetValue(FrameSessionStateProperty);

            if (frameState == null)
            {
                var frameSessionKey =
(String)frame.GetValue(FrameSessionStateKeyProperty);
                if (frameSessionKey != null)
                {
                    // Registered frames reflect the corresponding session state
                    if (!_sessionState.ContainsKey(frameSessionKey))
                    {
                        _sessionState[frameSessionKey] = new Dictionary<String,
Object>();
                    }
                    frameState = (Dictionary<String,
Object>)_sessionState[frameSessionKey];
                }
                else
                {
                    // Frames that aren't registered have transient state
                    frameState = new Dictionary<String, Object>();
                }
                frame.SetValue(FrameSessionStateProperty, frameState);
            }
            return frameState;
        }

        private static void RestoreFrameNavigationState(Frame frame)
        {
            var frameState = SessionStateForFrame(frame);
            if (frameState.ContainsKey("Navigation"))
            {
                frame.SetNavigationState((String)frameState["Navigation"]);
            }
        }

        private static void SaveFrameNavigationState(Frame frame)
        {
            var frameState = SessionStateForFrame(frame);
            frameState["Navigation"] = frame.GetNavigationState();
        }
    }
    public class SuspensionManagerException : Exception
    {
        public SuspensionManagerException()
        {
        }

        public SuspensionManagerException(Exception e)
            : base("SuspensionManager failed", e)
        {

        }
    }
```

To leverage it, the app needs to do three things. First, the SuspensionManager needs to register the main frame of the application, which is done in theOnLaunched event. If necessary, the manager should also restore the previous state:

```
protected async override void OnLaunched(LaunchActivatedEventArgs e)
{
        // ...

        Frame rootFrame = Window.Current.Content as Frame;
        if (rootFrame == null)
        {
                rootFrame = new Frame();
        SuspensionManager.RegisterFrame(rootFrame, "AppFrame");

                if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
                {
                    await SuspensionManager.RestoreAsync();
                }

                // ...
        }

        // ...
}
```

Next, the SuspensionManager needs to store the state when the app actually suspends, which occurs conveniently enough in the OnSuspended event. Note that since the operation is asynchronous, we need to first get then complete the suspension deferral:

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
        var deferral = e.SuspendingOperation.GetDeferral();
        await SuspensionManager.SaveAsync();
        deferral.Complete();
}
```

The application is now configured to store and load state at the application level. However, the last step is to configure each view to also save and load the state of individual pages.

The 8.1 Universal Hub App does this through the NavigationHelper class, which we could certainly reuse in our app. However, MvvmLight already includes a NavigationService, which we'll explore in the next chapter. Instead, we want to extract all the relevant code from the helper and wire it directly to the page itself.

## ViewBase

Since we need this to execute on every page load (and unload), we'll put this in a new ViewBase base class, from which all our pages will now derive. This allows us to override the ***OnNavigatedTo*** and ***OnNavigatedFrom*** events of every View to ensure that both the individual page state, as well as the complete navigation history, is serialized and saved so it can be reloaded for the user.

### ViewModel State Events

There is one more advantage to this approach. Recall that the DataContext every page is now bound directly to its ViewModel, each of which derives from the same ***ViewModelBase*** class. By adding public Load and Save events to this class, we can fire them automatically from the page by adding a reference to this base ViewModel in our ViewBase and calling the events at the appropriate time.

Here is the modified ViewModelBase class with virtual events to Load and Save State. Each ViewModel will override these method (if needed) to load and save the state of their associated page.

```
public class BaseViewModel : ViewModelBase
{
        public BaseViewModel()
        {
            if (this.IsInDesignMode)
            {
                LoadDesignTimeData();
            }
        }

        private bool isLoading;
        public virtual bool IsLoading
        {
            get { return isLoading; }
            set
            {
                isLoading = value;
                RaisePropertyChanged();
            }
        }

        #region State Management

        public virtual void LoadState(object navParameter, Dictionary<string, object>
state) { }

        public virtual void SaveState(Dictionary<string, object> state) { }

        protected virtual T RestoreStateItem<T>(Dictionary<string, object> state, string
stateKey, T defaultValue = default(T))
        {
            return state != null && state.ContainsKey(stateKey) && state[stateKey] !=
null && state[stateKey] is T ? (T)state[stateKey] : defaultValue;
        }

        #endregion

        protected virtual void LoadDesignTimeData() { }
}
```

Note that this also has an additional generic helper method ***RestoreStateItem*** to make it easy restore individual properties from the state; we'll see this in action shortly.

Since we're passing the state object to the ViewModel, we can add to it to save properties for later use if the application is suspended and restarted. For example, here we've added a simple text property to the SecondPage ViewModel, saving and loading its value from the state as the view is unloaded and loaded again.

```
public class SecondPageViewModel : BaseViewModel
{
        private TestItem selectedItem;
        public TestItem SelectedItem
        {
            get { return selectedItem; }
            set
            {
                selectedItem = value;
                RaisePropertyChanged();
            }
        }

        private string stateText;
        public string StateText
        {
            get { return stateText; }
            set
            {
                stateText = value;
                RaisePropertyChanged();
            }
        }

        #region State Management

        public override void LoadState(object navParameter, Dictionary<string, object>
state)
        {
            base.LoadState(navParameter, state);

            // load test items again; in production this would retrieve the live item by
id or get it from a local data cache
            var items = GetFakeRuntimeItems();

            SelectedItem = items.FirstOrDefault(i => i.Id == (int)navParameter);

            if (state != null)
            {
                StateText = this.RestoreStateItem<string>(state, "STATETEXT");
            }
        }

        public override void SaveState(Dictionary<string, object> state)
        {
            base.SaveState(state);

            state["STATETEXT"] = StateText;
        }

        #endregion

        private List<TestItem> GetFakeRuntimeItems()
        {
            var items = new List<TestItem>();
            for (var i = 1; i <= 5; i++)
            {
                var color = string.Join("", Enumerable.Repeat(i.ToString(), 6));
                var testItem = new TestItem() { Id = i, Title = "Runtime Item " + i,
Subtitle = "Subtitle " + i, HexColor = string.Concat("#", color) };
                items.Add(testItem);
            }
```

```
            return items;
        }

        protected override void LoadDesignTimeData()
        {
            base.LoadDesignTimeData();

            SelectedItem = new TestItem() { Title = "Design Time Selected Item", Subtitle
= "Design subtitle", HexColor = "#333333" };
        }
}
```

### Calling ViewModel Events from the Page

As previously mentioned, we want to fire the ***LoadState*** and ***SaveState*** events from the view automatically, which we can do by adding a reference to the ViewModel, which again is automatically stored in the DataContext.

Once we have that reference, we can wire into the LoadState and SaveState of the ViewBase class, pushing the event right through the pipeline.

```
public class ViewBase : Page
{
    private BaseViewModel PageViewModel
    {
        get { return this.DataContext as BaseViewModel; }
    }

    public ViewBase()
    {
        this.LoadState += ViewBase_LoadState;
        this.SaveState += ViewBase_SaveState;
    }

    void ViewBase_SaveState(object sender, SaveStateEventArgs e)
    {
        if (PageViewModel != null) PageViewModel.SaveState(e.PageState);
    }

    void ViewBase_LoadState(object sender, LoadStateEventArgs e)
    {
        if (PageViewModel != null)
        {
            var view = this.GetType().Name;
            PageViewModel.LoadState(e.NavigationParameter, e.PageState);
        }
    }

        // ...
}
```

Now all that is left is to add the code pulled from the original Windows 8.1 NavigationHelper class related to saving navigation state, and we have a complete system to manage both the application and navigation state.

Here is the complete ViewBase code:

```
public class ViewBase : Page
{
        private BaseViewModel PageViewModel
        {
            get { return this.DataContext as BaseViewModel; }
        }

        private String _pageKey;

        public ViewBase()
        {
            this.LoadState += ViewBase_LoadState;
            this.SaveState += ViewBase_SaveState;
        }

        void ViewBase_SaveState(object sender, SaveStateEventArgs e)
        {
            if (PageViewModel != null) PageViewModel.SaveState(e.PageState);
        }

        void ViewBase_LoadState(object sender, LoadStateEventArgs e)
        {
            if (PageViewModel != null)
            {
                var view = this.GetType().Name;

                PageViewModel.LoadState(e.NavigationParameter, e.PageState);
            }
        }

        /// <summary>
        /// Register this event on the current page to populate the page
        /// with content passed during navigation as well as any saved
        /// state provided when recreating a page from a prior session.
        /// </summary>
        public event LoadStateEventHandler LoadState;

        /// <summary>
        /// Register this event on the current page to preserve
        /// state associated with the current page in case the
        /// application is suspended or the page is discarded from
        /// the navigaqtion cache.
        /// </summary>
        public event SaveStateEventHandler SaveState;

        protected override void
OnNavigatedTo(Windows.UI.Xaml.Navigation.NavigationEventArgs e)
        {
            base.OnNavigatedTo(e);

            var frameState = SuspensionManager.SessionStateForFrame(this.Frame);
            this._pageKey = "Page-" + this.Frame.BackStackDepth;

            if (e.NavigationMode == NavigationMode.New)
            {
                // Clear existing state for forward navigation when adding a new page to
the
                // navigation stack
                var nextPageKey = this._pageKey;
                int nextPageIndex = this.Frame.BackStackDepth;
                while (frameState.Remove(nextPageKey))
                {
                    nextPageIndex++;
```

```
                    nextPageKey = "Page-" + nextPageIndex;
                }

                // Pass the navigation parameter to the new page
                if (this.LoadState != null)
                {
                    this.LoadState(this, new LoadStateEventArgs(e.Parameter, null));
                }
            }
            else
            {
                // Pass the navigation parameter and preserved page state to the page,
using
                // the same strategy for loading suspended state and recreating pages
discarded
                // from cache
                if (this.LoadState != null)
                {
                    this.LoadState(this, new LoadStateEventArgs(e.Parameter,
(Dictionary<String, Object>)frameState[this._pageKey]));
                }
            }
        }

        protected override void OnNavigatedFrom(NavigationEventArgs e)
        {
            base.OnNavigatedFrom(e);

            var frameState = SuspensionManager.SessionStateForFrame(this.Frame);
            var pageState = new Dictionary<String, Object>();
            if (this.SaveState != null)
            {
                this.SaveState(this, new SaveStateEventArgs(pageState));
            }
            frameState[_pageKey] = pageState;
        }
    }

    /// <summary>
    /// Represents the method that will handle the <see
cref="NavigationHelper.LoadState"/>event
    /// </summary>
    public delegate void LoadStateEventHandler(object sender, LoadStateEventArgs e);
    /// <summary>
    /// Represents the method that will handle the <see
cref="NavigationHelper.SaveState"/>event
    /// </summary>
    public delegate void SaveStateEventHandler(object sender, SaveStateEventArgs e);

    /// <summary>
    /// Class used to hold the event data required when a page attempts to load state.
    /// </summary>
    public class LoadStateEventArgs : EventArgs
    {
        /// <summary>
        /// The parameter value passed to <see cref="Frame.Navigate(Type, Object)"/>
        /// when this page was initially requested.
        /// </summary>
        public Object NavigationParameter { get; private set; }
        /// <summary>
        /// A dictionary of state preserved by this page during an earlier
        /// session.  This will be null the first time a page is visited.
        /// </summary>
```

```
        public Dictionary<string, Object> PageState { get; private set; }

        /// <summary>
        /// Initializes a new instance of the <see cref="LoadStateEventArgs"/> class.
        /// </summary>
        /// <param name="navigationParameter">
        /// The parameter value passed to <see cref="Frame.Navigate(Type, Object)"/>
        /// when this page was initially requested.
        /// </param>
        /// <param name="pageState">
        /// A dictionary of state preserved by this page during an earlier
        /// session.  This will be null the first time a page is visited.
        /// </param>
        public LoadStateEventArgs(Object navigationParameter, Dictionary<string, Object>
pageState)
            : base()
        {
            this.NavigationParameter = navigationParameter;
            this.PageState = pageState;
        }
    }
    /// <summary>
    /// Class used to hold the event data required when a page attempts to save state.
    /// </summary>
    public class SaveStateEventArgs : EventArgs
    {
        /// <summary>
        /// An empty dictionary to be populated with serializable state.
        /// </summary>
        public Dictionary<string, Object> PageState { get; private set; }

        /// <summary>
        /// Initializes a new instance of the <see cref="SaveStateEventArgs"/> class.
        /// </summary>
        /// <param name="pageState">An empty dictionary to be populated with serializable
state.</param>
        public SaveStateEventArgs(Dictionary<string, Object> pageState)
            : base()
        {
            this.PageState = pageState;
        }
}
```

The last step is to simply update all our views to inherit from this base class:

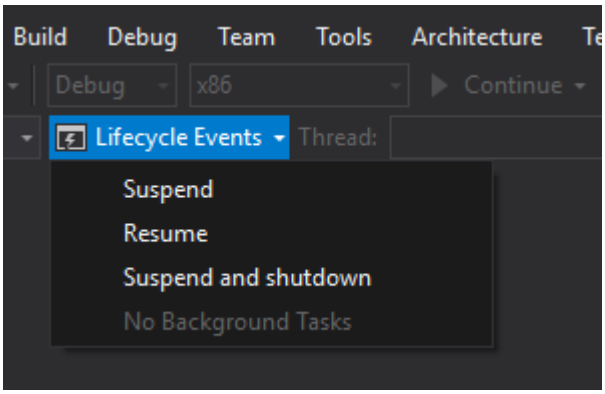
```
public sealed partial class MainPage : ViewBase
{
      // ...
}

public sealed partial class SecondPage : ViewBase
{
     // ...
}
```

Note that since pages are partial classes you need to make sure that both the XAML and the code behind inherit from ViewBase:

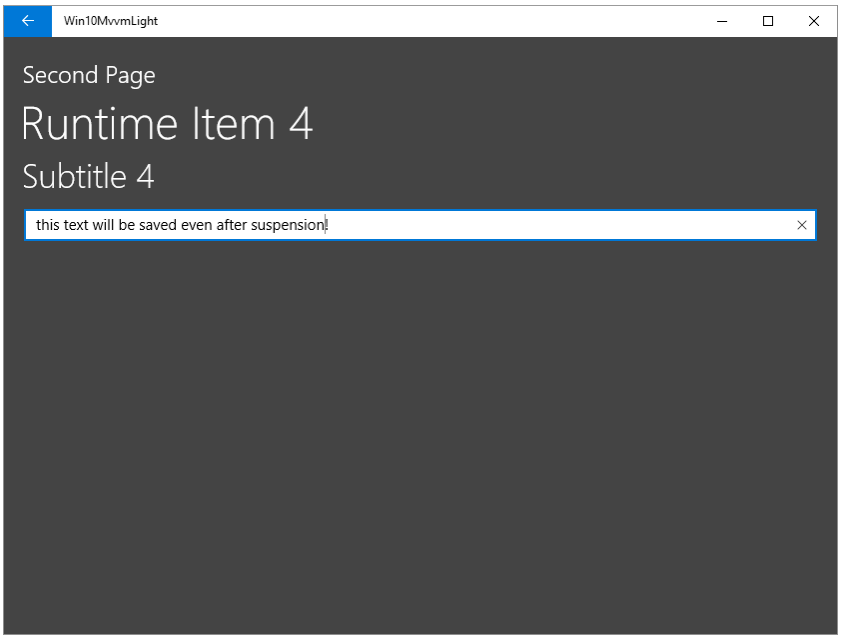
```
<local:ViewBase
    x:Class="Win10MvvmLight.Views.SecondPage"

    <!-- ... -->

    DataContext="{Binding Path=SecondPage, Source={StaticResource ViewModelLocator}}">

    <!-- ... -->
</local:ViewBase>
```

Now we can remove the code from the code-behind related to state management so that it is fully driven by the LoadState and SaveState methods in each individual ViewModel. Best of all, since these events only fire during actual navigation, our Design Time data still remains available in both Visual Studio and Blend.

And of course, if we now run the app, navigate to the second page and enter some text, we can use the Visual Studio Lifecycle events menu to suspend the app:




Then launch the app again to verify that not only is our navigation state fully restored, but our TextBox content was also preserved:




## Generics in XAML

Since each ViewModel derives from a specific type, and we have a generic base class from which all our pages will derive, you might be tempted to define ViewBase as a generic <T> where T inherits from ViewModelBase. Unfortunately, although XAML does appear to have support for generics, this support does not yet appear to extend to Windows Store apps, so you must leave the reference to the ViewModel in the page DataContext as the base version.

However, if you do need to add a reference to your ViewModel in the code-behind, you can easily add a ViewModel property to the page, and cast the base to the appropriate type:

```
public sealed partial class SecondPage : Page
{
        SecondPageViewModel ViewModel
        {
                get
                {
                        return PageViewModel as SecondPageViewModel;
                }
        }

        // ...
}
```

That way the ViewBase can still call all the state management events automatically underneath the covers, but at the individual page level, you can have a strongly-typed reference to the specific ViewModel associated with the page.

## Wrapping Up and Next Steps

We now have a complete system that allows a Windows 10 application to manage navigation and application state, as well as an intuitive mapping of ViewModels to Pages including design time data to aid in laying out the application.

However, we have one more step to complete this framework, which is to leverage the NavigationService in MvvmLight to facilitate the navigation between pages. We'll see how to achieve this, including supporting use of the ViewModels in other platforms in the next chapter.

# MvvmLight NavigationService and the Behaviors SDK

Following from the previous chapter, we now have a solid framework for our app to handle state and lifecycle. However, at this point we are still navigating the app directly from the code-behind of the Views, which ties the navigation to the platform code. In addition to cluttering up our code, this also restricts us from fully taking advantage of the cross-platform opportunities offered by MvvmLight.

In this chapter we'll see how to centralize this navigation code, removing the platform-specific definition and moving from the code-behind to the ViewModels, allowing maximum reuse of code. We'll begin with the code related to navigation.

## NavigationService

MvvmLight includes a cross-platform implementation of a navigation service which provides a device-agnostic way to perform navigation, allowing us to refactor that code into the ViewModels. The result is a fully portable, cross-platform solution.

Up until now we've been navigating directly against the Frame container for the application, which accepts as an argument the type of the intended page, as well as a single parameter. Obviously this implementation cannot be cross platform, since the pages are defined only in the Windows project.

Instead, the NavigationService in MvvmLight accepts a key of type string to identify the intended destination, which is resolved at runtime to the desired page. This is accomplished by first registering each page at startup, associating it with the specific key. You would do this once within each platform, reusing the same key when registering the destination so that they can be called consistently from the ViewModel.

For convenience, and to avoid typos, by convention I use the ViewModels themselves as the key, which intuitively links each one to its appropriate page. Here's an example of the navigation registration code for our sample project:

```
protected INavigationService InitNavigationService()
{
    var service = new NavigationService();

    service.Configure(typeof(MainPageViewModel).FullName, typeof(MainPage));
    service.Configure(typeof(SecondPageViewModel).FullName, typeof(SecondPage));

    return service;
}
```

This registration needs to happen at startup so that it is available immediately throughout the project, and it's perfectly acceptable to do this in the ***OnLaunched*** or similar event. However, since the ***ViewModelLocator*** we previously created is defined as a static resource in the App.xaml file, it is automatically instantiated and registered at application startup, so this seems like the perfect place to register the views.

Unfortunately, our current ViewModelLocator is in the portable project, obviously so that it can be leveraged on other platforms. Instead, we'll create an inherited version of the locator and extend it with the platform-specific code to register the ***NavigationService*** and register all the pages.

Reusing the navigation code in other platforms is as simple as creating a new version of the locator for that platform and registering it with that the NavigationService implementation for that platform.

## WindowsViewModelLocator

Implementing the locator for Windows is fairly simple; we only need to inherit from the ***BaseViewModelLocator*** we already defined, and of course, ensure that the inherited base constructor executes (which we need to register all the ViewModels).

In the inherited constructor, we can proceed to register the NavigationService, and any other platform-specific code that needs to be in place at startup.

We register the NavigationService by binding it to the ***INavigationService*** interface from the portable project, and pass to it a factory that initializes the service with the registered views. Here's the complete code for the inherited locator.

```
public class WindowsViewModelLocator : BaseViewModelLocator
{
    public WindowsViewModelLocator() : base()
    {
        if (ViewModelBase.IsInDesignModeStatic)
        {
            SimpleIoc.Default.Register<INavigationService, NavigationService>();
        }
        else
        {
            var navigationService = InitNavigationService();
            SimpleIoc.Default.Register<INavigationService>(() => navigationService);
        }
    }

    protected INavigationService InitNavigationService()
    {
        var service = new NavigationService();

        service.Configure(typeof(MainPageViewModel).FullName, typeof(MainPage));
        service.Configure(typeof(SecondPageViewModel).FullName, typeof(SecondPage));

        return service;
    }
}
```

Notice we are again checking if we are in the designer, and if so, skip the navigation registration and register the default NavigationService. This is mostly due to a bug that causes multiple instances to be registered when using a factory, resulting in the error "INavigationService is already registered".

Skipping the navigation registration frees the designer from any clutter not necessary at design-time so it's a good practice to follow.

The last thing we need to do with the new locator is register it as a static resource, replacing the previous base locator in App.xaml:

```
<Application
    x:Class="Win10MvvmLight.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Win10MvvmLight"
    xmlns:vm="using:Win10MvvmLight.ViewModels"
    RequestedTheme="Light">
    <Application.Resources>
        <ResourceDictionary>
            <vm:WindowsViewModelLocator x:Key="ViewModelLocator" />
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

This way the registration happens automatically as soon as the app fires up.

## Refactoring Navigation to the ViewModels

Now that we have a centralized, cross-platform way to navigate to a screen, we can proceed to refactor our code to the ViewModels to maximize code reuse. The first thing we need to do is add a reference to the NavigationService to the BaseViewModel class, so it is accessible throughout the app.

However, we cannot add the platform-specific NavigationService we just setup, as that version is only for Windows. Instead, we want to once again take advantage of the ***ServiceLocator*** to instead define the reference by its interface, which will be resolved at runtime to the platform-specific version.

We can do that by simply adding the following property to the BaseViewModel:

```
protected INavigationService NavigationService { get { return
ServiceLocator.Current.GetInstance<INavigationService>(); } }
```

From there can call ***NavigateTo*** on the service to execute the navigation to a specified page. It would look something like this:

```
NavigationService.NavigateTo(typeof(SecondPageViewModel).FullName, itemId);
```

To make things simpler I added a generic helper method to make it even easier to call by simply specifying the type and optional parameter to the method:

```
public void Navigate<T>(object argument = null)
{
    if (argument == null)
        NavigationService.NavigateTo(typeof(T).FullName);
    else
        NavigationService.NavigateTo(typeof(T).FullName, argument);
}
```

Now that we have this in order, we can easily execute navigation from the individual ViewModels using another helpful feature of MvvmLight

### RelayCommand

The RelayCommand in MvvmLight implements the ***ICommand*** interface to allow you to fire events including strongly-typed parameters, exposing them to your views to be executed by UII events (such as a button click).

A thorough discussion of Commands is outside the scope of this book, but if you are new to this concept I highly recommend you take a look at this extensive article on MSDN that goes into incredible detail on the pattern: Commands, RelayCommands and EventToCommandCommands, RelayCommands and EventToCommand.

The RelayCommand in MvvmLight can be either fire a generic method, or send a strongly typed parameter via the generic form RelayCommand. In additon, a RelayCommand can be initialized with a seperate delegate to determine whether or not a command should be allowed to execute.

This is helpful if you want to disable a command for a specific reason, such as disabling a “Refresh” button while a ViewModel is in the “Loading” state, as this would likely mean that the command is currently already executing.

We don’t have any need to disable such commands in our example; we just need a way to select an item from the list on the MainPage and navigate to its details. Since this requires a specific item, we want to use the generic RelayCommand with our TestItem type so that we can use its ID property to properly navigate. Here’s what the command looks like:

```
private RelayCommand<TestItem> selectItemCommand;
public RelayCommand<TestItem> SelectItemCommand
{
    get
    {
        return selectItemCommand ?? (selectItemCommand = new
RelayCommand<TestItem>((selectedItem) =>
        {
            if (selectedItem == null) return;

            Navigate<SecondPageViewModel>(selectedItem.Id);
        }));
    }
}
```

Now that we have a command to navigate, we need a way to trigger it. One perfectly acceptable way to do this would be to replace our previous code-behind that fires on event-click to execute the command instead. It might look something like this:

```
private void listView_ItemClick(object sender, ItemClickEventArgs e)
{
    var vm = DataContext as MainPageViewModel;
    vm.SelectItemCommand.Execute(null);
}
```

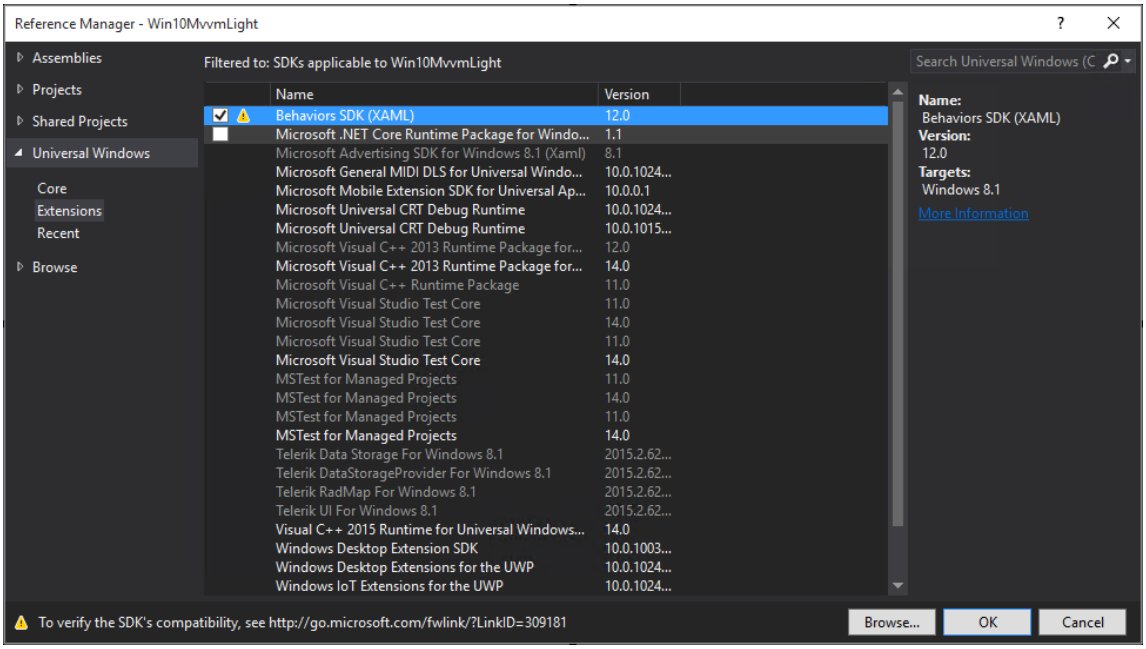
While this gets the job done, there's a more elegant solution made possible by leveraging the XAML Behaviors SDK.

## Behaviors SDK

The Behaviors SDK has its roots in the Expression Blend SDK that allowed such behaviors and actions in XAML. For an in-depth tour of the SDK and its features, I recommend taking a look at this this post on Behaviors SDK by Timmy Kokke, which even describes how to build your own behaviors and actions.

But for our simple project, we simply want to leverage the ***EventTriggerBehavior*** to associate a specific event — in this case ItemClick of the ListView on the MainPage — with the command.

First, we need to make sure we add a reference to the SDK to the project:



Then we need to add the appropriate namespaces for the behavior we want to use from the SDK, which are ***Interactivity*** and Core:

```
<local:ViewBase
    <!-- ... -->

    xmlns:Interactivity="using:Microsoft.Xaml.Interactivity"
    xmlns:Core="using:Microsoft.Xaml.Interactions.Core"

    <!-- ... -->
>
```

At last we can proceed to use these to attach the behaviors to the ListView, associating the ***ItemClick*** event with the following XAML:

```
<ListView x:Name="listView"
  ItemTemplate="{StaticResource TestItemTemplate}"
  ItemsSource="{Binding TestItems}"
  Margin="19,12,19,0"
  IsItemClickEnabled="True">
     <Interactivity:Interaction.Behaviors>
          <Core:EventTriggerBehavior EventName="ItemClick">
               <Core:InvokeCommandAction Command="{Binding SelectItemCommand}"
InputConverter="{StaticResource ItemClickEventArgsConverter}"/>
          </Core:EventTriggerBehavior>
     </Interactivity:Interaction.Behaviors>
</ListView>
```

There's one very important property that we haven't yet covered, which is the **InputConverter** property. If you leave this out and attempt to call the command without converting the arguments, you'll get an error similar to this:

> Unable to cast object of type 'Windows.UI.Xaml.Controls.ItemClickEventArgs' to type 'Win10MvvmLight.Portable.Model.TestItem'

The reason this happens is that the argument of the **ItemClick** event is of type **ItemClickEventArgs**, but the **SelectItemCommand** is expecting it to be of type **TestItem**. The InputConverter property lets you specify a class that will convert the arguments to the appropriate type.

This is simply an implementation if ***IValueConverter*** and in this case simply gets the clicked item out of the arguments from the click event, and passes it — cast to the appropriate type of course — to the command. Here's what it looks like:

```
public sealed class ItemClickEventArgsConverter : IValueConverter
{
        public object Convert(object value, Type targetType, object parameter, string
language)
        {
            var args = value as ItemClickEventArgs;
            if (args == null)
                throw new ArgumentException("Value is not ItemClickEventArgs");
            if (args.ClickedItem is TestItem)
            {
                var selectedItem = args.ClickedItem as TestItem;
                return selectedItem;
            }
            else
                return null;
        }

        public object ConvertBack(object value, Type targetType, object parameter, string
language)
        {
            throw new NotImplementedException();
        }
}
```

Be sure that this class is also registered as a static resource in App.xaml, so that it can be referenced in the XAML above.

With this last piece of the puzzle in place, we can now clear out ALL the code-behind from both pages, as all of the code necessary for loading the pages AND navigating between them is entirely defined in the portable project!

## Wrapping Up and Next Steps

By registering our views with the NavigationService offered by Mvvm Light, we have a portable, cross-platform way to perform app navigation, separating the code from the platform-specific views. Since this service is implemented on various platforms, including Xamarin, we can use the exact same ViewModels across different devices without changing a single line of code in the portable project.

We'll come back to this in a later publication, showing how we can extend this project to other platforms with Xamarin Forms. In the meantime, we'll take a break from the sample project to look closer at some of the new controls available in Windows 10, as well as how we leveraged them (and built new ones!) in our Falafel2Go app.

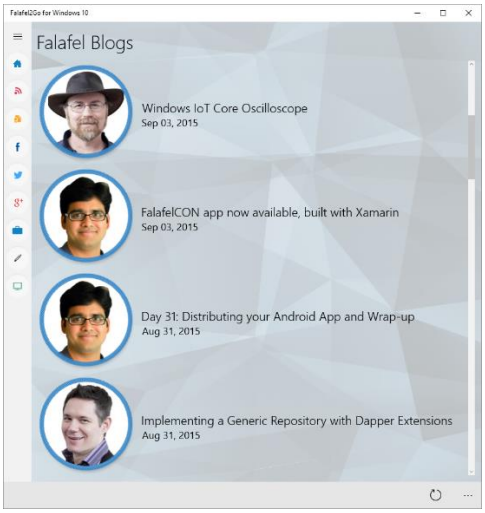
# The SplitView Control

Now that we have a simple but solid foundation for a complete Windows 10 app, let's take a tour through some of the brand new controls available on the platform. Today we'll look at the SplitView control, which enables you to quickly create a consistent, intuitive navigation UI that can automatically adjust to different screen sizes and device platforms.

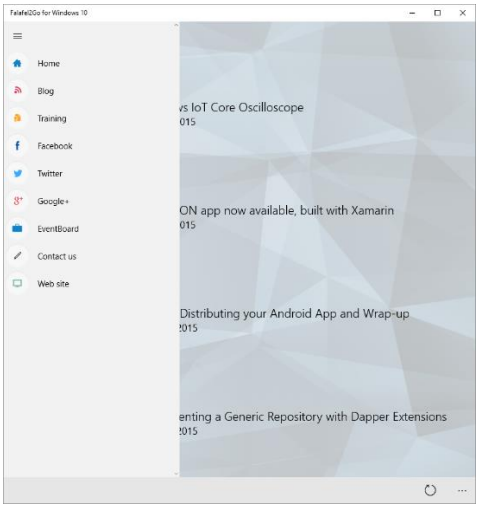
## SplitView

The most common function of the SplitView control is to provide a familiar, responsive navigation structure to an application. As the controls name suggests, it consists of two separate views, with a larger Content view to contain the main content of the screen, and a smaller, docked Pane section generally reserved for your navigation menu.

The Windows 10 version of Falafel2Go takes full advantage of this control and serves as a great example of how you can use the SplitView for navigation. Here's what it looks like on the desktop, with the Pane section collapsed to a traditional vertical sidebar menu.

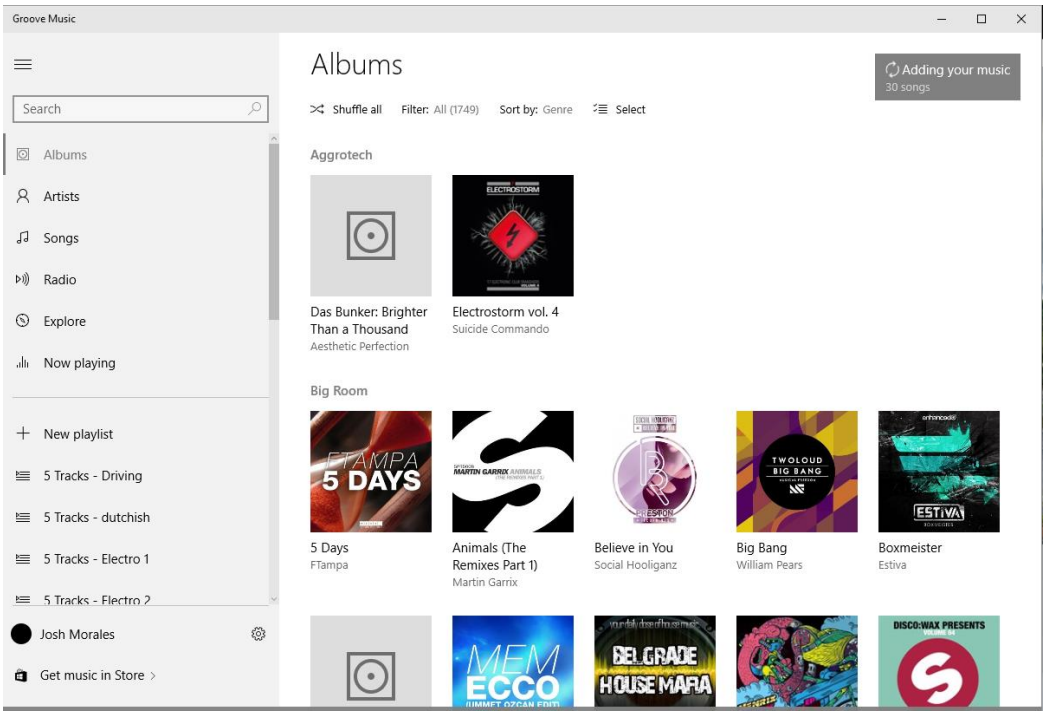



The menu button shown triggers the menu to expand and reveal more context about the navigation options:

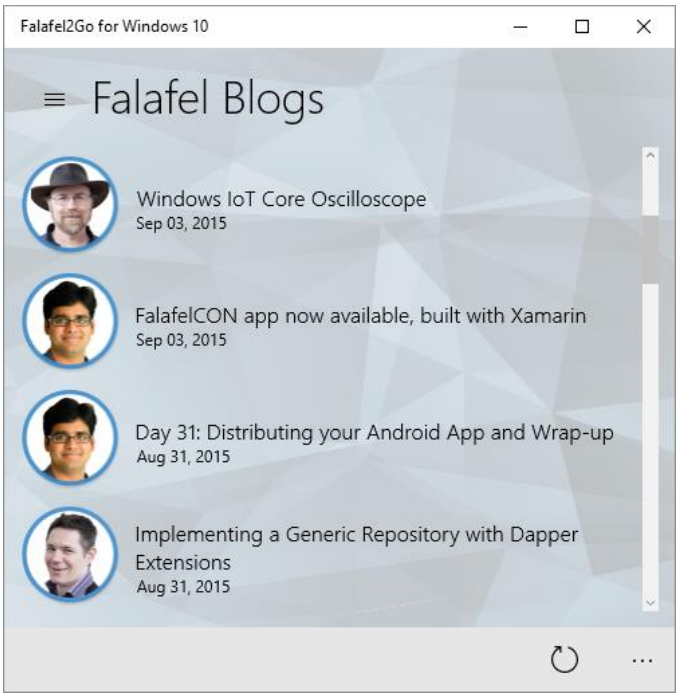



However, the SplitView offers many configuration options to present the control in different ways depending on the UI required by your app.

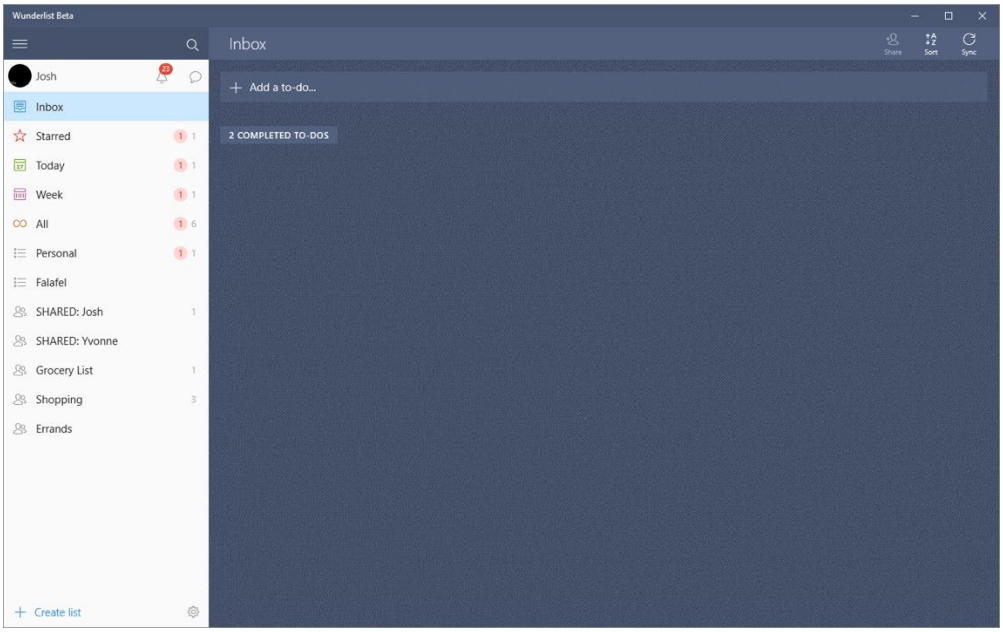
For example, the Pane can be expanded by default to expose a more interactive side bar, as demonstrated by the Groove Music app:




Or collapsed down to a button to support smaller and mobile devices:




However, the SplitView isn't limited to just navigation. Many apps make creative use of the Pane section to add more context and interaction to the main Content section, as shown here in the Wunderlist app:




### XAML

Defining and using a SplitView with XAML is intuitive and easy as shown here with a minimal declaration of the control markup:

```
<SplitView>
  <SplitView.Content>

  </SplitView.Content>
  <SplitView.Pane>

  </SplitView.Pane>
</SplitView>
```

The Content and Panel sections represent the two respective panes for the main content of your app and the collapsible area usually reserved for navigation.

Configuring the layout simply involves setting a few of the available properties for each panel.

### Display Mode

The Panel exposes several properties to control how it appears to the user, which can be configured by setting the appropriate DisplayMode property, combined with the IsPaneOpen property to show or hide it. For the Panel section, there are currently four options:

#### *Inline*

In this mode, the side bar is fully expanded, giving a full, side-by-side view of both areas of content.

#### *CompactInline*

This option collapses the panel to a configurable compact size when closed, but when expanded it “pushes” the main Content area aside, revealing both areas fully, similar to the Inline mode.

#### *Overlay*

This mode hides the Pane area completely when it is closed. When expanded, it will display the Pane over the Content, covering the area below. Tapping outside the opened Pane area will dismiss and close the Panel, hiding it from view.

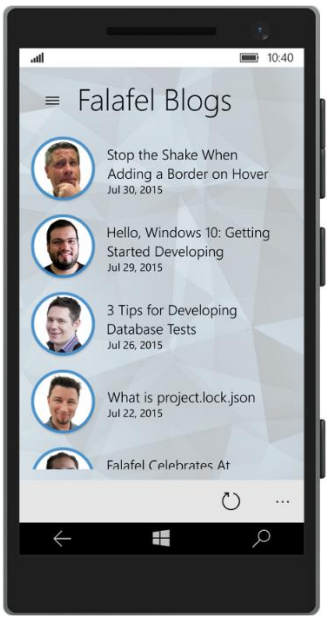
Usually this display mode is paired with a menu button to toggle the visibility of the Panel, as we’ll see shortly.

### *CompactOverlay*

Finally, this mode is similar to CompactInline, in that it collapses the menu to a smaller pane. However, when the Pane is opened, it will overlay the Content area instead of pushing it aside.

### Responsive Layouts

Since the control supports all these different visual arrangements, you can configure it to adjust itself on the fly using Visual States to automatically adjust itself to best accommodate the screen size, device type, or application window size, as we did with Falafel2Go, all the way down to mobile.




We'll take a closer look at Visual States and how we achieved these different layouts for Falafel2Go in a future chapter.

## Adding a Hamburger Menu

The SplitView control lends itself to the common UI pattern of a menu button, often referred to as the "hamburger button" to toggle the visibility of the Panel menu. This is especially useful in mobile devices which have limited real estate as shown in the Falafel2Go image above.

The SplitView does not natively expose such a control to toggle its visibility; instead the developer is responsible for creating this control on the canvas to trigger the events. In Falafel2Go we achieved by simply adding a Button control to the Pane section:

```
<SplitView.Pane>
    <StackPanel>
        <Button x:Name="MenuButton" FontFamily="Segoe MDL2 Assets" Content="·"
Style="{StaticResource SplitViewButton}" Click="MenuButton_Click" />
                <!-- ... -->
        </StackPanel>
</SplitView.Pane>
```

Notice that we're using the Segoe MDL2 Assets font, which contains a wide variety of glyph assets you can use for icons in your app.

In the Click event for the button, we simply update the DisplayMode properties of the SplitView to reveal or hide the menu as appropriate:

```
private void MenuButton_Click(object sender, RoutedEventArgs e)
{
        PageSplitView.IsPaneOpen = !PageSplitView.IsPaneOpen;
}
```

Since on mobile, the Pane section is completely hidden when collapsed, we added another copy of the button to the canvas of the Content section, and use Visual States to show it only on mobile devies.

We also could have also leveraged the VisualStateManager to define this behavior declaratively in the XAML, but we'll look at that in a future chapter.

## Wrapping Up and Next Steps

The SplitView is a handy and welcome new control to the Windows developer toolbox, which makes it easy to create navigation menus and sidebars for your application that can automatically adjust to accommodate different platforms, orientations, and screen sizes. The two areas of content are clearly and intuitively declared in XAML and the Pane properties make it easy to customize the control to suit your application.

# The RelativePanel Control

Another new control in the Windows 10 Developer toolbox is the RelativePanel, a layout container which enables flexible positioning between the elements it contains. In this chapter we'll take a closer look at this control and how we used it to build the UI of the Falafel2Go app for Windows 10.

## RelativePanel Positioning

Using the RelativePanel, each element it contains can specify its position in relation to either another element, or docking to the container itself. By leveraging these position properties, you can easily create dynamic layouts in a wide variety of configurations, without having to modify the structure of your original XAML or having to create multiple versions for different layouts.

The layout options are exposed as attached properties to the parent RelativePanel (similar to the Grid.Row and Grid.Column properties for organizing elements within a grid).

### Alignment Options

There are two ways you can align elements inside a relative panel. The first is to use the properties that bind an elements position relative to the container itself, such as:

- RelativePanel.AlignTopWithPanel
- RelativePanel.AlignBottomWithPanel
- RelativePanel.AlignLeftWithPanel
- RelativePanel.AlignRightWithPanel
- RelativePanel.AlignVerticalCenterWithPanel
- RelativePanel.AlignHorizontalCenterWithPanel

These properties are equivalent to the VerticalAlignment and HorizontalAlignment properties, but represent the positioning of elements specifically in the RelativePanel.

The real power of the RelativePanel comes from combining these options with the properties that specify positioning in relation to other elements within the control. These are:

- Above
- Below
- LeftOf
- RightOf

Besides relational positioning of the controls, you can also specify an alignment so that the controls instead "line up" other items, using these properties:

- AlignLeftWith
- AlignRightWith
- AlignTopWith
- AlignBottomWith
- AlignVerticalCenterWith

- AlignHorizontalCenterWith

In order for these properties to work, each control must have its ***x:Name*** property set, so that you can specify which control to which a control should align its position.

For example, in Falafel2Go, we have an ActivityControl that represents the menu buttons on the home screen of the app:


This is made up of two elements (Image and TextBlock) inside of a RelativePanel, and here is the markup for the control:

```
    <RelativePanel d:DataContext="{Binding Blog}"
ScrollViewer.VerticalScrollBarVisibility="Disabled">
        <Image x:Name="Icon" Source="{Binding Icon}"
RelativePanel.AlignHorizontalCenterWithPanel="True" RelativePanel.Above="Text"
Stretch="Uniform" RelativePanel.AlignTopWithPanel="True" />
        <TextBlock x:Name="Text" RelativePanel.AlignBottomWithPanel="True"
RelativePanel.AlignHorizontalCenterWithPanel="True"
                Text="{Binding Title}" RelativePanel.AlignVerticalCenterWithPanel="True"
VerticalAlignment="Center" />
    </RelativePanel>
```

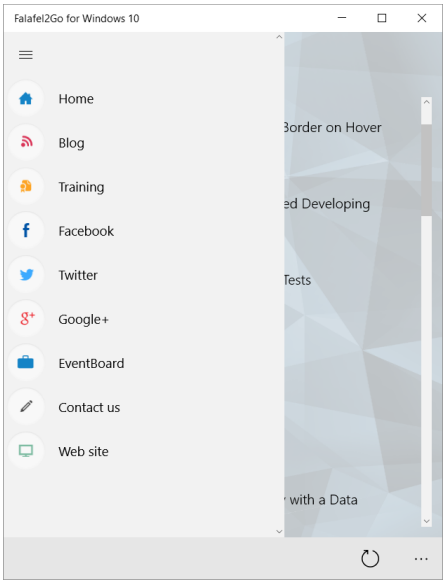
Notice that the Icon control uses the ***x:Name*** property of the “Text” control to position itself above it. By doing this we ensure that both the Text control locks to the bottom (using the ***AlignBottomWithPanel***) and that the Icon control then lies directly above it, and uses the remaining area of the control, stretching to fill it.

## Changing Position

Now at this point you’re probably wondering why we bothered to use the RelativePanel when a Grid would have worked just as easily (and probably been a lot cleaner with simpler markup). While it is true that the Grid can be a better choice for a control like this, the Grid relies on Rows and Columns to position items. This makes it difficult to change positioning, since the new locations must also align within the defined grid.

The RelativePanel on the other hand, makes it easy to move elements around by simply changing the appropriate position properties of the controls to be moved.

This is best demonstrated by the navigation control in Falafel2Go:



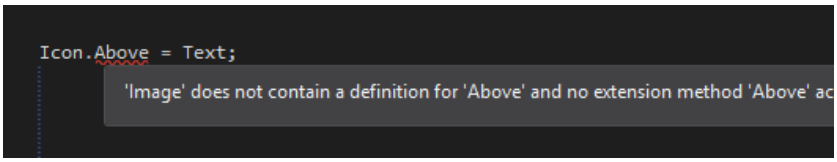

If you look closely, you'll notice that the menu items are identical in content to the activity buttons on the main screen (shown in the screen shot above). The only difference is that the icon is smaller, and positioned to the left of the label, instead of above.

Since the only difference between these controls is the position it made sense to use the RelativePanel, and now we can reuse the exact same controls for both the home screen and navigation.

However, the ActivityControl is itself a UserControl, as there are multiple instances of it on a page (one for each activity). As a result, we don't have a simple way to orient the position from the parent Page. One way might be to expose custom properties for each control (Icon and Text) to pass the layout positioning from the page to the control. However, this proved to be an awkward and unintuitive solution, and certainly wouldn't scale if we decided to add more controls to the Activity buttons later.

Instead, we started by adding a new ***Orientation*** property to the ActivityControl, giving the page a simple way to specify the layout position of the controls.

But how do we actually use this to control the layout? Recall that since the properties related to the RelativePanel positioning are actually attached properties, we can't simply refer to them directly in code. So trying to do something like ***Image.Above*** would result in a compiler error.




We solved this problem by taking advantage of the way that XAML uses dependency properties.

### Dependency Properties

A full discussion of dependency properties is outside the scope of this article, but you can certainly learn more by starting here on MSDN: Dependency properties overview.

All you need to know for the purposes of what we're doing here is that dependency properties are what are responsible for binding the values of things like databinding and attached properties. A dependency is registered and the framework handles keeping everything in sync, allowing you to simply declare the binding or attached property in XAML…

But how do we set these properties in code?

The answer is found here: Attached properties overview. By using the SetValue, ***GetValue***, and ***ClearValue*** methods on a control, we can programmatically change these values, which we do in the setter of the ***Orientation*** property:

```
private Orientation orientation = Orientation.Vertical;
public Orientation Orientation
{
        get { return orientation; }
        set
        {
            orientation = value;
            if (orientation == Orientation.Horizontal)
            {
                Text.Margin = new Thickness(12, 0, 0, 0);
                Icon.ClearValue(RelativePanel.AboveProperty);
                Icon.ClearValue(RelativePanel.AlignHorizontalCenterWithPanelProperty);
                Text.SetValue(RelativePanel.RightOfProperty, Icon);
                Text.ClearValue(RelativePanel.AlignBottomWithPanelProperty);
            }
            else
            {
                Text.Margin = new Thickness(0);
                Icon.SetValue(RelativePanel.AboveProperty, Text);
                Icon.SetValue(RelativePanel.AlignHorizontalCenterWithPanelProperty,
true);
                Text.ClearValue(RelativePanel.RightOfProperty);
                Text.SetValue(RelativePanel.AlignBottomWithPanelProperty, true);
            }
        }
}
```

Since we defaulted the value to Vertical, on the main screen we simply display the control:

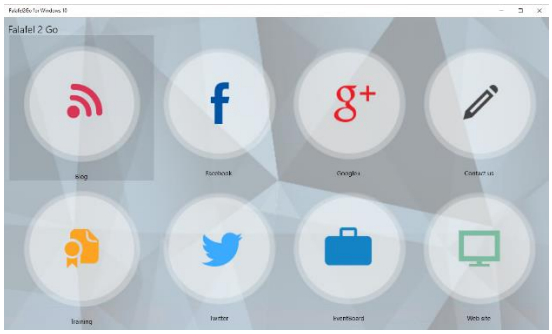
```
<ctrl:ActivityControl DataContext="{Binding Facebook}"   />
<ctrl:ActivityControl DataContext="{Binding Twitter}"    />
<ctrl:ActivityControl DataContext="{Binding Google}"     />
<ctrl:ActivityControl DataContext="{Binding Eventboard}" />
<ctrl:ActivityControl DataContext="{Binding Contact}"    />
<ctrl:ActivityControl DataContext="{Binding Website}"    />
```

While in the NavigationControl, we set the layout to be horizontal (note we also exposed a size property to render the image smaller):

```
<ctrl:ActivityControl DataContext="{Binding Home}"
Orientation="Horizontal"  IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Blog}"
Orientation="Horizontal"  IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Training}" Orientation="Horizontal"
IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Facebook}" Orientation="Horizontal"
IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Twitter}" Orientation="Horizontal"
IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Google}" Orientation="Horizontal"
IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Eventboard}" Orientation="Horizontal"
IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Contact}" Orientation="Horizontal"
IconSize="50" />
<ctrl:ActivityControl DataContext="{Binding Website}" Orientation="Horizontal"
IconSize="50" />
```

By setting the appropriate properties based on the orientation we now have a slick, reusable control that fits two completely different layout needs without having changed a single line of XAML in the original control.

Falafel2Go uses the RelativePanel to create a reusable and responsive navigation, maximizing code reuse by changing the relative properties to fit the required layout. However, if you have a sharp eye, you probably noticed that the main screen itself is also a relative panel, allowing the activity controls to position themselves based on the screen size and orientation.




In order for this to work, however, we need to ensure that each control is the exact same size. We'll see how this was done in the next chapter.

## Wrapping Up and Next Steps

The new RelativePanel in Windows 10 is a powerful and flexible container control, allowing you to specify positioning of its elements relative to each other and/or the container itself. The attached properties offer an intuitive, declarative syntax for laying out the UI, and automating these properties allows your layouts to be flexible and dynamic.

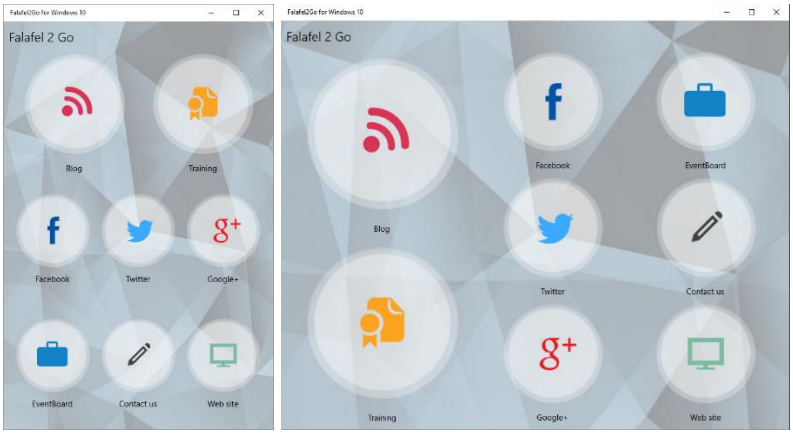
In the next chapter, we'll create a UniformGrid that allowed us to orient the activity controls on the main screen so that they scale to accommodate any screen size.

# Creating a UniformGrid Container

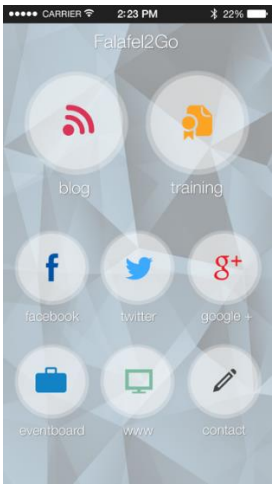
This chapter describes the need for and implementation of a UniformGrid layout control that, when used with a ListView, allows the repeated elements to appropriately stretch to a consistent size to achieve a grid-like layout.

## Grid Layouts: Simple but Static

In the previous chapter we looked at the new RelativePanel control, which we used in the Falafel2Go app for Windows 10 to create a dynamic, responsive layout for the home screen:




You'll notice that in most of the orientations, the Blog and Training elements are featured and grouped together, and therefore render a bit larger than the others. A static layout like this isn't difficult to achieve. In fact, the previous version of this app (built with Xamarin Forms) used nested Grid controls to achieve the layout.




You can read more about this approach here: [Beyond the ListView: Fancy Layouts with Xamarin Forms]

However, as we touched on in the earlier chapter on the RelativePanel, the Grid doesn't easy lend itself to dynamic and flexible layouts. Even something as simple as offering a landscape mode can be a challenge:


Another limitation of using Grid controls was that we could not bind to the list of activities; instead each activity was hard coded into a specific cell in the Grid layout.

It's not obvious from the Falafel2Go for Windows 10 screenshots, but the activities are actually bound to seperate ListView controls, one for the featured Blog and Training sections, and the other to the remaining activities.

This is what it looks like in the XAML, using a RelativePanel to allow the featured list to reorient itself:

```
<RelativePanel Grid.Row="1">
          <ListView x:Name="MainPanel" Style="{StaticResource ActivitiesListView}"
ScrollViewer.VerticalScrollMode="Disabled"
                    RelativePanel.AlignTopWithPanel="True">
                <ctrl:ActivityControl DataContext="{Binding Blog}"    />
        <ctrl:ActivityControl DataContext="{Binding Training}"    />
      </ListView>
          <ListView x:Name="OtherActivitiesPanel" Style="{StaticResource
ActivitiesListView}" ScrollViewer.VerticalScrollMode="Disabled"
                    RelativePanel.Below="MainPanel"
RelativePanel.AlignBottomWithPanel="True" RelativePanel.AlignRightWithPanel="True">
                <ctrl:ActivityControl DataContext="{Binding Facebook}"    />
        <ctrl:ActivityControl DataContext="{Binding Twitter}"    />
        <ctrl:ActivityControl DataContext="{Binding Google}"     />
        <ctrl:ActivityControl DataContext="{Binding Eventboard}" />
        <ctrl:ActivityControl DataContext="{Binding Contact}"    />
        <ctrl:ActivityControl DataContext="{Binding Website}"    />
      </ListView>
</RelativePanel>
```

The problem with this XAML is that we declared the ActivityControl in such a way that the Icon would fill the area above the label, so without any kind of constraints, it's going to stretch to fill the entire screen, yielding an unexpected layout for the screen:


We experimented with different solutions, such as adding properties to set the width and height values of the images. However, since the dimensions would vary wildly based on the screen orientation and size, this proved very

cumbersome. Ultimately we were unable to find a consistent way to handle all possible orientations using the default toolbox for Windows 10.

What we needed was a way for the items in the ListView to automatically calculate their size based on the orientation and available space.

## Changing the Container

We made some progress by overriding the ItemsPanelTemplate, which represents the container panel for the ListView By default this is defined as an ItemsStackPanel, which we can modify by adding a property in the XAML:

```
        <ListView x:Name="MainPanel" Style="{StaticResource ActivitiesListView}"
ScrollViewer.VerticalScrollMode="Disabled" RelativePanel.AlignTopWithPanel="True">
                <ListView.ItemsPanel>
            <ItemsPanelTemplate>
                <ItemsStackPanel Orientation="Horizontal" />
            </ItemsPanelTemplate>
        </ListView.ItemsPanel>
        <ctrl:ActivityControl DataContext="{Binding Blog}"   />
        <ctrl:ActivityControl DataContext="{Binding Training}"    />
    </ListView>
```

However, simply changing the orientation to Horizontal wasn't enough, since we still have the problem of the activity control wanting to stretch to fill the height. This leaves no room for the other ListView beneath and almost no room for the second Training button on the right:


We needed a different container, one that can resize its contents automatically, something like the UniformGrid which was available in WPF, but unfortunately was not ported to WinRT.

## Creating a UniformGrid

The solution was to build our own version of the UniformGrid. We found several helpful resources along the way, including Greg Stoll's Universal Wrap Panel, available on GitHub here: https://github.com/gregstoll/UniversalWrapPanel

Another excellent source of help was this article from Olivier Matis: Create a panel to give all GridView items the maximum width/height which had the breakthrough revelation that by default, ListViewBase controls like ListView and GridView will always use the first item in the list to auto-size the remaining items.

To overcome this, we need to inherit our control from Panel and override the MeasureOverride and ArrangeOverride methods, which are the two key events fired by Layout controls to render the children items in the container.

```
public class UniformGrid : Panel
{
        protected override Size MeasureOverride(Size availableSize)
        {
            double finalWidth, finalHeight;

            // ...

            return new Size(finalWidth, finalHeight);
        }

        protected override Size ArrangeOverride(Size finalSize)
        {
            // ...
            return finalSize;

        }
}
```

We need to use the ***MeasureOverride*** event to actual size of the individual activity controls that each ListView will present, finally returning the overall size of the container (and therefore the parent ListView). Next, the ***ArrangeOverride*** will take those controls and position them in the control, laying them out at specific coordinates calculated from the size of each control.

The result is a uniformly sized and spaced arrangement that takes up exactly the right amount of room based on the screen size.

At this point you're probably wondering how to perform the actual calculation. That is, what are we using as a guide to size the controls in the overrides?

There are many ways to do this, including the default way of simply calculating the rendered size of the first item in the list. Alternatively, the GridView sample linked above calculates the resulting grid when each item is placed in a column, moving items to a new column when the maximum height of the container is reached. The final size of that resulting container is then used to render the controls within based on that calculated size.

In the case of Falafel2Go, while we wanted the controls to size dynamically, we also knew that the grouping of items was fixed: 2 items in the top container, and rows of three in the bottom. So instead of letting the controls drive the calculation, we added properties to specify either the number of Columns (when using a Horizontal Orientation) or the number of ***Rows*** (when using a Vertical Orientation).

```
public int Columns
{
            get { return (int)GetValue(ColumnsProperty); }
            set { SetValue(ColumnsProperty, value); }
}

public int Rows
{
            get { return (int)GetValue(RowsProperty); }
            set { SetValue(RowsProperty, value); }
}

public Orientation Orientation
{
            get { return (Orientation)GetValue(OrientationProperty); }
            set { SetValue(OrientationProperty, value); }
}

public static readonly DependencyProperty ColumnsProperty =
        DependencyProperty.Register("Columns", typeof(int), typeof(UniformGrid), new
PropertyMetadata(1, OnColumnsChanged));


public static readonly DependencyProperty RowsProperty =
        DependencyProperty.Register("Rows", typeof(int), typeof(UniformGrid), new
PropertyMetadata(1, OnRowsChanged));

public static readonly DependencyProperty OrientationProperty =
DependencyProperty.Register("Orientation", typeof(Orientation), typeof(UniformGrid), new
PropertyMetadata(1, OnOrientationChanged));

static void OnColumnsChanged(DependencyObject obj, DependencyPropertyChangedEventArgs e)
{
            int cols = (int)e.NewValue;
            if (cols < 1)
                ((UniformGrid)obj).Columns = 1;
}

static void OnRowsChanged(DependencyObject obj, DependencyPropertyChangedEventArgs e)
{
            int rows = (int)e.NewValue;
            if (rows < 1)
                ((UniformGrid)obj).Rows = 1;
}

static void OnOrientationChanged(DependencyObject obj, DependencyPropertyChangedEventArgs
e)
{
}
```

Using these dependency properties, we can replace the ItemsPanelTemplate with our UniformGrid and tell it exactly how many rows or columns it should use when rendering the list.

```
<RelativePanel Grid.Row="1">
    <ListView x:Name="MainPanel" Style="{StaticResource ActivitiesListView}"
ScrollViewer.VerticalScrollMode="Disabled"
                RelativePanel.AlignTopWithPanel="True">
        <ListView.ItemsPanel>
        <ItemsPanelTemplate>
            <ctrl:UniformGrid Columns="2" Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </ListView.ItemsPanel>
    <ctrl:ActivityControl DataContext="{Binding Blog}"   />
    <ctrl:ActivityControl DataContext="{Binding Training}"    />
    </ListView>
    <ListView x:Name="OtherActivitiesPanel" Style="{StaticResource ActivitiesListView}"
ScrollViewer.VerticalScrollMode="Disabled"
                RelativePanel.Below="MainPanel" RelativePanel.AlignBottomWithPanel="True"
RelativePanel.AlignRightWithPanel="True">
        <ListView.ItemsPanel>
            <ItemsPanelTemplate>
                <ctrl:UniformGrid Columns="3" Orientation="Horizontal" />
            </ItemsPanelTemplate>
        </ListView.ItemsPanel>
        <ctrl:ActivityControl DataContext="{Binding Facebook}"   />
        <ctrl:ActivityControl DataContext="{Binding Twitter}"    />
        <ctrl:ActivityControl DataContext="{Binding Google}"     />
        <ctrl:ActivityControl DataContext="{Binding Eventboard}" />
        <ctrl:ActivityControl DataContext="{Binding Contact}"    />
        <ctrl:ActivityControl DataContext="{Binding Website}"    />
    </ListView>
</RelativePanel>
```

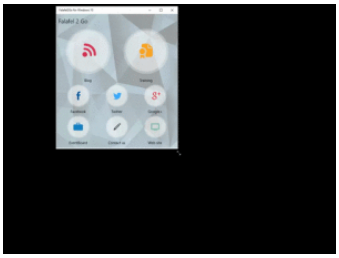
At last we can implement the ***MeasureOverride*** and ***ArrangeOverride*** to calculate the controls based on the number of items that should go in each row or column.

```
protected override Size MeasureOverride(Size availableSize)
{
    double finalWidth, finalHeight;

    if (this.Orientation == Orientation.Horizontal)
    {
        finalWidth = availableSize.Width;
        var itemWidth = Math.Floor(availableSize.Width / Columns);
        var actualRows = Math.Ceiling((double)Children.Count / Columns);
        var actualHeight = Math.Floor((double)availableSize.Height / actualRows);
        var itemHeight = Math.Min(actualHeight, itemWidth);

        foreach (var child in Children)
        {
            child.Measure(new Size(itemWidth, itemHeight));
        }

        finalHeight = itemHeight * actualRows;
    }
    else
    {

        finalHeight = availableSize.Height;
        var itemHeight = Math.Floor(availableSize.Height / Rows);
        var actualColumns = Math.Ceiling((double)Children.Count / Rows);
        var actualWidth = Math.Floor((double)availableSize.Width / actualColumns);
        var itemWidth = Math.Min(actualWidth, itemHeight);
        finalWidth = itemWidth * actualColumns;

        foreach (var child in Children)
        {
            child.Measure(new Size(itemWidth, itemHeight));
        }
    }

    return new Size(finalWidth, finalHeight);
}

protected override Size ArrangeOverride(Size finalSize)
{
    if (this.Orientation == Orientation.Horizontal)
    {
        var actualRows = Math.Ceiling((double)Children.Count / Columns);
        var cellWidth = Math.Floor(finalSize.Width / Columns);
        var cellHeight = Math.Floor(finalSize.Height / actualRows);
        Size cellSize = new Size(cellWidth, cellHeight);
        int row = 0, col = 0;
        foreach (UIElement child in Children)
        {
            child.Arrange(new Rect(new Point(cellSize.Width * col, cellSize.Height *
row), cellSize));
            var element = child as FrameworkElement;
            if (element != null)
            {
                element.Height = cellSize.Height;
                element.Width = cellSize.Width;
            }

            if (++col == Columns)
            {
                row++;
                col = 0;
            }
```

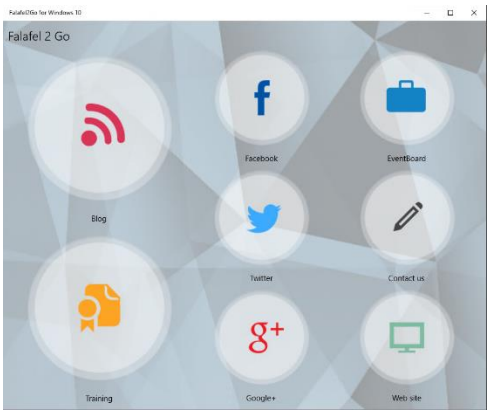
```
        }
    }
    else
    {
        var actualColumns = Math.Ceiling((double)Children.Count / Rows);
        var cellWidth = Math.Floor(finalSize.Width / actualColumns);
        var cellHeight = Math.Floor(finalSize.Height / Rows);
        Size cellSize = new Size(cellWidth, cellHeight);
        int row = 0, col = 0;
        foreach (UIElement child in Children)
        {
            child.Arrange(new Rect(new Point(cellSize.Width * col, cellSize.Height *
row), cellSize));
            var element = child as FrameworkElement;
            if (element != null)
            {
                element.Height = cellSize.Height;
                element.Width = cellSize.Width;
            }

            if (++row == Rows)
            {
                col++;
                row = 0;
            }
        }
    }
    return finalSize;

}
```

Notice here that we're using the same values from the calculated width (for horizontal layouts) to populate the height, so that the actual space occupied is a square (and therefore, uniform). The same goes for the calculated height (for vertical layouts) to assign an equal width.

As a result, because the featured ListView for Blogs and Training only have two items, they will use half the width of the screen (as opposed to one-third for the ListView items below) and we are guaranteed that they will render larger, no matter what the size of the screen is, even as we drag it freely (click the image below to see the animated gif)!



When the screen is oriented horizontally (with width being larger than height), the relative panel adjusts to place the featured buttons to the left, and orients the list vertically, giving 50% of the height to each button.




We'll see how we achieved this automation in the next chapter.

## Wrapping Up and Next Steps

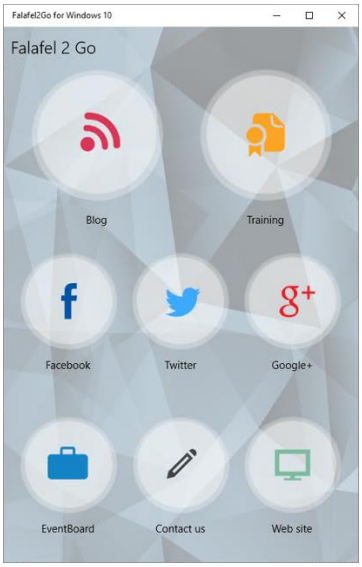
By default, the ListViewBase controllers like ListView and GridView size the item containers based on the first item in the list. This is problematic if you have variable-sized items in the list, or if you want the items to be uniformly sized relative to the container (rather than their contents).

Creating a UniformGrid control solves this by customizing the ***MeasureOverride*** and ArrangeOverride events of the Panel that are responsible for sizing and arranging its contents.

We can take this one step further and allow the control to automatically re-orient itself between columns and rows while maintaining the fixed distribution in either direction. This can be done by leveraging the VisualStateManager, which is the subject of the following chapter.

# UI Automation with Blend and VisualStateManager

In the previous chapter we built a UniformGrid container for the ListView, resulting in an evenly-spaced, grid-like view for the main screen of Falafel2Go:




This screen is actually made up of two such ListViews in a RelativePanel, allowing it to reposition the lists to support a landscape view, while still keeping the larger size for the featured Blog and Training activities.


This layout switch happens automatically as the screen size changes (or, on a phone, if you rotate the screen to its side). In this chapter, we'll see how we can achieve similar automation using Blend and the VisualStateManager.

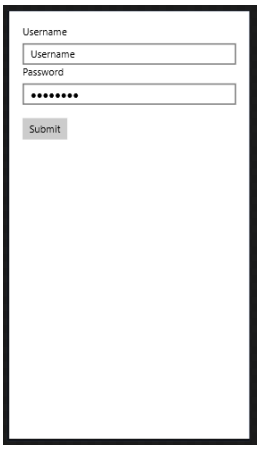
## VisualStateManager

A thorough review of the ***VisualStateManager*** is outside the scope of this book; for that I recommend you start with MSDN: VisualStateManager class.

But for the purposes of our example, what you need to know is that the **VisualStateManager** is used to define, manage, and transition between different states of controls on a page. Specifically, a **VisualState** refers to the collection of XAML control properties which together define a particular state of the application.

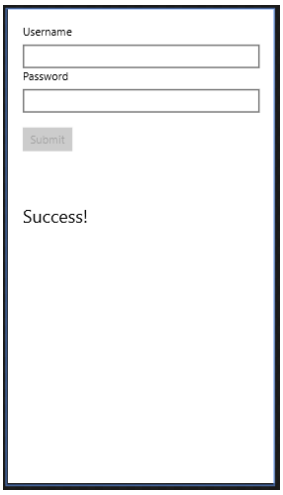
For example, a Registration screen might show Textbox controls to input the desired Username and Password, as well as a Button control to submit. When the Textbox controls are empty, the Button should be disabled since the form is not valid. Having that button initially disabled would represent one **VisualState**.


After the user fills in both Textbox controls with values, the Button can then be enabled, representing a different **VisualState**.




Once the button is pressed, the application my reject the credentials, perhaps because the Username has already been registered. The Submit button could then be disabled again, and the Username Textbox would change to reveal a Red border indicating the error in yet another **VisualState**:


Finally, when the registration is successful, both Textbox controls can be cleared, the Button disabled, and a success message can be shown, revealing the finalVisualState for the application:




Now that we understand how Visual States can help define, organize, and transition the UI, let's see how easy it is to create them.

Certainly we could use the code-behind to manipulate the individual properties, but not only is this tedious and error prone, thanks to Blend, it's also completely unnecessary.

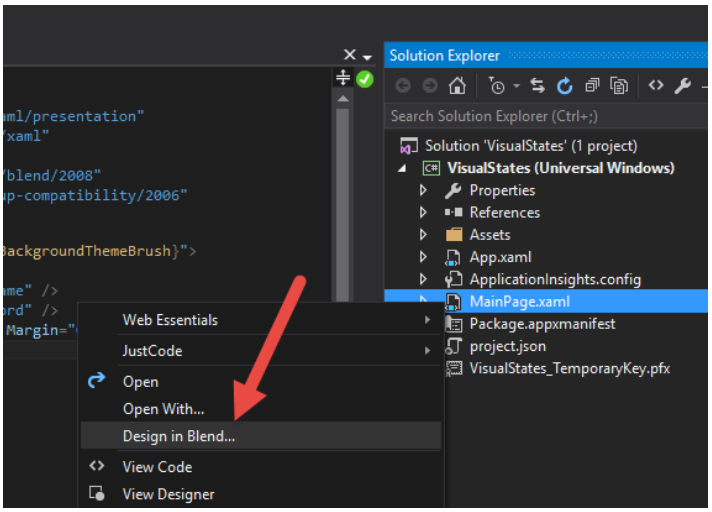
## Creating Visual States with Blend

As we'll see shortly, Visual States are defined in XAML, and as such it is certainly also possible to construct the complete array of Visual Statues or your application requires manually.

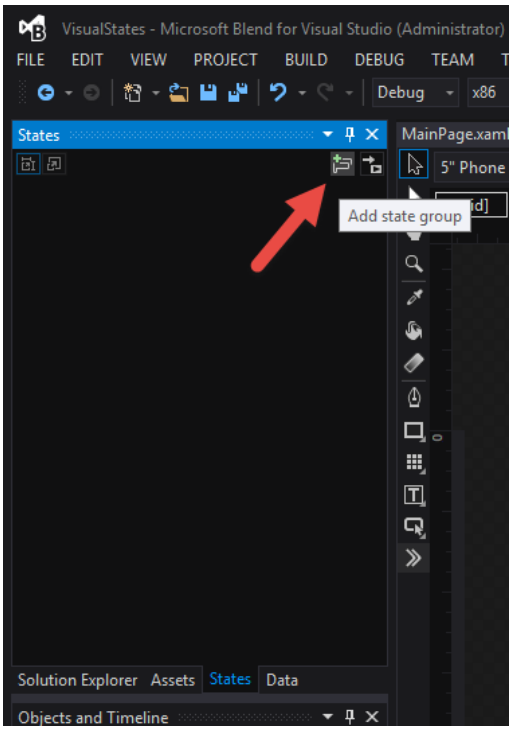
However, you're definitely going to want to instead take advantage of the Blend for Visual Studio companion app that comes with all versions of Visual Studio. As we've seen before, this application is similar to Visual Studio in that you can open the same project and solution for your app to view and edit code, XAML and other resources. But while Visual Studio is optimized for editing code and developer needs, Blend is tailored specifically to managing the design experience.

Arguably one of the most powerful features of Blend is its ability to easily define and manage the Visual States of your application.

To begin simply open the page you wish to manage in Blend. This can be launched directly from Visual Studio from the context menu:

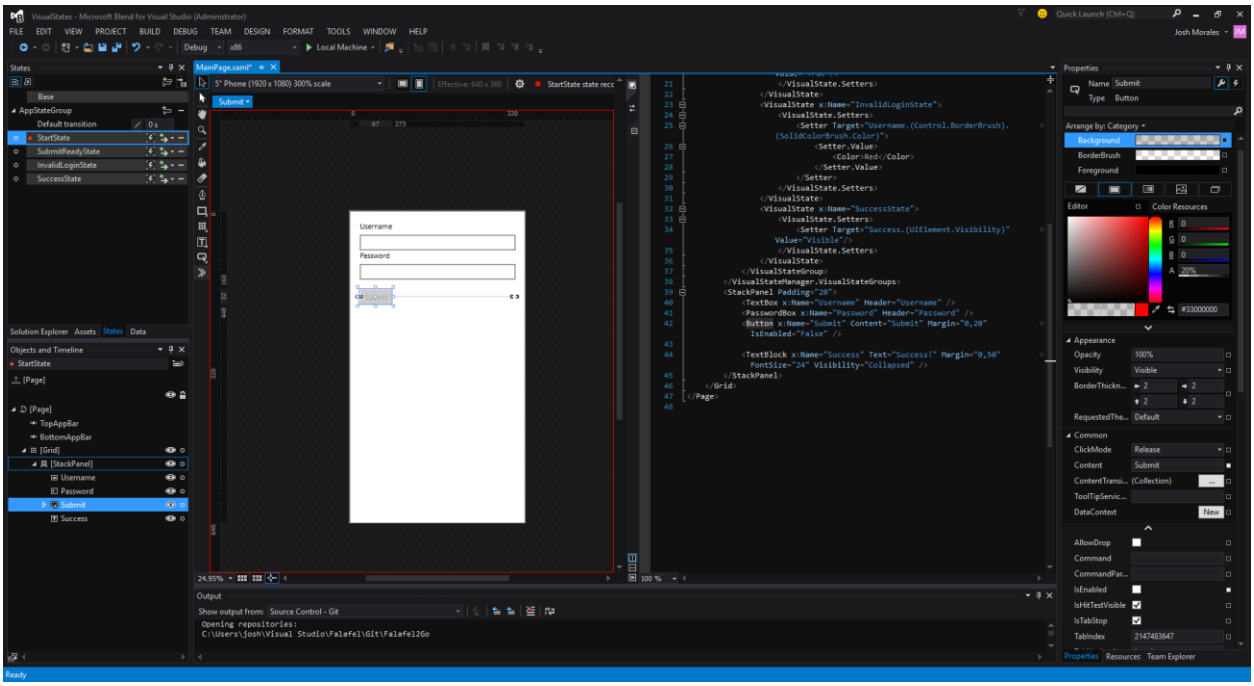



Then open the ***States*** window within the Blend interface and add a new state group:




A ***StateGroup*** is a container for the different states of the page. The states within a StateGroup are mutually exclusive; that is, only one can be active at any given time.

Once a StateGroup is declared you can proceed to add one or more VisualStates to the group. For the example app described above, we'll define four states:



Selecting a state from the menu activates a "recorder", indicated by the red border shown around the designer in the screenshot above. When this is active, any changes you make to the controls on the page with be recorded and saved as part of the currently selected state.

In the screenshot above we've selected the *StartState* which if you'll look closely, sets the ***IsEnabled*** property of the button to false. Similar property changes are made for the remaining states to complete the required UI changes.

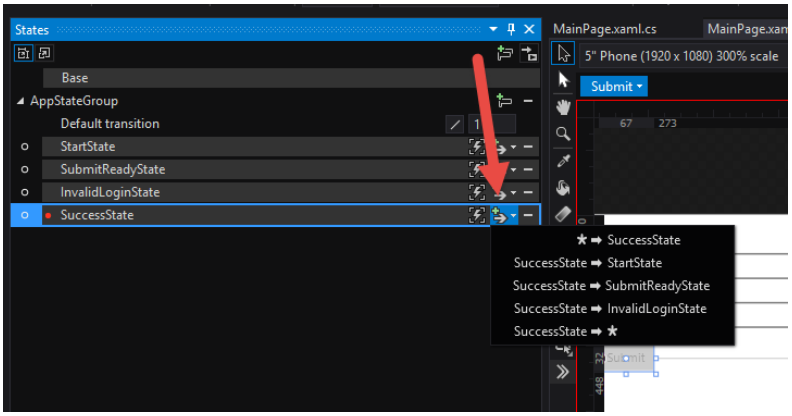
And as mentioned before, ultimately, these changes are saved as XAML, with Blend doing all the heavy lifting to write out the correct syntax as shown here:

```
  <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="AppStateGroup">
            <VisualState x:Name="StartState">
                <VisualState.Setters>
                    <Setter Target="Submit.(Control.IsEnabled)" Value="False"/>
                    <Setter Target="Success.(UIElement.Visibility)" Value="Collapsed"/>
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="SubmitReadyState">
                <VisualState.Setters>
                    <Setter Target="Submit.(Control.IsEnabled)" Value="True"/>
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="InvalidLoginState">
                <VisualState.Setters>
                    <Setter
Target="Username.(Control.BorderBrush).(SolidColorBrush.Color)">
                        <Setter.Value>
                            <Color>Red</Color>
                        </Setter.Value>
                    </Setter>
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="SuccessState">
                <VisualState.Setters>
                    <Setter Target="Success.(UIElement.Visibility)" Value="Visible"/>
                    <Setter Target="Submit.(Control.IsEnabled)" Value="False"/>
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

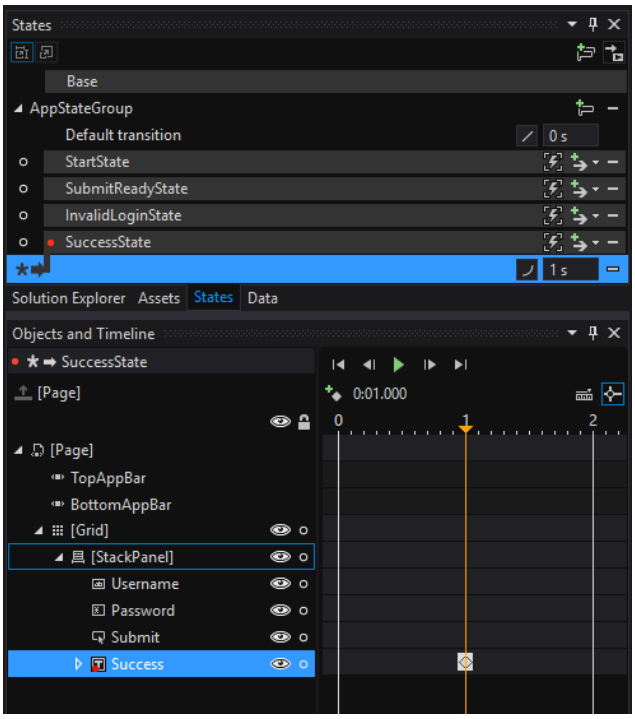
Note that for this to work, the affected controls do need to have their ***x:Name*** properties set, as this is the identifier used by the VisualStateManager to assign the property changes.

## Animations

Blend can also help you animate transitions between states, making your app feel more alive by highlighting and focusing user attention on a specific control or activity. In the State window each state has a button to set its transition:




Notice that you can define a separate to and from each state. Selecting one adds a field to set its duration, and also creates a new Storyboard, visible in the Objects and Timelines window in Blend.

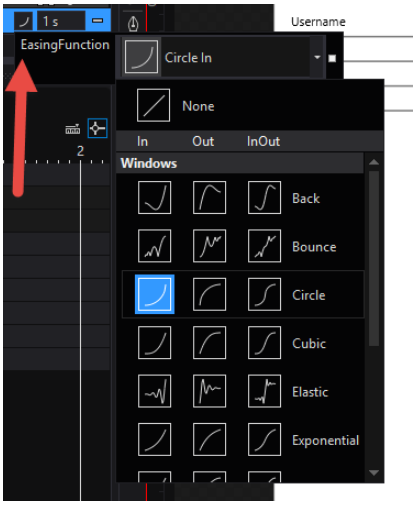



Just as before with static states, the record indicator allows you to set the properties you wish to change, but this time over time. You select the point at which the property should change, then change the property in the designer, which is recorded for playback. You can even preview the animation with the transport controls in the Timeline window.

For our sample, we'll simply animate the Success message by changing it's Opactiy value from 0 to 100 over the period of 1 second, resulting in some new XAML for our page:

```
    <VisualStateGroup.Transitions>
        <VisualTransition GeneratedDuration="0"/>
        <VisualTransition GeneratedDuration="0:0:1" To="SuccessState">
            <VisualTransition.GeneratedEasingFunction>
                <CircleEase EasingMode="EaseIn"/>
            </VisualTransition.GeneratedEasingFunction>
            <Storyboard>
                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="(UIElement.Visibility)" Storyboard.TargetName="Success">
                    <DiscreteObjectKeyFrame KeyTime="0">
                        <DiscreteObjectKeyFrame.Value>
                            <Visibility>Visible</Visibility>
                        </DiscreteObjectKeyFrame.Value>
                    </DiscreteObjectKeyFrame>
                    <DiscreteObjectKeyFrame KeyTime="0:0:1">
                        <DiscreteObjectKeyFrame.Value>
                            <Visibility>Visible</Visibility>
                        </DiscreteObjectKeyFrame.Value>
                    </DiscreteObjectKeyFrame>
                </ObjectAnimationUsingKeyFrames>
                <DoubleAnimationUsingKeyFrames
Storyboard.TargetProperty="(UIElement.Opacity)" Storyboard.TargetName="Success">
                    <EasingDoubleKeyFrame KeyTime="0" Value="0"/>
                    <EasingDoubleKeyFrame KeyTime="0:0:1" Value="1"/>
                </DoubleAnimationUsingKeyFrames>
            </Storyboard>
        </VisualTransition>
    </VisualStateGroup.Transitions>
```

Finally, you can also specify “easing” functions for your transitions, which can help them to appear more natural than a simple linear change.




Now that we have the four visual states for our app defined, let’s look at one simple way to transition between them.

## Changing States with Code

Transitioning states is quite simple; you need only reference the static ***VisualStateManager*** object and call the ***GoToState*** method, passing in the parent control, the name of the state, and a boolean to indicate whether or not to use transitions (which we’ll look at shortly).

We do this in the constructor to go to the default (disabled button) state:

```
public MainPage()
{
    this.InitializeComponent();

    VisualStateManager.GoToState(this, StartState.Name, false);
}
```

Adding additional event handlers to transition between states completes the experience:

```
private void TextChanged(object sender, TextChangedEventArgs e)
{
        if (string.IsNullOrEmpty(Username.Text) ||
string.IsNullOrEmpty(Password.Password)) return;

        VisualStateManager.GoToState(this, SubmitReadyState.Name, false);
}

private void PasswordChanged(object sender, RoutedEventArgs e)
{
        if (string.IsNullOrEmpty(Username.Text) ||
string.IsNullOrEmpty(Password.Password)) return;

        VisualStateManager.GoToState(this, SubmitReadyState.Name, false);
}

private void Submit_Click(object sender, RoutedEventArgs e)
{
        if (Username.Text == "Username")
            VisualStateManager.GoToState(this, InvalidLoginState.Name, false);
        else
            VisualStateManager.GoToState(this, SuccessState.Name, true);
}
```

Note that for the Submit button, when we navigate to the SuccessState, we want to make sure to enable the transitions so that the state is revealed with the animations we expected.

## Wrapping Up and Next Steps

Blend makes it easy to create various visual states for your application, and even lets you quickly add animations and transitions to help your app feel more alive and natural. You can easily change between different states, grouping the properties and behaviors together that define the various states of your application.

# Responsive Design with AdaptiveTriggers

In the previous chapter we looked at the basics of UI automation in Windows 10 apps using Blend to declare the XAML for Visual States. However, while last time we used event handlers in the code-behind of a page to trigger different states, this time we'll see how to use the new AdaptiveTriggers in Windows 10 to let the application handle state transitions automatically.

## StateTriggers

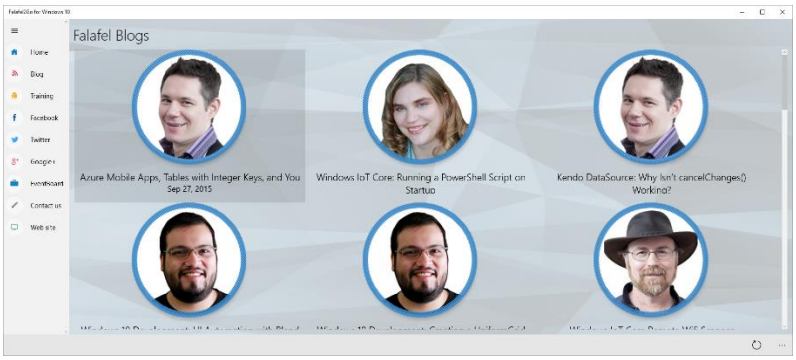
Each VisualState exposes a StateTriggers property, which is a collection of triggers that it uses to determine whether or not the state should be activated. These triggers inherit from the StateTriggerBase class, which uses the SetActive() method to set the active status of the parent VisualState, based on specific properties of the trigger in relation to the running application.

If all triggers resolve their value to true, the parent VisualState is activated.

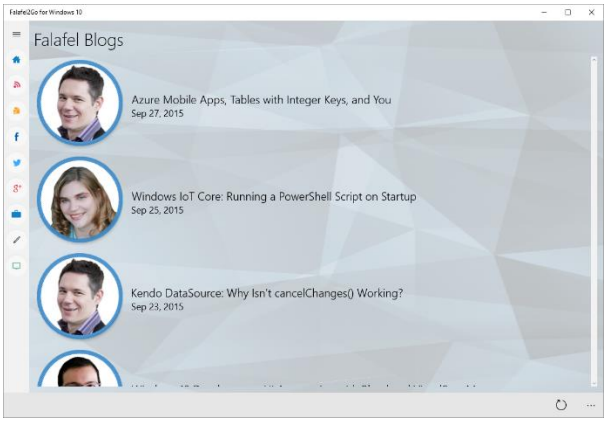
### AdaptiveTrigger

Windows 10 currently offers only a single trigger in the Framework: the AdaptiveTrigger. This trigger exposes two properties (***MinWindowWidth***and ***MinWindowHeight***) that serve as screen size threshold values. When the screen is larger than the specified property (or properties, if they are both specified), the adaptive trigger sets active to true.

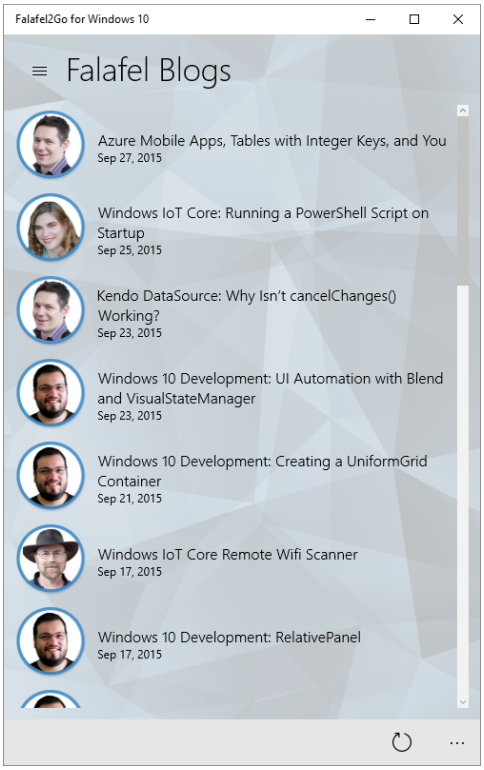
This behavior is demonstrated in Falafel2Go with the SplitView, which adapts across various sizes from fully open on a wide desktop:




a medium, collapsed panel for medium screens:



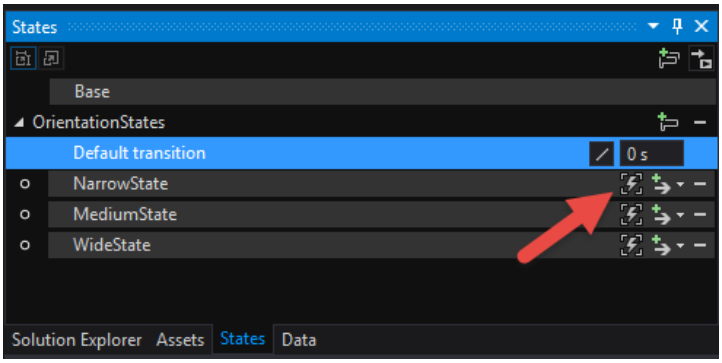

and finally hidden for narrow screens and mobile devices:



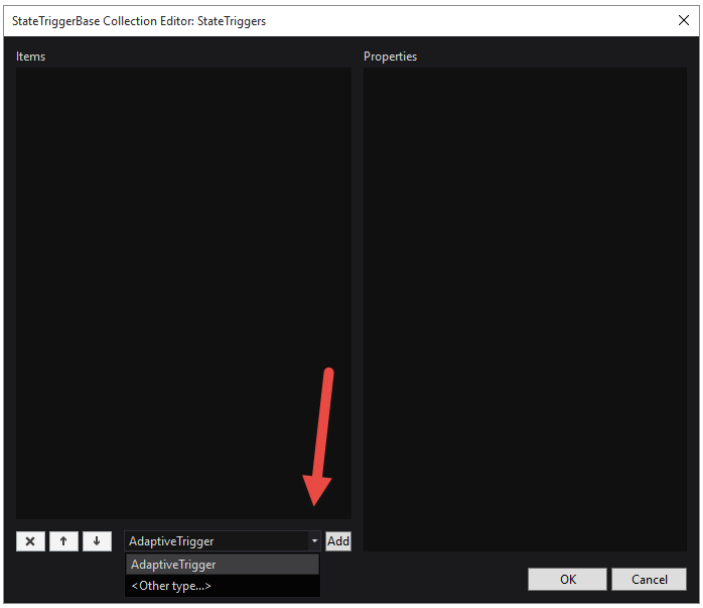

## Defining Triggers with Blend

Once again we can leverage the design features and tools in Blend for Visual Studio to simplify the process of defining triggers by creating them visually in the designer.

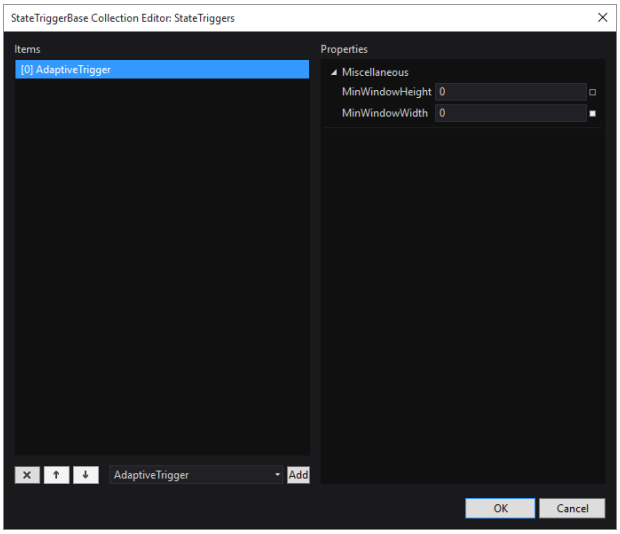
In the States window, to the right of each declared VisualState is a button to launch a dialog to manage the triggers for that state.




In this dialog you can select the native AdaptiveTrigger (as well as a custom trigger, which we'll see in the next chapter) and add it to the collection.




Once the trigger is added you can select it in the list above, which exposes the related properties, allowing you to specify the conditions which enable the trigger:




Repeating this process of adding triggers to each VisualState results in the following XAML for the VisualStateManager, which describes the desired behavior:

```
   <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="OrientationStates">
            <VisualState x:Name="NarrowState">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="0" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="PageSplitView.DisplayMode" Value="Overlay" />
                    <Setter Target="MobileMenuButton.Visibility" Value="Visible" />
                    <Setter Target="BlogsListView.ItemTemplate" Value="{StaticResource
SmallTemplate}" />
                    <Setter Target="BlogsListView.ItemsPanel" Value="{StaticResource
VerticalTemplate}" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="MediumState">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="720" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="PageSplitView.DisplayMode" Value="CompactOverlay" />
                    <Setter Target="MobileMenuButton.Visibility" Value="Collapsed" />
                    <Setter Target="BlogsListView.ItemTemplate" Value="{StaticResource
MediumTemplate}" />
                    <Setter Target="BlogsListView.ItemsPanel" Value="{StaticResource
VerticalTemplate}" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="WideState">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="1280" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="PageSplitView.DisplayMode" Value="Inline" />
                    <Setter Target="PageSplitView.IsPaneOpen" Value="True" />
                    <Setter Target="MobileMenuButton.Visibility" Value="Collapsed" />
                    <Setter Target="BlogsListView.ItemTemplate" Value="{StaticResource
LargeTemplate}" />
                    <Setter Target="BlogsListView.ItemsPanel" Value="{StaticResource
HorizontalTemplate}" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

Notice in this case we're only specifying the width, and that the smallest definition for mobile has a minimum value of zero.

Although all states fulfill this condition, the VisualStateManager is smart enough to prioritize the definitions so that the wider values appropriately resize the control as the thresholds are reached.

It is also worth noting that you are not limited to simple properties when automating UI with VisualStates and Triggers. As you see in the screenshots above, while the medium and mobile screens orient the list of blog posts vertically, the largest state arranges them horizontally to take advantage of the larger space.

This is achieved by simply defining two ItemsPanelTemplates, using the VisualState and Triggers to swap them out on the fly:

```
<Page.Resources>
    <ItemsPanelTemplate x:Key="VerticalTemplate">
        <VirtualizingStackPanel />
    </ItemsPanelTemplate>
    <ItemsPanelTemplate x:Key="HorizontalTemplate">
        <WrapGrid Orientation="Horizontal" />
    </ItemsPanelTemplate>
    <!-- ... -->
</Page.Resources>
```

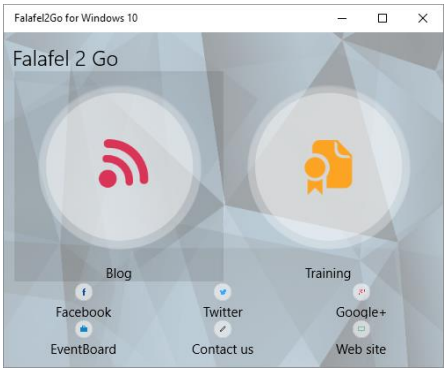
Notice that in this case for the HorizontalTemplate I opted to use a WrapGrid since it will wrap items to the next line while the VirtualizingStackPanel does not.

The same strategy is used with the ListView ItemTemplate, swapping out to a different as needed for the alternative layouts as needed to support the different screen sizes.

## Wrapping Up and Next Steps

We used this strategy to achieve a responsive layout on all the screens, including the main activities page, which updates the RelativePanel attached properties to rearrange the buttons.

However, you may notice that we have a slight discrepancy in our responsive layout. While we have set a fixed width to trigger the different states, we haven't taken into consideration the height. As a result, while the main screen sizes appropriately if the layout is portrait, on a narrow screen we don't quite get the result we expect if the height is smaller than the width (landscape).




While we can certainly add additional AdaptiveTriggers to cover the different orientation options, we can simplify this process by instead creating our own custom trigger to combine these properties and reduce the complexity. We'll take a look at this in the final chapter.

# Creating Custom StateTriggers

We've seen how the Windows 10 AdaptiveTrigger can help you create a dynamic responsive UI based on the height or width of the device or screen. However, this trigger is limited to a specific value for each dimension, and offers no way to respond to changes in the relationship between them, such as to determine the orientation or aspect ratio of the app.

Fortunately, you are not limited to only the AdaptiveTrigger in Windows 10; you can create your own custom statetriggers based on virtually any property, even those not at all related to the UI. In our last chapter, we'll look at how we combined both the width and orientation properties of the app to create a custom OrientationSize trigger to switch between visual states using these properties.

## StateTriggerBase

Custom triggers in Windows 10 inherit from StateTriggerBase which exposes the key method SetActive(). This method accepts a boolean parameter indicating whether or not the associated VisualState should be activated.

The custom trigger simply needs to call this method, passing in whatever value is appropriate based on any condition or custom code to which the app has access, including window size, network connectivity, time of day, etc.

In our case, we are concerned with both the Orientation and the Window size, the cross product of which results in six possible options. I've encapsulated these values into a custom enumeration:

```
public enum OrientationSize
{
    NarrowPortrait,
    NarrowLandsape,
    Portrait,
    Landscape,
    WidePortrait,
    WideLandscape
}
```

The goal is to create a trigger that can identify if it is in one of these states so that the layout can adjust accordingly.

We begin then by inheriting from the required base class, with a property that represents the desired orientation enumeration value for the trigger:

```
public class OrientationSizeTrigger : StateTriggerBase
{
       public OrientationSize Orientation { get; set; }
}
```

Now, the trigger we create obviously needs to be able to respond to changes in the application size. This can be done by listening to the SizeChanged event of the current window, wired up in the constructor:

```
public OrientationSizeTrigger()
{
    Window.Current.SizeChanged += Current_SizeChanged;
}

private void Current_SizeChanged(object sender,
Windows.UI.Core.WindowSizeChangedEventArgs e)
{
    SetTrigger();
}
```

Any change in the window size or orientation will launch a custom method which I've called ***SetTrigger*** to compare the current state of the window size to the state required by the defined trigger instance properties:

```
private void SetTrigger()
{
    var currentView = ApplicationView.GetForCurrentView();
    bool active = false;
    switch (Orientation)
    {
        case OrientationSize.NarrowPortrait:
            active = currentView.Orientation == ApplicationViewOrientation.Portrait &&
currentView.VisibleBounds.Width <= 720;
            break;

        case OrientationSize.NarrowLandsape:
            active = currentView.Orientation == ApplicationViewOrientation.Landscape &&
currentView.VisibleBounds.Width <= 720;
            break;

        case OrientationSize.Portrait:
            active = currentView.Orientation == ApplicationViewOrientation.Portrait &&
currentView.VisibleBounds.Width > 720 && currentView.VisibleBounds.Width <= 1280;
            break;

        case OrientationSize.Landscape:
            active = currentView.Orientation == ApplicationViewOrientation.Landscape &&
currentView.VisibleBounds.Width > 720 && currentView.VisibleBounds.Width <= 1280;
            break;

        case OrientationSize.WidePortrait:
            active = currentView.Orientation == ApplicationViewOrientation.Portrait &&
currentView.VisibleBounds.Width > 1280;
            break;

        case OrientationSize.WideLandscape:
            active = currentView.Orientation == ApplicationViewOrientation.Landscape &&
currentView.VisibleBounds.Width > 1280;
            break;
    }

    this.SetActive(active);
}
```

Here I've used the three threshold values of 0, 720, and 1280 to define the boundaries of the different sizes, combined with the existing Orientation property of the ***ApplicationView*** state to determine the overall state of the application. If it is a match for the trigger's expected property, we set the associated VisualState to be active.

Although we have defined six possible states, as we saw in the previous chapter, we only need three for the different states of Falafel2Go for small, medium, and wide views. However, as we also saw the views aren't defined strictly by the width or height; the screen could be small but also landscape, in which case we do not want to use a vertical layout.

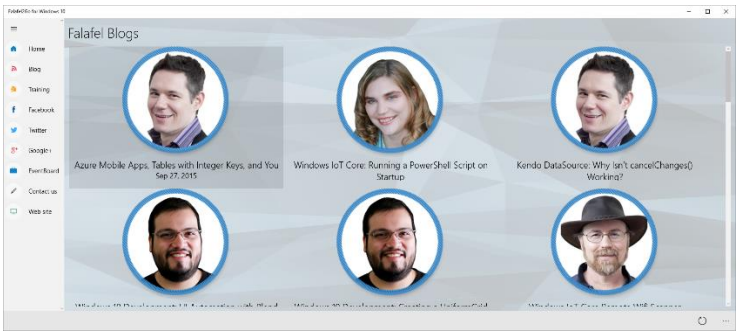
Instead, we can simply combine the states, mixing and matching their arrangements to properly enable the states that layout the app properly given the resulting state.

Here's the XAML that does exactly that:

```
<VisualStateManager.VisualStateGroups>
<VisualStateGroup x:Name="OrientationStates">
    <VisualState x:Name="NarrowState">
        <VisualState.StateTriggers>
            <triggers:OrientationSizeTrigger Orientation="NarrowPortrait"/>
            <triggers:OrientationSizeTrigger Orientation="WidePortrait"/>
        </VisualState.StateTriggers>
        <VisualState.Setters>
            <Setter Target="MainPanel.(RelativePanel.AlignRightWithPanel)" Value="True"
/>
            <Setter Target="MainPanel.ItemsPanel">
                <Setter.Value>
                    <ItemsPanelTemplate>
                        <ctrl:UniformGrid Columns="2" Orientation="Horizontal" />
                    </ItemsPanelTemplate>
                </Setter.Value>
            </Setter>
            <Setter Target="OtherActivitiesPanel.(RelativePanel.Below)" Value="MainPanel"
/>
            <Setter Target="OtherActivitiesPanel.(RelativePanel.RightOf)" Value="" />
        </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="LandscapeState">
        <VisualState.StateTriggers>
            <triggers:OrientationSizeTrigger Orientation="NarrowLandsape"/>
            <triggers:OrientationSizeTrigger Orientation="Landscape"/>
        </VisualState.StateTriggers>
        <VisualState.Setters>
            <Setter Target="MainPanel.(RelativePanel.AlignRightWithPanel)" Value="False"
/>
            <Setter Target="MainPanel.ItemsPanel">
                <Setter.Value>
                    <ItemsPanelTemplate>
                        <ctrl:UniformGrid Rows="2" Orientation="Vertical" />
                    </ItemsPanelTemplate>
                </Setter.Value>
            </Setter>
            <Setter Target="OtherActivitiesPanel.ItemsPanel">
                <Setter.Value>
                    <ItemsPanelTemplate>
                        <ctrl:UniformGrid Rows="3" Orientation="Vertical"/>
                    </ItemsPanelTemplate>
                </Setter.Value>
            </Setter>
            <Setter Target="OtherActivitiesPanel.(RelativePanel.AlignRightWithPanel)"
Value="True" />
            <Setter Target="OtherActivitiesPanel.(RelativePanel.Below)" Value="" />
            <Setter Target="OtherActivitiesPanel.(RelativePanel.RightOf)"
Value="MainPanel" />
        </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="WideState">
        <VisualState.StateTriggers>
            <triggers:OrientationSizeTrigger Orientation="WideLandscape" />
        </VisualState.StateTriggers>
        <VisualState.Setters>
            <Setter Target="MainPanel.(RelativePanel.AlignRightWithPanel)" Value="False"
/>
            <Setter Target="MainPanel.ItemsPanel">
                <Setter.Value>
                    <ItemsPanelTemplate>
                        <ctrl:UniformGrid Rows="2" Orientation="Vertical" />
                    </ItemsPanelTemplate>
```

```
                </Setter.Value>
            </Setter>
            <Setter Target="OtherActivitiesPanel.ItemsPanel">
                <Setter.Value>
                    <ItemsPanelTemplate>
                        <ctrl:UniformGrid Rows="2" Orientation="Vertical" />
                    </ItemsPanelTemplate>
                </Setter.Value>
            </Setter>
            <Setter Target="OtherActivitiesPanel.(RelativePanel.AlignRightWithPanel)"
Value="True" />
            <Setter Target="OtherActivitiesPanel.(RelativePanel.Below)" Value="" />
            <Setter Target="OtherActivitiesPanel.(RelativePanel.RightOf)"
Value="MainPanel" />
        </VisualState.Setters>
    </VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Notice here that we combine the the layouts that make sense (such as Landscape and NarrowLandscape) while separating out the unique WideLandscape to accommodate the unique, full-screen desktop views, like the tiled view for blogs:

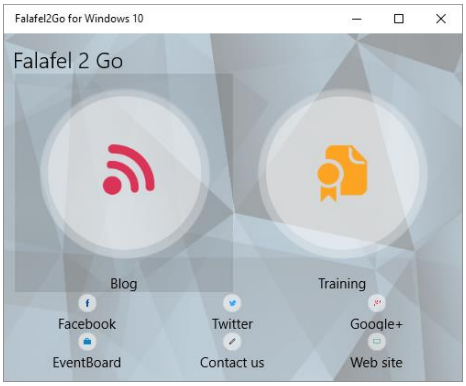



You might be wondering why there isn't an entry for the regular Portrait enumeration; the reason this is missing is that this is the "default" state of the XAML page, so if none of the triggers are satisfied, it will fall back to that state, completing all of the possible required views for the app.

## Firing Triggers on Startup with Dependency Properties

There's still one last missing piece of the puzzle here, which you will discover if you exit the app while not in the default ("Portrait") state. Exiting a Windows 10 app preserves the window size and orientation for the next launch. As you recall, the default state for our app is Portrait, and we only update our custom triggers when the window is resized.

This means that if we exit our app in a different VisualState, then relaunch it, it will attempt to restore the default "Portrait" state, even though that state is not correct for the launched orientation, resulting in an unexpected layout:




You might think, as I sure did, that the fix is to simply force the trigger to evaluate in the constructor:

```
public OrientationSizeTrigger()
{
        Window.Current.SizeChanged += Current_SizeChanged;
        SetTrigger();
}
```

Unfortunately, this does not appear to be the correct fix, as in my testing, the ***Orientation*** property appeared to be fixed to the default, first value of the enumeration (in this case “NarrowPortrait”) for all of the triggers, disregarding what we set in the XAML properties.

The reason for this is that we need to instead configure our trigger to rely on Dependency Properties which, as previously discussed, contain all the plumbing and infrastructure to manage binding at runtime.

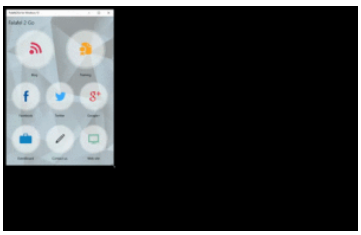
Replacing our OrientationSize with an implentation using Dependecy Properties, and using the changed event to call the trigger yields the following complete definition for our custom trigger:

```
public class OrientationSizeTrigger : StateTriggerBase
{
    public OrientationSizeTrigger()
    {
        Window.Current.SizeChanged += Current_SizeChanged;
    }

    private void Current_SizeChanged(object sender,
Windows.UI.Core.WindowSizeChangedEventArgs e)
    {
        var result = SetTrigger(Orientation);
        SetActive(result);
    }

    private static bool SetTrigger(OrientationSize orientation)
    {
        if (GalaSoft.MvvmLight.ViewModelBase.IsInDesignModeStatic) return false;

        var currentView = ApplicationView.GetForCurrentView();
        bool active = false;
        switch (orientation)
        {
            case OrientationSize.NarrowPortrait:
                active = currentView.Orientation == ApplicationViewOrientation.Portrait
&& currentView.VisibleBounds.Width <= 720;
                break;

            case OrientationSize.NarrowLandsape:
                active = currentView.Orientation == ApplicationViewOrientation.Landscape
&& currentView.VisibleBounds.Width <= 720;
                break;

            case OrientationSize.Portrait:
                active = currentView.Orientation == ApplicationViewOrientation.Portrait
&& currentView.VisibleBounds.Width > 720 && currentView.VisibleBounds.Width <= 1280;
                break;

            case OrientationSize.Landscape:
                active = currentView.Orientation == ApplicationViewOrientation.Landscape
&& currentView.VisibleBounds.Width > 720 && currentView.VisibleBounds.Width <= 1280;
                break;

            case OrientationSize.WidePortrait:
                active = currentView.Orientation == ApplicationViewOrientation.Portrait
&& currentView.VisibleBounds.Width > 1280;
                break;

            case OrientationSize.WideLandscape:
                active = currentView.Orientation == ApplicationViewOrientation.Landscape
&& currentView.VisibleBounds.Width > 1280;
                break;
        }

        return active;
    }

    public OrientationSize Orientation
    {
        get { return (OrientationSize)GetValue(OrientationSizeProperty); }
        set { SetValue(OrientationSizeProperty, value); }
    }
```

```
    public DependencyProperty OrientationSizeProperty =
DependencyProperty.Register("OrientationSize", typeof(OrientationSize),
typeof(OrientationSizeTrigger), new PropertyMetadata(OrientationSize.NarrowPortrait,
OnOrientationSizeChanged));

    private static void OnOrientationSizeChanged(DependencyObject sender,
DependencyPropertyChangedEventArgs args)
    {
        var trigger = (OrientationSizeTrigger)sender;
        var newVal = (OrientationSize)args.NewValue;
        var result = SetTrigger(newVal);
        trigger.SetActive(result);
    }
}
```

Note that I've pre-fixed the ***SetTrigger*** method with the design detection feature of MvvmLight to avoid throwing any errors in the designer.

With that, we've completed the custom trigger that allows the fully dynamic, responsive layout of Falafel2Go (click to open and animate the GIF):



We applied the same triggers in different combinations throughout Falafel2Go to achieve the different layouts required, automatically triggered by any change in device or screen orientation.

## Wrapping Up

Custom triggers allow your Windows 10 applications to respond to a wide variety of conditions to create dynamic, responsive layouts. Whether it's simply reflowing the UI to accommodate different sizes and orientations, or responding to changes in network connectivity, these triggers help your apps feel more alive, and more importantly, in-tune to your user's expectations and experiences.

With this chapter we've completed a trip through the basics of app development, from setting up an MVVM framework, to managing state and lifecycle, to the different UI controls and strategies for adapting your layout, and is everything you need to build a simple app for Windows 10.

In a future publication, we'll take a look at more advanced topics, really diving into the platform-specific features of Windows 10, including Live Tiles, background tasks, local storage, and even cloud services like push notifications and mobile apps.

If there's any topic you'd like to see be sure to get in touch, and of course, we at Falafel would be glad to work with you on your next Windows 10 project! Take a look at our consulting and training packages today and let us help you take your Windows apps to the next level!


falafel.com
FALAFEL
software