

# Windows 10 Development for Absolute Beginners

*Based on the video series originally recorded by*

Bob Tabor, <http://www.LearnVisualStudio.NET>

Edited and revised by Bob Tabor and Steven Nikolic

## About

This PDF contains what I call a “transliteration” of the original video series. After recording each video, I had them transcribed and edited for closed captioning, then took those closed captioning files, stripped out the special formatting, and used them as the basis for the text in this document. I added screenshots, re-wrote copious amounts of text to make it read less like a transcript and more like a book. The result is nearly 400 pages that should suffice as a close representation of what is discussed in each video, cleaned up dramatically. In other words, don’t expect to watch the video AND read this document together at the same time. Some topics are discussed out of order, in less / greater length and detail, etc. It’s provided for those who want don’t watch to wade through 17.5 hours of video to get at the material, or who do not speak English as their primary language.

I always underestimate the amount of effort these large projects will require. I usually forget the “double-it-and-add-10” rule when creating estimates. After realizing I was falling way behind schedule and already fatigued physically and mentally, I called in a friend for help. Fortunately, Steven Nikolic, a customer-turned-ally on my website stepped up and took on the editorial tasks for a large number of lessons to help me complete this as quickly as possible. Now we both need to schedule a trip to the chiropractor! If you are enjoying this PDF, please take a moment and thank him via his Twitter handle (@steven\_nikolic).

Please note that this was prepared hastily and I’m confident it has grammatical mistakes and perhaps, occasionally, technical mistakes. If you are easily offended by mistakes, I’d recommend you purchase a book by a major technical publisher where they utilize the skills of several editors to verify the correctness – grammatically, technically, etc. – of the material. As the old saying goes, “you get what you pay for”. If you find serious mistakes that need to be corrected, please be kind and point that out to me via my email address: [bob@learnvisualstudio.net](mailto:bob@learnvisualstudio.net). I’m offering that email address only for mistakes – I don’t give technical support. Please don’t ask me to help you with your project. I’m sorry ... too many requests, too little time.

I sincerely hope this work helps you realize your goals.

Best wishes and warm regards,



Bob Tabor

October, 2015

## Table of Contents

About.....	2
Table of Contents.....	3
UWP-001 - Series Introduction .....	6
UWP-002 - Creating your First Universal Windows Platform App.....	10
UWP-003 - Overview of Topics Related to Universal Windows Platform App Development .....	21
UWP-004 - What Is XAML? .....	26
UWP-005 - Understanding Type Converters.....	31
UWP-006 - Understanding Default Properties, Complex Properties and the Property Element Syntax ...	33
UWP-007 - Understanding XAML Schemas and Namespace Declarations.....	41
UWP-008 - XAML Layout with Grids .....	43
UWP-009 - XAML Layout with StackPanel .....	51
UWP-010 - Cheat Sheet Review: XAML and Layout Controls .....	59
UWP-011 - Laudable Layout Challenge.....	62
UWP-012 - Laudable Layout Challenge: Solution .....	64
UWP-013 - Legendary Layout Challenge.....	67
UWP-014 - Legendary Layout Challenge: Solution .....	69
UWP-015 - Laborious Layout Challenge.....	71
UWP-016 - Laborious Layout Challenge: Solution .....	73
UWP-017 - XAML Layout with RelativePanel.....	76
UWP-018 - XAML Layout with the SplitView .....	85
UWP-019 - Working with Navigation.....	90
UWP-020 - Common XAML Controls - Part 1.....	99
UWP-021 - Implementing a Simple Hamburger Navigation Menu.....	107
UWP-022 - Cheat Sheet Review: Windows 10 Layout Hamburger Navigation and Controls.....	113
UWP-023 - Hamburger Heaven Challenge.....	117
UWP-024 - Hamburger Heaven Challenge: Solution .....	121
UWP-025 - Common XAML Controls - Part 2.....	128
UWP-026 - Working with the ScrollViewer.....	138
UWP-027 - Canvas and Shapes .....	141

UWP-028 - XAML Styles .....	146
UWP-029 - XAML Themes.....	152
UWP-030 - Cheat Sheet Review: Controls, ScrollViewer, Canvas, Shapes, Styles, Themes .....	165
UWP-031 - Stupendous Styles Challenge.....	170
UWP-032 - Stupendous Styles Challenge Solution - Part 1: MainPage.....	175
UWP-033 - Stupendous Styles Challenge Solution - Part 2: Navigation and DonutPage .....	178
UWP-034 - Stupendous Styles Challenge Solution - Part 3: CoffeePage .....	182
UWP-035 - Stupendous Styles Challenge Solution - Part 4: SchedulePage .....	186
UWP-036 - Stupendous Styles Challenge Solution - Part 5: CompletePage .....	187
UWP-037 - Utilizing the VisualStateManager to Create Adaptive Triggers.....	189
UWP-038 - Working with Adaptive Layout.....	196
UWP-039 - Adaptive Layout with Device Specific Views .....	201
UWP-040 - Data Binding to the GridView and ListView Controls .....	203
UWP-041 - Keeping Data Controls Updated with ObservableCollection .....	210
UWP-042 - Utilizing User Controls as Data Templates .....	213
UWP-043 - Cheat Sheet Review: Adaptive Layout, Data Binding .....	218
UWP-044 - Adeptly Adaptive Challenge .....	222
UWP-045 - Adeptly Adaptive Challenge Solution - Part 1: Setup and MainPage Layout .....	226
UWP-046 - Adeptly Adaptive Challenge Solution - Part 2: Creating the Data Model and Data Binding ..	230
UWP-047 - Adeptly Adaptive Challenge Solution - Part 3: Creating a User Control as the Data Template .....	235
UWP-048 - Adeptly Adaptive Challenge Solution - Part 4: Adaptively Resizing .....	237
UWP-049 - UWP SoundBoard – Introduction .....	239
UWP-050 - UWP SoundBoard - Setup and MainPage Layout.....	240
UWP-051 - UWP SoundBoard - Creating the Data Model & Data Binding.....	244
UWP-052 - UWP SoundBoard - Playing Sounds with the Media Element.....	250
UWP-053 - UWP SoundBoard - Adding Drag and Drop .....	253
UWP-054 - UWP SoundBoard - Finishing Touches .....	256
UWP-055 - UWP SoundBoard - Add Assets with Package.AppXManifest .....	258
UWP-056 - UWP SoundBoard - Submitting to the Windows Store .....	262
UWP-057 - UWP Weather - Introduction .....	269
UWP-058 - UWP Weather - Setup and Working with the Weather API.....	271
UWP-059 - UWP Weather - Accessing the GPS Location.....	280

UWP-060 - UWP Weather - Testing Location in the Phone Emulator .....	283
UWP-061 - UWP Weather - Updating the Tile with Periodic Notifications .....	286
UWP-062 - UWP Weather - Finishing Touches .....	299
UWP-063 - Album Cover Match Game – Introduction .....	303
UWP-064 - Album Match Game – Setup, Working with Files & Folders .....	304
UWP-065 - Album Match Game - Layout, Data Binding & Game Setup.....	309
UWP-066 - Album Cover Match Game - Employing Game Logic.....	313
UWP-067 - Album Match Game - User Input & Tracking Progress.....	318
UWP-068 - Album Cover Match Game - Enabling the Play Again Feature .....	322
UWP-069 - Album Cover Match Game - Monetizing with Ads .....	323
UWP-070 - Album Cover Match Game - In App Purchase for Ad Removal .....	333
UWP-071 - Hero Explorer - Introduction .....	339
UWP-072 - Hero Explorer - Accessing the Marvel Web API .....	340
UWP-073 - Hero Explorer - Creating an MD5 Hash and Calling the API .....	348
UWP-074 - Hero Explorer - DataBinding & Navigating the Object Graph .....	353
UWP-075 - Hero Explorer – Displaying Character Details.....	360
UWP-076 - Hero Explorer - Displaying Comic Books for a Character .....	365
UWP-077 - Hero Explorer – Displaying Comic Book Details .....	374
UWP-078 - Hero Explorer – Displaying Comic Book Details .....	380
UWP-079 – Hero Explorer – Cortana Integration .....	384
UWP-080 - Wrap Up .....	394
Closing Thoughts and Acknowledgements .....	396

## UWP-001 - Series Introduction

In this series of 80 lesson I'll demonstrate how to build apps that can be sold or downloaded for free on the Windows Store for Windows desktop, or phone, or wherever Universal Windows Platform apps can be used. Which will soon include devices like the Xbox One, Microsoft Hub, and even the HoloLens. Now that term "Universal Windows Platform" I just used it describes tools and APIs that you can utilize to build apps that run universally across all new Windows devices. And the beauty is that you can write one application and it'll look great on many different screen resolutions and device form factors. And really that's one of the most important things that discuss and demonstrate in this series of lessons. So this series is intended for an absolute beginner audience.

Having said that, you should already have some familiarity with C# and Visual Studio. And I want to assume that you're watching this after making sure that you've either watched or at least you understand all the concepts that I discuss in the C# Fundamentals for Absolute Beginners series on Microsoft Virtual Academy and Channel9.

<http://bit.do/csharp-fundamentals>

If you're already an experienced developer then -- fair warning -- quite honestly here, this course is going to move very slow for you. And we did that on purpose. Honestly, there are probably some better resources out there where you can spend your time. I'd recommend that you watch Andy Wigley and Shen Chauhan in a series that they created called A Developers Guide to Windows 10. Great, up to date, it's awesome. And I'd recommend you start there if you're already an experienced developer.

<http://bit.do/developers-guide-to-windows-10>

This is the fourth version of this particular video /PDF series that I've created. And I started way back on the Windows Phone 7 in about 2009 / 2010. Each time that I release a version of this series I'm asked a few questions repeatedly. So I want to answer those right up front so that there are no misunderstandings.

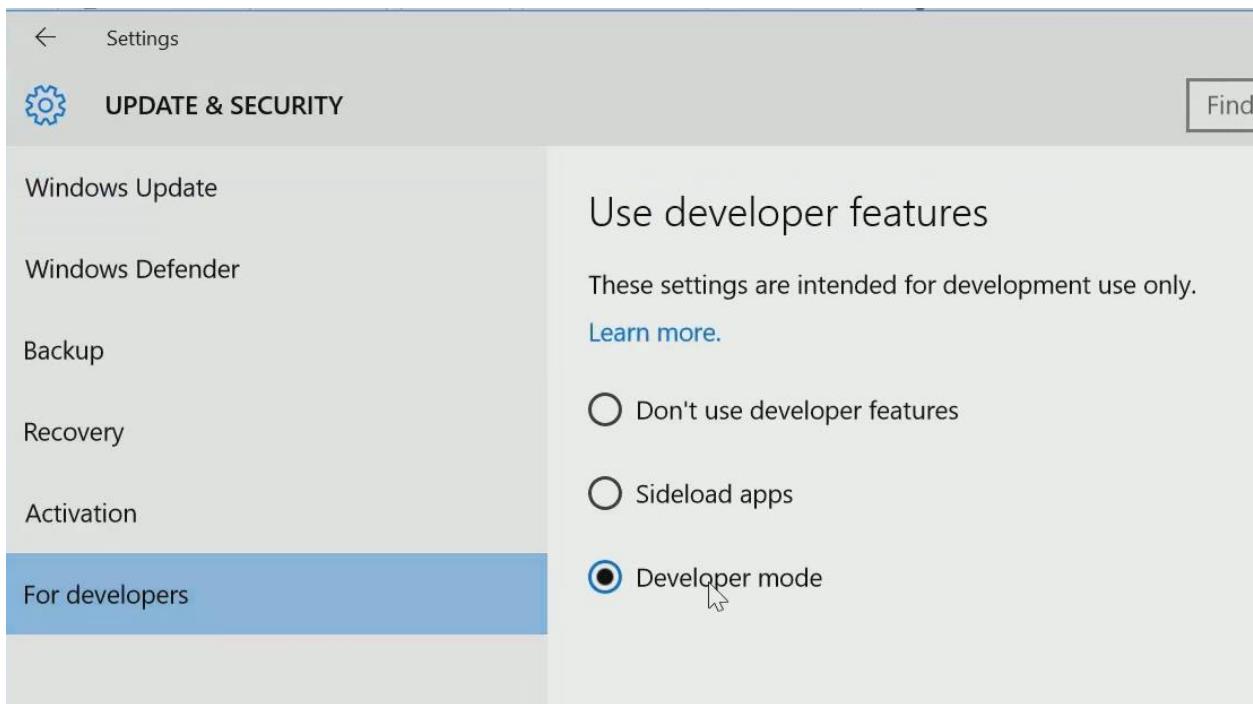
First of all, you must use Windows 10 to build apps for Windows 10, to build Universal Windows Platform apps. You cannot use Windows 8.1, or Windows 8, or Windows 7, or Windows XP. Alright, you have to use Windows 10 (specifically Pro, explained below).

Second, you have to use Visual Studio 2015. Now I recommend that you use Visual Studio 2015 Community Edition, which is a free edition with all the features of Pro edition but it's intended for individuals who are just learning or creating a community based projects. Now frankly you can use other editions of Visual Studio but not previous versions. You must use version 2015 to build Universal Windows Platform apps.

Third, you're going to see me use the phone emulator in order to run and test my applications occasionally. It'll look like a little phone running on my screen with a little menu off to the right hand side. It's actually running software called Hyper-V which is a platform for running virtual machines on your desktop. And so it's running the Windows Phone 10 operating system in a little virtual machine that looks like a phone. You will need Windows 10 Pro and possibly special hardware to run the phone emulator since that is a requirement for Hyper-V. Specifically, your motherboard and your chip must

support a technology called SLAT, or rather "Second Level Address Translation". Now most modern motherboards will support this, however not all motherboards and chips support this. Most importantly, I can't help you with this. I understand very little about it. If you get errors during install of Visual Studio 2015 then you can search Microsoft's forums for help, but again I can't help you. I've tried to help in the past and honestly I've probably confused more people than I've actually helped. Now the worst case scenario if you can't get the emulator running on your local machine, is that you might need to deploy your apps to a physical phone device running either the full edition or a beta edition of Windows Phone 10 for the purpose of testing. It's simple to do, however I don't demonstrate how to do that in this series of videos / PDF. There are articles online That will show you how.

Now if you have all these things in place then, fourth, you're going to need to turn on Developer Mode. And the way that you get that is you open up Settings in Windows 10 and you go to Update & Security. And then on the left hand side you select "For Developers", and make sure to choose "Developer mode". And if it asks you to save then go ahead and save. I'm not sure, I can't remember that particular part. But make sure that you have that set. Visual Studio will probably give you an error if you first run an application if you don't have that setting set up, okay?



This series of 80 videos is close to 17.5 hours, and in this print version around 400 pages. So, it's fairly comprehensive. One of the most important new features of this training series is that give you homework assignments called "challenges". And I'll give you all the tools that you need to build an app to my specifications and then I'm even going to give you the solution to the challenge in case you get stuck. And that's a great way for you, and you should definitely do these challenges where you "get your hands dirty in the code" as I like to say it.

We're also going to build a little cheat sheet for review purposes, and then you can reference it after you finish the series. Feel free to add to the cheat sheet anything that you think might be useful as you go forward and start building your own applications.

And then finally in the last half of the series, we will build four entire applications and even one of them for inclusion in the Windows Store. Now these apps are going to show you how to think like an app developer from concept through implementation. And we'll use a variety of different techniques, and APIs, and tools and approaches to learn how to interact with sensors on a given device. How to access media libraries, how to access online services that provide weather updates. And even allow us to tap into fun services like Marvel comics web API, that allow us to retrieve back all their characters and look through them and look through the artwork.

Many of the lessons in the series will have a zip file associated with them that contains the code that I wrote while recording the video. Or in the case of challenges like I just described a moment ago, it will contain the images that you need, any instructions, or any other files that will be required in order to actually perform that challenge. Now the zipped resource folder will be on the page where you're currently watching the video, or where you originally downloaded the video from. So please before you ask in the comments, "I can't find the download link, where's the download link, the download link!" Please hit Control + f on your keyboard in your web browser and search for the term "download". If there are no link to download a file then that particular lesson does not have files to download. Please search for the term "download" first.

While this is a comprehensive set of lessons and videos this is still really just an introduction. I can't possibly show you everything that the Universal Windows Platform contains. You should treat this as a gentle introduction only, but you should always refer back to Microsoft's own documentation at the Windows Dev Center for comprehensive explanation how to get the most out of Universal Windows Platform. And you can access that at:

<http://dev.windows.com>

If you are going to watch the video version of these lessons keep in mind that you can't just watch a screen cast tutorial training series the way that you would watch a movie or a sitcom on TV. You're going to need to become an active learner. Don't be afraid to rewind or even re-watch an entire video or a portion of the video if at first something doesn't really make a lot of sense to you. Or look at the documentation at the Windows Dev Center at the link I just shared with you. For more detail surrounding the given topic that we're discussing at that moment in the videos.

You learn best whenever you use different modalities to learn the same idea, the same content. And ultimately the videos / PDF that I'm presenting are just one tool to help you realize your aspirations of building apps for sale in the Windows Store.

On a personal note if you like what I do here please visit me at <http://www.LearnVisualStudio.NET> where I help beginners get their first software development job building Windows and web apps into world's best companies. There are tons of challenge exercises there and deeper insights into writing software using Visual Studio, and C#, and ASP.NET and more.

Alright now finally I'd like to take a moment and thank the hundreds of thousands of people who have watched the previous versions of this series. And for those who took the time to actually tell Microsoft

that you wanted more of this type of training. Your feedback made this happen, so thank you very much.

Also I want to thank Andy Wigley (Twitter handle: @andy\_wigley) who patiently answered all of the questions that I had and gave me a ton of advice while I was building this series. I'm very thankful for his guidance as I worked on these lessons. And ultimately this series was championed primarily by Clint Rutkas (Twitter handle: @clintrutkas) who has been involved in almost all of the video projects that I have worked on for Microsoft. Without Clint, none of this would be possible. So please reach out to him on Twitter and let him know how much you appreciate his good work.

Okay so enough setup. You've got Visual Studio 2015 running on Windows 10 and you've turned on Developer mode in Settings like we looked at just a moment ago, and you're wondering what comes next. Well we will get started in the very next lesson.

## UWP-002 - Creating your First Universal Windows Platform App

Near the end of the C# Fundamentals for Absolute Beginners series on Microsoft Virtual Academy I demonstrated how Events work in C#.

And so, to demonstrate that, I created two very different types of applications, but I followed the same workflow and got a similar result. First, I created a ASP.NET Web Forms application, and then second I created a Windows Presentation Foundation (WPF) application. I took the same basic steps to create those two applications, and I got essentially the same result, even though one result was presented in a web browser and the other was presented in a Windows form. I placed a button on the form, the end user clicks the button which was handled by the Click event, and the code programmatically changed the value of Label control to the words "Hello World".

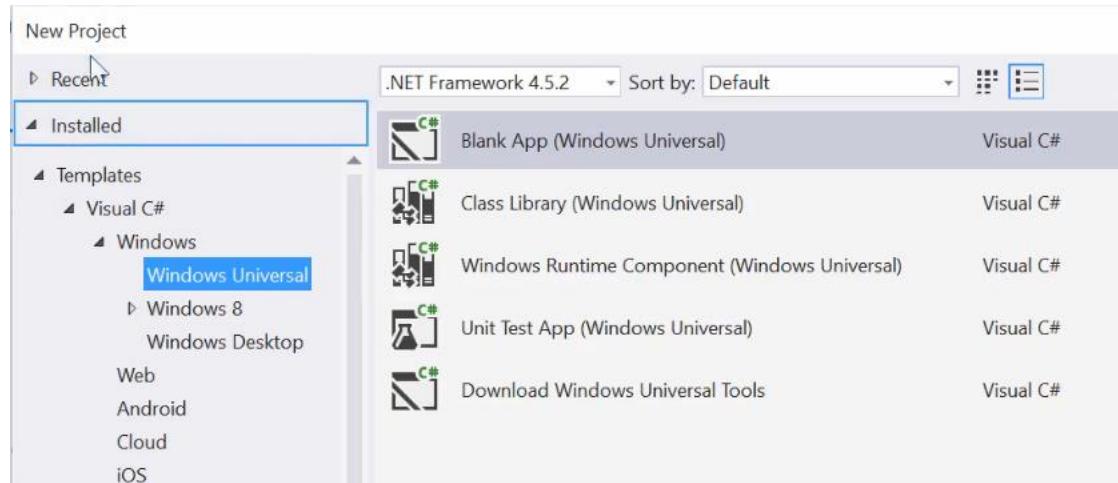
That example illustrated how C# Events work, and also to give us confidence we could leverage the same techniques and workflow to build another type of application.

The good news is that we can re-purpose that knowledge to create Universal Windows Platform apps. In this lesson I want to re-create that same application a basic "Hello World" application, but this time, I'll do it creating a simple, Universal Windows application and I encourage you to follow along.

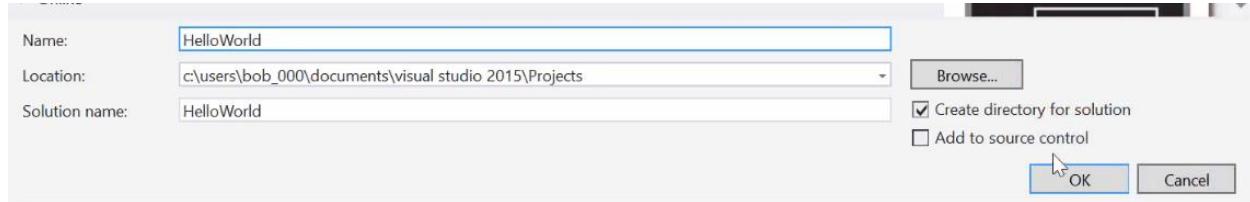
Before getting started please make sure that you already have Visual Studio 2015 installed. Any edition will do: Express, Community, or one of the paid editions. Secondly, you're going to need to make sure that you have the Windows 10 Emulators installed, just like we talked about in the previous lesson.

Assuming that you've got all of that installed and we're ready to go, we will get started by creating a new project. There are many ways in Visual Studio to do this. But create a new project by clicking on the New Project link in the Start page. That will open up the New Project dialog in Visual Studio.

On the left-hand side, select: Installed templates > Visual C# > Windows > Windows Universal, and then in the center a series of templates are displayed. Choose the top option, "Blank App (Windows Universal)" template.

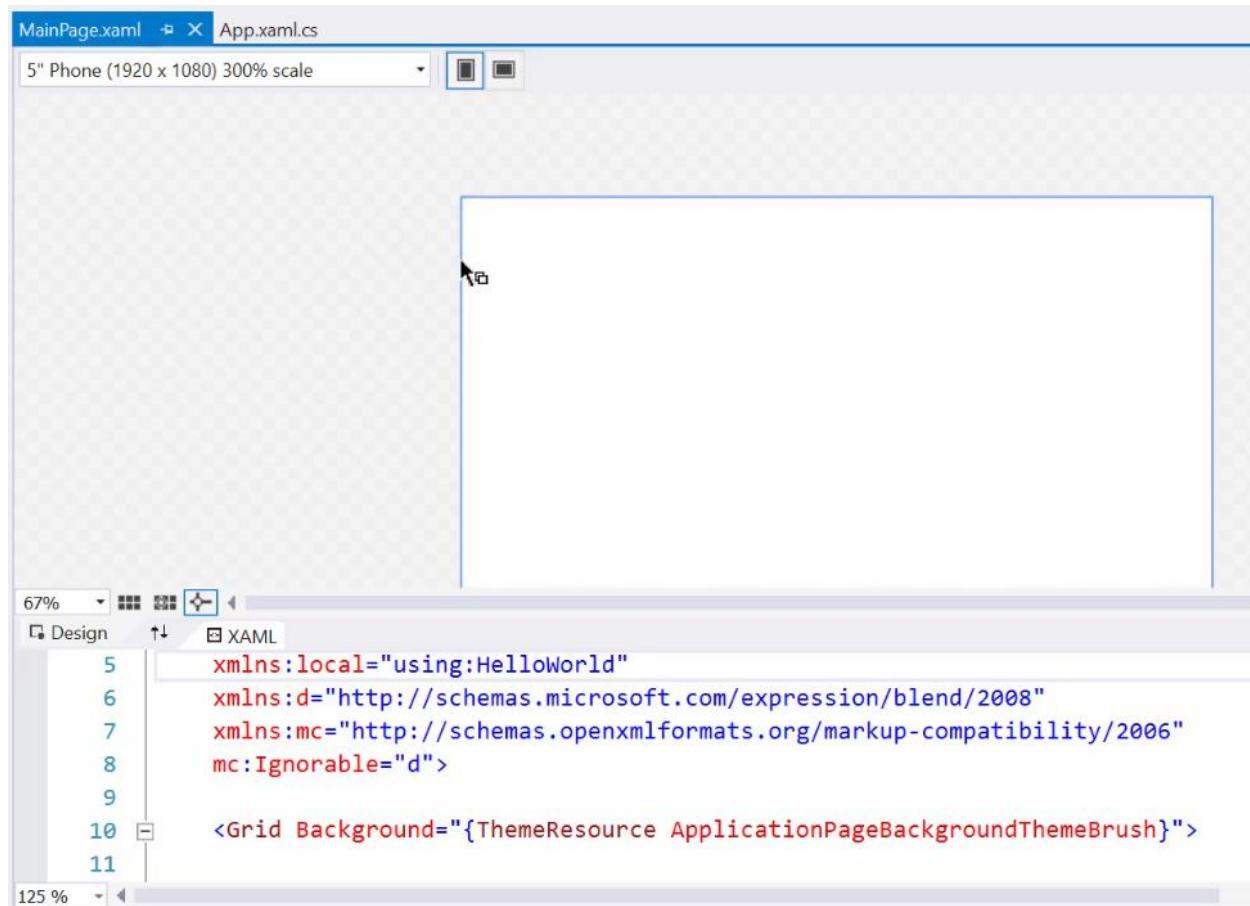


Next, change the project name from whatever the default is to simply "HelloWorld", and click the OK button in the lower right-hand corner.



It may take Visual Studio a few moments to set up the new project.

First, in the Solution Explorer window, double click the MainPage.xaml file. That will open up that file in a special type of designer that has two panes. The top pane is a visual designer, and the lower pane is a textual code-based designer. This designer allows us to edit the code that will be used to generate objects on screen. The lower pane displays the XAML that correlates to the upper pane. So I'll refer to this as the "XAML Editor".



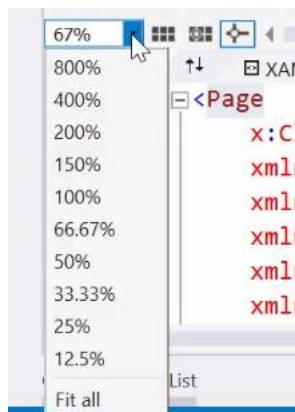
In the top pane, you can see that we get a visual representation of our application. And you can see that we're actually looking at a rendering of how it would look if we were to design our application or run our application on a five-inch phone screen with a resolution of 1920 by 1080.

The visual designer is displaying at a 300% scale. This may be too large for our purposes and we will change that in just a moment.

Notice that we can also view what our application's user interface would look like in portrait or in landscape mode by toggling the little buttons at the top.

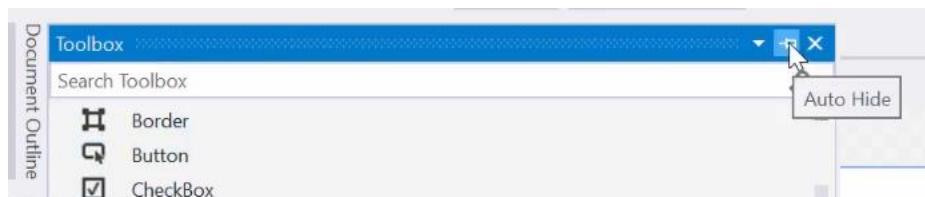


But again, this is a little bit large, we have to scroll around on the screen just to see the entire design surface. So we can make it a little bit smaller by going to the zoom window in the lower left-hand corner. And I'm just going to choose like 33%.

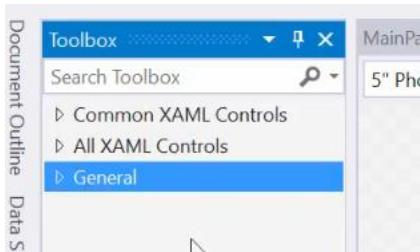


Now that won't completely compact it down for our viewing pleasure. However, it makes it much smaller and manageable.

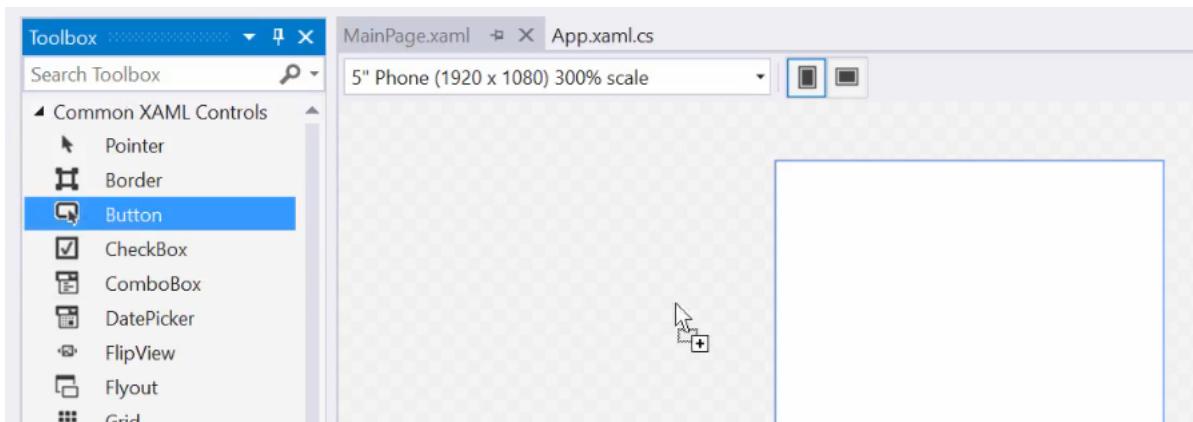
What we will do is start off by adding some controls onto this designer surface, just to break into exactly how this all works. So, over on the left-hand side, there should be a little tab sticking out called Toolbox. And if you click it, the Toolbox window will jettison out from the right-hand side. I'm actually going to use this little pin in the upper right-hand corner of that window. And I click it once, and that will turn off auto-hide. So now that Toolbox window is automatically docked over here on the left-hand side. And if you don't like the way that it's positioned, you can just, again, position it any way you want to using your mouse and the little border area between the Toolbox and the main area.



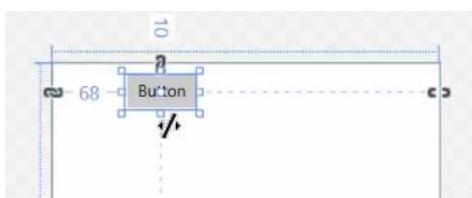
And you'll notice that if we were to roll up here to the very top, that there are a couple of categories of controls that can be added, common XAML controls, or all XAML controls. We'll discuss many of these throughout this series. We basically want to work with just some common ones to start off with.



So drag and drop a Button control from the Toolbox, onto the design surface.



So here we go, dragging and dropping. And notice when I do, it drops it right where my mouse cursor was.



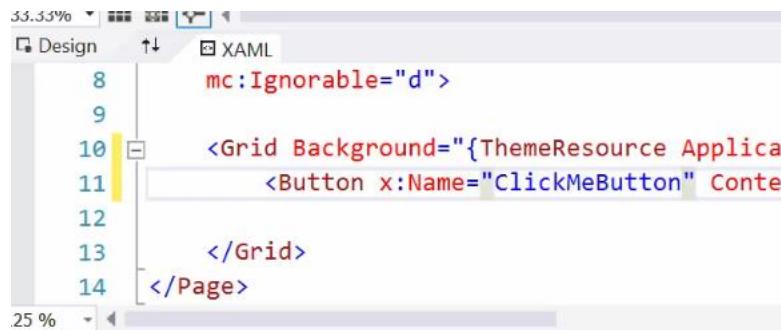
And, hopefully you notice also, here, in this XAML editor that it added this line of code in line number 11. It created a tag, which is an opening and closing angle bracket.



Very similar to what you would see in HTML, and yet this is not HTML. But similar in the sense that you use HTML to lay out the design of a web page. Same is true with XAML. use the XAML programming language to easily lay out the various controls and the layout for our application.

Notice that we created a button control and it has a number of different properties like the name of that control, the content of that control. Also the alignment, the horizontal alignment and the vertical alignment, and then margin. And that margin, as you can see, is based on, here, this little visual view of it. It's 10 pixels from the top and 68 pixels from the left. And so, that's where the alignments come into play, the horizontal alignment left, that's what we're aligning to and then 10 pixels from the top. And then, for the right-hand side and the bottom, well, those are set to zero because we're really not worried about those.

However, just notice that there are attributes or properties of the elements that we add in our XAML editor. So now if I wanted to change the name of this button for programmatic access in C#, I can just call this the ClickMeButton.



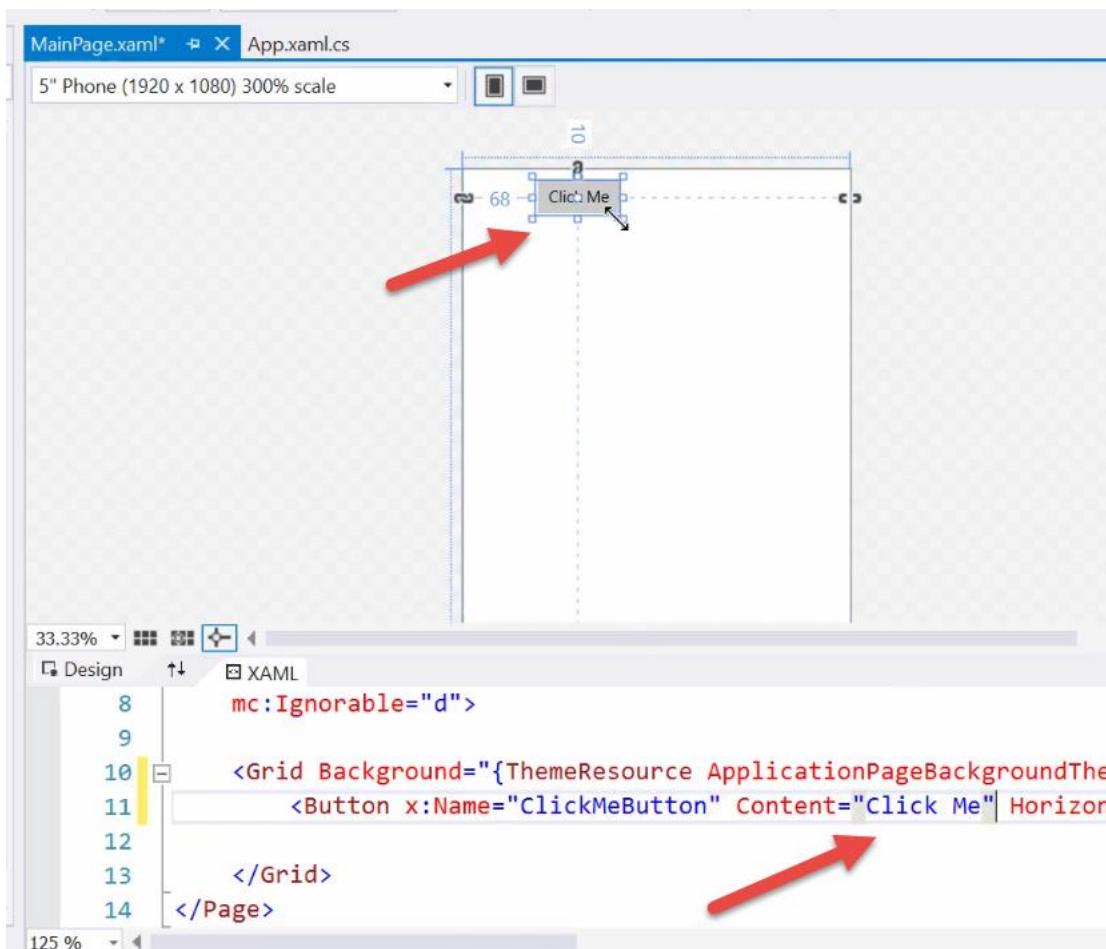
The screenshot shows the Visual Studio interface with the XAML editor open. The title bar says "Design" and "XAML". The XAML code is displayed:

```
53.33%  Design XAML
8
9
10
11
12
13
14
```

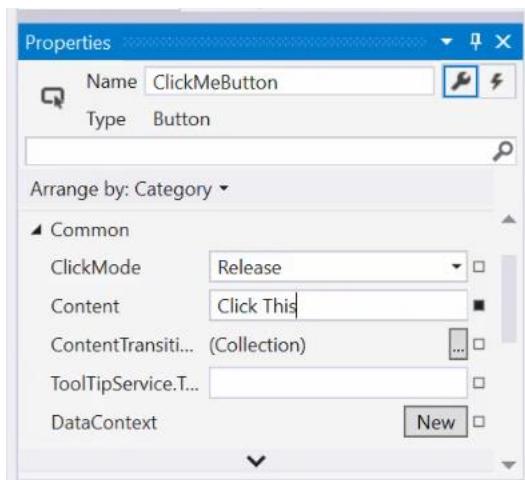
```
mc:Ignorable="d">
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}" HorizontalAlignment="Left" Margin="10,68,0,0" VerticalAlignment="Top">
<Button x:Name="ClickMeButton" Content="Click Me"/>
</Grid>
</Page>
```

The line "Content="Click Me"" is highlighted in yellow, indicating it is selected for editing.

And I can also change the content here as well, by typing in Click Me to the Content area. Notice that when I do that, it changes the visual attribute of the button to "Click Me" instead of whatever was in there before.



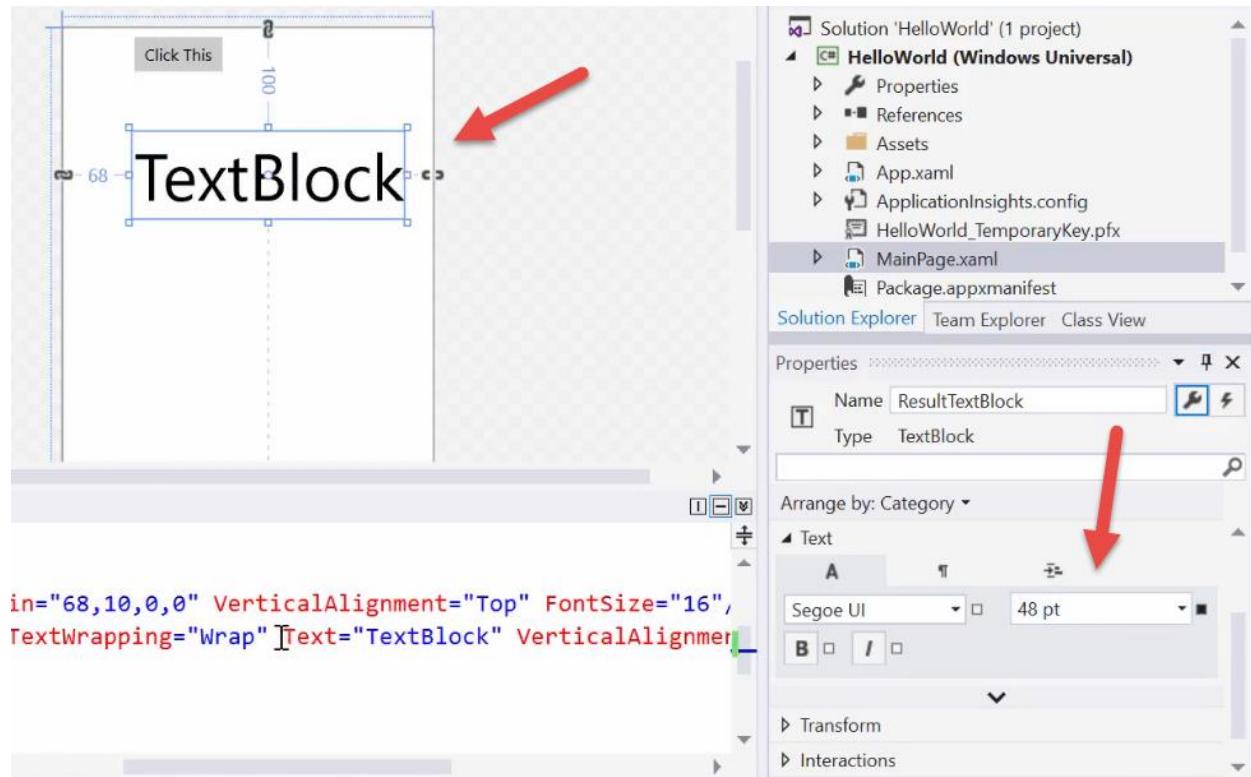
Now there are also a third way that I can make modifications to this object, to this button object. We see it here in the designer. We see it here in the XAML editor. But then also here in the right-hand side of the Properties window, we can make changes by finding a property that modify like, for example, the text property or some common properties like the content. And we can change it by just modifying the little Property editor and hitting enter on our keyboard.



Notice when I did that, it modified not only what we see here in the designer, but then also what we see here in the XAML editor. Change the content property to the value "Click This".

Notice that these are essentially three different views of the same object. One is visual, one is textual, and the other is a Property editor. But we can make changes in any of these and they'll show up in the other one, so they're all connected together.

Next I'll drag and drop a TextBlock control from the Toolbox to the visual Designer surface. The XAML code is also displayed in the XAML editor. I'll change the name to "ResultTextBlock" and change the text property to 48 point.



A TextBlock is a label: somewhere where we can display text.

When reviewing the XAML notice that the font size is set to 64.



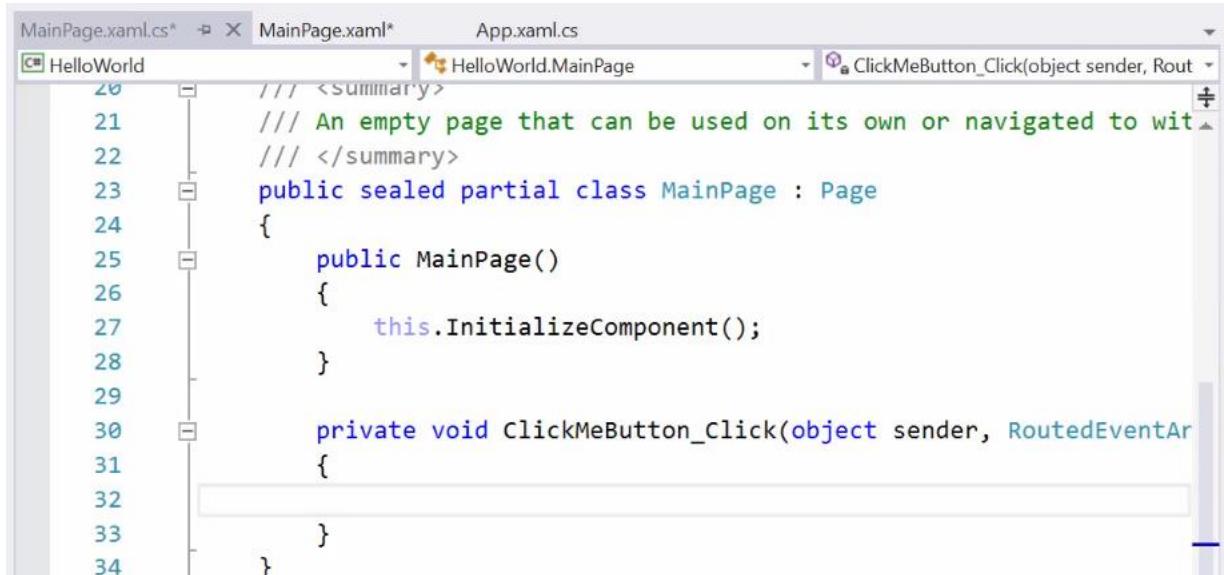
And you might say, "Wait a second here. I see that it's 48 point over here in the Properties editor, but it's 64 units here in the Font Size attribute in the XAML editor. I thought those were supposed to be the same."

Well actually, they are the same. The XAML editor is rendering values in device independent pixels, or rather DIPs. The Properties window is actually representing values in points, which are a fixed size of 1/72nd of an inch. So, it's a very complicated topic, but we will revisit this when we take into consideration the various screen sizes and resolutions on the various form factors that Windows 10 will run on as we're building our applications. As you know you can have a large screen with a low resolution, or vice versa. And so, units of measurement like inches are irrelevant. Instead we need a new unit of measurement that accounts for both screen size and resolution to determine the sizes of things like controls that we add to our apps.

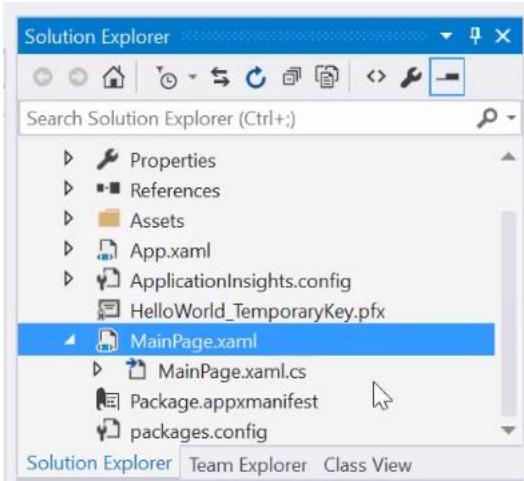
Next I'll select the Button control and then, in the Properties window, I'll select the little "bolt of lightning" button to display events that can be handled for the button.



The very first event is the Click event. I'll double-click inside of that TextBox in the Property editor and open up the MainPage.xaml.cs file.



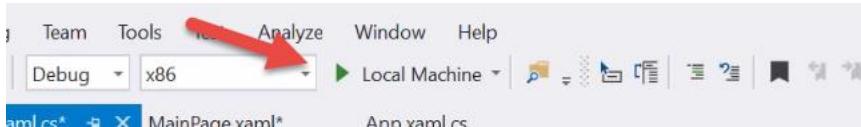
Now notice here in our Solution Explorer, that the MainPage.xaml is related to the MainPage.xaml.cs file. They're actually two parts of the same puzzle. Two pieces of the same puzzle. talk about that more later.



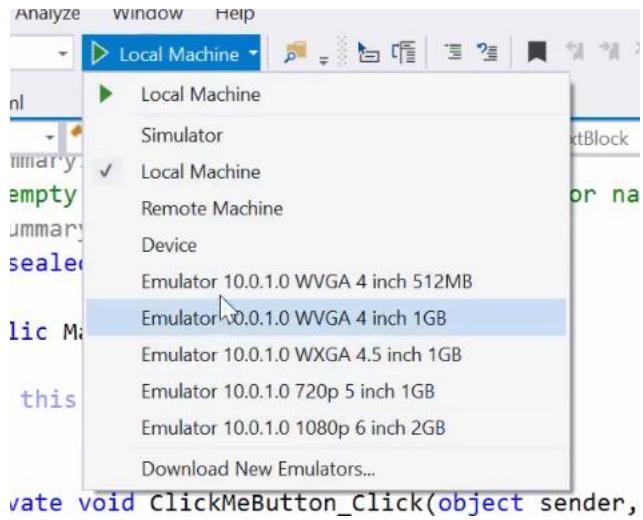
I'll write code that will display "Hello World" in the ResultTextBlock when a user clicks the ClickMeButton.

```
30         private void ClickMeButton_Click(object sender, RoutedEventArgs
31         {
32             ResultTextBlock.Text = "Hello World";
33         }
```

Next I'll test the application by clicking the Run Debug icon in the toolbar at the top.

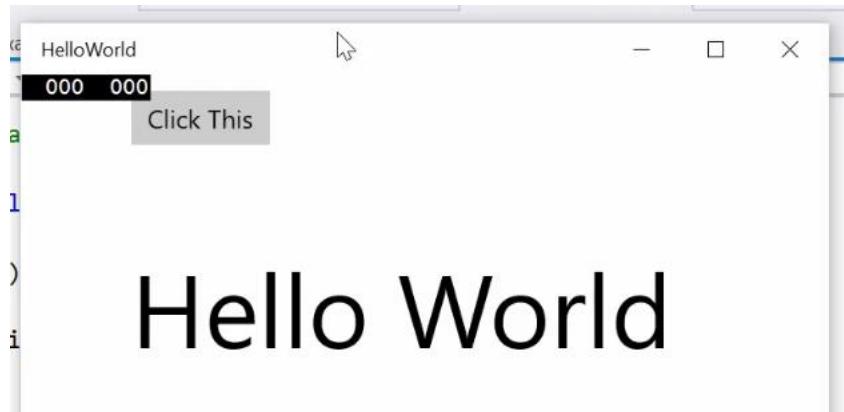


Notice that I'm running it on my local machine, but if I use this little drop-down arrow, you can see that there are some other options as well, including a Simulator, and an Emulator, as well as an actual physical device.



```
    vate void ClickMeButton_Click(object sender,
```

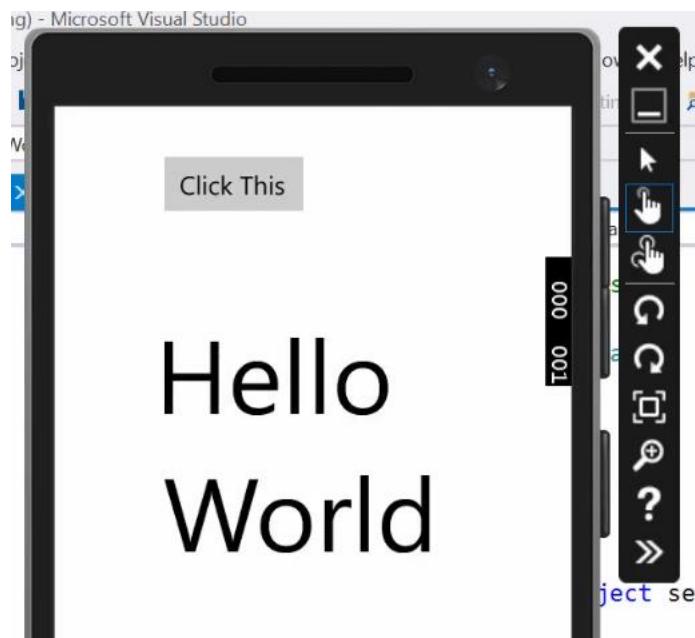
So the first time that we run this, actually just run it on the local machine. And it opens up our application, and it's actually quite large, but resize this down. And if we click the Click This button you can see the text "Hello World" appears, as we would expect.



And so, working in debug mode is great because we're able to set break points like we learned about in the C# Fundamentals for Absolute Beginners series, and inspect values as the application is currently running. We're able to also see our app, our Universal Windows Platform app running without having to deploy our app to an actual, physical Windows device. It's just running on our desktop.

Now for most of this course, be running the app on my local machine, like we've done here. Because really it's the fastest way to test what we've done. However, at times want to test on emulators. And so, an emulator is a virtual machine that emulates, or mimics, a physical device. And that physical device could be a phone or a tablet, or even an Xbox One, although we don't have that emulator just yet to work with.

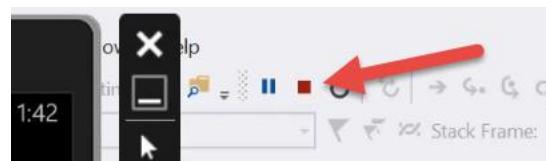
I'll run the application again, but this time choose this Emulator 10.0.1.0 WVGA. That option specifies which physical phone that we will be emulating. So, if we take a look, we can see that our application has been loaded up in this emulator that looks just like a phone.



To stop debugging using the emulator, I could click the close button in the upper right-hand corner of the emulator.

However, you should get in the habit of just leaving the emulator up and running, even when you're not debugging. There are a couple of reasons for this. First of all, the emulator is running as a virtual machine and it takes time to essentially reboot. To shut it down and then to turn it back on. And then, secondly, any app data that you save to the phone between debugging sessions will be lost whenever you shut down the simulator. So if you have data that you want to keep around between debugging sessions, do not close down the emulator.

The other option you have is to actually click the Stop Debugging button in the Toolbar.



Stop Debugging will allow the Emulator to continue running for the next debugging session.

The purpose of this lesson was to explain the relationship between the XAML editor, the design view and the Properties window. Second, the purpose was to demonstrate how C# in the code behind file can handle events and set attributes of those visual objects. Finally, how to debug the application as a desktop application or as a mobile application using the Emulators.

## UWP-003 - Overview of Topics Related to Universal Windows Platform App Development

There were some key takeaways from that previous lesson that are important enough to elaborate on as we're getting started.

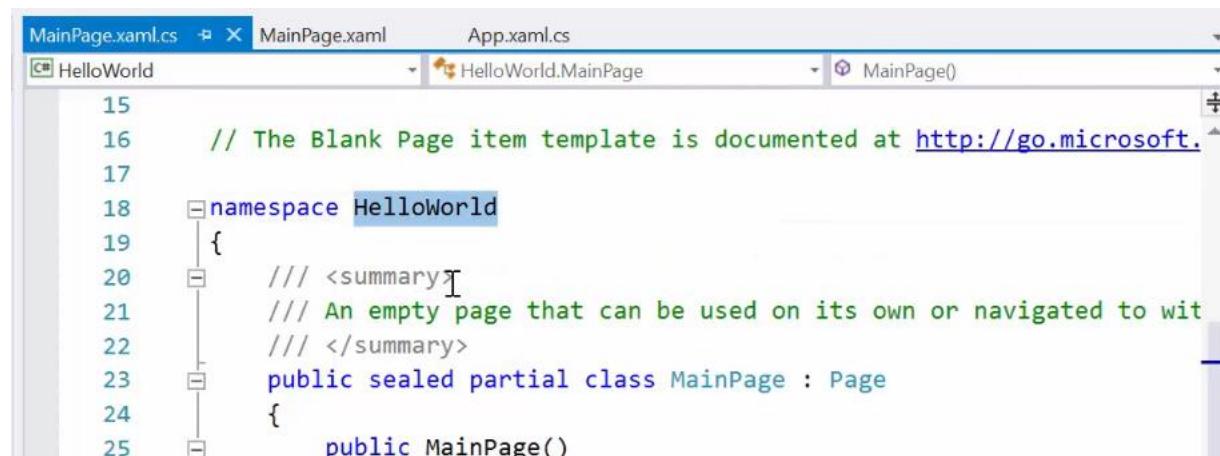
First of all, I hope you realize that you're going to be able to leverage your existing knowledge of creating user interfaces and doing so in Visual Studio, and apply that towards creating Universal Windows Platform apps. Did you notice how similar it was to creating WPF and ASP.NET Web Forms? That was no accident. It's a very similar experience and a very similar workflow. And as a result, it's just as fun, and it's just as easy, and while there is a lot to learn, there's certainly nothing to be intimidated about.

There are several options for designing the user interface of our application. First, by dragging and dropping controls from the Toolbox onto the design surface, resizing those controls and positioning them on the design surface.

The Properties window can be used to modify the various attributes of the controls that we see on the designer surface. However, I think you're going to find that, just like HTML, a visual designer won't give you the level of control that you're really going to want over the various properties and how they relate to each other.

So, there is really no way around learning the actual XAML syntax itself. And we will begin learning the XAML syntax in earnest in the next lesson.

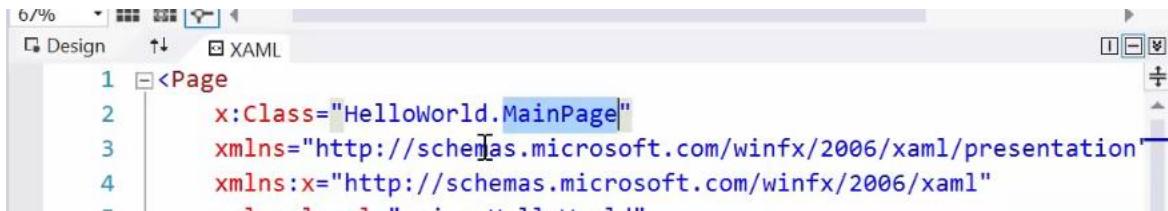
Furthermore, just like ASP.NET Web Forms and WPF applications, every page has a code behind file that's associated with it where we can write event handler code in C#. We see that same relationship between the MainPage.xaml and the MainPage.xaml.cs in Windows 10 apps. In fact, in the MainPage.xaml.cs, you can see that we're in the namespace: HelloWorld and we're creating a class called MainPage, notice that it says "partial class".



```
15
16     // The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=237112.
17
18     namespace HelloWorld
19     {
20         /// <summary>
21         /// An empty page that can be used on its own or navigated to within a Frame.
22         /// </summary>
23         public sealed partial class MainPage : Page
24         {
25             public MainPage()
26             {
27                 InitializeComponent();
28             }
29         }
30     }
31 }
```

Using the keyword partial allows developers to create multiple class definitions and all have them be partial definitions of a single class, stored in different files. But as long as that file has the same class name, and is in the same namespace, and it has the keyword "partial", we can create many different files to represent a single class.

Notice that this MainPage derives from an object called Page. If we were to hover over, you can see that this is a class called Windows.UI.Xaml.Controls.Page. If you go to the MainPage.xaml, and look here at the very top, notice here that we're working with a Page object and notice that the class name is also HelloWorld, namespace, MainPage, class.

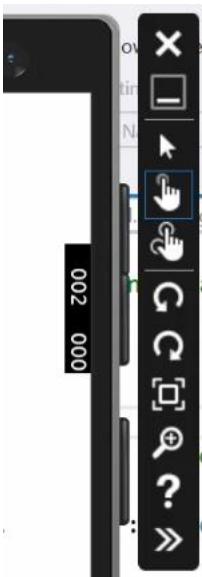


The MainPage.xaml and MainPage.xaml.cs are two different files that represent the same class. One of these files represents the class from a visual perspective and the other represents it from a behavioral aspect.

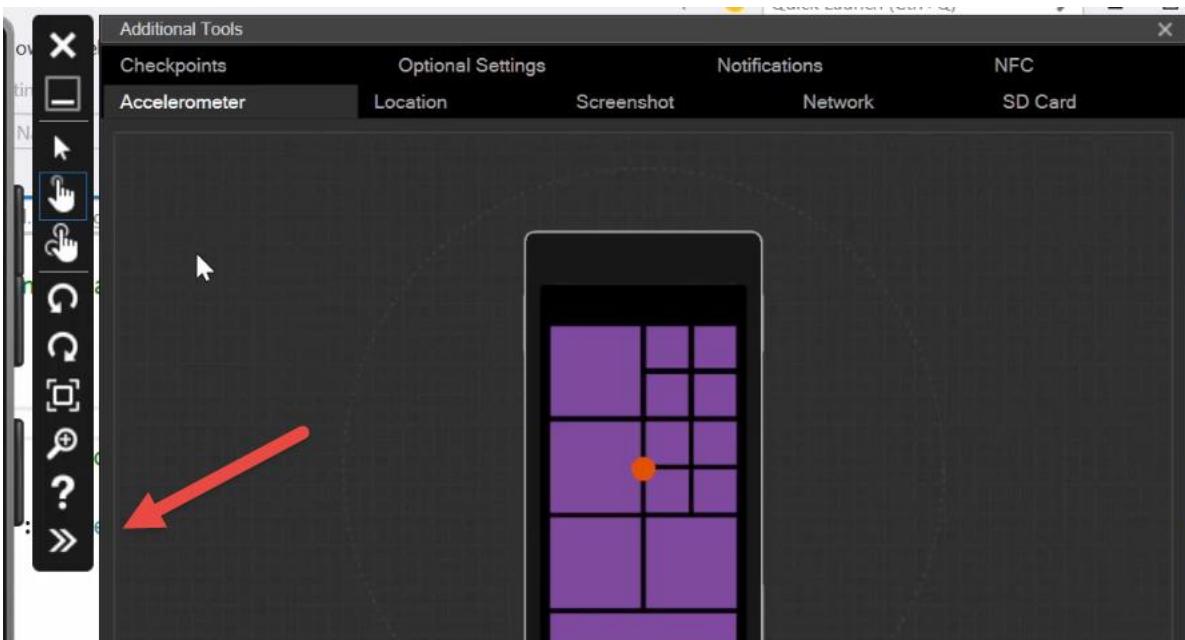
Next, in the previous lesson I ran the application in Debug mode. When building Windows 10 applications you can use all the debugging tools you've already learned about. You can set break points, step line-by-line through code, evaluate property values, variables, etc. as the application is running.

You may be wondering why there are so many different emulators available in debug mode. Not only do they emulate the different screen sizes and resolutions, but they also emulate different memory constraints for the different phone hardware available today. So we can test our application in a low memory environment or test our application in a high memory environment, as well as screen resolutions and sizes.

Furthermore, regarding the Emulators, there are a number of actions on the toolbar docked to the right side. Many of the icons and actions are obvious as to what they do: screen rotation, multi-touch input, zoom, etc.



There are also tools that we can access through this double chevron icon at the bottom. This opens up a whole dialogue of additional tools. So for example, if test our application's accelerometer feature, we can record a shaking motion, or some sort of gesture on the phone and then test it and see how our application responds to it if we are actually handling events for those type of gestures.



And we can also do the same with GPS. Later in this series we'll use the Emulator to simulate as if we are in a different part of the world to test Location services.

Back in Visual Studio on the MainPage.xaml, we also built our app utilizing various XAML controls from the Toolbox. A control has both a visual quality and a behavior, and both of these can be modified by you and I as developers by either changing properties of the control, like we did whenever we modified the XAML code or by using the Properties window, or by writing code in response to certain key events, at key moments in time in the application's life.

I demonstrated how to use the Property window's Events tab (lightning bolt icon) to handle a particular event for the button.

The toolbox has dozens of other controls, there are simple controls, input controls like the Button and TextBox. There are display controls like the TextBlock. There are input selection controls like DropDownListBoxes, or like the Date Control for selecting dates and times. There grids that can be used to display data, there are grids and other layout controls that can be used to help us to position controls in our application.

You can see that there's a Grid used in the page template for MainPage.xaml. We'll spend a lot more time than you might think learning about layout controls; they're very important to the adaptive story of Universal Windows Platform apps. Layout controls allow a single code base to be utilized across many different devices of device form factors.

The Universal Windows Platform API provides this rich collection of visual controls that work across all Windows devices. They allow input via mouse in some cases, or via finger in other cases. But that same API also provides us with thousands of methods across hundreds of classes of namespaces that allow you to do really cool stuff with your application.

For example, if you need to access the Internet to go retrieve some sort of resource, or if you want to work with a file on the file system, whether it be on a phone, or a tablet, or a desktop, or even the Xbox. Or if you want to play a media file, like a song or a video, there are methods in the UWP API, that make all of those things possible and a lot more, as you're going to learn throughout this lesson, a series of lessons.

These the layout and visual qualities of controls may need to change based on the screen size of the device they're running on. Creating adaptive triggers allows me to modify the layout and the scale of items in our application based on the size of the screen. Again, this will be another topic that I'll be demonstrating often throughout this series because it too, is one of the most important new features available in the Universal Windows Platform.

So, just backing up a little bit here, whenever I sit down to learn something new, a new technology, new API, I spend a lot of time just trying to organize things in my mind, making key distinctions, putting things in buckets, I guess you could say. And learning UWP for me, was no different. Categorizing the topic matter helps me understand its purpose and how it relates to all other topics.

First we will need to learn XAML. XAML allows us to layout controls on our app's forms. XAML is not specific to the Universal Windows Platform; it's been around for about a decade now. But building a Universal Windows Platform app really all starts with a fundamental understanding of the XAML language and how to mold it and shape it to do what we want it to do for our application. So, learning XAML is number one and we will start that in the very next lesson.

Second, we will need to learn how to use C# to call methods of classes in the Universal Windows Platform API itself. So, there are the programming languages: XAML and C#. Additionally, there is an API -- a library of functionality that Microsoft created -- that we can tap into and utilize for our applications. Developers write to code to call UWP API methods to do meaningful things in an application like load, and save data in a file, or access the network, etc.

Third, there are new C# features that will become important to learn like the keywords `async`, `await`, and `Task`. They are used extensively whenever you're working with the Universal Windows Platform API and I'll explain why when we get to that point.

Fourth, there are data formats like XML, which is the basis for XAML. Also JSON, which is short for "JavaScript Object Notation". We'll need to learn these because we will call web-based APIs to get current weather from the open weather map API, or to get comic book character data from Marvel comics and they'll deliver back their data in JSON or XML, so we will need to learn how to de-serialize JSON into a format that we can work with in C#.

Fifth, we'll learn about the tooling that helps us build Universal Windows Platform apps. So, we're talking about things like Visual Studios Designers and its wizards, and the project template itself, learn about the Emulators, etc.

Finally, we will learn the design patterns that we should follow whenever we're building our applications. So, patterns are guidance, they're good solutions to common problems. There are patterns that help us design our application's user interface, so that it looks and it behaves like other Windows 10 applications. Then there are patterns that apply to navigation, so that users who have used other Windows 10 applications feel comfortable navigating through the various pages in our application. And then there are also coding patterns, especially when working with data in C# and displaying updates to that data, back to the user.

Hopefully we took a few moments here and talked about all these topics that are very high level and my intent was to orient you in the right direction. Believe it or not we've already conquered some of the biggest conceptual hurdles that actually face when learning how to build Universal Windows Platform apps. All we need to do is just incrementally add the details ... "How do I do this? How do I do that?" It's not difficult, it just requires time to cover the details.

## UWP-004 - What Is XAML?

It's easy to figure out the absolute basics of XAML just by staring at the syntax. I imagine you were able to figure out the correlation between the tags, the properties, and what you saw in the visual designer in the previous lessons.

This lesson will discuss XAML's features and functions that may not be so obvious at first glance. Hope to take a couple of passes at this in subsequent lessons that, when combined together, will give us a pretty thorough understanding of XAML and how it all works.

First, we will talk about the purpose and the nature of XAML, especially in comparison to C#. Then in the next few lessons, we will talk about the special features of XAML; hidden features of the language that, again, may not be obvious when you first start looking at it.

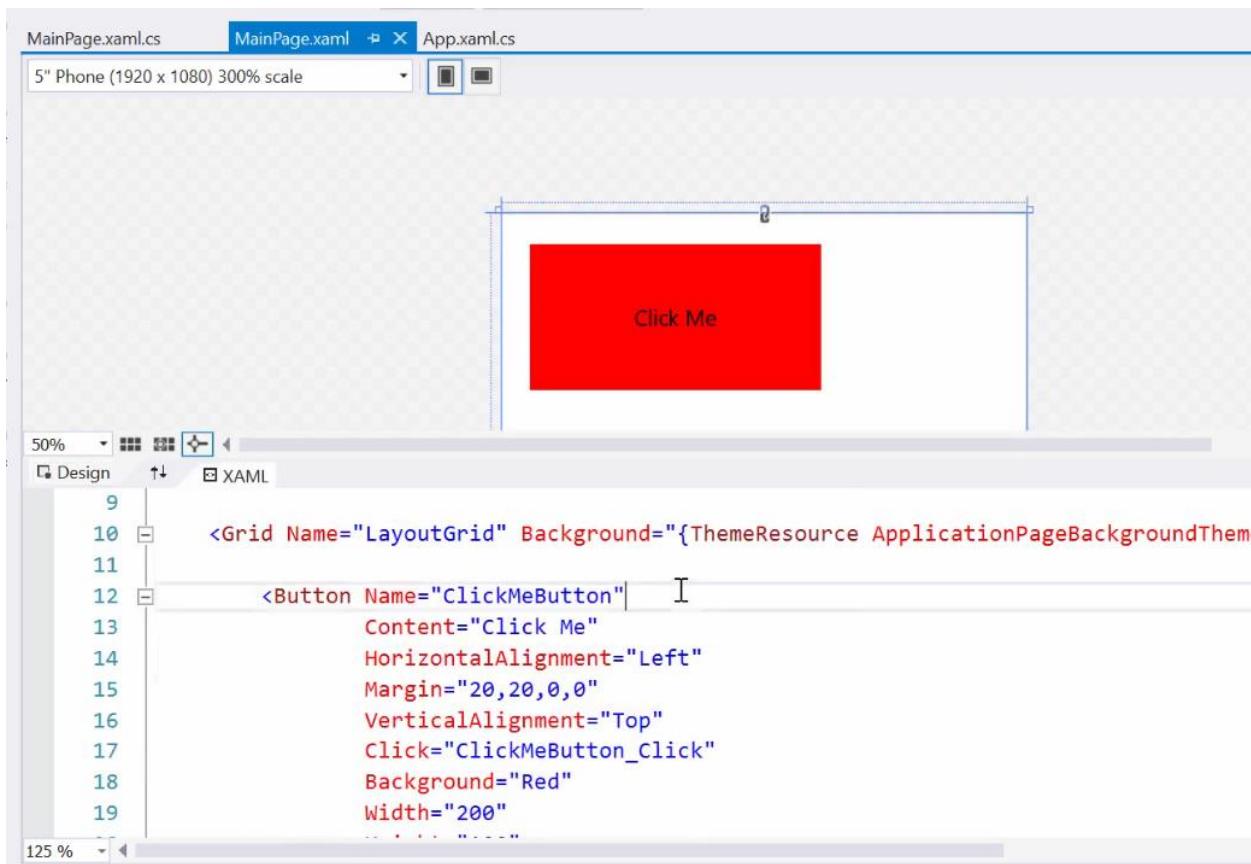
By the end of these first set of lessons my hope is that you'll have enough knowledge that you can look at the XAML that we write together in the remainder of this series and take a pretty good guess at what it's trying to do before I even explain what it does.

In the previous lesson, I made a passing remark about how XAML looks similar to HTML and that's no accident. XAML is really just XML, the eXtensible Markup Language. I'll explain that relationship in just a moment, but at a higher level, XML looks a lot like HTML insomuch that they share a common ancestry. Whereas HTML is specific to structuring a web page document, XML is a little bit more generic in nature. And by generic I mean that you can use it for any purpose that you devise, and you can define the names and the elements and the attributes to suit your needs.

In the past, developers have used XML for things like storing application settings or using it as a means of transferring data between two systems that were never intended to work together. And to use XML, you start out by defining what's called a Schema which declares the proper names of the elements and their attributes. A schema is similar to a contract that two parties agree to. Everybody agrees, both the producer of the XML and the consumer of the XML, to write and read the XML, to conform to those rules set forth in the schema. And now that they both agree that they're working in the same rules and the same contract, they can communicate with each other. So a schema is a really important part of XML. And just keep that in mind, because we will come back to that in just a moment.

XAML is a special usage of XML. Obviously we see, at least in this case, XAML has something to do with defining a user interface for our application. So in that regard, it feels a lot like HTML, however there is a big difference. XAML is actually used to create instances of classes and set values of their properties.

For example, in the previous lesson, we created a little Hello World application. I expanded on that for this lesson, just to add a little more design. I added a button. Notice that I put all the attributes on separate lines so now we can see, hopefully, the definition a little bit better. There's a red button called "ClickMeButton", and there are also a MessageTextBlock that will appear below it.



It is very similar to the previous example, however I simply started over from scratch.

When a user clicks the button the Click event in the code behind will set the TextBlock's Text property equal to "What is XAML?"

The screenshot shows the Windows Phone 8.1 code editor with three tabs at the top: MainPage.xaml.cs, MainPage.xaml, and App.xaml.cs. The MainPage.xaml.cs tab is selected. The code editor shows the following C# code:

```
1 //<summary>  
2 // An empty page that can be used on its own or navigated to within a Frame.  
3 //</summary>  
4 public sealed partial class MainPage : Page  
5 {  
6     public MainPage()  
7     {  
8         this.InitializeComponent();  
9     }  
10    private void ClickMeButton_Click(object sender, RoutedEventArgs e)  
11    {  
12        MessageTextBlock.Text = "What is XAML?";  
13    }  
14 }  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36
```

To demonstrate how XAML works I will comment out the Button control in the XAML editor. To comment out XAML you use the following syntax:

```
<!--
```

Your XAML here

```
-->
```

Visual Studio's code editor displays comments with a green foreground by default.

Next, I want to give the Grid control a name to access it programmatically. I give it the name LayoutGrid because that is indicative of what it's actually doing for us.

```
 /  
 8   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 9     mc:Ignorable="d">  
10   <Grid Name="LayoutGrid" Background="{ThemeResource ApplicationPageBackgroundBrush}"  
11     <!--  
12     <Button Name="ClickMeButton"  
13       Content="Click Me"
```

Next, I'll handle the Loaded event for the Page by typing:

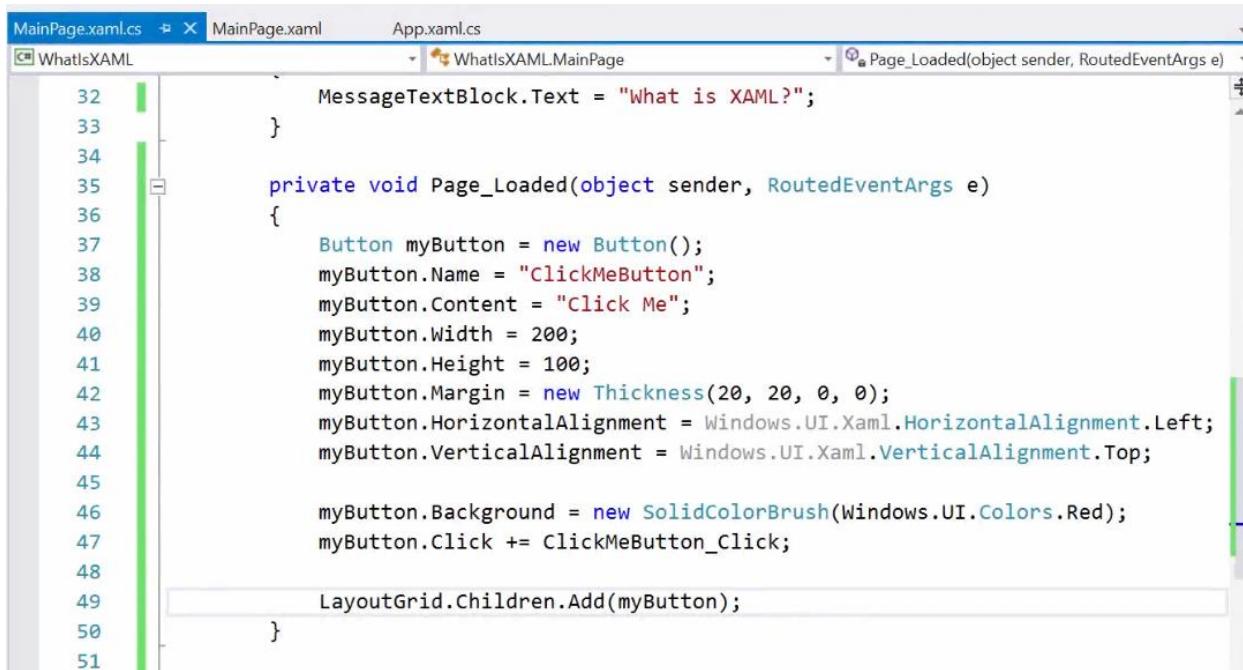
```
Loaded="Page_Loaded"
```



... inside of the Page tag.

Next, with my mouse cursor on the term Page\_Loaded I press the F12 key on my keyboard which will open the Page's code behind to that event handler's method stub. This event handler will execute whenever the page has loaded into memory.

Next, I'll add C# code that will perform the same function as the XAML that I commented out previously.



The screenshot shows the Visual Studio IDE with the MainPage.xaml.cs file open. The code defines a button named 'ClickMeButton' with the text 'Click Me'. It sets the button's width to 200, height to 100, and background color to red. The button is aligned to the left and top of its container. The Click event is handled by the ClickMeButton\_Click method.

```
32     MessageTextBlock.Text = "What is XAML?";
33 }
34
35 private void Page_Loaded(object sender, RoutedEventArgs e)
36 {
37     Button myButton = new Button();
38     myButton.Name = "ClickMeButton";
39     myButton.Content = "Click Me";
40     myButton.Width = 200;
41     myButton.Height = 100;
42     myButton.Margin = new Thickness(20, 20, 0, 0);
43     myButton.HorizontalAlignment = Windows.UI.Xaml.HorizontalAlignment.Left;
44     myButton.VerticalAlignment = Windows.UI.Xaml.VerticalAlignment.Top;
45
46     myButton.Background = new SolidColorBrush(Windows.UI.Colors.Red);
47     myButton.Click += ClickMeButton_Click;
48
49     LayoutGrid.Children.Add(myButton);
50 }
51 }
```

To verify that functionality has not changed, I run the application in Debug mode.

The point of this exercise is that it took about 12 lines of C# code to do what I was able to do in just one line of XAML code. Admittedly, I spaced the XAML out over several lines, but you can see that the C# version of this is much more verbose.

Second, this provides insight into what's going on as defining elements and attributes in XAML: we're creating new instances of classes in the Universal Windows Platform API and defining and setting their attributes, their properties, just like we're doing here in the C# code.

The important takeaway is this; XAML is simply a way to create instances of classes and set those objects' properties in a much more simplified, succinct syntax.

Furthermore, when using XAML, I get this automatic feedback here in the visual designer. In the preview pane. So I can see the impact of my changes instantly if I choose to work like that.

So in the case of the procedural C# that I wrote, I have to run the application each time that I wanted to see how my tweaks to the code actually worked.

To recap, we learned about the basics and purpose of XAML. First, XAML is just a specific flavor of XML. It follows all the rules of XML. Somebody defines a schema, a contract, and then both the producer and consumer of XML agree to the contract, and then they can begin to work together knowing that they're pretty much on the same page. Now in this case, the contract is XAML and it was defined by Microsoft. The producer of the XML is you, me, in Visual Studio, and then the consumer of the XML is the compiler, which will turn our code in an executable that will run in Windows 10.

Second, XAML is an easy way to create instances of classes and set their properties. And sure, you could do it all in C#, but it's much more verbose and you would lose the design-time tooling that we've become accustomed to in here in two or three lessons.

## UWP-005 - Understanding Type Converters

In this lesson I want to explain one curious little feature called Type Converters that we see in play in the example we created in the previous lesson.

If you took a few moments to examine the project, and if you employed a keen eye, you may have noticed that the HorizontalAlignment property is set to a string with the value "Left".

```
11
12     <Button Name="ClickMeButton"
13         Content="Click Me"
14         HorizontalAlignment="Left"
15         Margin="20,20,0,0"
16         VerticalAlignment="Top"
17         Click="ClickMeButton_Click"
18         Background="Red"
19         Width="200"
20         Height="100"
21     />
```

However, the C# version is a little bit different. (You see a similar situation when observing the VerticalAlignment property set to "Top".)

```
54
55     private void Page_Loaded(object sender, RoutedEventArgs e)
56     {
57         Button myButton = new Button();
58         myButton.Name = "ClickMeButton";
59         myButton.Content = "Click Me";
60         myButton.Width = 200;
61         myButton.Height = 100;
62         myButton.Margin = new Thickness(20, 20, 0, 0);
63         myButton.HorizontalAlignment = HorizontalAlignment.Left;
64         myButton.VerticalAlignment = VerticalAlignment.Top;
65
66         myButton.Background = new SolidColorBrush(Windows.UI.Colors.Red);
67         myButton.Click += ClickMeButton_Click;
68
69         LayoutGrid.Children.Add(myButton);
70     }
```

If you take a look at when we set that HorizontalAlignment property, we're actually using a strongly typed enumeration (not a string) of type Windows.UI.Xaml.HorizontalAlignment and the particular enumeration is also Left -- however it is strongly typed.

How does this work? Why is it that we can use a string here, but we have to use a strong type in C#? The reason why this works is because the XAML parser will perform a conversion to turn this string value into a strongly typed version of that value of type Windows.UI.Xaml.HorizontalAlignment.left through the use of a feature called a Type Converter.

A Type Converter is simply a class that has one function and that is to translate a string value into a strong type. And there are several of these that are built into the Universal Windows Platform API that we use throughout this series.

So, in this example of the HorizontalAlignment property, when it was developed by Microsoft's developers it was marked with a special attribute in the source code, which signals to the XAML parser that looks through our code and makes sure everything is okay and that it would compile to run that string value "Left" through a Type Converter and try to match the literal string "Left" with one of the enumeration values defined in this Windows.UI.Xaml.HorizontalAlignment enumeration.

Just for fun, let's take a look at what happens if we were to misspell the word "Left". Let's get rid of the "t" on the end.



The screenshot shows a code editor with a vertical line of numbers on the left (11 to 21). A button element is defined with various properties. The 'HorizontalAlignment' property is set to 'Lef', which is underlined with a red wavy line, indicating a spelling error. A tooltip box appears over the 'Lef' text with the message 'Requested value 'Lef' was not found.' This visual cue serves as a type converter, alerting the developer that the intended value 'Left' was not found in the enumeration.

```
11
12     <Button Name="ClickMeButton"
13         Content="Click Me"
14         HorizontalAlignment="Lef" // Error here
15         Margin="20,20,0,0"
16         VerticalAlignment="Top"
17         Click="ClickMeButton_Click"
18         Background="Red"
19         Width="200"
20         Height="100"
21     />
```

Immediately the XAML parser says, "Wait a second. I can't find a match for this literal string, lef, (without the T), with any of the enumerations that are defined in this Windows.UI.Xaml.HorizontalAlignment enumeration."

This will cause a compilation error because the Type Converter can't find the exact match to convert it to a strong type.

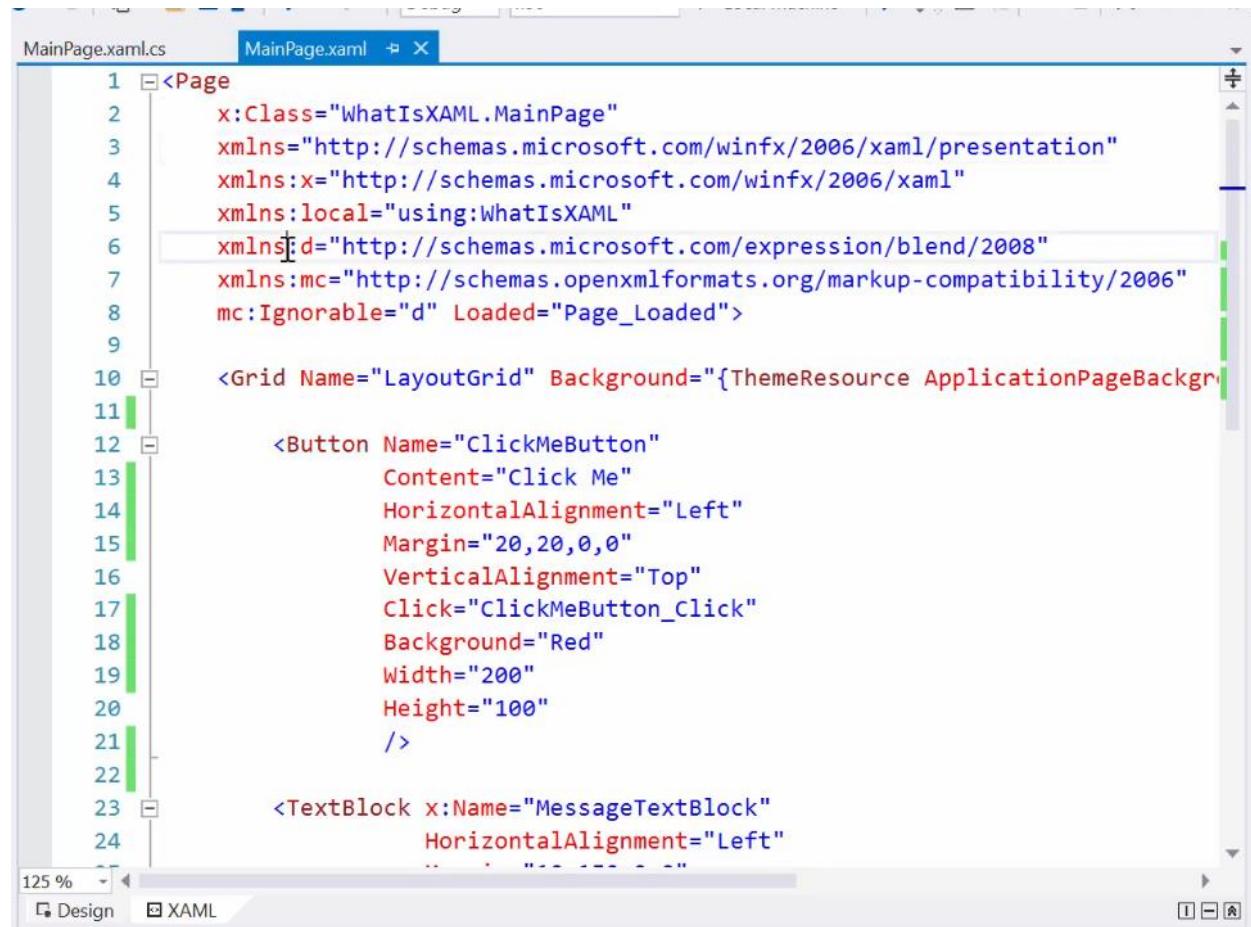
So, one of the first characteristics of XAML is that it is a succinct means of creating instances of classes and setting properties in Type Converters. This is one little trick that XAML uses to accomplish this so that we don't have to waste characters and that we can actually just use a very succinct syntax to set values that might be actually representing longer class and enumeration names behind the scenes.

## UWP-006 - Understanding Default Properties, Complex Properties and the Property Element Syntax

This lesson will discuss other XAML syntax features, specifically Default Properties, Complex Properties, and the Property Element Syntax. Finally, it will discuss how an intelligent XAML parser reduces extraneous XAML code because it can infer the relationships between the elements by their context

To begin, notice the containment relationship between objects defined in XAML. Since XAML is essentially an XML document, we can embed elements inside of other elements and that implies a containment relationship.

Observing the MainPage.xaml at the top-most level there's Page object and inside of the opening and closing Page tag, there's a Grid object, and inside of the opening and closing Grid element, there's a both a Button and a TextBlock.



```
MainPage.xaml.cs MainPage.xaml < X
1 <Page
2     x:Class="WhatIsXAML.MainPage"
3     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:local="using:WhatIsXAML"
6     xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8     mc:Ignorable="d" Loaded="Page_Loaded">
9
10    <Grid Name="LayoutGrid" Background="{ThemeResource ApplicationBackground}"
11        Margin="20">
12        <Button Name="ClickMeButton"
13            Content="Click Me"
14            HorizontalAlignment="Left"
15            Margin="20,20,0,0"
16            VerticalAlignment="Top"
17            Click="ClickMeButton_Click"
18            Background="Red"
19            Width="200"
20            Height="100"/>
21
22        <TextBlock x:Name="MessageTextBlock"
23            HorizontalAlignment="Left"
24            Text="Hello World"/>
25    
26
27 </Page>
```

The proper parlance in XAML is that the Page's content property is set to a new instance of the Grid control. The Grid control has a property collection called Children. In this case two instances a Button and a TextBlock are added to the Children collection.

Admittedly those properties (I.e., Content, Children) are not explicitly defined in the XAML. However, what is going on behind the scenes is that those properties are being set. I'll address how this is possible at the very end of this lesson. For now, understand that there's more than meets the eye.

By using XAML to indicate containment we're actually setting the Grid's Children property and we're setting the Pages' content property. If you take a look at the C# version of our code, it more accurately shows that relationship between the LayoutGrid and the Button and TextBlock. The LayoutGrid has a Children collection of controls, and we're adding to that Children collection an instance of the Button, OK. We just don't see that here in our XAML.

Next, Default Properties are employed in the example's XAML. Depending on the type of control that you're working with, the Default Property for that control can be populated by using this embedded style syntax where we have XAML elements defined inside of other XAML elements. For example, the Button control's Default Property is Content, so I can remove that property from the properties or the attributes of that element, and then I can also just remove that self-enclosing syntax there for the element, and just create a proper closing tag for the Button.

```
11
12     <Button Name="ClickMeButton"
13         HorizontalAlignment="Left"
14         Margin="20,20,0,0"
15         VerticalAlignment="Top"
16         Click="ClickMeButton_Click"
17         Background="Red"
18         Width="200"
19         Height="100"
20         >Hello World</Button>
21
```

By using that Default Property syntax for the Button, we can actually see in the Design View that we've changed the Content property for the Button.

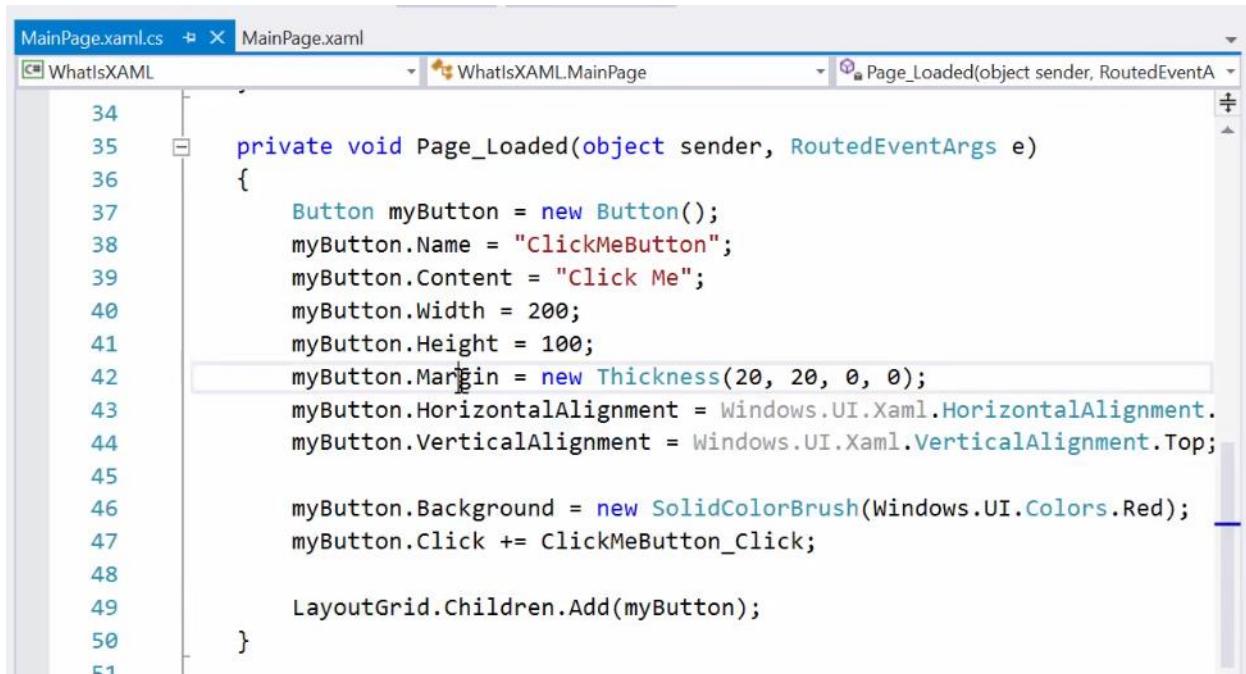
Next, let's revisit Type Converters for a moment. In last lesson, I only highlighted some simple Type Converters like the HorizontalAlignment property's string value to perform the type conversion to an instance of an enumeration Windows.UI.Xaml.HorizontalAlignment.Left.

However, there are more complex versions of Type Converters, like we see here in the Margin.

```
11
12     <Button Name="ClickMeButton"
13         HorizontalAlignment="Left"
14         Margin="20,20,0,0" ←
15         VerticalAlignment="Top"
16         Click="ClickMeButton_Click"
17         Background="Red"
18         Width="200"
19         Height="100"
20         >Hello World</Button>
```

Here, our Margin property in XAML is set to a literal string of "20, 20, 0, 0", and if you take a look at the Background property, it also is set to just a simple string, a literal string of "Red".

But if you take a look at the C# version, there are a little bit more going on here. The Type Converters have to work a little bit harder than they did with the Horizontal and Vertical Alignment. In the case of the Margin, notice that we're just not giving it an enumeration but we're actually creating a new instance of an object called Thickness, and that Thickness has a constructor that accepts four input parameters that happen to be the left margin, top margin, right margin, bottom margin.



```
MainPage.xaml.cs  MainPage.xaml
MainPage.xaml.cs  MainPage.xaml
WhatIsXAML  WhatIsXAML.MainPage  Page_Loaded(object sender, RoutedEventArgs e)
34
35     private void Page_Loaded(object sender, RoutedEventArgs e)
36     {
37         Button myButton = new Button();
38         myButton.Name = "ClickMeButton";
39         myButton.Content = "Click Me";
40         myButton.Width = 200;
41         myButton.Height = 100;
42         myButton.Margin = new Thickness(20, 20, 0, 0);
43         myButton.HorizontalAlignment = Windows.UI.Xaml.HorizontalAlignment.Center;
44         myButton.VerticalAlignment = Windows.UI.Xaml.VerticalAlignment.Top;
45
46         myButton.Background = new SolidColorBrush(Windows.UI.Colors.Red);
47         myButton.Click += ClickMeButton_Click;
48
49         LayoutGrid.Children.Add(myButton);
50     }
51 }
```

Likewise, if you take a look at the Background property here, we're actually setting it to a new instance of an object called a SolidColorBrush, and we're passing in as an input parameter to the constructor, a enumeration of type, Windows.UI.Colors.Red.

In some situations, there are property values that are simply too complex to be represented merely by Type Converters, or handled by Type Converters. When a control's property is not easily represented by just a simple XAML attribute like we see in these examples that we've been looking at, then it's referred to as a Complex Property.

To demonstrate this, I will remove the Background attribute from the Button definition. I will also remove this Default Property and I'll reset the Content property here back to what it was.

```

11
12     <Button Name="ClickMeButton"
13         HorizontalAlignment="Left"
14         Content="Click Me"
15         Margin="20,20,0,0"
16         VerticalAlignment="Top"   ←
17         Click="ClickMeButton_Click"
18         Width="200"
19         Height="100"
20     ></Button>
21

```

I'll comment out the C# code in the Page\_Loaded event:

```

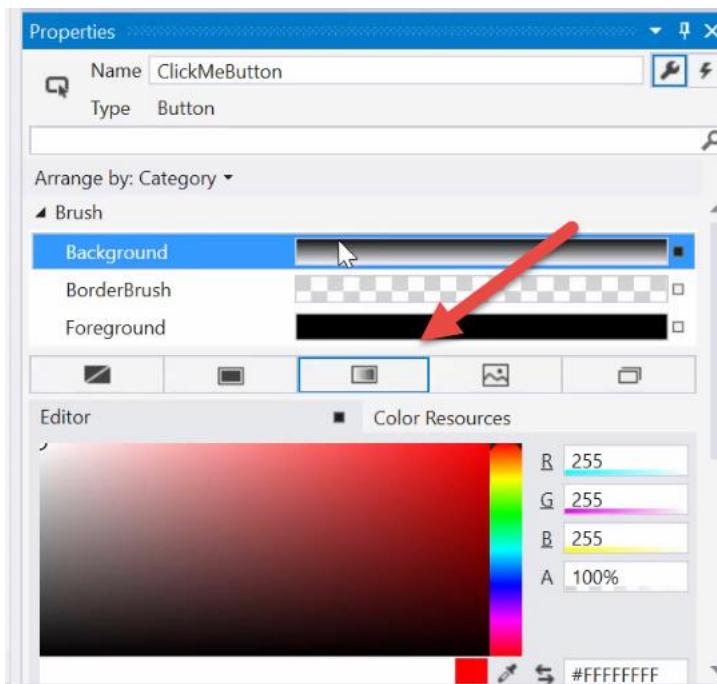
MainPage.xaml.cs  X MainPage.xaml*
WhatIsXAML          WhatIsXAML.MainPage          Page_Loaded(object sender, RoutedEventArgs e)
36     {
37         /*
38         Button myButton = new Button();
39         myButton.Name = "ClickMeButton";
40         myButton.Content = "Click Me";
41         myButton.Width = 200;
42         myButton.Height = 100;
43         myButton.Margin = new Thickness(20, 20, 0, 0);
44         myButton.HorizontalAlignment = Windows.UI.Xaml.HorizontalAlignment.Left;
45         myButton.VerticalAlignment = Windows.UI.Xaml.VerticalAlignment.Top;
46
47         myButton.Background = new SolidColorBrush(Windows.UI.Colors.Red);
48         myButton.Click += ClickMeButton_Click;
49
50         LayoutGrid.Children.Add(myButton);
51     */
52 }

```

Selecting the Button with my mouse cursor, I look at the Properties window and ensure that the Name is set to ClickMeButton. That lets me know that I'm in the right context for the Properties window. I want to set that Background property again using the Property dialog here, and I want to change the Background property.



Now since I removed the XAML, the Background property is not being set at all. I will change from a SolidColorBrush that to a GradientBrush to use a gradient.



By default, the gradient starts out as black and then it slowly fades into the color white.

Notice the XAML that was generated:

```

11
12     <Button Name="ClickMeButton"
13         HorizontalAlignment="Left"
14         Content="Click Me"
15         Margin="20,20,0,0"
16         VerticalAlignment="Top"
17         Click="ClickMeButton_Click"
18         Width="200"
19         Height="100"
20     >
21     <Button.Background>
22         <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
23             <GradientStop Color="Black" Offset="0"/>
24             <GradientStop Color="White" Offset="1"/>
25         </LinearGradientBrush>
26     </Button.Background>
27 </Button>
28

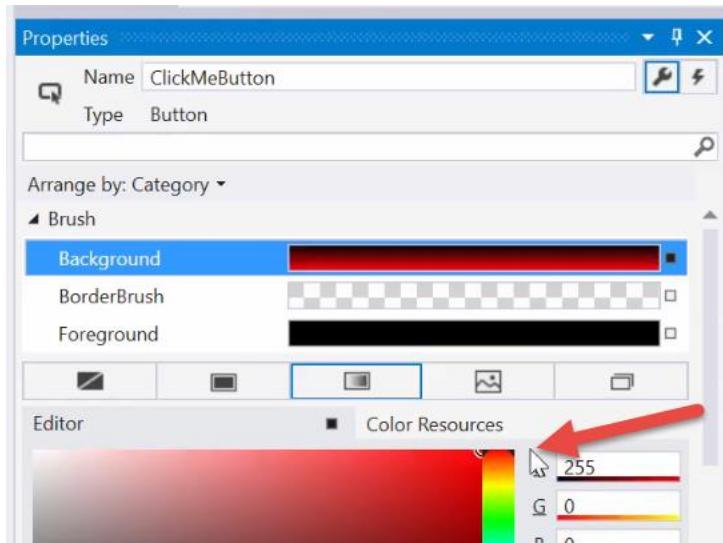
```

Quite a bit of XAML has been added to the project between the opening and closing Button tags. Here, the Button.Background property is set to an instance of LinearGradientBrush.

So whenever you see this type of syntax -- this type of element embedded inside of another element -- this is called a Property Element Syntax. In other words, there's an Object.Property and XAML is added inside of the Property Element Syntax that will define the values for this Complex Property.

You may wonder what is a LinearGradientBrush? First whenever you see the term "brush" think "paint brush"; you're just thinking about color or colors that will be painted on an object. In this case, we're looking at a paintbrush that can paint a color starting at the top of the wall, at black, and by the time we paint the bottom of the wall, it'll be white.

However, I want to modify the colors. Making sure that I'm working with the Button control, I go to the Properties window. I want it to change the color White to the color Red. I select the little circle in the upper left hand corner and I drag it all the way to the right hand side.



Notice that it changed the GradientStop from White to Red.



```
21     <Button.Background>
22         <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
23             <GradientStop Color="Black" Offset="0"/>
24             <GradientStop Color="Red" Offset="1"/>GradientStop.Color
25         </LinearGradientBrush>
26     </Button.Background>
27 </Button>
```

Let's save that and take a look at what it looks like here in the designer, and you can see how that's represented. At the top of the Button, it's black, the bottom of the button is red, great.



Note: You probably never, ever want to do this because it just doesn't follow the same aesthetic as the rest of the applications that your users are going to see in Windows 10, but let's pretend for now that you want to express your individuality or there is some branding that you want to do that your company is known for and you need that gradient.

In the XAML editor, if you do want to define a LinearGradientBrush, you have to supply quite a bit of information here. You have to not only give it the colors that you want to use, you also have to give it a collection of GradientStops and their Offsets, where one Color starts and the other Color stops. While this does look like a lot of additional XAML just to represent a Color, the code snippet here that I have highlighted is actually shortened automatically by Visual Studio. Let me take just a moment here and type out what the full XAML should be if Visual Studio didn't try to compact it for us



```
20     >
21     <Button.Background>
22         <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
23             <LinearGradientBrush.GradientStops>
24                 <GradientStopCollection>
25                     <GradientStop Color="Black" Offset="0"/>
26                     <GradientStop Color="Red" Offset="1"/>
27                 </GradientStopCollection>
28             </LinearGradientBrush.GradientStops>
29         </LinearGradientBrush>
30     </Button.Background>
31 </Button>
```

I've added a couple lines of code here in Line 23 and 24, and then the closing tags in Lines number 28 and 29. I'm setting the Button's Background to a new instance of the LinearGradientBrush class. The LinearGradientBrush class has a property called GradientStops. The GradientStops property is of type GradientStopCollection, so we create a new instance of GradientStopCollection and add two instances of the GradientStop object to that collection.

So, as you can see, that we were able to omit lines 23, 24, as well as 27, or rather 28, or yeah, 27 and 28, and we were able to omit this, or actually, Visual Studio, when it generated the XAML for us, was able to omit it because it wanted to make the XAML more concise and more compact, and it's made possible by an intelligent XAML parser.

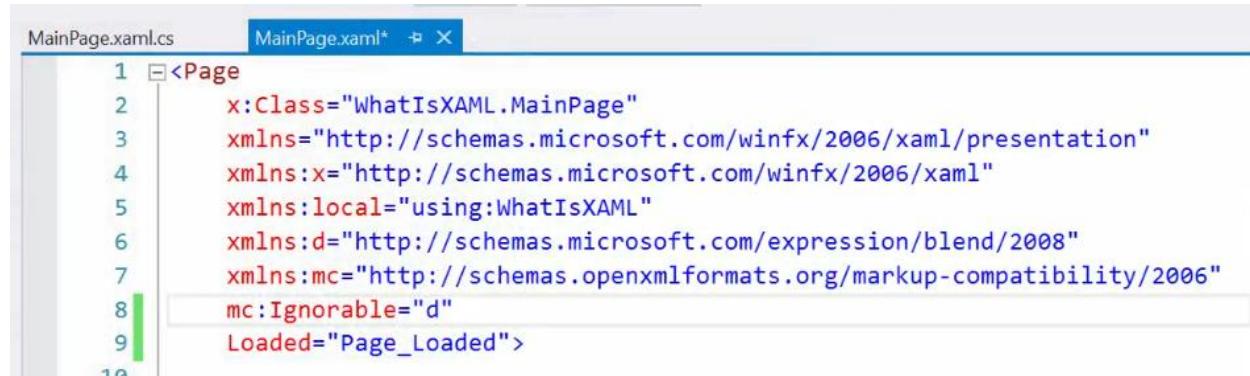
We talked about Default Properties at the very outset of this lesson. The GradientStops property is the Default Property for the LinearGradientBrush. The GradientStops property is of type GradientStopCollection, so the only thing that we can put in a GradientStopCollection are instances of GradientStop. So since we just put two GradientStop objects inside of a LinearGradientBrush, it knows that we're dealing with the Default Property and that Default Property is of type GradientStopCollection, so we don't even need to put that in there. That's already implied. We can supply the two GradientStops and it can infer the rest that it needs from its context.

So the moral of the story is that the XAML parser is intelligent, and it doesn't require us to include redundant code that it can infer from the context. As long as it has enough information to create that object graph correctly, it'll do it and we don't have to give it any more than it needs. And furthermore, Visual Studio will emit concise code if we use the Properties window or other tooling support inside of Visual Studio.

So just to recap this lesson, we talked about a number of different things again that all fit together. We talked about Default Properties, we talked about Complex Properties, then we talked about the Property Element Syntax like we saw here, like we saw, we've actually removed some of those lines of code between Line 22 and Line 25, and we also talked about how an intelligent XAML parser allowed us to remove those lines of code because it allows us to keep our XAML compact by inferring from context what inner elements should be used for. And finally, whenever possible, Visual Studio will generate concise XAML for us.

## UWP-007 - Understanding XAML Schemas and Namespace Declarations

Up to now we've avoided all of this ugly code here at the very top of our MainPage.xaml.



```
MainPage.xaml.cs MainPage.xaml* X
1 <Page
2     x:Class="WhatIsXAML.MainPage"
3     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:local="using:WhatIsXAML"
6     xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8     mc:Ignorable="d"
9     Loaded="Page_Loaded">
10
```

A few lessons ago I said that XAML is actually just a flavor of XML, or rather that XAML uses XML to implement its syntax. I also said that in order to use XML (or at least use it properly) you have to create schemas. And a schema is like a contract that both the producer of the XML and the consumer of the XML agree upon so that they can work together.

So if that's the case, then where do we find the schema for XAML, for the page that we're currently working on? Well, actually, you might have guessed it, that's lines number three through seven, specifically, here at the very top of our file.

There are actually five schemas that the MainPage.xaml adheres to and it promises that it will follow the rules set forth in those schemas. Furthermore, it associates each schema with a namespace. A XAML namespace is just like we would use in a C# namespace.

For example, the very first schema is defined in the namespace associated with the letter "x", and that adheres to a schema defined here at <http://schemas.microsoft.com/winfx/2006/xaml>.

So when we prefix some XAML in our code with a namespace, we're saying that the rules for this particular XAML is specified by the associated schema.

That also means that everything else in this document, everything that doesn't have any prefix in front of it, whether it be the "x", the "local", the "d", the "mc", and so on, that means it adheres to this default namespace that's defined at the very top: <http://schemas.microsoft.com/winfx/2006/xaml/presentation>

You may wonder if you can see the schema if you were to open up that URL (above) into a web browser. However, you'll get a 404 error "Page Not Found". Schemas are not really defined at location in the sense that you can go out and you can look at the schemas online. It's more of just a unique name, just like namespaces are unique names, just intended to disambiguate. When Microsoft implemented Visual Studio and Blend and the XAML parser and compiler, they adhered to these "unwritten" rules -- or they might be written somewhere, but we can't get to them, at least not from this particular URL. In fact, it's not really even a URL, it's more of a URI, a Uniform Resource Indicator that's used as a namespace, in a sense, to define schemas that we can use in our document.

The intricacies of this may be confusing, however there is really just one main takeaway in this lesson: everything in our XAML Page follows a namespace, or rather a schema, an XML schema that's defined in one of these URIs.

You might wonder what each of these schemas are used for. For example, what's really the difference between this topmost schema and the second schema? The difference is really subtle, but essentially the top schema defines all of the UI elements, so the Grid, the Button, and all their attributes whereas the second schema is for the general rules for XAML.

Next up you see that we have a local namespace. This is actually not a schema, per se. It's actually a namespace just like we would use a using statement in C# -- so that we can reference classes without their full namespace hierarchy. So say for example I create a Car class in my application. I would be able to reference it inside of my XAML because as long as I prefix that with local:car, I'd be able to reference it here in my XAML document.

The last two are used by the Designer tools of Visual Studio and Blend. Actually you can see that the schema's URI actually uses: expressionblend/2008.

However, if you look at this little line number 8, it said that the mc: namespace prefix is ignorable. So at runtime, just ignore this namespace - we don't need it. The only reason why this ignorable attribute exists is because of its definition at the URI <http://schemas.openxmlformats.org/markup-compatibility/2006>.

Now that you understand what these do, you can completely forget this. They have no actionable result until we get a little bit deeper into our software development careers. We may have to use these namespace prefixes, especially the local one, and we may have to create one when we get into working with data, when we get to working with custom classes that we've created that represent data. But that's later. Just know that you should never ever modify this code here at the very top of a XAML file unless you have a really good reason to do so. And you almost never will.

The moral of the story is that, yes, XAML is XML. These are the schemas that it adheres to. Don't make any changes to this. But it's what makes everything work together seamlessly between Visual Studio and the parser and the compiler.

## UWP-008 - XAML Layout with Grids

In this lesson, I want to begin talking about layout, or rather, the process of positioning visual controls and other elements on your application's user interface. There are several different XAML controls that exist for the purpose of layout, and cover most of the popular ones in this series of lessons.

In the past, layout was relatively simple. After all, you were typically only laying out an application for a single form factor -- a single device like a phone or a desktop application. However, there are a few new wrinkles introduced as we begin to build applications that can adaptively resize based on the device that we run our app on. And this is one of the new key features in app development on the Windows platform.

We'll start with simple concepts then build up to the more challenging concepts in the lessons that follow.

Before we begin in earnest, I want to point out one thing regarding all XAML controls that are intended for the purpose of layout. Most controls have a Content property, so your Button control has a Content property, for example. And the Content property can only be set to an instance of another object. So in other words, I can set the Content property of a Button to a TextBlock:

```
9
10 <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11   <Button>
12     <TextBlock>Hi</TextBlock>
13     <Image Source="Assets/Square44x44Logo.scale-200.png" />
14   </Button>
15 </Grid>
16 </Page>
```

... but then I also have added an Image inside of that Button control's default property as well. Whenever I attempt to put more than one control inside of the Content property, I get the error: The property "Content" can only be set once.

However, layout controls are intended to host more than one control. And so as a result, they do not have a Content property. Instead, they usually have a Children property that is of a special data type, a collection data type that can hold XAML controls called UIElementCollection.

In XAML, as we add new instances of controls inside of the definition of our layout control, we're actually calling the Add method of our layout control's UIElementCollection, or rather, just the collection property. So here again, XAML hides a lot of the complexity for us and makes our code very concise by inferring our intent by how we write our XAML.

So we will begin learning about layout in this lesson by looking at the Grid control. Like any grid, it allows you to define both rows and columns to create cells. And then each of the controls that are used by your application can request which row and which column that they want to be placed inside of. So whenever you create a new app using the blank app template, you're provided very little guidance. You get a single empty Grid with no rows or no columns defined.

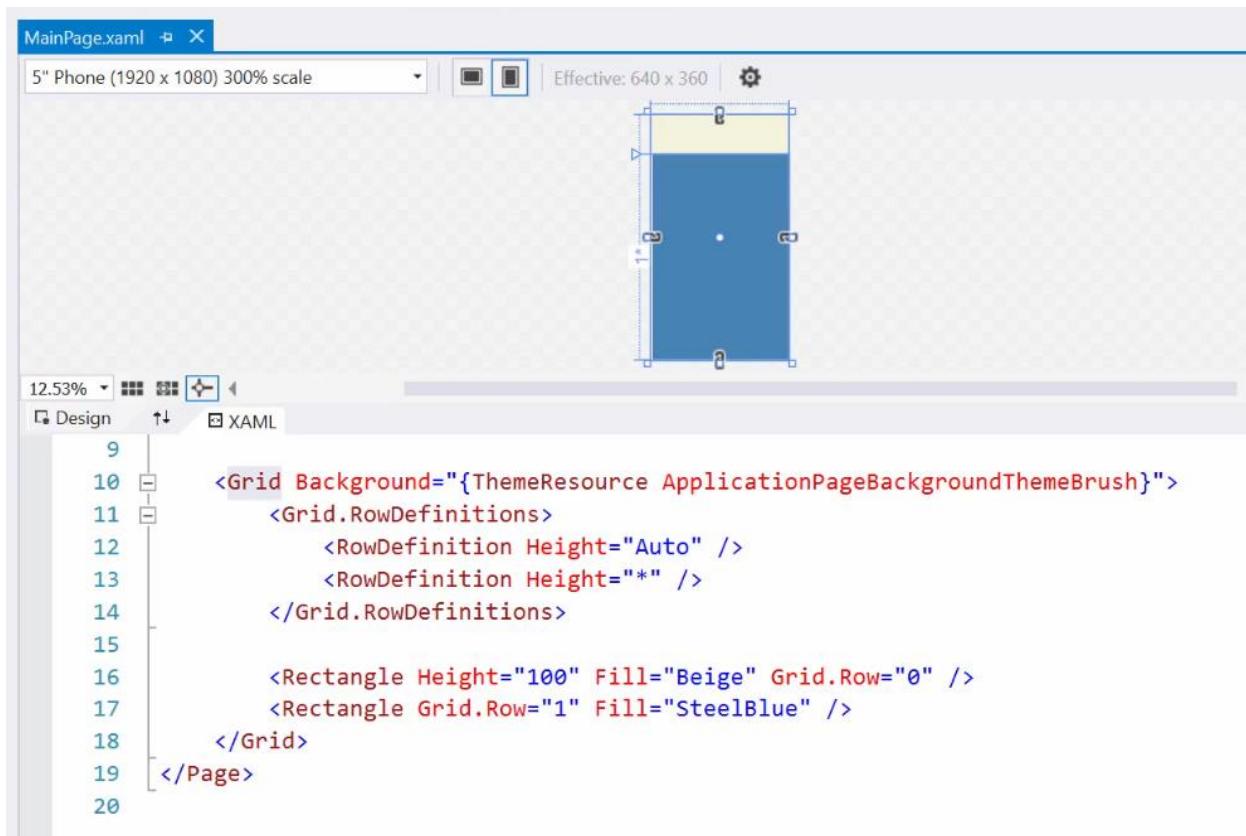
```

9
10    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11
12    </Grid>
13  </Page>

```

However, by default, there is always one row definition and one column definition, even if it is not explicitly defined in your XAML. These take up the full vertical and horizontal space available to represent one large cell in the grid. Any items that are placed between that opening and closing Grid element are understood to be inside of that single, implicitly defined cell.

I created a quick example of a grid that defines two rows, just to illustrate two primary ways of creating rows and setting their heights. See the associated project called: RowDefinitions.



So here you can see that I want you to notice that I have two rectangles. There is an upper rectangle and a lower rectangle. And those are defined through a series of RowDefinition objects here. So you can see that I have inside of the grid this property element syntax to define a collection of RowDefinitions with instances of RowDefinition created with their Height property set.

The first RowDefinition has its Height property set to Auto and its second RowDefinition object has its Height set to star (\*).

And then there are two Rectangle objects. Notice how I'm setting the Row property that this Rectangle object wants to put itself inside of. It wants to put itself inside of the row zero, the first row (since when referencing Rows and Columns you apply a zero-based counting.)

The second Rectangle wants to put itself inside of the Grid.Row="1".

Notice is how the Rectangles are putting themselves into the various Grid Rows, and then also how you reference both Rows and Columns using a zero-based numbering scheme.

Next, observe the weird syntax: Grid.Row and Grid.Column. And these are called "Attached Properties". Attached Properties enable an object (in this case, a rectangle) to assign a value for a property (in this case, the row property, but it could apply to the column property as well), to assign a value for a property that its own class does not define. So, nowhere in the Rectangle class definition is there a Grid.Row property, or even a Row property. These are all defined inside of the Grid object.

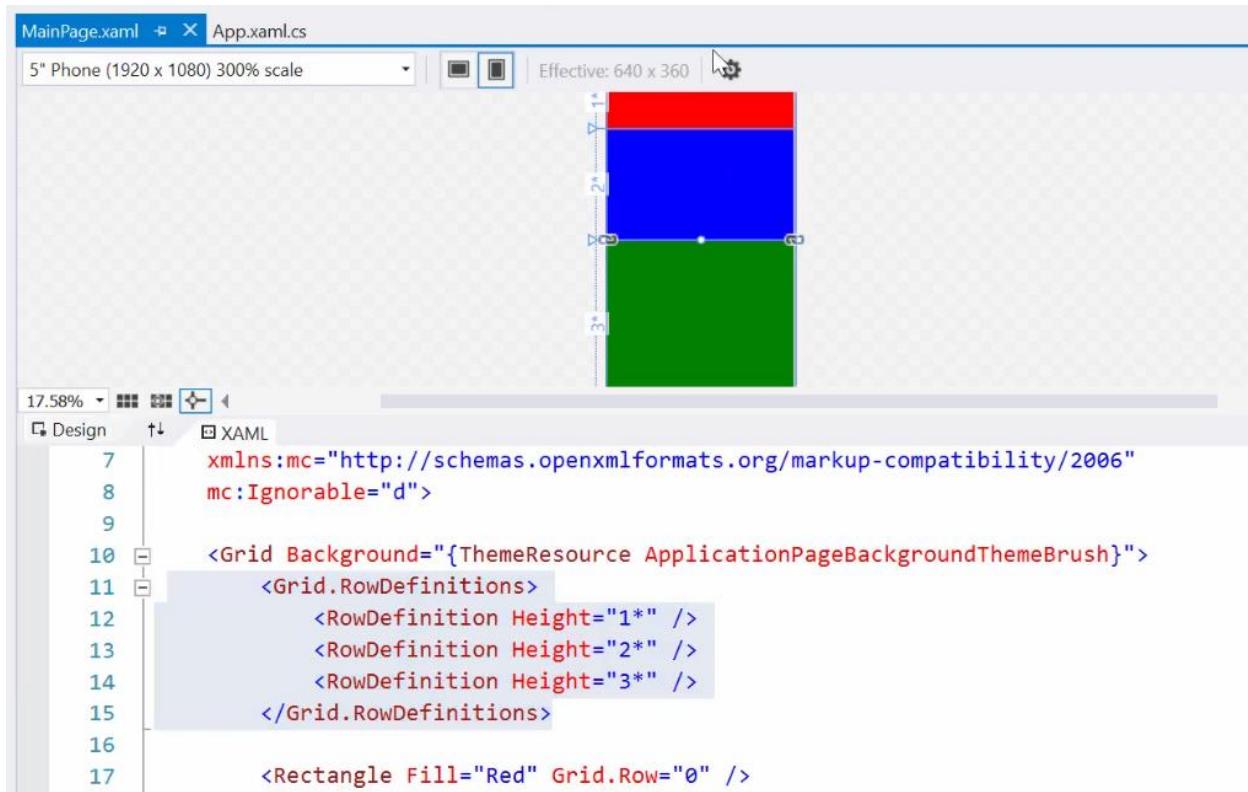
The reason why Attached Properties exist is an advanced XAML topic that is not actionable at this point in your introduction to XAML. If you want to get deeper into the internals of XAML, then you should search MSDN for articles about both "Attached Properties", as well as the loosely related topic of "Dependency Properties". In a nutshell, Attached Properties keep your XAML simple.

Third, notice in this example that there are the two different row heights. The first height, we set to Auto and the second row height we set to star (\*). There are three syntaxes that you can use to help persuade the sizing for each row and each column. I use the term "persuade" intentionally. With XAML layout, heights and widths are relative and can be influenced by a number of different factors. All these factors are considered by the layout engine at run time to determine the actual placement of items on your given page, or your screen.

So for example, the term "Auto" means that the height for the row should be tall enough to accommodate all of the controls that are placed inside of that row. If the tallest control (in this case, you can see the Rectangle has its height explicitly set to 100 pixels) is 100 pixels tall, then that's the actual height of the row, 100 pixels. If we were to change the height of the Rectangle to 50 pixels, you can see that the height of the Row changes now to be 50 as well. "Auto" means that the height is relative to the controls that are inside of that given row or column, or whatever the case might be.

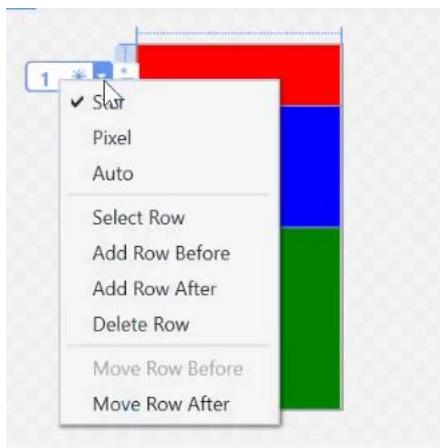
Secondly, the asterisk character is also known as "star sizing", and it means that the height of the row should take up all of the rest of the available height available.

I created a separate project called StarSizing that has three Rows defined in the Grid.



Notice the heights of each one of them. By adding a number before the asterisk I am saying of all the available space, give me one “share” of all the available space, or two or three “shares” of all the available space. The sum of all of those rows adds up to six. So each “one star” is equivalent to 1/6th of the height that is currently available. Therefore, three star would get half of the height that is available as depicted in the output of this example.

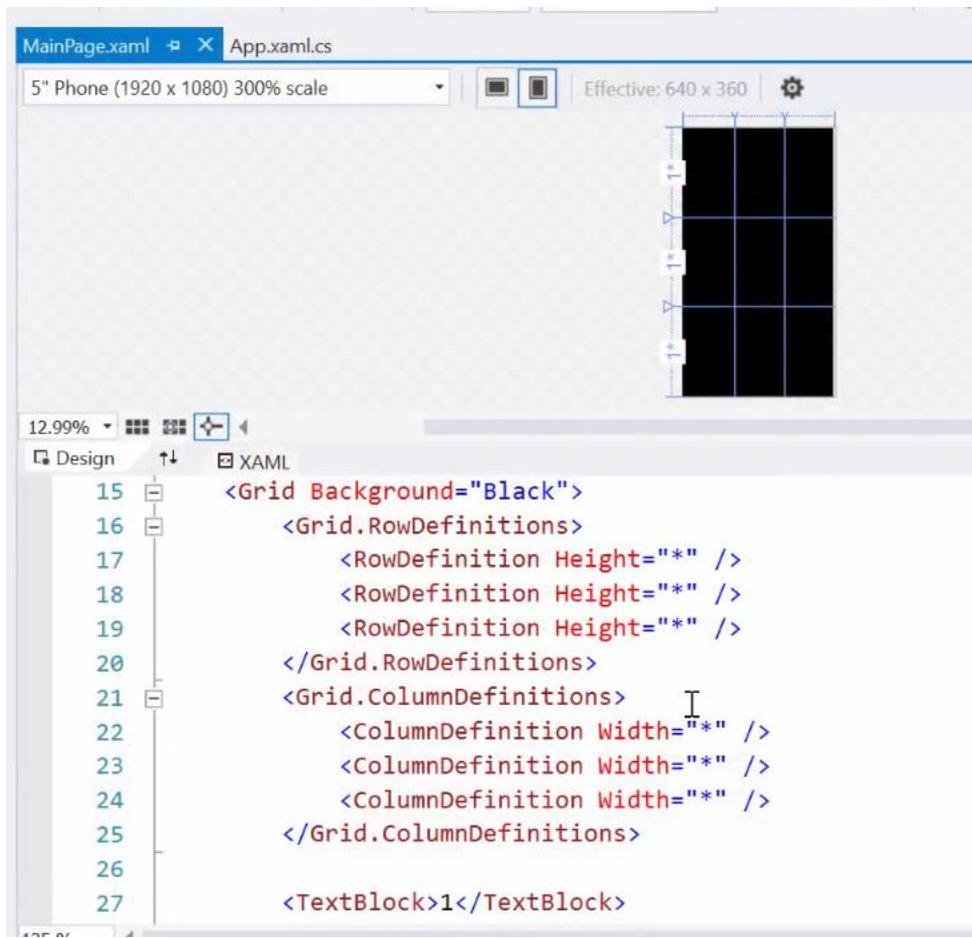
Also notice off to the left-hand side of the visual designer’s depiction of the Grid that there are some visual tools that we can use to change the sizing. For example, I can change from star sizing to auto, or to pixel. I can also just type in the given value here, and that would set the Height property.



Besides auto and star sizing, you can also specify widths and heights, as well as margins in terms of pixels. So in fact, when only numbers are present, it represents that number of pixels for the width or the height. Generally, it is **not** a good idea to use exact pixels in layouts for widths and heights because of the likelihood that various screens will be larger or smaller, so there are several different types of phones or several different form factors for tablets and desktops. You do not want to specify exact numbers or else it is not going to look correct on a different form factor. Instead, it is preferable to use relative layouts like auto and star sizing for layout.

Notice that the Widths and Heights, are assumed to be 100% unless otherwise specified, especially for rectangles. And while that's generally true for many XAML controls, other controls like the Button are not treated this way. Their size is based on the content inside of them.

A Grid can have a collection of ColumnDefinitions. In another example named "GridsRowsAndColumns" you can see that I created a 3 x 3 grid, three RowDefinitions and a ColumnDefinitions collection that contains three ColumnDefinitions.



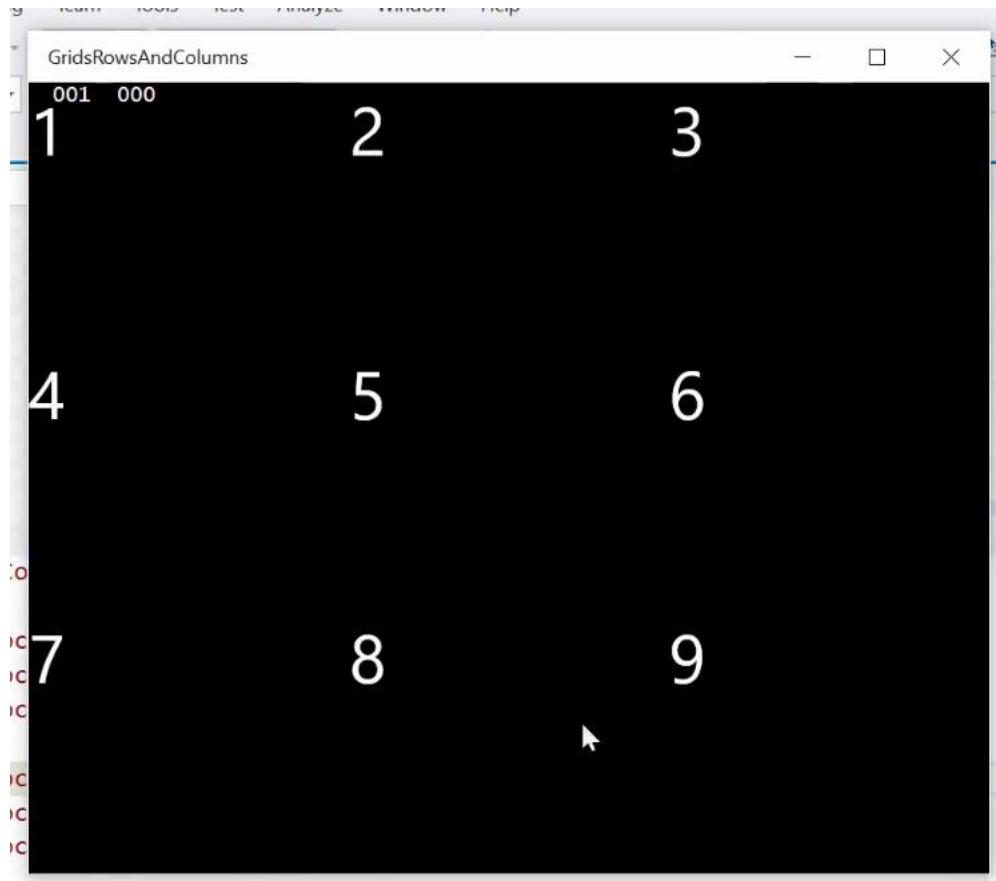
Furthermore, I added a TextBlock inside of each of the cells.

```

26
27     <TextBlock>1</TextBlock>
28     <TextBlock Grid.Column="1">2</TextBlock>
29     <TextBlock Grid.Column="2">3</TextBlock>
30
31     <TextBlock Grid.Row="1">4</TextBlock>
32     <TextBlock Grid.Row="1" Grid.Column="1">5</TextBlock>
33     <TextBlock Grid.Row="1" Grid.Column="2">6</TextBlock>
34
35     <TextBlock Grid.Row="2">7</TextBlock>
36     <TextBlock Grid.Row="2" Grid.Column="1">8</TextBlock>
37     <TextBlock Grid.Row="2" Grid.Column="2">9</TextBlock>

```

Now unfortunately, you the designer is not displaying them correctly, but if we were to run the application, you would be able to see that we get a different number in each cell.

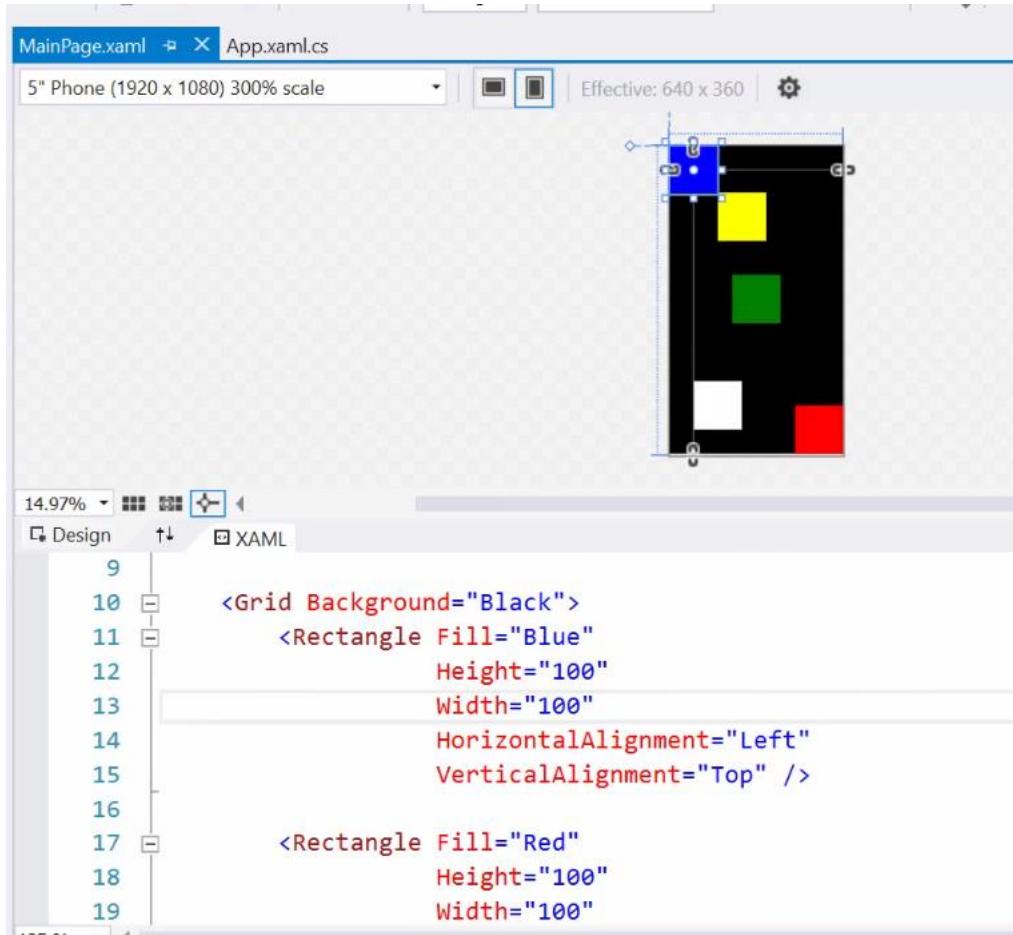


In this very first cell in the upper left-hand corner I am not setting a row, nor am I setting a column. By default, if you do not supply that information, it is assumed to be zero. So we're assuming that we're putting a TextBlock containing "1" in row zero, column zero.

Furthermore, if you take a look at the next TextBlock containing "2", I am setting the grid column equal to one, but I'm not setting the row, meaning that I am assuming that to be zero.

Relying on defaults keeps your code, again, more concise, but you have to understand that there is a convention being used.

I have yet another example called AlignmentAndMargins.



This example illustrates how `VerticalAlignment` and `HorizontalAlignment` work even in a given grid cell. And this will hold true in a `StackPanel` as well as we talk about it in the next lesson.

The `VerticalAlignment` or `HorizontalAlignment` property attributes pull controls towards their boundaries. By contrast, the `margin` attributes push controls away from their boundaries.

In this example, you can see that the `HorizontalAlignment` is pulling this blue rectangle towards the left-hand side, and the `VerticalAlignment` is pulling it towards the top side.

Next, look at the White Rectangle. The `HorizontalAlignment` is pulling it towards the left, and the `VerticalAlignment` is pulling it towards the bottom. But then I'm setting the `Margins` equal to `50, 0, 0` and `50`. As a result you can see that the margin will now push the rectangle away from the left-hand boundary by 50 pixels and away from the bottom boundary by 50 pixels as well.

Also notice the odd way in which margins are defined. Margins are represented as a series of numeric values that are separated by commas. This convention was borrowed from Cascading Style Sheets. So the numbers represent the margin pixel values in a clockwise fashion, starting at the left-hand side.

A bit earlier, I said that it is generally a better idea to use relative sizes like auto or star sizing whenever you want to define heights and widths. So why is it then that margins are defined in exact pixels? Usually margins are just small values to provide spacing or padding between two relative values and so they can be a fixed size without negatively impacting the overall layout of the page. If you want a small amount of spacing between two rectangles 50 pixels should suffice whether you have a large or a smaller size. And if it is not, then you can change it through other techniques that I'll demonstrate in this series.

To recap, in this lesson, we talked about layout controls and how they allow you to define areas of your application where other visual XAML controls will be hosted. In this lesson, we specifically learned about the Grid and how to define columns and rows, how to define their relative sizes using star and auto, and then how to specify which row and column a given control would request to be inside of by setting Attached Properties (I.e., Grid.Row, Grid.Column) on that given XAML Control.

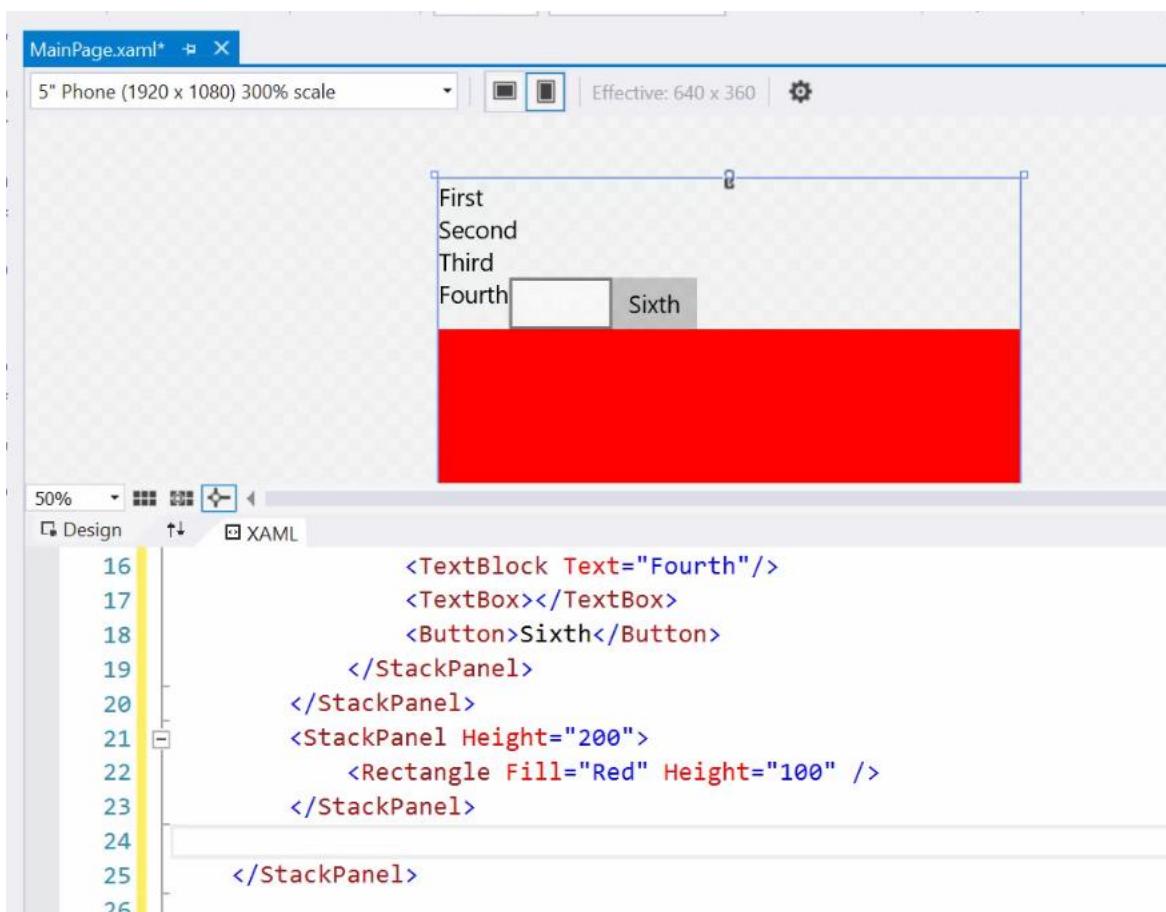
We also talked about how to set the alignment and the margins of those controls inside of a given cell and more.

## UWP-009 - XAML Layout with StackPanel

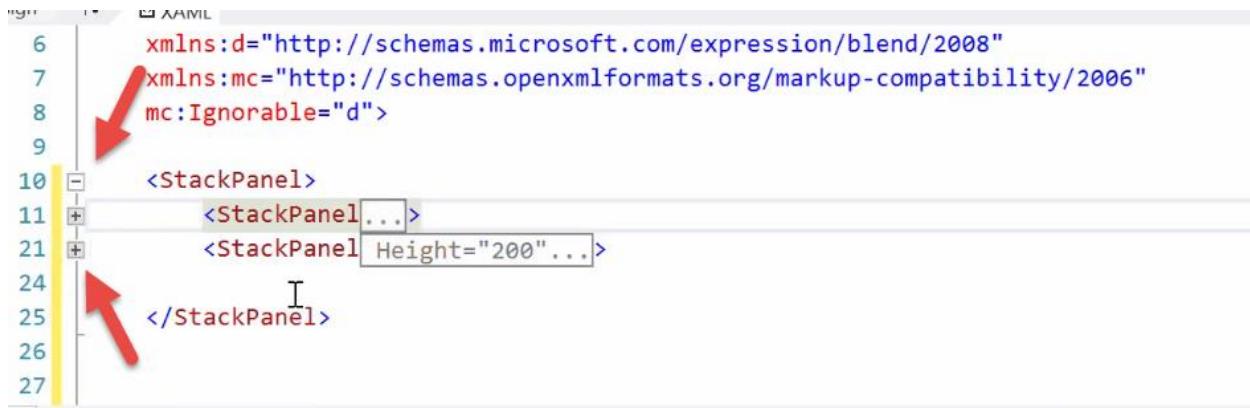
The StackPanel layout control allows you to arrange your XAML Controls in a flow from top to bottom by default. Or we can change the Orientation property to flow from left to right in a horizontal fashion. We can even change the flow to go from left-to-right to right-to-left.

Creating a StackPanel is simple. It's simply a panel (almost like a div tag, if you're familiar with HTML development), and inside of the opening and closing tags you can add XAML controls which will be stacked on top of each other or side by side.

I created a very simple application named SimpleStackPanel.



As a side note: notice that I use a little plus and minus symbols in the left-most column to roll up my code so we can see the overall structure of this application.



```
6      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8      mc:Ignorable="d">
9
10     <StackPanel>
11         <StackPanel ...>
12             <StackPanel Height="200" ...>
13                 ...
14             ...
15         ...
16     ...
17
18
19
20
21
22
23
24     ...
25
26
27
```

And you might be wondering, well why do we have StackPanels inside of StackPanels? Why can't we just do something like this?



```
Design XAML
6      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8      mc:Ignorable="d">
9
10
11     <StackPanel ...>
12         <TextBlock ...>
13             ...
14         ...
15     ...
16
17
18
19
20
21
22
23
24
25
26
27
```

If I remove the outer-most StackPanel I get the blue squiggly line with the familiar exception that the property "Content" is set more than once. A Page only has a Content property and it does not have a Children property collection. So that's why we need an outermost StackPanel and then inside of that we can put as many StackPanels or other XAML controls as needed.

Inside of the first StackPanel, notice that I have three TextBlocks that are stacked on top of each other.

```
Design XAML
10 <StackPanel>
11   <StackPanel>
12     <TextBlock>First</TextBlock>
13     <TextBlock>Second</TextBlock>
14     <TextBlock>Third</TextBlock>
15     <StackPanel Orientation="Horizontal">
16       <TextBlock Text="Fourth"/>
17       <TextBox></TextBox>
18       <Button>Sixth</Button>
19     </StackPanel>
20   </StackPanel>
```

Notice they take up the full width because we haven't specified a width and by default, and that they're arranged in a vertical fashion. The item at the top will be the first item stacked, the item at the bottom will be the last item stacked. So the stacking is performed in order.

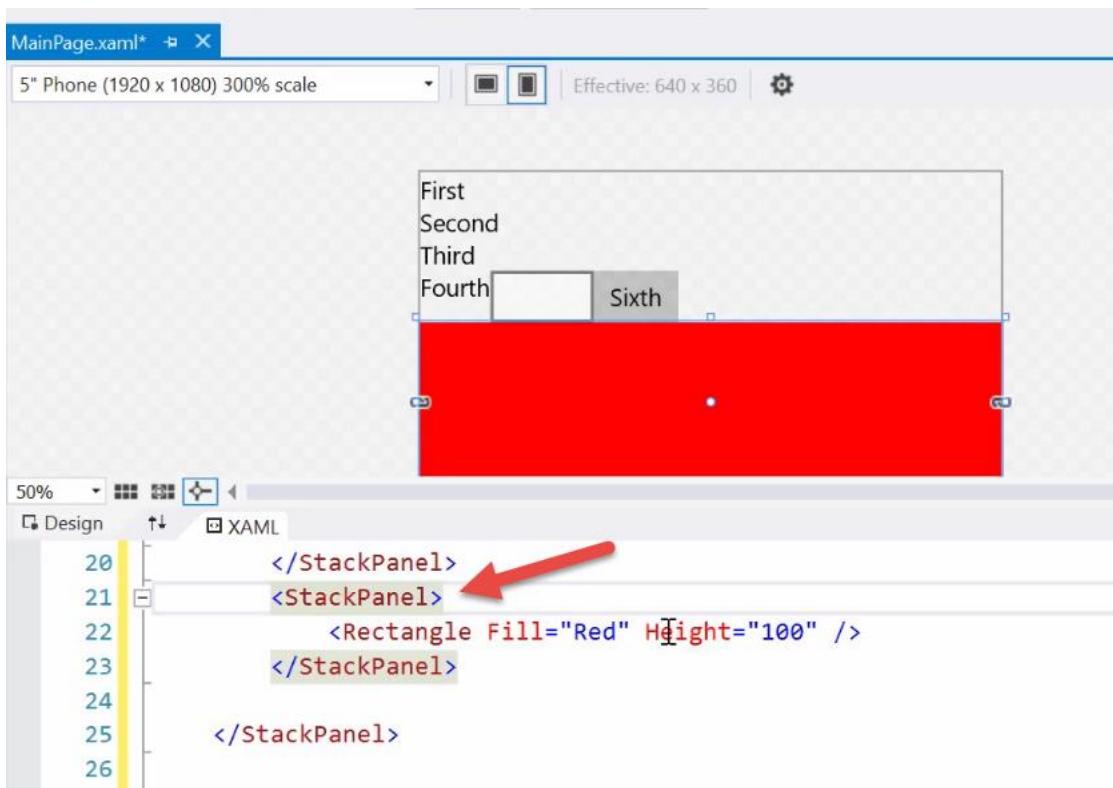
Furthermore, underneath the third TextBlock, I've added a second StackPanel. I set its Orientation to Horizontal. I'm using the StackPanel here as another "row" that's stacked beneath the TextBlocks. Inside of the inner-most StackPanel, I stacked several XAML Controls in a horizontal fashion. The leftmost stacked item is a TextBlock that has the word "Fourth" in it, the next item that is stacked next to it is a TextBox, then the next item is a button with the content, "Sixth", in it.

We can achieve almost anything by simply using this technique of stacking StackPanels inside of StackPanels. I tend to use StackPanel actually more than I use the Grid for layout. Personally, I think that it allows me to get the design that I want and it gives me the flow that I want regardless of the size of the device that the app is actually running on.

You can also see that I've got a StackPanel defined beneath it with a simple rectangle.

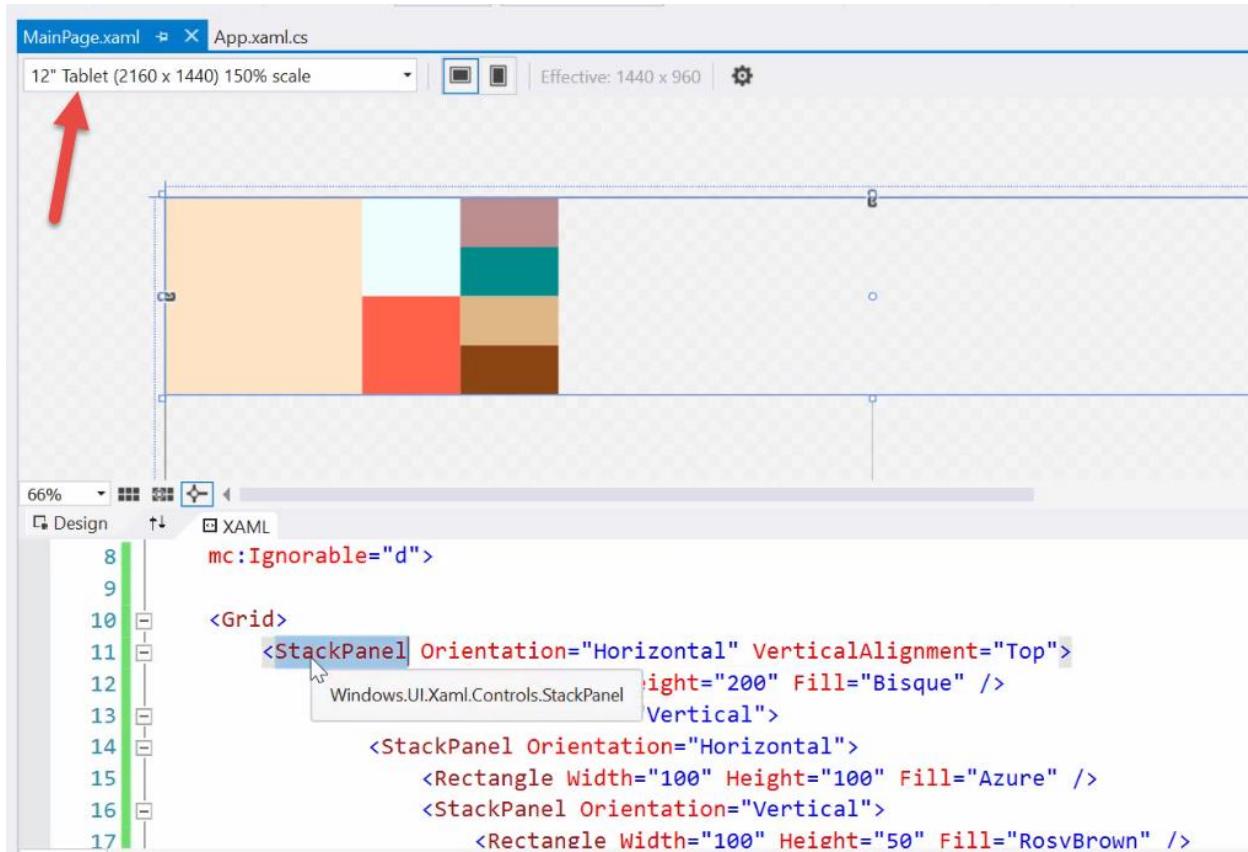
```
Design XAML
20 </StackPanel>
21 <StackPanel Height="200">
22   <Rectangle Fill="Red" Height="100" />
23 </StackPanel>
24
25 </StackPanel>
26
```

The only thing I want to illustrate here is that the height of the StackPanel can be set, and yet items inside of the StackPanel can have their own height independent of the parent. If I were to remove the height, notice what happens:



Essentially, the Height of the StackPanel is set to Auto. If we were try to set it to star (\*), it wouldn't work but we can set it to a numerical pixel value or we can just leave it at the default, which is Auto.

I created another example called ComplexStackPanel. Note that I have changed from the default viewer, that would be a five-inch phone, to a 12" Tablet to accommodate the larger width of this design that I created.



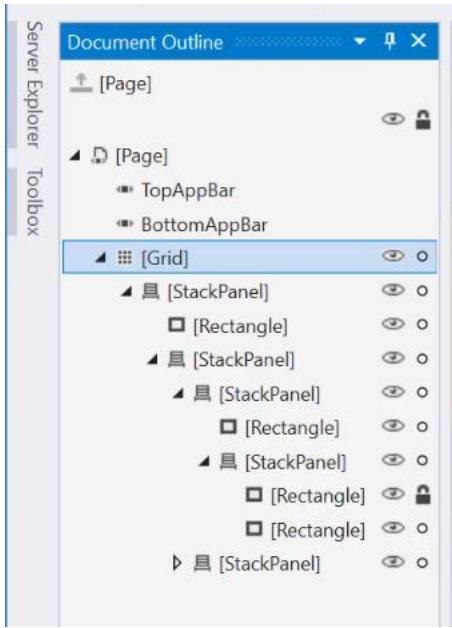
This example takes the concept of adding StackPanels inside of StackPanels to an extreme. Admittedly, it can become confusing to read the code and understand what code is producing a given StackPanel, however, hopefully, between the color descriptions and holding your mouse cursor on it and seeing the little boundary box that is selected you can make heads or tails of how this works.

The topmost StackPanel defines a “row” of sorts. The height of the top-most StackPanel is actually dictated by the items that are inside of it ... specifically rectangle with the Fill="Bisque". Its height is set to 200 so that sets the height for the entire StackPanel since nothing else inside of it has a greater height.

Also, you'll notice that I set the VerticalAlignment of this top-most StackPanel, and the reason I did that, because if you change that, it will now get moved to the vertical middle of the grid cell, and that's just the difference in how grids work and StackPanels work.

So the result of StackPanels inside of StackPanels produces the desired intricacy until I have the series of rectangles that resemble something like an ornate pane of Frank Lloyd Wright designed glass.

One of the things that can help you out whenever you are working through an intricate amount of XAML and it's difficult to find your way around is this little Document Outline. If you do not see it by default on this left-most next to the Toolbox, you can go to the View menu > Other Windows > Document Outline.



You can expand each node in the Document Outline to reveal the hierarchy of items in the Designer. This is a representation of a “visual tree”. By selecting a given leaf in the tree you can see the little boundary box selection around the associated XAML Control in the Designer.

Furthermore, the Document Outline is very convenient whenever you want to make changes in the Properties window and it's difficult to find the exact item that you are looking for just by clicking around the Designer and the XAML is too intricate.

The Document Outline also can be used to hide items by clicking the left-column on the right-hand side to remove certain items from view. We can also lock items which means that they can't be selected and therefore they cannot easily be changed in the Properties window. You can see that it adds this `IsLocked="True"` XAML to the design time experience.

The final example project named GridAndStackPanel demonstrates a “gotcha” when using StackPanels. Furthermore it should help us to have a better understanding of the difference between Grids and StackPanels, and something that you need to watch out for and completely understand.

```
9
10    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11        <StackPanel ...>
12            <StackPanel Height="200" ...>
13                ...
14            </StackPanel>
15        </StackPanel>
16    </Grid>
17</Page>
```

This is almost identical to the first example that we used earlier in this lesson. This time, we have a Grid that surrounds two StackPanels. The top-most StackPanel contains three TextBlocks, first, second and third, and then another embedded StackPanel whose Orientation property is set to Horizontal. This time I just have three TextBlocks that will be stacked horizontally, so we got fourth, fifth and sixth.

```

11   <StackPanel>
12     <TextBlock>First</TextBlock>
13     <TextBlock>Second</TextBlock>
14     <TextBlock>Third</TextBlock>
15     <StackPanel Orientation="Horizontal">
16       <TextBlock>Fourth</TextBlock>
17       <TextBlock>Fifth</TextBlock>
18       <TextBlock>Sixth</TextBlock>

```

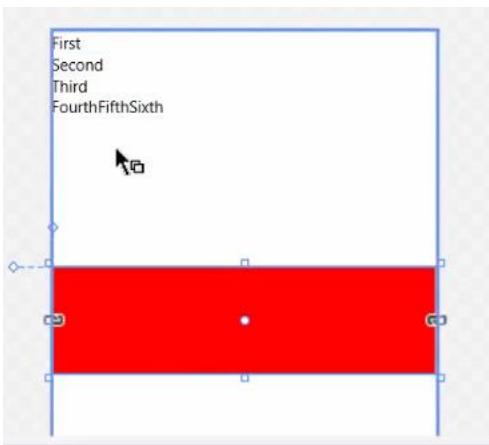
Below that, we have another StackPanel with a rectangle inside of it.

```

20   </StackPanel>
21   <StackPanel Height="200">
22     <Rectangle Fill="Red" Height="100" />
23   </StackPanel>
24 </Grid>

```

Now in this case, you'll look at it and say, why is there so much space in between this Rectangle and the bottom of this StackPanel?



And you might attempt to resolve this by setting the `VerticalAlignment="Top"`. It would appear as though you lost the top-most StackPanel. You didn't lose it, actually. It is sitting behind the second StackPanel (the one with the red Rectangle).

There are three things going on here at the same time, and this will hopefully help you to understand the difference between Grids and StackPanels.

First, some controls like image and rectangle controls that set themselves inside of a grid cell will be set to 100 width and 100 height by default. So that is also true of StackPanels. When you put a StackPanel inside of a grid cell, in this case the default grid cell, cell zero or row zero, column zero, then the two StackPanels are essentially, well, the first StackPanel has set itself to 100 height and 100 width.

Now the second StackPanel has set itself to 200 height, but the second thing that you need to understand is that by default the content in a cell of a grid will be vertically and horizontally centered, so

we have the second StackPanel, and let's remove the VerticalAlignment. The default here would be center. And so you can see that it is actually sitting, the entire StackPanel is in the center of that cell.

Third, you can easily overlap items in a grid cell. So if two or more controls are set to reside in the exact same grid cell, and their HorizontalAlignment equals top and their VerticalAlignment equals top, then they will literally sit on top of each other. We're actually looking at two StackPanels. This first StackPanel takes up the entire length, and the items inside of it, however, are aligned to the top of that StackPanel, but the StackPanel itself takes up the whole frame. And then the second StackPanel is set into the middle of that cell, but it's only 200 tall. However, whenever we change the VerticalAlignment and now we move that up to the very top of that single cell inside the grid, that's when we overlapped.

So I just want you to understand that about grids and about StackPanels, that StackPanels will never ever allow you to put two of them on top of each other where you cannot see the one that is underneath it. Whereas grid cells will absolutely allow you to do that.

There are a couple of different ways that we could rectify this. The easiest one is just to use StackPanels as the outermost container here in this particular layout. However, you could also create two rows instead of just one row and put this first StackPanel in the top row, put the second StackPanel in the bottom row, and That will ensure that they don't overlap. You could also set the StackPanels, you can set its VerticalAlignment to the top but then use a margin to push it down. That seems a little more fragile, I don't know if I like that idea.

There are a couple of different ways to still make this work even though you are working with a grid. But ideally, you'll probably switch to just using StackPanels. In fact, like I said earlier, I typically use the StackPanel just about for everything. I don't use Grids as much as I used to, except to give the page an overall structure. And with everything else, I try to use StackPanels.

There is this special technique, however, that I learned when we talk about adaptive triggers and adaptive layout that utilizes Grids to adapt from a desktop to a mobile layout, that wouldn't be possible with a StackPanel. The Grid still does have its uses. I just prefer, in most cases, to stick with the StackPanel.

## [UWP-010 - Cheat Sheet Review: XAML and Layout Controls](#)

Let you in on a little secret, nobody memorizes all this stuff, at least not at first. It would probably take you several months of working and building applications in order to really internalize a lot of this. IntelliSense is awesome, and it really helps you through a lot of the rough spots. Little code snippets online are great, I use cheat sheets, which allow me to, they're just basically notes that I've gathered from watching a video series, or from reading articles online. It helps me to organize the information, see it visually, and then organize it in my mind as well. We will use this as a form of, of review throughout this series of lessons.

This is what I promote on LearnVisualStudio.NET, and it's a great way to index in to the content to remind yourself of what the major ideas where, the crux of those videos. You can always remove the content you don't need and you can add content that you might think might be helpful in the future to customize this resource for how you will use it.

Note: In the video I narrate through the creation of the cheat sheet, however in this PDF version I will simply paste in the cheat sheet review content for the sake of brevity.

### [UWP-04 - What is XAML?](#)

XAML - XML Syntax, create instances of Classes that define the UI.

### [UWP-05 - Understanding Type Converters](#)

Type Converters - Convert literal strings in XAML into enumerations, instances of classes, etc.

### [UWP-06 - Understanding Default Properties, Complex Properties and the Property Element Syntax](#)

Default Property ... Ex. sets Content property:

```
<Button>Click Me</Button>
```

Complex Properties - Break out a property into its own element syntax:

```
<Button Name="ClickMeButton"
       HorizontalAlignment="Left"
       Content="Click Me"
       Margin="20,20,0,0"
       VerticalAlignment="Top"
       Click="ClickMeButton_Click"
       Width="200"
       Height="100">
```

```
<Button.Background>
  <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="Black" Offset="0"/>
    <GradientStop Color="Red" Offset="1"/>
  </LinearGradientBrush>
</Button.Background>
</Button>
```

### UWP-07 - Understanding XAML Schemas and Namespace Declarations

Don't touch the schema stuff - it's necessary!

Schemas define rules for XAML, for UWP, for designer support, etc.

Namespaces tell XAML parser where to find the definition / rules for a given element in the XAML.

### UWP-08 - XAML Layout with Grids

Layout controls don't have a content property ... they have a Children property of type UIElementCollection.

By embedding any control inside of a layout control, you are implicitly calling the Add method of the Children collection property.

```
<Grid Background="Black">
  <Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
</Grid>
```

Sizes expressed in terms of:

Explicit pixels - 100

Auto - use the largest value of elements it contains to define the width / height

\* (Star Sizing) - Utilize all the available space

1\* - Of all available space, give me 1 "share"

2\* - Of all available space, give me 2 "shares"

3\* - Of all available space, give me 3 "shares"

6 total shares ... 3\* would be 50% of the available width / height.

Elements put themselves into rows and columns using attached property syntax:

...

...

```
<Button Grid.Row="0" />  
</Grid>
```

- When referencing Rows and Columns ... 0-based.
- There's always one default implicit cell: Row 0, Column 0
- If not specified, element will be in the default cell

#### UWP-09 - XAML Layout with StackPanel

```
<StackPanel>  
  <TextBlock>Top</TextBlock>  
  <TextBlock>Bottom</TextBlock>  
</StackPanel>
```

- Vertical Orientation by default.
- Left-to-right flow by default when Horizontal orientation.
- Most layouts will combine multiple layout controls.
- Grid will overlap controls. StackPanel will stack them.

## UWP-011 - Laudable Layout Challenge

The challenges I created on LearnVisualStudio.NET are easily the most popular feature of the site. The challenges force you to learn by doing, to exercise what you just learned in the previous lessons. This will force you to struggle a bit with these new concepts. And struggle is good. It's how you learn. No struggle, no learning.

It might take you 30 minutes, an hour. You might have to go back and review some lessons or reference MSDN for some help but I think you can do this based on what we've already learned so far. There's actually three challenges in a row and they're all related to the topic of layout what we've already learned.

The first challenge I'm calling the "Laudable Layout Challenge". If you can see that we're building a simple app -- only the user interface. So there's no real functionality in this app whatsoever.

First of all, this example requires a single grid only. That's part of the requirements we'll talk about them in a moment.



And you can see that we're just capturing in TextBoxes the first name, last name, and the email address for a sales prospect and then we have a save button and again, it doesn't do anything at all.

The first column holds the TextBlocks, the second column holds the TextBoxes and Button, the third column is used for padding. You can also see there's a number of rows but I'm going to let you figure out how many to use.

See the document in the zipped folder associated with this lesson called UWP-11-Instructions.txt.

First of all, you can only use a grid control.

Secondly, this is more of a tip, whenever I was designing the application, I used the designer of a five-inch phone 1920 by 1080 300% scale.

The large TextBlock at the top is 48 points.

Most of the margins that you see here on the left and here between this top TextBlock and all the rest of the TextBoxes are either 10 pixels or 20 pixels ... again, I'll let you figure that out.

And then the TextBlocks for the first name, last name and email address should be centered vertically in the row. Presumably, this is a row here at the very top and you can see that the TextBox is butted up at what appears to be the top of the row and then there's some spacing below it and if you were to use your mouse cursor and trace along, you can see that the TextBlock in the left-most column is centered between the top line of the TextBox and the start of the next TextBox in the next row.

The tricky part: requirement number six. You're going to need to figure out how to allow your TextBlock to span multiple columns. When you have a need to tell the grid how to do something, what do you use? Use an attached property, so you should use IntelliSense to figure out the correct attached property to make your TextBlocks span multiple columns.

If you ever get stuck, you can take a look at a screenshot that I provide as guidance: UWP-011-Screenshot.png.

You should attempt to solve this without my help whatsoever. If you get stuck, I don't want you to struggle too much and get frustrated. Only review enough of the next lesson to get unstuck and then continue to move forward on your own to the extent that it's possible.

And then after you finish and successfully complete the challenge, you will not only feel victory but you will also then want to review the solution lesson in total, so that you can compare the way that you approached this challenge versus the way that I approached it. Maybe you have a more elegant way of approaching something than I did. Maybe I use a different technique than you did so you can compare and contrast and hopefully by doing that, learn a little bit more.

Don't cheat, don't go on directly to the next lesson. Make sure you struggle by attempting to solve this challenge first. Spend a half hour, an hour maybe, if it takes that. Re-read the lesson, or re-watch the videos if you have to.

## UWP-012 - Laudable Layout Challenge: Solution

If you're reading this immediately after reading the previous lesson and you've not cracked open Visual Studio and written a single line of code yet, then stop what you're doing. Shame on you! Open up Visual Studio, do this for yourself. Nobody learns by just reading books or watching videos. You must get your hands "dirty in the code". That's how you learn. So, stop reading this lesson. Go off and attempt to solve the challenge. If you get stuck, come back here and read just enough of this lesson to get unstuck.

Step 1: Create a new "Blank App (Universal Windows)" project and name it LaudableLayout.

Step 2: Add RowDefinitions and ColumnDefinitions into the default Grid:

```
11 <Grid>
12     <Grid.RowDefinitions>
13         <RowDefinition Height="Auto" />
14         <RowDefinition Height="Auto" />
15         <RowDefinition Height="Auto" />
16         <RowDefinition Height="Auto" />
17         <RowDefinition Height="Auto" />
18         <RowDefinition Height="*" />
19     </Grid.RowDefinitions>
20     <Grid.ColumnDefinitions>
21         <ColumnDefinition Width="2*" />
22         <ColumnDefinition Width="3*" />
23         <ColumnDefinition Width="1*" />
24     </Grid.ColumnDefinitions>
25 </Grid>
26 </Page>
```

Step 3: Add the TextBlocks, TextBoxes and Button to the Grid's Children collection, assigning each one to the appropriate Row or Column:

```

<TextBlock Text="ACME Sales Corp" FontSize="48" Grid.ColumnSpan="3" />
    <TextBlock Grid.Row="1" Text="First Name: " />
    <TextBox Grid.Row="1" Grid.Column="1" />

    <TextBlock Grid.Row="2" Text="Last Name: " />
    <TextBox Grid.Row="2" Grid.Column="1" />

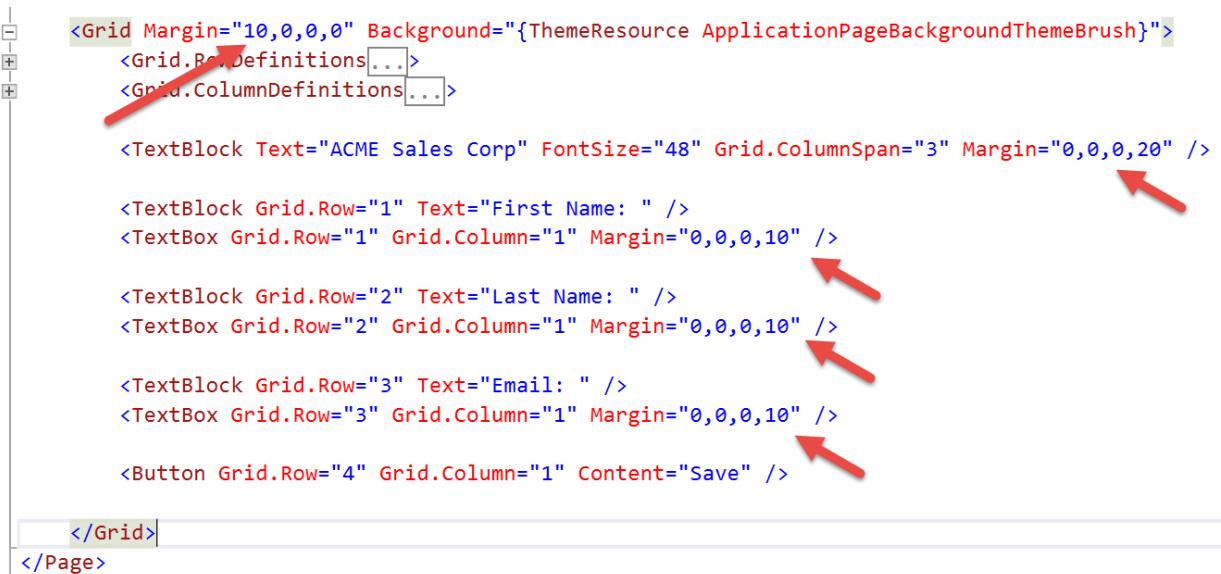
    <TextBlock Grid.Row="3" Text="Email: " />
    <TextBox Grid.Row="3" Grid.Column="1" />

    <Button Grid.Row="4" Grid.Column="1" Content="Save" />
</Grid>
</Page>

```

Notice the Grid.ColumnSpan Attached Property applied to the top-most TextBlock.

#### Step 4: Add Margins to the Grid and the XAML Controls



The screenshot shows the XAML code in the editor with several red arrows pointing to specific margin settings. One arrow points to the `Margin="10,0,0,0"` attribute on the `<Grid>` element. Another arrow points to the `Margin="0,0,0,20"` attribute on the first `<TextBlock>` element. Three more arrows point to the `Margin="0,0,0,10"` attribute on each of the three `<TextBlock>` elements in rows 2, 3, and 4. The XAML code is as follows:

```

<Grid Margin="10,0,0,0" Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>...</Grid.RowDefinitions>
    <Grid.ColumnDefinitions>...</Grid.ColumnDefinitions>

    <TextBlock Text="ACME Sales Corp" FontSize="48" Grid.ColumnSpan="3" Margin="0,0,0,20" />
        <TextBlock Grid.Row="1" Text="First Name: " />
        <TextBox Grid.Row="1" Grid.Column="1" Margin="0,0,0,10" />

        <TextBlock Grid.Row="2" Text="Last Name: " />
        <TextBox Grid.Row="2" Grid.Column="1" Margin="0,0,0,10" />

        <TextBlock Grid.Row="3" Text="Email: " />
        <TextBox Grid.Row="3" Grid.Column="1" Margin="0,0,0,10" />

        <Button Grid.Row="4" Grid.Column="1" Content="Save" />
    </Grid>
</Page>

```

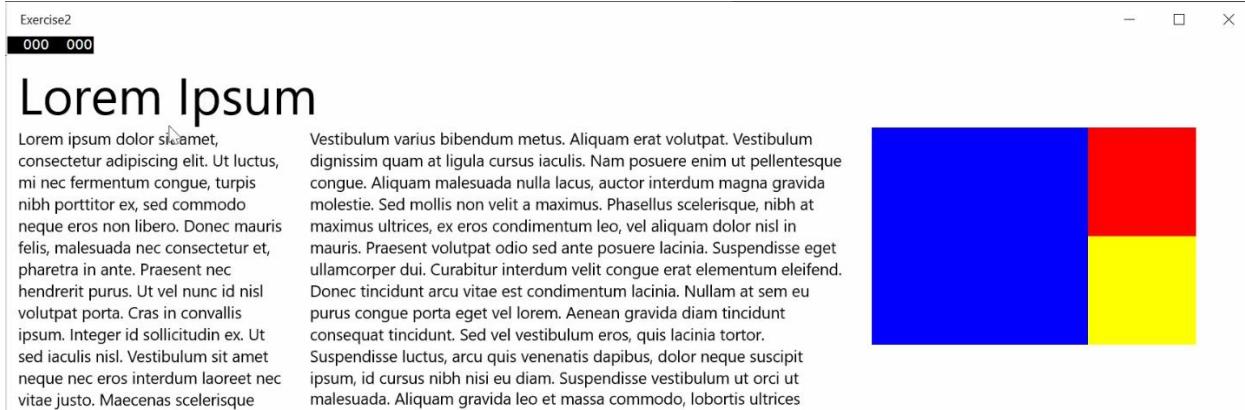
#### Step 5: Align TextBlocks vertically to the Center.

```
<TextBlock Text="ACME Sales Corp" FontSize="48" Grid.ColumnSpan="3" Margin="0,0,0,20" />  
<TextBlock Grid.Row="1" Text="First Name: " VerticalAlignment="Center" /> ←  
<TextBox Grid.Row="1" Grid.Column="1" Margin="0,0,0,10" />  
  
<TextBlock Grid.Row="2" Text="Last Name: " VerticalAlignment="Center" /> ←  
<TextBox Grid.Row="2" Grid.Column="1" Margin="0,0,0,10" />  
  
<TextBlock Grid.Row="3" Text="Email: " VerticalAlignment="Center" /> ←  
<TextBox Grid.Row="3" Grid.Column="1" Margin="0,0,0,10" />  
  
<Button Grid.Row="4" Grid.Column="1" Content="Save" />  
  
</Grid>  
</Page>
```

Please note: If this is something you struggled with then maybe you should come back tomorrow and try it again just to cement these ideas in your mind.

UWP-013 - Legendary Layout Challenge

The second challenge is called the "Legendary Layout Challenge". In this layout challenge, you can only use StackPanel. Here's what you're going to build:



There's a large title here at the top with three columns below. The first column is about half the size of this second column. Both of these columns contain "Lorem Ipsum" text, which is just dummy text, fake Latin text, has no meaning. (If it does, I don't know what it means.)

In the third column, there are a series of squares that almost resemble a flag. Keep in mind that you can only use StackPanels.

The instruction file is available in the zipped folder of resources for this lesson named UWP-013-Instructions.txt.

The first requirement is that only StackPanel controls can be used – no Grid controls.

The designer I set to 13.3 Desktop (1280 x 720) 100% Scale. That might help you as you're building this. You're going to need a little bit more horizontal space to work with.

The TextBlock at the very top is 48 pixels. The TextBlock in the first column will be 250 pixels wide. The TextBlock in the second column will be 500 pixels wide. The third column is 300 pixels wide. The third area looks like it's split 200 pixels for the large blue box and then 100 pixels for the red and the yellow boxes.

Side note: yes, I know that I said don't use absolute pixel values, rather use relatives whenever you're working with Universal Windows Platform apps. However, this is just an exercise just to flex your layout muscles, so don't use this in the real world.

You should get the text for these Lorem Ipsum columns from this file that I'm supplying called loremipsum.txt. Just copy and paste what you need in there. The first TextBlock only needs two paragraphs. The second TextBlock requires several paragraphs from that file.

Margins are either 10 or 20 pixels.

Important: Use the TextBlock's TextWrapping property to allow the Lorem Ipsum text to wrap properly. If you don't set the TextWrapping property equal to Wrap then you're only going to see the first line of the Lorem Ipsum in each of your TextBlocks.

You can use the image named UWP-013-Screenshot.png to reference while you're building the application.

And just like I said before, try to go through this challenge without my help, without peeking at the solution lesson. Struggle a little bit. Don't get frustrated. If you get to the point where you don't know what you need to do next then review just enough of the solution lesson to get unstuck and then finally, if you do solve it without my help or once you solve this on your own and you're curious as to how I would solve it, if I took different techniques or whatever, then absolutely read the entire solution lesson, maybe there are something different about how I did it from how you did it.

## UWP-014 - Legendary Layout Challenge: Solution

Step 1: Create a new “Blank App (Universal Windows) project called “LegendaryLayout”.

Step 2: Create the overall layout using StackPanels. Set the Width and TextWrapping properties of the inner TextBlocks, Set the FontSize of the outer TextBlock.

```
<StackPanel>
    <TextBlock Text="Lorem Ipsum" FontSize="48" />
    <StackPanel Orientation="Horizontal">
        <TextBlock Width="250" TextWrapping="Wrap">
            Lorem ipsum dolor sit amet, consectetur adipiscing elit.
            Vestibulum sit amet neque nec eros interdum laoreet nec \
        </TextBlock>
        <TextBlock Width="500" TextWrapping="Wrap">
            Vestibulum varius bibendum metus. Aliquam erat volutpat.
            Praesent volutpat odio sed ante posuere lacinia. Suspendi
            Nunc a vehicula velit. Vivamus rutrum elementum massa in
        </TextBlock>
    <StackPanel Orientation="Horizontal">
```

Step 3: Add the Lorem Ipsum to columns 1 and 2. Copy and paste the Lorem Ipsum from the provided file into the empty TextBlocks.

```
<StackPanel>
    <TextBlock Text="Lorem Ipsum" FontSize="48" />
    <StackPanel Orientation="Horizontal">
        <TextBlock Width="250" TextWrapping="Wrap">
            Lorem ipsum dolor sit amet, consectetur adipiscing elit.
            Vestibulum sit amet neque nec eros interdum laoreet nec \
        </TextBlock>
        <TextBlock Width="500" TextWrapping="Wrap">
            Vestibulum varius bibendum metus. Aliquam erat volutpat.
            Praesent volutpat odio sed ante posuere lacinia. Suspendi
            Nunc a vehicula velit. Vivamus rutrum elementum massa in
        </TextBlock>
    <StackPanel Orientation="Horizontal">
```

Step 4: Create the “flag” in column 3.

```

<StackPanel Orientation="Horizontal">
    <Rectangle Fill="Blue" Width="200" Height="200" VerticalAlignment="Top" />
    <StackPanel>
        <Rectangle Fill="Red" Width="100" Height="100" />
        <Rectangle Fill="Yellow" Width="100" Height="100" />
    </StackPanel>
</StackPanel>

```

Step 5: Add Margins

```

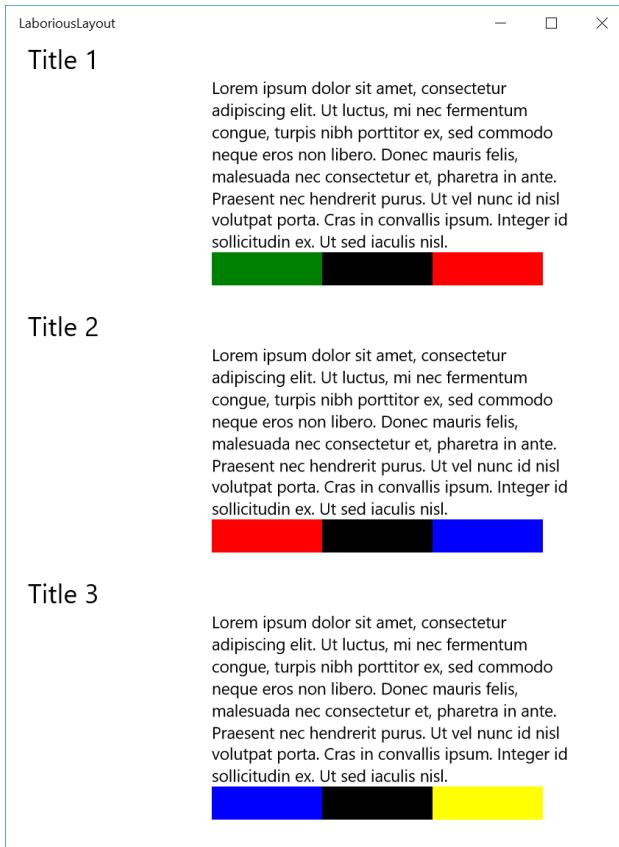
<StackPanel Margin="10,20,0,0"> ←
    <TextBlock Text="Lorem Ipsum" FontSize="48" Margin="0,0,0,20" /> ←
    <StackPanel Orientation="Horizontal">
        <TextBlock Width="250" TextWrapping="Wrap" Margin="0,0,20,0"> ←
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut luctus, mi nec ...
            Vestibulum sit amet neque nec eros interdum laoreet nec vitae justo. Maecen ...
        </TextBlock>
        <TextBlock Width="500" TextWrapping="Wrap" Margin="0,0,20,0"> ←
            Vestibulum varius bibendum metus. Aliquam erat volutpat. Vestibulum digniss ...
            Praesent volutpat odio sed ante posuere lacinia. Suspendisse eget ullamcor ...
            Nunc a vehicula velit. Vivamus rutrum elementum massa in consectetur. Nunc ...
        </TextBlock>
        <StackPanel Orientation="Horizontal">
            <Rectangle Fill="Blue" Width="200" Height="200" VerticalAlignment="Top" />
            <StackPanel>
                <Rectangle Fill="Red" Width="100" Height="100" />
                <Rectangle Fill="Yellow" Width="100" Height="100" />
            </StackPanel>
        </StackPanel>
    </StackPanel>
</StackPanel>

```

## UWP-015 - Laborious Layout Challenge

The "Laborious Layout Challenge" is a little trickier because you'll need to use both multiple Grids and multiple StackPanels. Furthermore, there are probably a number of different ways to solve this, not just one way.

You'll be building the following User Interface:



There are three distinct sections. Each section starts with a title. The title is offset vertically from the text that it's associated with.

Below each section of text, there are a little flag-like area with three rectangles of different colors.

Note that there is some margin space on the left and between each of the titled areas.

Please review the rules for this challenge as outlined in the file inside of the zipped folder associated with this lesson called UWP-015-Challenge.txt.

First, you must use both Grids and StackPanels. Even if you think you could just use StackPanels or Grids to accomplish this, the requirement is to use both.

Second, I've set the designer at the Desktop, not the phone size, just so I get more horizontal space.

Third, the width of the entire layout should be 500 pixels. Notice that this gives creates a lot of extra space over on the right-hand side, that's okay. Again, this is just a challenge. This is not real life.

Fourth, use the text from the LoremIpsum.txt file. I think all you need is that very first paragraph of Lorem Ipsum.

Next, most of the margins are either 10 or 20 pixels.

The Title TextBlocks have a font size of 24 points.

The Rectangles should be 100 wide by 30 tall.

The color of the rectangles should be:

- Green, Black, Red
- Red, Black, Blue,
- Blue, Black, Yellow

Use the UWP-015-Screenshot.png image as your guide as you're working on this.

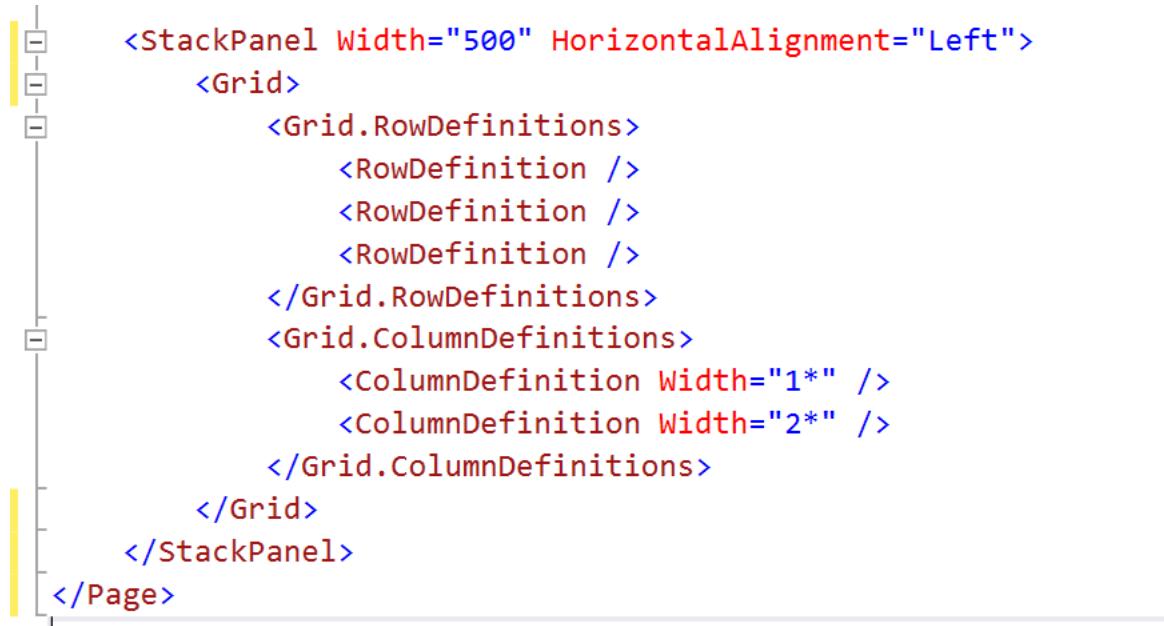
## UWP-016 - Laborious Layout Challenge: Solution

There is no one correct way to complete this challenge since there are many possible combinations of Grids and StackPanels that could be employed. I offer only my solution.

As an overall strategy, I will focus on building one of the three sections. After it looks good, I'll copy and paste twice, then replace the variable information as needed.

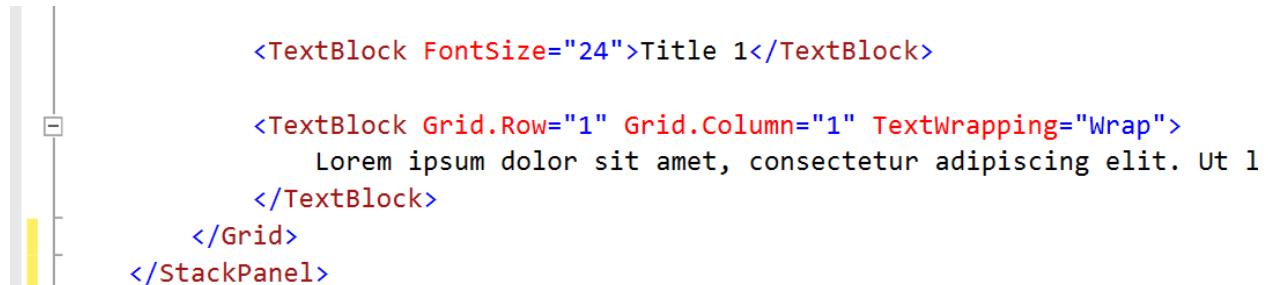
Step 1: Create a new project named "LaboriousLayout".

Step 2: Create a Grid for the first section with RowDefinitions and ColumnDefinitions.



```
<StackPanel Width="500" HorizontalAlignment="Left">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*" />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>
    </Grid>
</StackPanel>
</Page>
```

Step 3: Add the TextBlocks for the Title and Lorem Ipsum paragraph using the supplied file.



```
<TextBlock FontSize="24">Title 1</TextBlock>

<TextBlock Grid.Row="1" Grid.Column="1" TextWrapping="Wrap">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut l
</TextBlock>
</Grid>
</StackPanel>
```

Step 4: Create the "flag" using a StackPanel and Rectangles.

```

    <StackPanel Orientation="Horizontal" Grid.Row="2" Grid.Column="1">
        <Rectangle Fill="Green" Width="100" Height="30" />
        <Rectangle Fill="Black" Width="100" Height="30" />
        <Rectangle Fill="Red" Width="100" Height="30" />
    </StackPanel>
</Grid>

```

Step 5: Add margins.

```

<StackPanel Width="500" Margin="20,0,0,0" HorizontalAlignment="Left">
    <Grid Margin="0,0,0,20">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>

```

Step 6: Copy the first section and paste twice below the first section.

Step 7: Replace required markup for section 2.

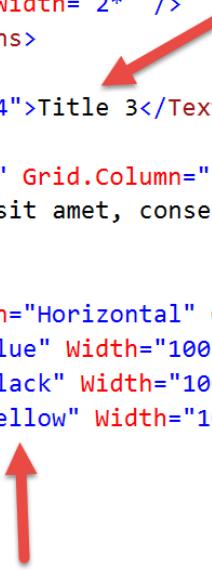
```

        <Grid Margin="0,0,0,20">
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="1*" />
                <ColumnDefinition Width="2*" />
            </Grid.ColumnDefinitions>
            <TextBlock FontSize="24">Title 2</TextBlock>
            <TextBlock Grid.Row="1" Grid.Column="1" TextWrapping="Wrap">
                Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut luctus,
            </TextBlock>
            <StackPanel Orientation="Horizontal" Grid.Row="2" Grid.Column="1">
                <Rectangle Fill="Red" Width="100" Height="30" />
                <Rectangle Fill="Black" Width="100" Height="30" />
                <Rectangle Fill="Blue" Width="100" Height="30" />
            </StackPanel>
        </Grid>
    </StackPanel>
</Page>

```

Step 8: Replace required markup for section 3.

```
<Grid Margin="0,0,0,20">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*" />
        <ColumnDefinition Width="2*" />
    </Grid.ColumnDefinitions>
    <TextBlock FontSize="24">Title 3</TextBlock>
    <TextBlock Grid.Row="1" Grid.Column="1" TextWrapping="Wrap">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut luctus,
    </TextBlock>
    <StackPanel Orientation="Horizontal" Grid.Row="2" Grid.Column="1">
        <Rectangle Fill="Blue" Width="100" Height="30" />
        <Rectangle Fill="Black" Width="100" Height="30" />
        <Rectangle Fill="Yellow" Width="100" Height="30" />
    </StackPanel>
</Grid>
</StackPanel>
</Page>
```



## UWP-017 - XAML Layout with RelativePanel

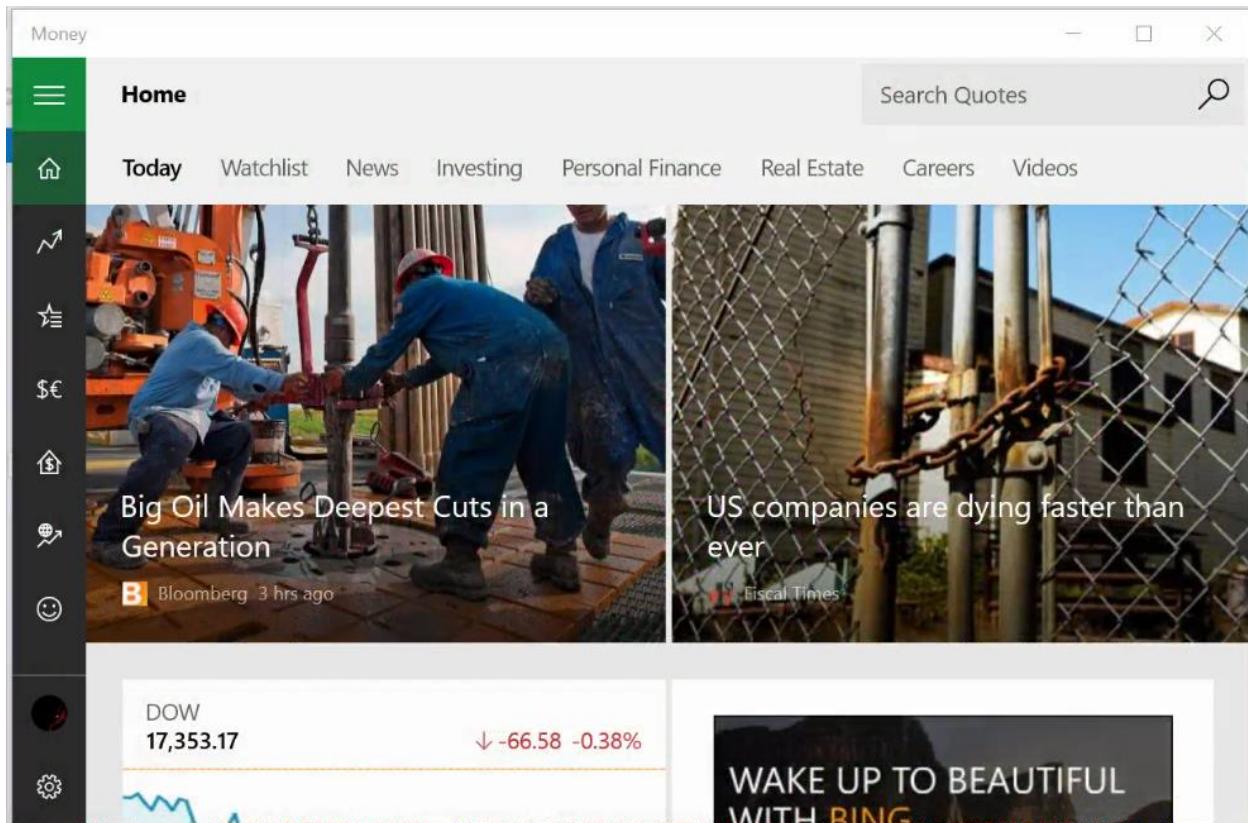
We're not done with Layout quite yet. Next we will talk about new Layout controls that were introduced with the Universal Windows Platform API.

You might wonder why are there new Layout controls? What has changed from previous APIs that we used to build apps for the Windows 8.1 store and the Windows Phone 8.1 store. There are two reasons why we need new Layout controls.

First of all, to help us build apps that look like they belong in Windows 10. I'll talk about that more in just a moment.

Secondly, to help us build applications that adapt to different device family and screen sizes and talk about that in an upcoming lesson.

If you're not familiar with the new Windows 10 aesthetic see the screenshot of the Money app, one of the stock applications that come preinstalled with Windows 10.



If you were to open the other stock apps, namely News, Weather, Sports, and a few others, you would see that they all share some common characteristics in their aesthetic and functionality that really identify themselves as Windows 10 applications. I want to take note of those features then, over the

course of the next few lessons, show how we can duplicate this ourselves with the built-in controls available in the Universal Windows Platform.

The aesthetic includes both the chrome and the “cards” in the main area. By “chrome”, I’m referring to the top and left-most area that provides navigation and other services to your application. The style of navigation which includes the icon with three horizontal lines is known as “hamburger navigation”. When you click that button in the upper left-hand corner it will show a SplitView control that will display navigation to the various areas or functionality of your application.

In expanded mode both an icon and a title for the given area of the application is displayed at the top. We can navigate around using the expanded view or the compacted view. When we’re in compacted mode, you can only see the icons on the left. The selected item is highlighted in one of the primary accent colors for the application.

Next, if you look at the very top of the application you’ll see a bar containing a search box. Each app every app like a search bar in the upper right hand corner. In this case, we’re searching for the current stock price for a given company. Also then to the left of that, docked over to the left-hand side is the title of the area that we’re currently in and then to the left of that a navigation button that allows us to go back through the navigation history to get back to the homepage.

There are some other features of these applications like the card-based layout, where you have all of these little panels of cards that will dynamically resize themselves and then depending on the view port (the size of the window) that has been resized by the user, they will either shift down, wrap down to the next line or they will expand and contract.

In this lesson I’ll introduce the RelativePanel which is a Layout container that allows you to create user interfaces that don’t have a clear linear pattern. When I say “linear pattern” what I mean is that it’s for layouts that are not fundamentally stacked or wrapped like that card layout or even tabular like in a grid. So these are layouts that you may not find as easy to reproduce using a StackPanel or a Grid.

Now certainly, what we saw in the Windows 10 application we probably could achieve that using a StackPanel and Grid combination, however, I think you’re going to find that these two new controls will help you achieve this a little bit more elegantly.

The RelativePanel defines an area where you position and align child objects, so you position other controls either in relation to each other or in relation to the parent panel itself. And there are three basic categories of attached properties that allow you to position the controls inside of the panel.

- There are panel alignment relationships. These have attached properties like align the top of my control with the panel. So AlignTopWithPanel, AlignLeftWithPanel, and so on.
- Then there are sibling alignment relationships. So AlignTopWith = (and then you give the name of the control that you want to be aligned top with).
- And then there are sibling positional relationships, so I want to be above my sibling, I want to be below my sibling, I want to be to the right and the left of my sibling.

I’ve created a project named RelativePanelExample which I’ll expand the example to learn about RelativePanel.

So first I’ll add some RowDefinitions and add a RelativePanel in the second RowGrid.Row="1".

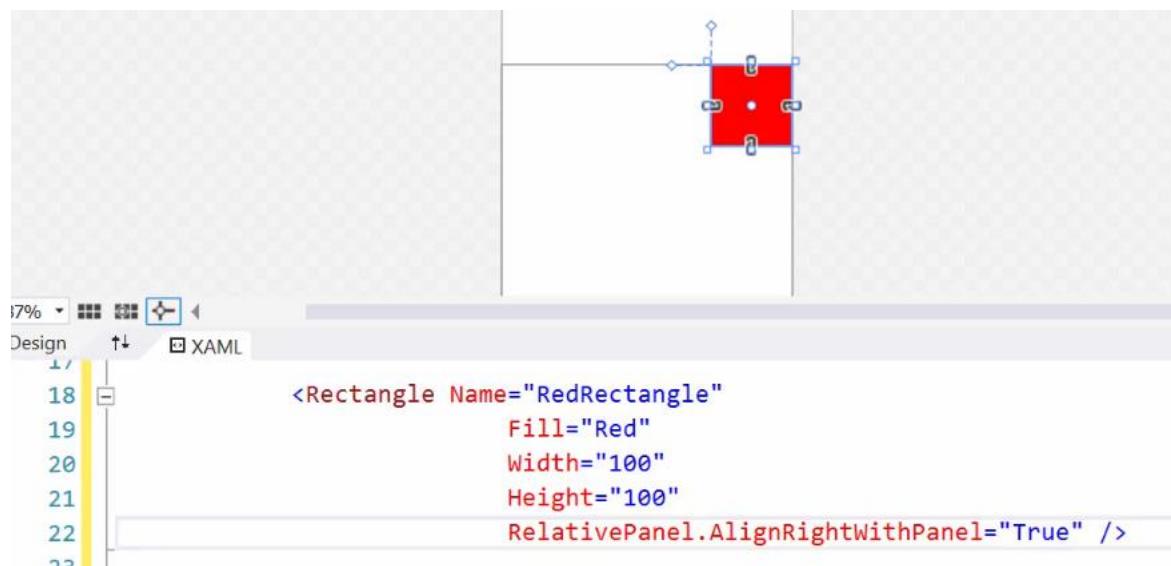
```

10   <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11     <Grid.RowDefinitions>
12       <RowDefinition Height="1*" />
13       <RowDefinition Height="2*" />
14       <RowDefinition Height="1*" />
15     </Grid.RowDefinitions>
16     <RelativePanel MinHeight="300" Grid.Row="1" >
17

```

I also set a minimum height, so no matter what I never want the height of this RelativePanel to be less than 300 pixels.

Next I'll add a series of rectangles.



The first rectangle will have a fill set to Red. Notice I've set the `RelativePanel.AlignRightWithPanel = "True"`. This aligns the right side of the Rectangle with the panel.

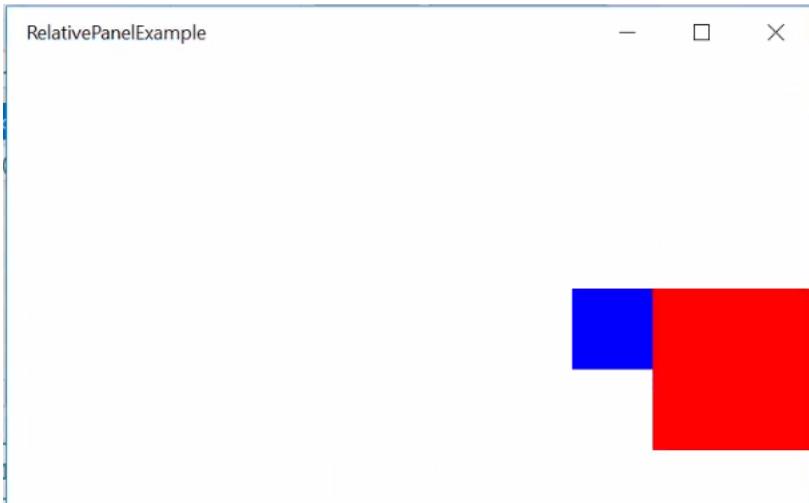
Next I'll add another Rectangle that will be positioned in relation to that sibling Rectangle.

```

24   <Rectangle Name="BlueRectangle"
25     Fill="Blue"
26     Width="50"
27     Height="50"
28     RelativePanel.LeftOf="RedRectangle" />

```

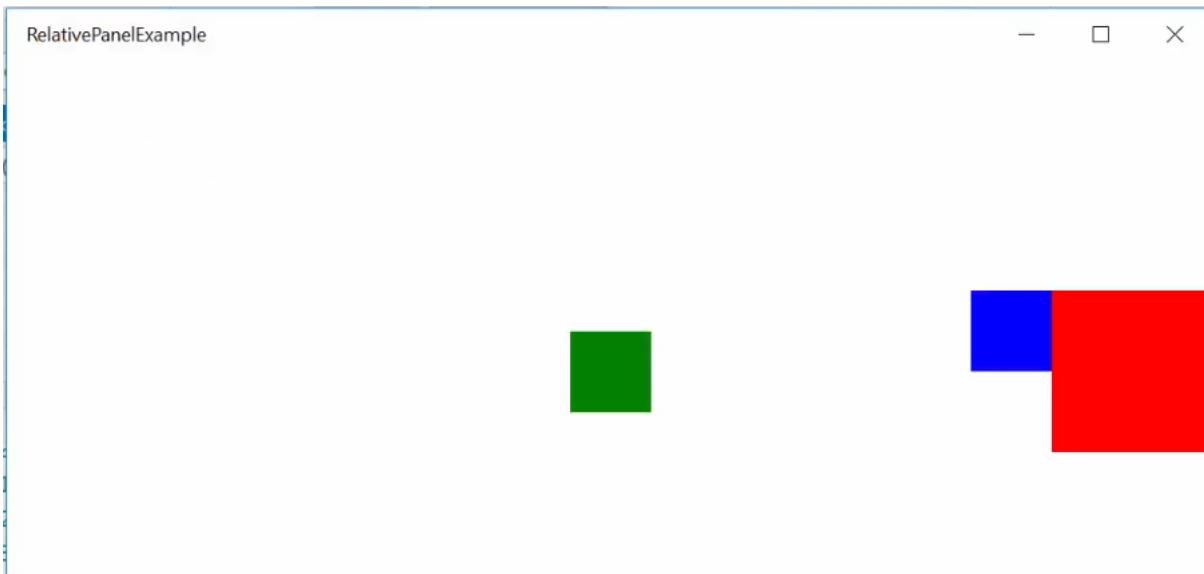
When running the application, even as we resize the application, the blue rectangle will always be to the left of the red rectangle. And the red rectangle will always be aligned to the right-hand side.



Next I'll demonstrate that you can also set multiple attached properties an XAML control.

```
30 <Rectangle Name="GreenRectangle"
31     Fill="Green"
32     Width="50"
33     Height="50"
34     RelativePanel.AlignVerticalCenterWith="RedRectangle"
35     RelativePanel.AlignHorizontalCenterWithPanel="True" />
```

On the green rectangle I've set the `AlignVerticalCenterWith="RedRectangle"`. So now the center of the red rectangle and the center of the green rectangle will be the same. Furthermore, I set the `AlignHorizontalCenterWithPanel="True"`. So show me where the center is of the horizontal size of the app and I want to be right there in the middle.



If you were to run the project, you could resize the height and width of the app and see how it responds to different screen sizes; the green rectangle with its center aligned vertically to the red rectangle and its horizontal alignment centered to the center of the application.

Next, I add a yellow Rectangle:

```
37 <Rectangle Name="YellowRectangle"
38     Fill="Yellow"
39     MinWidth="50"
40     MinHeight="50"
41     RelativePanel.AlignBottomWithPanel="True"
42     RelativePanel.AlignTopWith="PurpleRectangle"/>
```

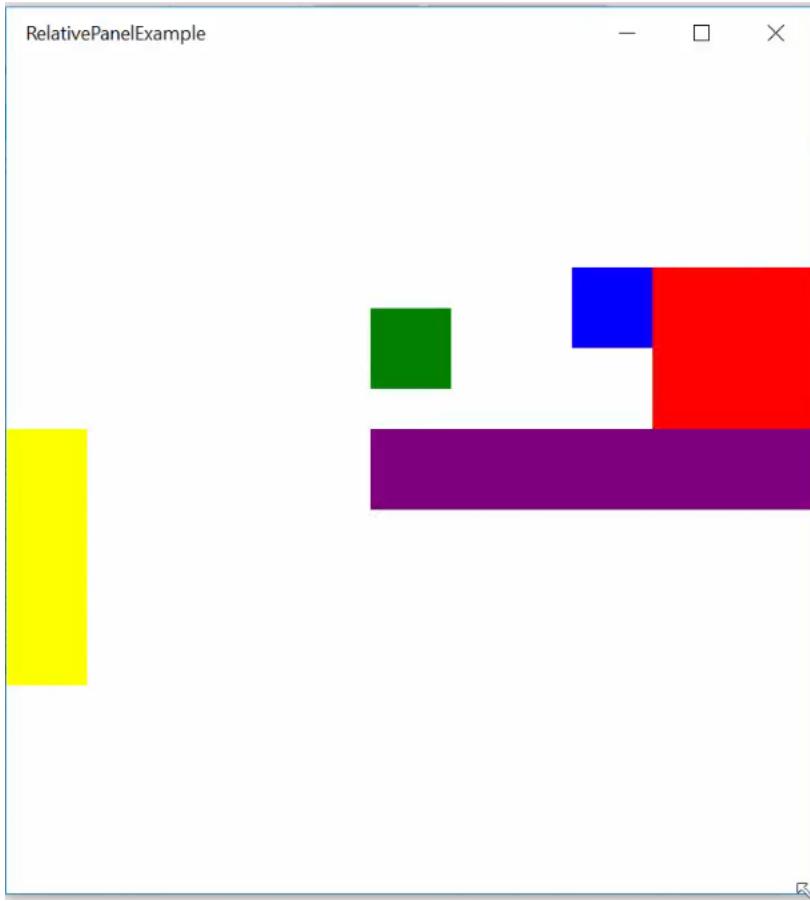
You can see that I set the minimum width to 50 and the minimum height to 50. And why this is important is because align the bottom of our rectangle with the bottom of the panel and align the top with the purple rectangle.

And so I'll define the purple rectangle above it:

```
45 <Rectangle Name="PurpleRectangle"
46     Fill="Purple"
47     MinWidth="50" MinHeight="50"
48     RelativePanel.Below="RedRectangle"
49     RelativePanel.AlignRightWith="RedRectangle"
50     RelativePanel.AlignLeftWith="GreenRectangle"
51 />
```

I want the purple rectangle to be positioned below the red rectangle. I also want its right aligned with the right of the red rectangle, and I want the left aligned with the green rectangle.

When I run the project we can observe the results:

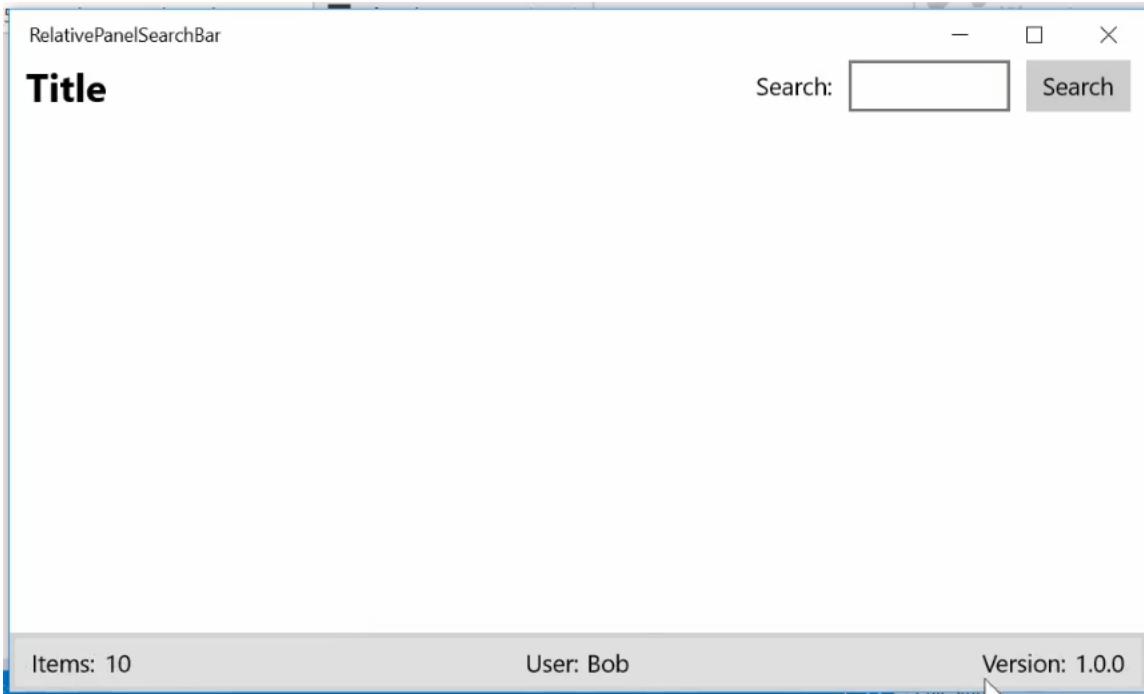


As I re-position the actual size of the Purple and Yellow rectangles will change. In both cases you'll see that by resizing, we will get a larger width or height, a larger rectangle.

This is why you can perform interesting positional logic with the RelativePanel because you can set sides relative to other controls or relative to the panel itself and things can automatically resize themselves.

Why is this important? Hopefully this will be evident in another project I created named RelativePanelSearchBar.

This project contains the beginnings of a Windows 10 application with a search bar on the right-hand side. It doesn't look finished yet, but you can see how no matter what, over on the left-hand side, or on the right-hand side rather, we have it always docked to the right and we have the title to the left.



Furthermore, I wanted to show that we can create a little status bar that's always docked to the bottom. Depending on the information, it's either aligned to the left, aligned to the right, or aligned center. So let's see how we achieve this effect in our XAML.

First, the Grid's RowDefinitions:

```
]   <Grid>
]     <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
```

The top and the bottom rows are Auto so just take up enough room, just what you need, and then the middle area will be star sizing and so we didn't do anything in there.

Next, I added a RelativePanel to sit in the top row. Inside of the RelativePanel a SearchButton whose right side is aligned with the panel, a TextBox that is to the left of the SearchButton, and so on.



Next in the third row a status bar creating using control called a Border and a Border. The Border just creates a little layout control that provides us the ability to style the background color and the stroke around the background color.

```
<Border BorderThickness="3" Background="#FFE0E0E0" Grid.Row="2" BorderBrush="#FFD2D2D2">
    <RelativePanel>
        <TextBlock Name="ItemsTextBlock"
            Text="Items:"
            RelativePanel.AlignLeftWithPanel="True"
            Margin="10,5,0,5" />
        <TextBlock Text="10"
            RelativePanel.RightOf="ItemsTextBlock"
            Margin="5,5,0,5" />

        <TextBlock Text="Version:"
            RelativePanel.LeftOf="VersionTextBlock"
            Margin="0,5,5,5" />
        <TextBlock Name="VersionTextBlock"
            Text="1.0.0"
            RelativePanel.AlignRightWithPanel="True"
            Margin="0,5,10,5" />
```

So we're just putting our RelativePanel inside of the Border and then we will do our work inside of the RelativePanel itself.

There are three sections to the RelativePanel. The first TextBlock on the left-hand side will just contain the text “Items:”. In this example, “items” could represent items that need to be addressed. It uses AlignLeftWithPanel. The next TextBlock aligns itself to the right of that first TextBlock and would contain a numeric value.

The right-hand side works the same way except with the text “Version”.

And then the third case – the middle --I actually create a StackPanel because if you were to set one of the TextBoxes using the RelativePanel.AlignHorizontalCenterWithPanel, then it will be in the absolute center and anything else that you need to the left of it or to the right of it would then be off center. Since we need both TextBlocks to be centered I put them in a StackPanel which ensures the TextBlocks are centered no matter the width of either one.

```
<StackPanel RelativePanel.AlignHorizontalCenterWithPanel="True"
            Orientation="Horizontal">
    <TextBlock Text="User:" Margin="0,5,5,5" />
    <TextBlock Text="Bob" Margin="0,5,0,5" />
</StackPanel>
```

One note of caution regarding the RelativePanel; you could potentially get yourself into a circular reference. Example: I want object number one to the left of object number two, object number three to the left of object number two, object number four to the left of object number three, object number one to the left of number four.

Furthermore, you can set multiple relationships that target the same edge of the element and when you do that, you might have conflicting relationships in your layout as a result.

So whenever this happens, there are actually an order of events just like whenever you are working in a math problem or even in code that parentheses can dictate the order of operation. There is an order of operation with how these relationships are deciphered in this order.

- So first priority will be panel alignment, so align me to the left or the right of the panel.
- Then the second one will be Sibling Alignment relationship, so align me with the top of this control, align me with the left of that control,
- Then the third and the lowest priority would be Sibling Positional relationships. Set me above this control, below this control, to the left of this control, okay.

## UWP-018 - XAML Layout with the SplitView

The second new layout control in the Universal Windows Platform API is the SplitView which allows us to create a panel that can be displayed or hidden. The SplitView features an animation that provides a sliding effect into view and out of view. The most practical use of the SplitView is to implement Windows 10 hamburger style navigation.

There are two parts to a SplitView. One might be optional. The other is required. The part that's hidden by default that you display, that's called the "Pane". The part that's already displayed and can either be overlapped, or it can be pushed over, that's called the "Content".

Since the SplitView is a layout control it is intended to host other XAML Controls in the Pane and the Content sections. In the SplitView Pane, you would create the hamburger navigation style by adding an icon for a section or feature of your application and add text next to it.

To demonstrate its usage, I created a project named "SplitViewExample." I add a SplitView and inside the SplitView I create a Pane, and a Content area.

```
10     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11         <SplitView >
12             <SplitView.Pane>
13             ~~~~~
14             </SplitView.Pane>
15             <SplitView.Content>
16             ~~~~~
17             </SplitView.Content>
18             </SplitView>
19
20     </Grid>
21 </Page>
```

Next, in both the Pane and Content I'll add a StackPanel to contain a series of TextBlocks. I'll set the Text to values that will help us visually distinguish where the Pane ends and the Content begins.

```
10     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11         <SplitView >
12             <SplitView.Pane>
13                 <StackPanel>
14                     <TextBlock Text="First" />
15                     <TextBlock Text="Second" />
16                     <TextBlock Text="Third" />
17                 </StackPanel>
18             </SplitView.Pane>
19             <SplitView.Content>
20                 <StackPanel>
21                     <TextBlock Text="Fourth" />
22                     <TextBlock Text="Fifth" />
23                     <TextBlock Text="Sixth" />
24                 </StackPanel>
```

At this early stage if I were to run the app we would only see the Content area:

```
SplitViewExample
Fourth
Fifth
Sixth
```

How do we display the pane? We do this programmatically using C# in response to some event such as a button's click event. To access the Pane programmatically we will have to first of all give our SplitView a name. So I'll name it "MySplitView".

Next I'll add a Button with the content "Click Me" and handle the Click event by adding the attribute Click="Button\_Click".

```
25      </SplitView.Content>
26    </SplitView>
27    <Button Content="Click Me" Click="Button_Click" />
28  </Grid>
29 </Page>
```

I put my mouse cursor inside the text "Button\_Click" and press F12 on my keyboard which opens the MainPage.xaml.cs code behind and puts my mouse cursor inside of the Button\_Click event handler method stub. I add one line of code that will set the SplitView's IsPaneOpen property to the opposite of its current value, so:

```
27
28  private void Button_Click(object sender, RoutedEventArgs e)
29  {
30      MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen;
31  }
```

Back in the MainPage.xaml, I'll change this outermost Grid to a StackPanel.

```
1  <Page
2    x:Class="SplitViewExample.MainPage"
3    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5    xmlns:local="using:SplitViewExample"
6    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8    mc:Ignorable="d">
9
10   <StackPanel Background="{ThemeResource ApplicationPageBackgroundThemeBrush}>
11     <SplitView Name="MySplitView">
12       <SplitView.Pane>
13         <StackPanel>
14           <TextBlock Text="First" />
15           <TextBlock Text="Second" />
```

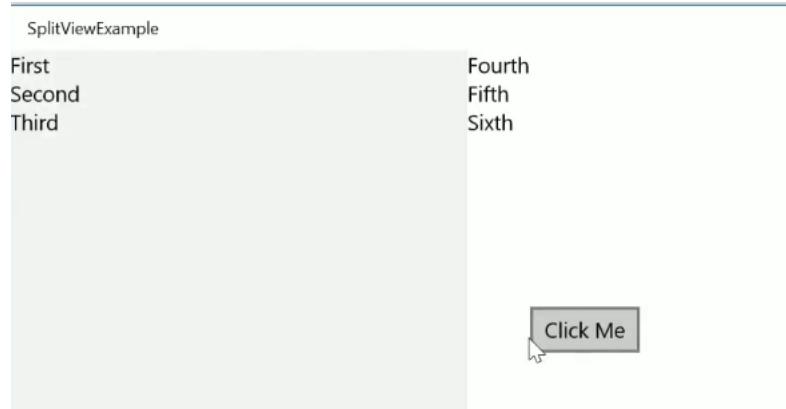
Now when I run my app in debug mode I can click the "Click Me" button to show and hide the Panel.

We're not finished configuring the SplitView's Panel. First, I'll modify the StackPanel's Orientation="Horizontal".

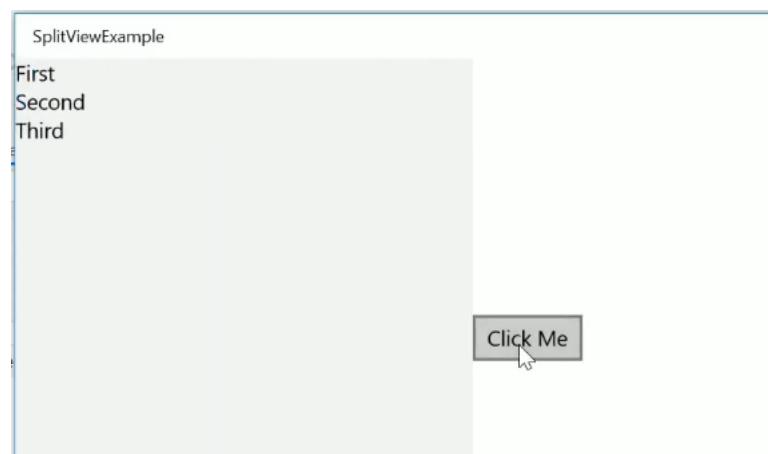
Next, I'll set the SplitView's DisplayMode. The DisplayMode is one of the most important properties that we can set on the SplitView, because it will dictate how it actually operates. There are four options here.



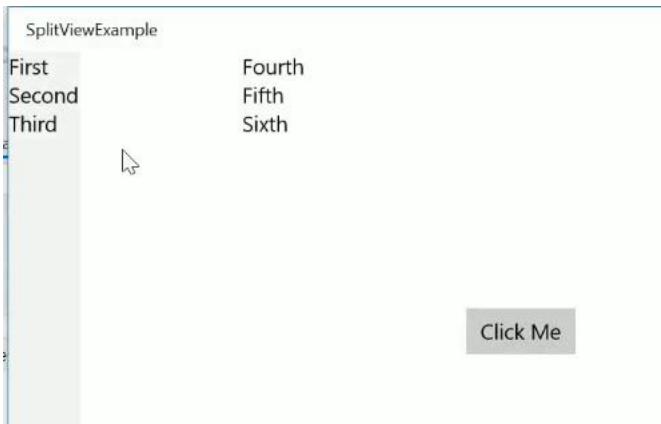
I'll set it to Inline then run the application.



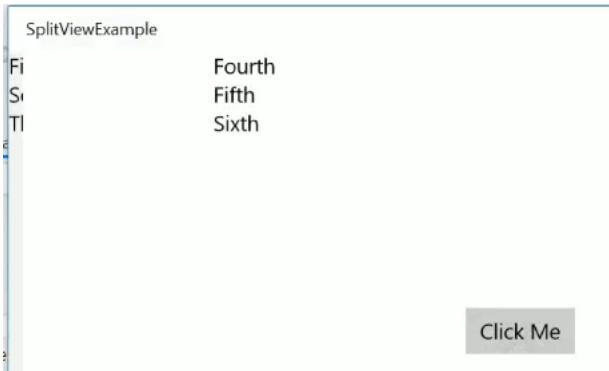
When set to Inline the Pane pushes the Content over. If I were to change the DisplayMode to Overlay the Pane would completely cover up the Content.



Next, I will set the CompactPaneLength to a relative small value, like 50. Just enough to display an icon. Then I'll set the DisplayMode to CompactOverlay. CompactOverlay is a variation on Overlay that will display just a little sliver of what's underneath in the Pane.



Let's change that even to just 10 pixels, just so we can barely see it creeping out.



You can just barely see First Second and Third creeping through there on the very most left side. Then when we click "Click Me," the Pane covers up whatever we see in the Content area.

CompactInline is exactly like Inline except it shows a sliver of Pane area as well. Just like Inline it will push the Content area over when it is opened.

We can also adjust a property called OpenPaneLength. If we were to set it to 50 pixels it would restrict the width of the Pane when it is in an opened state:

### SplitViewExample

First    Fourth  
Second Fifth  
Third   Sixth

Click Me

As you can see you're given a lot of latitude to build out your SplitView how you want to implement your Pane. In this case I just used TextBlocks however that's not the best choice. We would want to either use a Button or something that's clickable so that when we click it, we navigate to something in the main area (Content) of our application -- some feature or some different information. In a later lesson I'll demonstrate a simple implementation of the hamburger navigation featuring the SplitView.

## UWP-019 - Working with Navigation

Up until now we've only created apps with a single Page, the MainPage.XAML, and while that's fine for simple apps. However, it's likely that you will need to add additional pages and navigate to them.

Typically, you would want to load a new page into your app whenever you need new functionality, new layout, or new content. It's possible to create a complex app and change out the layout and the functionality all in a single page, but by adding additional pages it allows you to decompose functionality into smaller, more manageable units, and this will reduce the complexity of the application's development and its maintenance.

So Pages in the Universal Windows Platform are roughly equivalent to web pages in web development. As a web developer you might create several webpages and create hyperlinks between the pages so that users can navigate in between pages in the website. The end user would load up their frame (i.e., their web browser) and would navigate to your home page. From there they might then navigate to each of the pages using the hyperlinks, or they might navigate then, at some point, using the back and the forward buttons on the toolbar of the frame, (again, their web browser).

So in Universal Windows Platform your pages are hosted in Frame objects (that's why I keep using that word instead of the web browser). You may not realize this, but the Frame is critical to the application itself. As your application starts up, there is an Application object and then there is a Window inside of that, so that will give you all of the chrome in a Window's Desktop version -- the close, minimize, and the maximize buttons in the upper right-hand corner, the title, etc. along the top bar.

Inside of the Window sits that Frame which will host pages. As you can see in this new example that I've created named "NavigationExample", in the App.xaml.cs file in line 53, you can see where this all gets set up.

```
52
53     Frame rootFrame = Window.Current.Content as Frame;
54
55     // Do not repeat app initialization when the Window already has c
56     // just ensure that the window is active
57     if (rootFrame == null)
58     {
59         // Create a Frame to act as the navigation context and naviga
60         rootFrame = new Frame();
61
62         rootFrame.NavigationFailed += OnNavigationFailed;
63
64         if (e.PreviousExecutionState == ApplicationExecutionState.Ter
65         {
66             //TODO: Load state from previously suspended application
67         }
68
69         // Place the frame in the current Window
70         Window.Current.Content = rootFrame;
71     }
72 }
```

A lot of the hard work related to the interaction of the Application, Window and Frame is done for you automatically whenever you use this blank template. You can see where we get a reference to the Frame inside of the Window, and they call it rootFrame, and then in the rootFrame, you can navigate to a specific page in your application.

```
    }
}
if (rootFrame.Content == null)
{
    // When the navigation stack isn't restored navigate to the i
    // configuring the new page by passing required information as
    // parameter
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
}
// Ensure the current window is active
Window.Current.Activate();
}
```

In this case, we're navigating to MainPage, so we give it a type of a data type that we want it to navigate to, and then it creates the instance of that type, and it will pass in parameters as arguments.

While we're here, I'm going to delete this diagnostic ...

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
#if DEBUG
    if (System.Diagnostics.Debugger.IsAttached)
    {
        this.DebugSettings.EnableFrameRateCounter = true;
    }
#endif
}
```

At some point later on we will learn about loading new Pages into this topmost Frame, the rootFrame, but for getting started I think it would be easier to just talk about adding a Frame to our MainPage.xaml so that we can load individual Pages in and out of that Frame that's hosted by the MainPage.

To illustrate this, I'll add three new Pages by selecting the Project name in the Solution Explorer, then selecting the Project menu > Add New Item ... In the dialog, make sure that you select the Blank Page Template. Create three Pages named Page1, Page2 and Page3.

These will be the Pages that are loaded into and out of the MainPage's Frame.

In the MainPage.xaml I add a StackPanel to replace the top-most Grid.

I'll add another StackPanel and a Frame inside the top-most StackPanel, and I am going to give the Frame a name so I can access it programmatically. I'll name it "MyFrame". The StackPanel will be used to create a horizontal button bar, so I set the Orientation="Horizontal", and create three Buttons and set the Margin of each to "0,0,20,0".

```
9
10    <StackPanel>
11        <StackPanel Orientation="Horizontal">
12            <Button Margin="0,0,20,0" />
13            <Button Margin="0,0,20,0" />
14            <Button Margin="0,0,20,0" />
15        </StackPanel>
16        <Frame Name="MyFrame">
17
18            </Frame>
19        </StackPanel>
20    </Page>
21
```

The first Button will be named "HomeButton" and I'll set the Content="Home", and add a click event named "HomeButton\_Click".

The second Button will be named "BackButton" and I'll set the Content="Back", and add a click event named "BackButton\_Click".

The third Button will be named "ForwardButton" and I'll set the Content="Forward" and add a click event named "ForwardButton\_Click".

As our Frame loads along with the MainPage, I want to load Page1 into view automatically. In the code behind file MainPage.xaml.cs I'll located the constructor for the Page, I will access MyFrame, and call its Navigate method. Just like we saw in the App.xaml.cs, we will need to navigate to a typeof(Page1).

```
20
21    /// <summary>
22    /// An empty page that can be used on its own or navigated to
23    /// </summary>
24    public sealed partial class MainPage : Page
25    {
26        public MainPage()
27        {
28            this.InitializeComponent();
29            MyFrame.Navigate(typeof(Page1));
30        }
31    }
```

That should load Page1 during MainPage construction.

Next, I'll create method stubs for each of the three buttons by putting my mouse cursor in the Click property and pressing F12.

Next, I'll load pages into MyFrame with each button click.

```

31     private void HomeButton_Click(object sender, RoutedEventArgs e)
32     {
33         MyFrame.Navigate(typeof(Page1));
34     }
35
36     private void BackButton_Click(object sender, RoutedEventArgs e)
37     {
38         if (MyFrame.CanGoBack)
39         {
40             MyFrame.GoBack();
41         }
42     }
43
44     private void ForwardButton_Click(object sender, RoutedEventArgs e)
45     {
46         if (MyFrame.CanGoForward)
47         {
48             MyFrame.GoForward();
49         }
50     }

```

The HomeButton is easy. Whenever it is clicked the Frame will load Page1.

However, in the Back and Forward buttons I'll invoke the Frame's history, or rather the BackStack. The Frame will maintain the history of pages that the user navigates through, and it will replay those navigation requests whenever we go back or forward. To use this, we should ask the navigation's BackStack, "Can we go forward? "Can we go backward? And if we can, then go ahead and go back, or go ahead and go forward, respectively."

If MyFrame.CanGoBack is true then we'll call GoBack(). Similarly, if MyFrame.CanGoForward is true then we'll call GoForward().

Next I'll open Page1.xaml and replace the Grid with a StackPanel. Inside the StackPanel I'll add TextBlock, I'll make the font size large, and I'll set the Text equal to "Page1". Also, I'll add a HyperlinkButton. I could add any type of control where I can handle an event, and handle it to tell the Frame to navigate to a given Page. I use the HyperlinkButton because it connotes navigation, just like a web page's hyperlink connotes navigation. The HyperlinkButton has a special property that makes it different from a Button that's simply styled to look like a hyperlink. The HyperlinkButton has a NavigateURI property that will allow us to navigate out to a website. I'll add a second copy of the HyperlinkButton just to demonstrate that and I'll set the NavigateURI property to <http://www.microsoft.com>:

```

10     <StackPanel>
11         <TextBlock FontSize="48" Text="Page 1" />
12         <HyperlinkButton Content="Go to Page 2" Click="HyperlinkButton_Click" />
13         <HyperlinkButton Content="Go to Microsoft.com" NavigateUri="http://www.m
14     </StackPanel>
15 </Page>

```

And I'll handle the HyperlinkButton\_Click event:

```
29
30     private void HyperlinkButton_Click(object sender, RoutedEventArgs e)
31     {
32         Frame.Navigate(typeof(Page2));
33     }
34 }
```

I'll do the same on Page2.xaml:

```
10     <StackPanel>
11         <TextBlock FontSize="48" Text="Page 2" />
12         <HyperlinkButton Content="Go to Page 3" Click="HyperlinkButton_Click" />
13     </StackPanel>
14
15 </Page>
```

And I'll add the code to navigate to Page3:

```
29
30     private void HyperlinkButton_Click(object sender, RoutedEventArgs e)
31     {
32         Frame.Navigate(typeof(Page3));
33     }
34 }
```

Finally, on Page3 I'll simply add a TextBlock:

```
10     <StackPanel>
11         <TextBlock FontSize="48" Text="Page 3" />
12
13     </StackPanel>
14
```

If you run the application you will be able to click the "Go to Microsoft.com" to load that website into a web browser, or click the "Go to Page 2" and "Go to Page 3" links and navigate to those Pages in the Frame.



I should also be able to click the Home, Back and Forward buttons to traverse the BackStack.

You might wonder why am I referenced Frame and not MyFrame in the code above? After all, the Frame's name is MyFrame, right? True, whenever we call the Navigate method, and we pass in the type that create a new instance of, that Navigate method will create that instance and will act as a factory in

so much that it will set up the new instance of that object by setting its various properties. One of the properties that it sets for a given page is the Frame property, and sets it to whatever the parent Frame is, so that's how we get a reference to the current Frame that we're sitting inside of in order to call its Navigate method.

So that's the most basic scenario that we can cover. Let's add a wrinkle and pass values from Page2 to Page3.

I add a TextBox to Page2 and name that ValueTextBox.

```
10 <StackPanel>
11     <TextBlock FontSize="48" Text="Page 2" />
12     <TextBox Name="ValueTextBox" Width="200" />
13     <HyperlinkButton Content="Go to Page 3" Click="HyperlinkButton_Click" />
14 </StackPanel>
15
```

A screenshot of the Visual Studio code editor showing the XAML code for Page2.xaml. Line 13 contains a HyperlinkButton with a Click event handler. A red arrow points to the Click="HyperlinkButton\_Click" part of the code.

I add a TextBox to Page3 and also name that ValueTextBox.

```
9 <StackPanel>
10     <TextBlock FontSize="48" Text="Page 3" />
11     <TextBox Name="ValueTextBox" Width="200" />
12 </StackPanel>
13
14
```

A screenshot of the Visual Studio code editor showing the XAML code for Page3.xaml. It contains a StackPanel with a TextBlock and a TextBox, followed by a closing StackPanel tag on line 12.

Now what I want to do when I navigate from Page 2 to Page 3, I want to pass along the value that the user typed in the ValueTextBox on Page2 and then display that value in the ValueTextBox in Page 3. How do we do that?

If we take a look at Page2.xaml.cs, there is an overloaded version of the Navigate method that allows us to pass in anything we want as a parameter that is then retrieved by the other page that we're navigating to, so in this case just take ValueTextBox.Text, and pass it over to the other side.

```
30     private void HyperlinkButton_Click(object sender, RoutedEventArgs e)
31     {
32         Frame.Navigate(typeof(Page3), ValueTextBox.Text);
33     }
34 }
```

A screenshot of the Visual Studio code editor showing the C# code for Page2.xaml.cs. It contains a single method named HyperlinkButton\_Click that uses the Frame.Navigate method to navigate to Page3, passing the ValueTextBox.Text as a parameter.

Now retrieving it on the other side is a little bit trickier. On the Page3.xaml.cs we will have to override the OnNavigatedTo method which fires every time that we navigate to this Page.

```
29
30     protected override void OnNavigatedTo(NavigationEventArgs e)
31     {
32         var value = (string)e.Parameter;
33         ValueTextBox.Text = value;
34     }
35 }
```

A screenshot of the Visual Studio code editor showing the C# code for Page3.xaml.cs. It contains an override of the OnNavigatedTo method that retrieves the Parameter from the NavigationEventArgs and sets it to the ValueTextBox.Text.

Notice that we access the value using the NavigationEventArgs' Parameter property. Since the value we pass is passed as an Object data type, it can hold any type, therefore we must cast it to string.

Running the application again, we can send test from Page2, navigate to Page3 and display the value on Page3.

But what happens when click the Back button. The value is no longer in Page2's TextBox. However, if we then click the Forward button the value is still there. What happened?

Remember I said whenever you're navigating through the BackStack -- through the history of the Frame – it will replay the Navigation event. In our case, the step of navigating from Page 2 to Page 3 included some text, so it gets picked up as we replay that event and displayed into Page3's TextBox. However, when we go back to Page 2, when we navigated from Page 1 to Page 2, we didn't send anything over. It replays that event, and there is nothing in Page2's TextBox as a result of it.

But what if we wanted to maintain state between these pages? There are several different ways we can go about this. Some experienced developers might caution you against this approach, and they're probably right – if you were building an enterprise scale application, however I want to keep this as simple as possible, and as long as you don't do anything abusive to your application this technique should be fine. Just keep in mind that there are probably more elegant solutions than what I'm about to do.

So since we said that the Application sits at the very top of the stack when you launch an app, there is an Application object that hosts a Window, that hosts a Frame, that hosts MainPage. What we can do is treat this App class, this instance of Application, as a global variable. In other words, it is accessible throughout the application. So inside of the class's definition I can add a public (internal) field:

```
23  sealed partial class App : Application
24  {
25
26      internal static string SomeImportantValue;
27
28  /// <summary>
29  /// Initialize the singleton application object. This is
30  ///
```

SomeImportantValue is a static string. Internal means that anything inside of this app can access that variable, but nothing outside of the application.

Back in Page2.xaml.cs I modify the Hyperlinkbutton\_Click event to save the value typed into the ValueTextBox into the SomeImportantValue field.

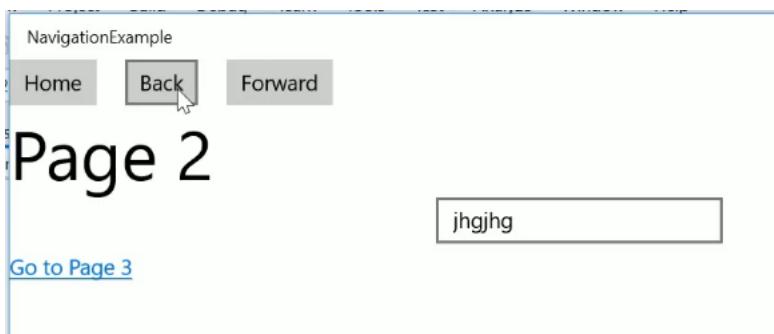
Next, I override the OnNavigatedTo() method and whenever navigation returns to this page I can see if the ValueTextBox.Text was ever set to any value by checking to see if it is not null or empty. If it was, then I can reset the ValueTextBox to the field SomeImportantValue:

```

30     private void HyperlinkButton_Click(object sender, RoutedEventArgs e)
31     {
32         App.SomeImportantValue = ValueTextBox.Text;
33         Frame.Navigate(typeof(Page3), ValueTextBox.Text);
34     }
35
36     protected override void OnNavigatedTo(NavigationEventArgs e)
37     {
38         if (!String.IsNullOrEmpty(App.SomeImportantValue))
39         {
40             ValueTextBox.Text = App.SomeImportantValue;
41         }
42     }

```

Replaying that scenario from earlier, we can retrieve the original value typed into Page2 when navigating back to that Page.



Next, I'll demonstrate how to navigate to a whole different page at the RootFrame level.

Let's go to MainPage.xaml and add one more Button named NavigateButton, and I'll set the Content="Navigate Root Frame", and set the Click equal to a new event handler.

I press F12 to work in the button's click event method stub. Remember we said that Page has a Frame property, so I'm just going to use the "this" keyword to access this instance of MainPage, then reference the Frame's Navigate() method and will pass in type of Page:

```

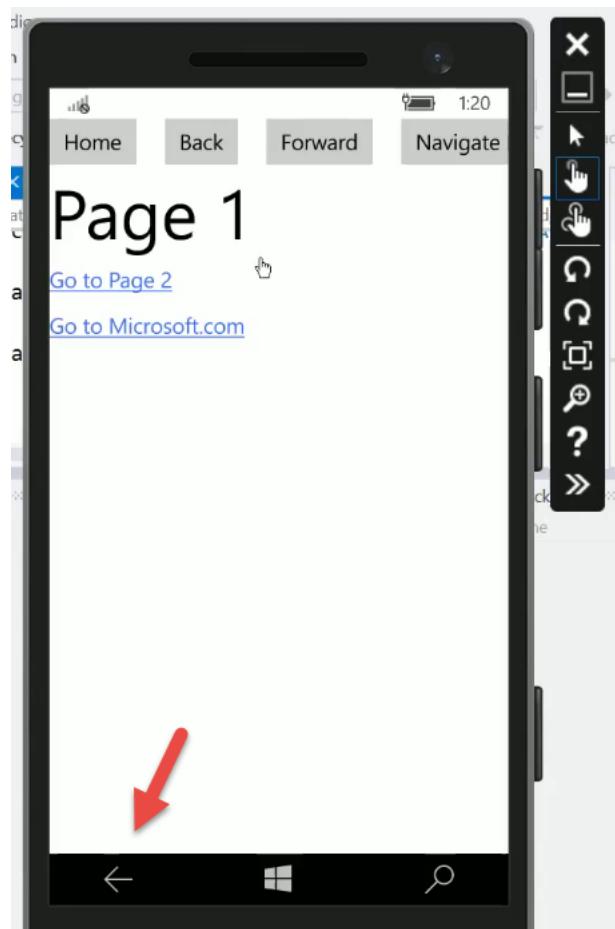
51     private void NavigateButton_Click(object sender, RoutedEventArgs e)
52     {
53         this.Frame.Navigate(typeof(Page2));
54     }
55 }
56

```

When I run the application I now automatically navigate to Page 2. However, as a result of that, I've lost the chrome, the outermost StackPanel, and I've lost my navigation buttons because I'm loading a new Page (Page2) to replace MainPage.

Finally, let's talk about what happens whenever we run this on a mobile device because the phone is unique regarding navigation when compared to other device families.

Below you can see that we have this representation of the back button here on the chrome of our emulator.



If we were to go to Page 2, type in data and go to Page 3, and then we can go back, and then we can go forward using the buttons we created at the top of the Page.

What if I were to tap this back arrow at the bottom? What we might want to happen is that go back to Page 2, then to Page One, but when I hit the back button, it completely escapes out of our application and goes back to the Home Page, and, unfortunately, how to fix this will require a little bit more explanation and we won't cover that in this series. There are lengthy articles that explain how to handle this scenario.

## UWP-020 - Common XAML Controls - Part 1

This lesson will cover some of the most basic input controls that we haven't already talked about up to this point. We will need some of these input controls for future projects. We will cover seven in rapid fire in this lesson.

I've created a new project named: "ControlsExamplePart1".

To setup this example I'll drag and drop an image from my desktop into the Assets folder. This image that should be included with the zipped folder associated with this lesson.

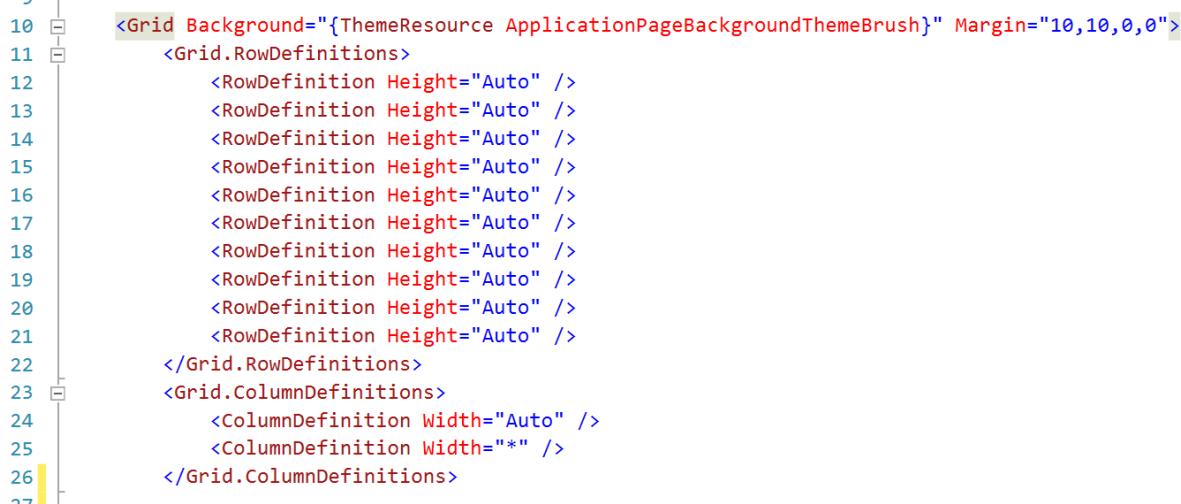
There are really three things that I want to learn about each control:

First of all, how do I display it on screen?

Secondly, how do I handle the major events that it will emit whether it be a tapped or a selection changed, whatever the case might be?

Third, how do I retrieve the value out of that control, the current selection, whatever has been typed in, whatever has been checked or unchecked, etc.?

I'll begin by adding a Grid, RowDefinitions and ColumnDefinitions:



```
10 <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}" Margin="10,10,0,0">
11   <Grid.RowDefinitions>
12     <RowDefinition Height="Auto" />
13     <RowDefinition Height="Auto" />
14     <RowDefinition Height="Auto" />
15     <RowDefinition Height="Auto" />
16     <RowDefinition Height="Auto" />
17     <RowDefinition Height="Auto" />
18     <RowDefinition Height="Auto" />
19     <RowDefinition Height="Auto" />
20     <RowDefinition Height="Auto" />
21     <RowDefinition Height="Auto" />
22   </Grid.RowDefinitions>
23   <Grid.ColumnDefinitions>
24     <ColumnDefinition Width="Auto" />
25     <ColumnDefinition Width="*" />
26   </Grid.ColumnDefinitions>
27 
```

The screenshot shows a code editor with XAML code. The code defines a Grid with a background color and margin. It contains two sections: RowDefinitions and ColumnDefinitions. The RowDefinitions section contains ten RowDefinition elements, each with a Height of "Auto". The ColumnDefinitions section contains two ColumnDefinition elements, one with a width of "Auto" and another with a width of "\*". The code is numbered from 10 to 27 on the left side. A vertical scroll bar is visible on the right side of the code editor window.

Next, I'll add the first input control is the CheckBox, a very simple "yes/no" control represented as a box that is empty (no / false) or has a checkmark inside of it (yes / true).

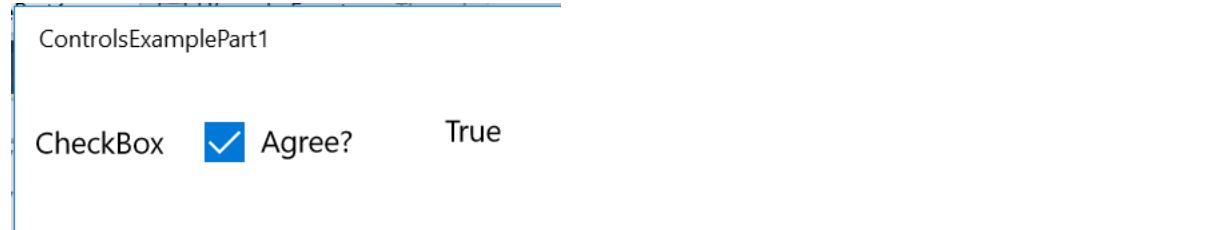
The CheckBox has a Content that will be displayed in the label next to the box. There's also a Tapped event that we will handle with C#.

```
28     <TextBlock Grid.Row="0" Text="CheckBox" VerticalAlignment="Center" />
29     <StackPanel Grid.Column="1"
30         Margin="20,10,0,10"
31         Orientation="Horizontal">
32         <CheckBox Name="MyCheckBox"
33             Content="Agree?"
34             Tapped="MyCheckBox_Tapped" />
35         <TextBlock Name="CheckBoxResultTextBlock" />
36     </StackPanel>
37
```

In the Tapped event we'll display the current value of the CheckBox by retrieving the IsChecked property:

```
30     private void MyCheckBox_Tapped(object sender, TappedRoutedEventArgs e)
31     {
32         CheckBoxResultTextBlock.Text = MyCheckBox.IsChecked.ToString();
33     }
```

When I run the app, you can see the result of tapping the checkbox:



Next, the RadioButton is similar to a CheckBox however it will allow you to add multiple potential values each grouped together by a GroupName property. In other words, all the radio buttons that belong together must have the same GroupName.

```

38     <TextBlock Grid.Row="2"
39         Text="RadioButton"
40         VerticalAlignment="Center" />
41     <StackPanel Grid.Row="2"
42         Grid.Column="1"
43         Orientation="Horizontal"
44         Margin="20,10,0,10">
45         <RadioButton Name="YesRadioButton"
46             Content="Yes"
47             GroupName="MyGroup"
48             Checked="RadioButton_Checked" />
49         <RadioButton Name="NoRadioButton"
50             Content="No"
51             GroupName="MyGroup"
52             Checked="RadioButton_Checked" />
53         <TextBlock Name="RadioButtonTextBlock" />
54     </StackPanel>
55

```

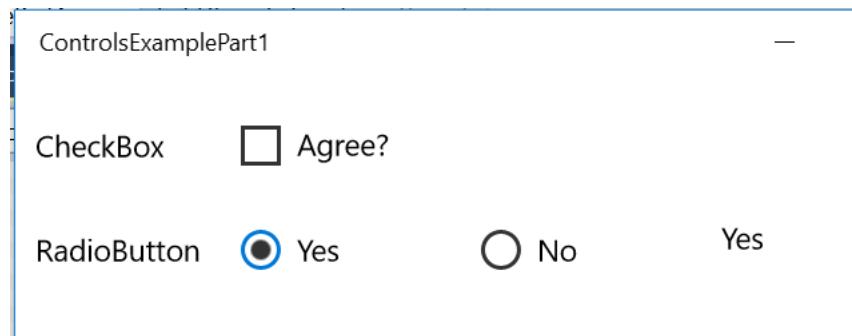
I can handle the Checked event, where I use the tertiary operator to display whether the Yes or No RadioButton IsChecked:

```

34
35     private void RadioButton_CheckedChanged(object sender, RoutedEventArgs e)
36     {
37         RadioButtonTextBlock.Text = (bool)YesRadioButton.IsChecked ? "Yes" : "No";
38     }

```

When I run the app and select the Yes RadioButton:



You can have two groups of radio buttons, and the selections will be unique within each of those groups. And we could add as many RadioButtons as we want but it typically only works well with four or five selections at most. If you have more than that, then you'll probably want to move onto a different type of control for input.

Next, the ComboBox which allows you to create a number of ComboBoxItems, each with their own value.

```
57     <TextBlock Grid.Row="3"
58         Text="ComboBox"
59         Name="MyComboBox"
60         VerticalAlignment="Center" />
61     <StackPanel Orientation="Horizontal"
62         Grid.Row="3"
63         Grid.Column="1"
64         Margin="20,10,0,10">
65         <ComboBox SelectionChanged="ComboBox_SelectionChanged" >
66             <ComboBoxItem Content="Fourth" />
67             <ComboBoxItem Content="Fifth" />
68             <ComboBoxItem Content="Sixth" IsSelected="True" />
69         </ComboBox>
70         <TextBlock Name="ComboBoxResultTextBlock" />
71     </StackPanel>
72
```

We're handling the SelectionChanged event so whenever somebody makes a different selection in the ComboBox, handle that and then take whatever the new selection was and display that in the label below or in the TextBlock below.

```
40     private void ComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
41     {
42         if (ComboBoxResultTextBlock == null) return;
43
44         var combo = (ComboBox)sender;
45         var item = (ComboBoxItem)combo.SelectedItem;
46         ComboBoxResultTextBlock.Text = item.Content.ToString();
47     }
```

When I run the app and select an ComboBoxItem:



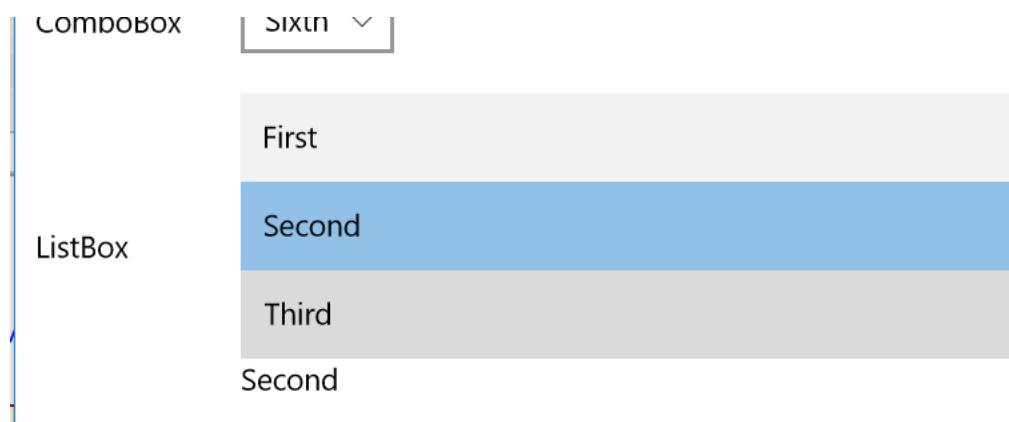
Next, the ListBox which will be a pretty important control when we are creating our hamburger navigation. There are a bunch of different ways that we can create that hamburger navigation that's popular in Windows 10. I choose the ListBox for reasons that I'll demonstrate in an upcoming lesson to follow up on a promise that I made earlier.

```
74      <TextBlock Grid.Row="4" Text="ListBox" VerticalAlignment="Center" />
75      <StackPanel Grid.Row="4" Grid.Column="1" Margin="20,10,0,10">
76          <ListBox Name="MyListBox"
77              SelectionMode="Multiple"
78              SelectionChanged="ListBox_SelectionChanged">
79              <ListBoxItem Content="First" />
80              <ListBoxItem Content="Second" />
81              <ListBoxItem Content="Third" />
82          </ListBox>
83          <TextBlock Name="ListBoxResultTextBlock" />
84      </StackPanel>
85
```

Notice that I set the SelectionMode to Multiple, meaning that the user can select more than one option. Then, I handle the SelectionChanged event and I get an array of selected items with a clever LINQ statement and join the items of the array together separated by a comma:

```
49      private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
50      {
51          var selectedItems = MyListBox.Items.Cast<ListBoxItem>()
52              .Where(p => p.IsSelected)
53              .Select(t => t.Content.ToString())
54              .ToArray();
55
56          ListBoxResultTextBlock.Text = string.Join(", ", selectedItems);
57      }
--
```

When I run the app, I can select one or more items to list them to see the selections in the TextBlock below.



Next, the Image control will be used in an upcoming exercise so I want to explain one of its most important properties. The Image control displays an image.

```
86             <TextBlock Grid.Row="5" Text="Image" VerticalAlignment="Center" />
87             <Image Source="Assets/logo.png"
88                 HorizontalAlignment="Left"
89                 Width="250"
90                 Height="50"
91                 Grid.Row="5"
92                 Grid.Column="1"
93                 Stretch="UniformToFill"
94                 Margin="20,10,0,10" />
95
```

First, I set the Source. In this case the Source is going to be relative to the root of our project. So we will look in the Assets folder for a logo.png. We could also supply it a location across the internet or some other folder in our application.

Next, I set the Stretch property. And there are a couple different options.

When the Stretch property is set to None, no stretching will be used. The image will be cropped to the available width and height of the control.

Image



When the Stretch property is set to Fill, the image will be stretched to use all available width and height of the control.

Image



When the Stretch property is set to Uniform, the image will be resized to fit completely in the available width and height meaning that it will be constrained by the smaller of the two dimensions so that there's no distorting the image:

Image



Finally, when the Stretch property is set to UniformToFill, the image will be resized to fit the in the available width OR height, but not both. In this case, the image is constrained by the height but it is displayed in full width.

Image



Next, the ToggleButton which is a “state-ful” Button, so you can click it on and you can click it off. There is also an option to make it a ThreeState button, so you can click it and give it no value, click it on or click it off. And so I've set the IsThreeState="True" so that we can see it at work.

```
96      <TextBlock Grid.Row="7" Text="ToggleButton" VerticalAlignment="Center" />
97      <StackPanel Orientation="Horizontal"
98          Grid.Row="7"
99          Grid.Column="1"
100         Margin="20,10,0,10" >
101         <ToggleButton Name="MyToggleButton"
102             Content="Premium Option"
103             IsThreeState="True"
104             Click="MyToggleButton_Click" />
105         <TextBlock Name="ToggleButtonResultTextBlock" />
106     </StackPanel>
107
```

Furthermore I handle the Click event where I grab the current value of IsChecked and display it in the TextBlock:

```
59     private void MyToggleButton_Click(object sender, RoutedEventArgs e)
60     {
61         ToggleButtonResultTextBlock.Text = MyToggleButton.IsChecked.ToString();
62     }
63 }
```

When I run the application I can toggle through the various states, such as the “on” state:

ToggleButton      Premium Option      True

Then the last control that we will look at is similar to the ToggleButton that except it's called the ToggleSwitch. The ToggleSwitch allows us to toggle between two states. And we see this control used often in the Settings panel in Windows 10. What you can do that's interesting is actually set a Content

area for what should be displayed whenever the Toggle is in an off position or an on position. You can also handle other events but this will be primarily how it's used to display the current state of the toggle.

```
108     <TextBlock Grid.Row="8"  
109         Text="ToggleSwitch"  
110         VerticalAlignment="Center" />  
111     <StackPanel Grid.Row="8"  
112         Grid.Column="1"  
113         Margin="20,10,0,10" >  
114         <ToggleSwitch>  
115             <ToggleSwitch.OffContent>  
116                 <TextBlock Text="I'm off right now." />  
117             </ToggleSwitch.OffContent>  
118             <ToggleSwitch.OnContent>  
119                 <TextBlock Text="I'm on!" />  
120             </ToggleSwitch.OnContent>  
121         </ToggleSwitch>  
122     </StackPanel>  
123
```

When I run the application and toggle the ToggleSwitch the TextBlock is set to either "I'm off right now." or "I'm on!"



## UWP-021 - Implementing a Simple Hamburger Navigation Menu

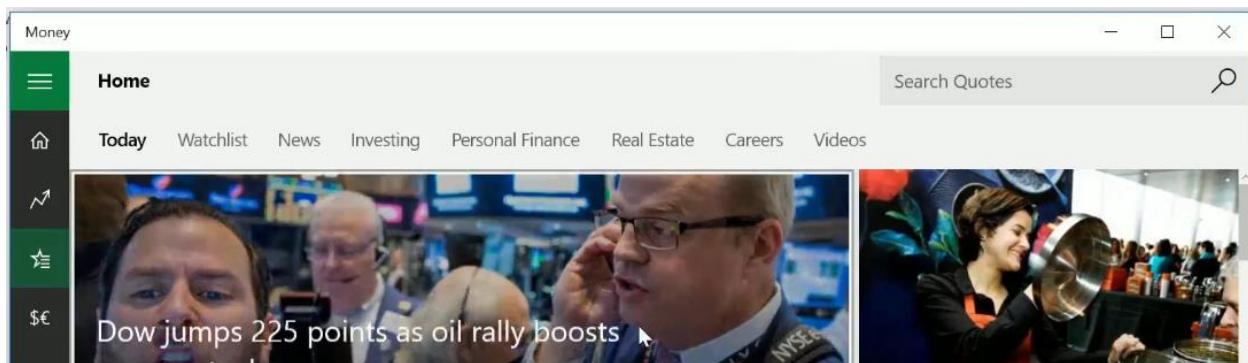
In this lesson we are going to combine several ideas that we covered in previous lessons to create what I call a "poor man's" hamburger navigation Layout. Why am I calling it a "poor man's" version? Because I'm trying to give you the simplest, cleanest, clearest path so that you can implement hamburger navigation right away in your Windows 10 apps.

There is a "rich man's" version of this detailed at Jerry Nixon's blog here at the shortcut on screen.

<http://bit.do/hamburger-nav>

Like I said, Jerry Nixon works for Microsoft. He has done an awesome job with this article: "Implementing an Awesome Hamburger Button with XAML's SplitView Control." The reason why I would call it a "rich man's" version is because it's more robust. It includes many additional features. But it's also a bit more complex if you are, indeed, an absolute beginner. Just keep that in mind. You might want to visit it and bookmark it for later, maybe after we go through several more lessons. You'll be able to use his version instead of mine.

What I want to do now is to recreate the hamburger layout that we saw in the Money app.



When I click on the hamburger little icon at the top, the SplitView opens up. When I click it again, it closes and all we can see is the icons.

Furthermore, whatever we currently select, you can see that one of the secondary colors, in this case, a darker green color, highlights the fact that we're no longer on the Home tab but rather we're on what I guess is the Watchlist.

I've created a new project named HamburgerExample. All I've done prior to creating this lesson is just open the App.xaml.cs and I removed the Frame counter.

In the MainPage.xaml I create a couple of RowDefinitions setting the Height="Auto" in the Row one and setting the Height="\*" on Row two. Next, I'll create a RelativePanel.

```

10   <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11     <Grid.RowDefinitions>
12       <RowDefinition Height="Auto" />
13       <RowDefinition Height="*" />
14     </Grid.RowDefinitions>
15     <RelativePanel>
16       | 
17
18     </RelativePanel>
19
20   </Grid>
21 </Page>

```

Inside of the RelativePanel I'll add a Button. I'll come back to that in a moment, but that will be our hamburger button.

Beneath that I'll add a SplitView and place it into Grid.Row="1" -- that second Row that takes up the \* size. This is where we're going to do the majority of our work.

```

10   <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11     <Grid.RowDefinitions>
12       <RowDefinition Height="Auto" />
13       <RowDefinition Height="*" />
14     </Grid.RowDefinitions>
15     <RelativePanel>
16       <Button />
17     </RelativePanel>
18     <SplitView Grid.Row="1">
19       | 
20     </SplitView>
21
22   </Grid>
23 </Page>

```

Inside of this SplitView I'm going to use a ListBox control for all the reasons I talked about in that previous lesson. You'll see it all come together here in just a moment. The ListBox is going to require us to create a number of ListBoxItems. I know we're going to have several of those as well. I'll just copy and paste a couple.

```

15   <RelativePanel>
16     <Button />
17   </RelativePanel>
18   <SplitView Grid.Row="1">
19     <ListBox>
20       <ListBoxItem />
21       <ListBoxItem />
22       <ListBoxItem />
23     </ListBox>
24   </SplitView>
25

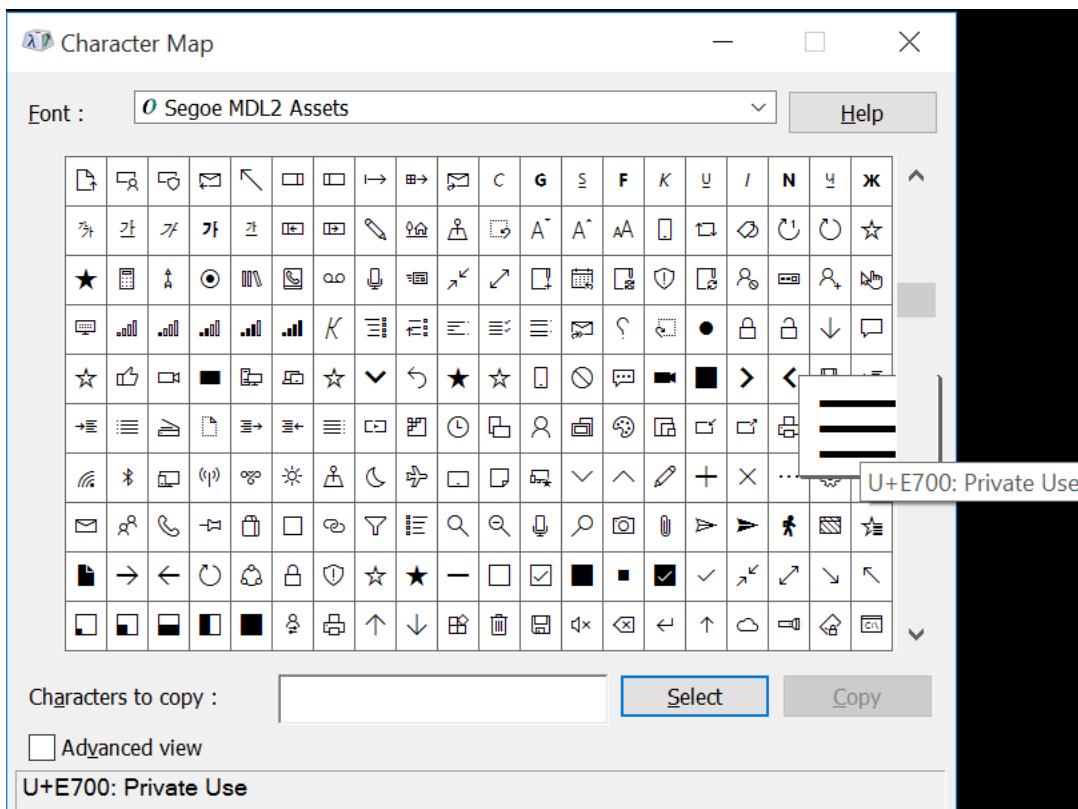
```

That's our general layout. Now we just have to fill in the details.

The ListBox will sit inside of the Pane of the SplitView. Each ListBox item will have an icon on the left and then Text on the right. As you select an item from the ListBox, we will handle the SelectionChanged event and if this were a real app, we would build our functionality or display our cards in the main Content area.

I'll start with the Button control and name it HamburgerButton. Then I'll choose an icon that represents the hamburger. You'll recall that it has three vertical lines stacked above each other. To do that we're going to find that there are a specific FontFamily available on Windows 10 devices called Segoe MDL2 Assets.

I open the Character Map application the Cortana search bar or and searching for "Character Map". Once it's open I'll select the font: Segoe MDL2 Assets. In the lower area I will search for the hamburger symbol. You see a capital U+E700 and it's marked for private use. The only thing that's important to us is the "E700" part.



Back in the MainPage.xaml, I set the FontFamily to Segoe MDL2 Assets the Content="E700". I'll add the prefix:

```
&#x
```

And a semi-colon as a suffix. So the entire content is:

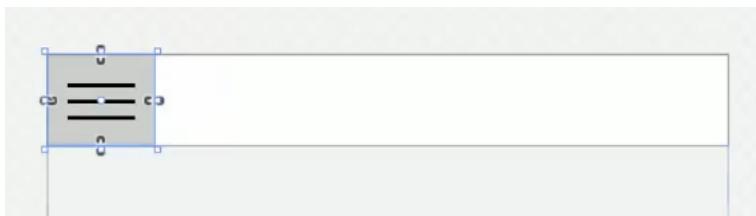
```
Content="꜀"
```

I'm also going to set the FontSize to 36. That might be a bit large for a real application. Finally I'll add a Click event handler.

```
16 <RelativePanel>
17   <Button Name="HamburgerButton"
18     FontFamily="Segoe MDL2 Assets"
19     Content="&#xE700;"
20     FontSize="36"
21     Click="HamburgerButton_Click" />
22 </RelativePanel>
```

I think we're going to add the Click EventHandler for it to open and close it.

In the designer we can see we've got our button with the hamburger icon on it.



We could style up that button to be one of the primary colors for our apps, however I'm not going to do that in this particular example.

Next, I'll focus on the SplitView. I give it the name "MySplitView". I set DisplayMode="CompactOverlay" because I want to show the icons beneath it.

When the Pane is open, I'll set the width to 200. When it's closed set the CompactPanelLength="56". Also, I set the HorizontalAlignment="Left".

```
24 <SplitView Name="MySplitView"
25   Grid.Row="1"
26   DisplayMode="CompactOverlay"
27   OpenPaneLength="200"
28   CompactPanelLength="56"
29   HorizontalAlignment="Left">
30
```

Next, I'll focus on the ListBox where I will display the ListBoxItems. I'll set the SelectionMode="Single" since I only want one item selected at a time. I'll set the Name="IconsListBox". Then I'll create a SelectionChanged event. I'll come back to that later.

Next, for each ListViewItem I'll add a StackPanel and two TextBlocks inside of the StackPanel. The StackPanel will be oriented horizontally. The first TextBlock will contain the icon. The second TextBlock will contain the text.

Where will we get the icon from? Once again, we're going to use the `FontFamily="Segoe MDL2 Assets"` then set the `FontSize="36"`. Finally, we're going to set the Text equal to and new icon from the Character Map. I'll choose E72D. So, I'll set the `Content=""`.

On the second TextBlock I'll set the `Text="Share"` (just random text ... don't read too much into that). I'll also give the ListViewItem the name "ShareListBoxItem". I'll set the `FontSize="24"` and add some margin to the left.

I'll do the same thing for the other ListViewItem. I'll copy and paste the first ListViewItem, and change out the `Text` property to "``" and "Favorites", respectively. I'll name this ListViewItem "FavoritesListBoxItem".

```
31 <SplitView.Pane>
32     <ListBox SelectionMode="Single"
33         Name="IconsListBox"
34         SelectionChanged="IconsListBox_SelectionChanged">
35         <ListBoxItem Name="ShareListBoxItem">
36             <StackPanel Orientation="Horizontal">
37                 <TextBlock FontFamily="Segoe MDL2 Assets"
38                     FontSize="36"
39                     Text="" />
40                 <TextBlock Text="Share"
41                     FontSize="24"
42                     Margin="20,0,0,0" />
43             </StackPanel>
44         </ListBoxItem>
45         <ListBoxItem Name="FavoritesListBoxItem">
46             <StackPanel Orientation="Horizontal">
47                 <TextBlock FontFamily="Segoe MDL2 Assets"
48                     FontSize="36"
49                     Text="" />
50                 <TextBlock Text="Favorites"
51                     FontSize="24"
52                     Margin="20,0,0,0" />
53             </StackPanel>
54         </ListBoxItem>
55     </ListBox>
56
57     </SplitView.Pane>
```

Now I'll back up to the HamburgerButton for a moment. Whenever someone clicks the button I will show or hide the SplitView's Pane. Using F12 on my keyboard I create an event handler method stub in the MainPage.xaml.cs code behind.

I write the following code:

```
~  
30     private void HamburgerButton_Click(object sender, RoutedEventArgs e)  
31     {  
32         MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen;  
33     }  
--
```

Nearing the end, I'll set the SplitView's Content. For our purposes I'm just going to put a TextBlock in the Content section, name it "ResultTextBlock".

To change the content of the ResultTextBlock I'll implement the IconsListBox\_SelectionChanged event. I'll put my mouse cursor on the name and press F12. In the code behind, I'll add the following:

```
~  
35     private void IconsListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)  
36     {  
37         if (ShareListBoxItem.isSelected) { ResultTextBlock.Text = "Share"; }  
38         else if (FavoritesListBoxItem.isSelected) { ResultTextBlock.Text = "Favorites"; }  
39     }  
40 }
```

Obviously this is the simplest case possible. If this were a real app would probably want to implement navigation and load in another Page into a Frame and that Frame would sit in the middle where we currently have the Content area.

## UWP-022 - Cheat Sheet Review: Windows 10 Layout Hamburger Navigation and Controls

### UWP-017 - XAML Layout with the RelativePanel

It basically defines an area within which you can position and align child objects

- in relation to each other
- or in relation to the parent panel.

Controls use attached properties to position themselves.

1. Panel alignment relationships (AlignTopWithPanel, AlignLeftWithPanel, ...) are applied first.
2. Sibling alignment relationships (AlignTopWith, AlignLeftWith, ...) are applied second.
3. Sibling positional relationships (Above, Below, RightOf, LeftOf) are applied last.

```
<RelativePanel MinHeight="300" Grid.Row="1">
    <Rectangle Name="RedRectangle" RelativePanel.AlignRightWithPanel="True" />
    <Rectangle RelativePanel.LeftOf="RedRectangle" />
</RelativePanel>
```

### UWP-018 - XAML Layout with the SplitPanel

The split view allows us to create a panel that can be displayed or hidden.

We would use the SplitView to implement hamburger navigation.

There are two parts to a SplitView:

1. The part that is hidden by default (Pane)
2. The part that is shown by default (Content)

You define other controls inside of the SplitView.Pane and SplitView.Content.

```
<SplitView Name="MySplitView"
    CompactPaneLength="50"
    IsPaneOpen="False"
    DisplayMode="CompactInline"
    OpenPaneLength="200" >
    <SplitView.Pane>
    </SplitView.Pane>
    <SplitView.Content>
    </SplitView.Content>
</SplitView>
```

DisplayMode Options:

**Inline** – Panel completely covers content. When expanded, panel pushes content.

**CompactInline** – Pane covers most of the Content. When expanded, panel pushes content.

**Overlay** – Panel completely covers content. When expanded, panel covers content.

**CompactOverlay** – Panel covers most of the content. When expanded, panel covers content.

Open / Close Pane in C#:

```
MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen;
```

### UWP\_019 - Working with Navigation

Hierarchy:

App > Window > Frame > MainPage

You can load pages into a child frame or into the root frame:

```
Frame.Navigate(typeof(Page2), additionalParameter);
```

You can retrieve additionalParameter on the page you navigated to:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    value = (string)e.Parameter;
}
```

Traverse back stack (history):

```
if (Frame.CanGoBack) {
    Frame.GoBack();
}

if (Frame.CanGoForward) {
    Frame.GoForward();
}
```

Create a global variable by declaring a static internal field in the App class definition.

### UWP-020 - Common XAML Controls - Part 1

```
<CheckBox Name="MyCheckBox" Content="Agree?" Tapped="MyCheckBox_Tapped" />
```

```
CheckBoxResultTextBlock.Text = MyCheckBox.IsChecked.ToString();
```

```

<RadioButton Name="YesRadioButton" Content="Yes" GroupName="MyGroup"
checked="RadioButton_Checked" />
<RadioButton Name="NoRadioButton" Content="No" GroupName="MyGroup"
Checked="RadioButton_Checked" />

RadioButtonTextBlock.Text = (bool)YesRadioButton.IsChecked ? "Yes" : "No";

<ComboBox SelectionChanged="ComboBox_SelectionChanged" >
  <ComboBoxItem Content="Fourth" />
  <ComboBoxItem Content="Fifth" />
  <ComboBoxItem Content="Sixth" IsSelected="True" />
</ComboBox>

if (ComboBoxResultTextBlock == null) return;
var combo = (ComboBox)sender;
var item = (ComboBoxItem)combo.SelectedItem;
ComboBoxResultTextBlock.Text = item.Content.ToString();

<ListBox Name="MyListBox" SelectionMode="Multiple"
SelectionChanged="ListBox_SelectionChanged">
  <ListBoxItem Content="First" />
  <ListBoxItem Content="Second" />
  <ListBoxItem Content="Third" />
</ListBox>

var selectedItems = MyListBox.Items.Cast<ListBoxItem>()
  .Where(p => p.IsSelected)
  .Select(t => t.Content.ToString())
  .ToArray();

ListBoxResultTextBlock.Text = string.Join(", ", selectedItems);

<Image Source="Assets/logo.png" Stretch="UniformToFill" />

<ToggleButton Name="MyToggleButton" Content="Premium Option" IsThreeState="True"
Click="MyToggleButton_Click" />

ToggleButtonResultTextBlock.Text = MyToggleButton.IsChecked.ToString();

```

```
<ToggleSwitch>
  <ToggleSwitch.OffContent>
    <TextBlock Text="I'm off right now." />
  </ToggleSwitch.OffContent>
  <ToggleSwitch.OnContent>
    <TextBlock Text="I'm on!" />
  </ToggleSwitch.OnContent>
</ToggleSwitch>
```

#### UWP-021 - Implementing a Simple Hamburger Navigation Menu

Jerry Nixon's Example: <http://bit.do/hamburger-nav>

Use Character Map to find the code to display icons using Segoe MDL2 Assets.

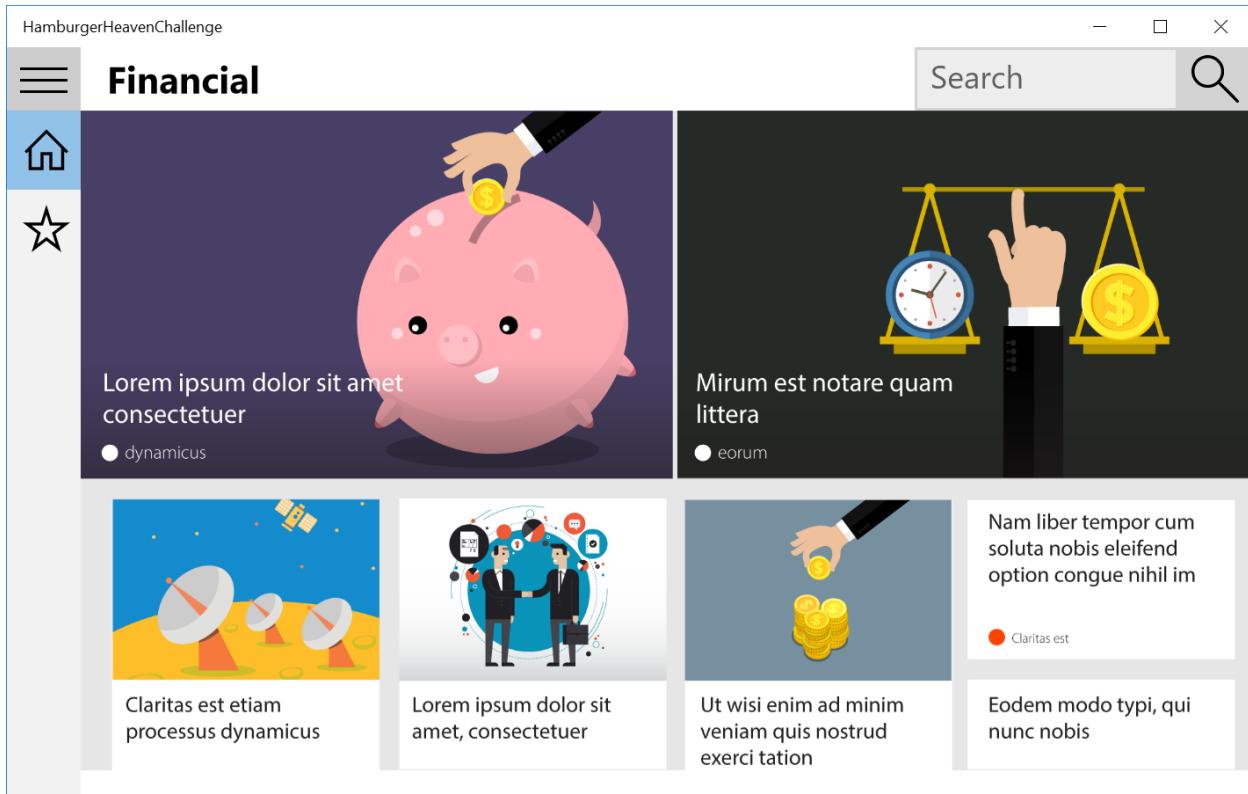
Hamburger: &#xE700;

Use ListBox and ListBoxItems for the navigation links inside of a SplitView.

## UWP-023 - Hamburger Heaven Challenge

The next challenge is called the “Hamburger Heaven Challenge”. This challenge will require you create a hamburger-style navigation for a fictitious Windows 10 application.

You will create two main navigation areas across the top. To the right, search area. To the left, hamburger button.



When I click it, the hamburger button SplitView's Panel will be fully displayed

HamburgerHeavenChallenge

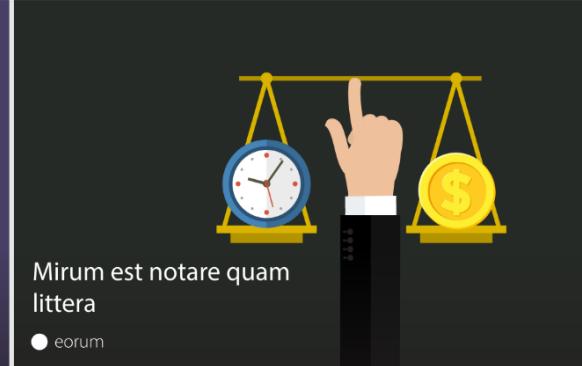
## Financial

Search 

-  Home
-  Favorites



Mirum est notare quam littera  
eorum



Nam liber tempor cum soluta nobis eleifend option congue nihil im  
Claritas est



Ut wisi enim ad minim veniam quis nostrud exerci tation



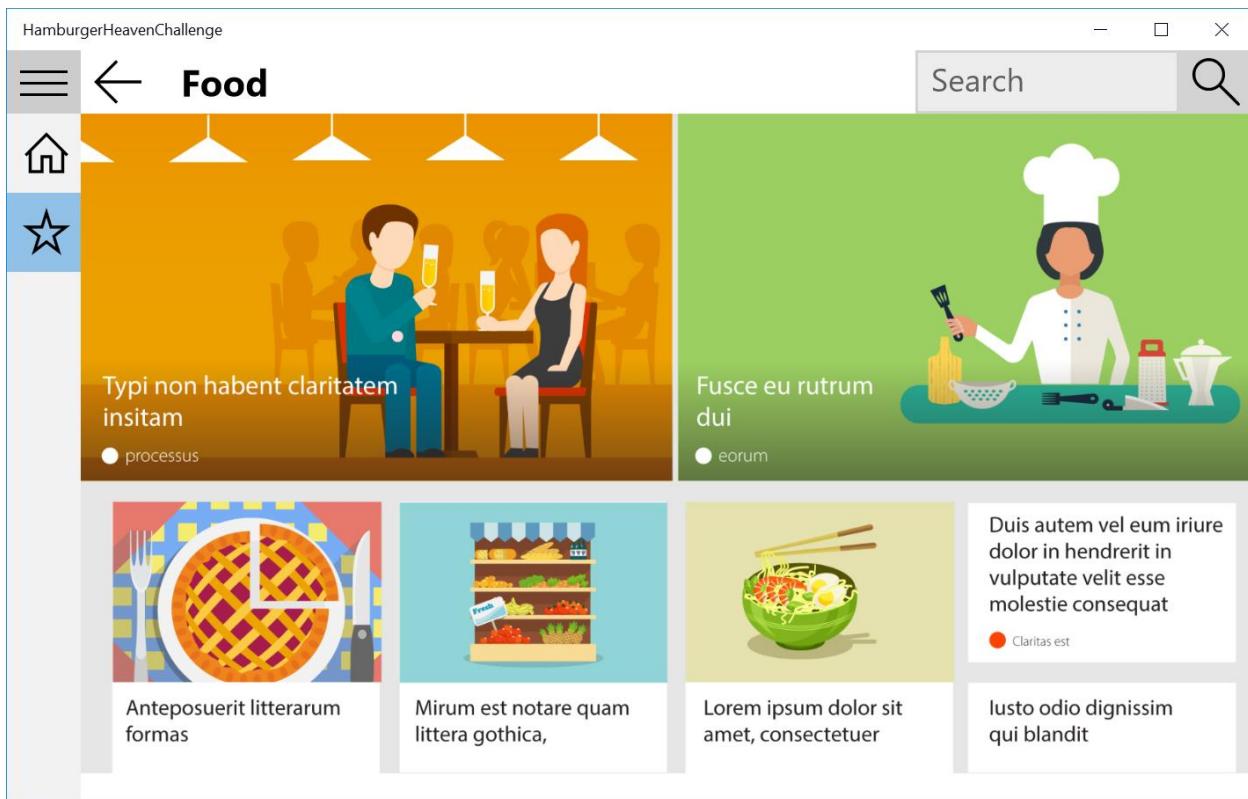
Claritas est

Eodem modo typi, qui nunc nobis

Color sit amet, consectetuer

n nicus

Furthermore I can select one of two primary sections of my application. When I go to the Financial page, you see that that's the Homepage for the application. When I change over to the Food page, notice that not only does the title change, but also I get this little icon that allows me to go back to the Financial page, the Homepage.



The colorful Content areas are a single image. All you'll need to do is display either the Financial.png or Food.png file that is available in the zipped folder associated with this lesson. You will host the images on Financial.xaml, and a Food.xaml, respectively. You'll use Pages in order to utilize the Frame's BackStack and the Navigation feature.

Also, in the zipped folder you will find the list of requirements in the text file named Instructions.txt. I'll discuss each of the items in that folder in the remainder of this lesson.

First, you'll create a folder named Assets and add the Food.png and the Financial.png to it.

Next, you'll add two Pages to the project named a Food.xaml and a Financial.xaml. And each of these pages will host a single Image control set to Food.png or Financial.png, respectively.

You'll add a Frame inside of this outermost MainPage.xaml

You'll implement the hamburger style navigation using a SplitView.

You'll display a Search bar in the upper right-hand corner and a hamburger button in the upper left-hand corner, and a back button next to that, and the title of the Page.

The Search area is for display only (it requires no C# code).

When the user navigates from Financial to Food the back arrow button should appear in the proper position on the Page. It should disappear once you return to the Financial page (which will be considered 'home').

In the Panel of the SplitView, choose any icon you want. Use the Character Map application to find an icon that you want to use.

The MainPage.xaml will host a Frame. When the application starts, the Financial.xaml page will be loaded into the Frame.

The search button should utilize the search icon. The back button must use a back arrow icon.

Try to figure out how to set the placeholder text "Search" in the TextBox. Note: I did not discuss this in the lesson material, however there is a special property that allows you to set that word "Search" in the TextBox but as soon as the user begins typing, that word will go away.

Font sizes that I used include 24, 28, and 36.

The colors that I use will include LightGrays, alright, and maybe a MediumGray.

Use these images as your guide for margins, font sizes and colors.

## UWP-024 - Hamburger Heaven Challenge: Solution

Step 1: Create a new Project using the “Blank App, Universal Windows” template named “HamburgerHeavenChallenge”

Step 2: Add an Assets folder to the project, drag and drop the Food.png and Financial.png to the Assets folder.

Step 3: Add two Pages to your app, Food.xaml and Financial.xaml. And each of these will host a single Image control. We will set the source to Food.png and Financial.png.

Go to the Project menu, Add New Item, make sure that I add a Blank Page. Name the first one Financial.xaml and the second one Food.xaml.

Add an Image control, set the Source="Assets/Food.png" and the VerticalAlignment="Top" just to make sure that it butts itself up against the top.

Do the same for Financial.xaml setting the Source="Assets/Financial.png"

Step 4: In the App.xaml.cs remove the Frame counter.

Step 5: In MainPage.xaml create two Rows. The first one height should be Auto, the second height should be "\*".

Step 6: Add a RelativePanel. Inside the RelativePanel:

First, add a Button named HamburgerButton.

Second, add a Button named BackButton.

Third, add a TextBlock named TitleTextBlock.

Fourth, add a Button named SearchButton.

Fifth, add a TextBox named SearchTextBox.

Sixth, set the HamburgerButton’s RelativePanel.AlignLeftWithPanel="True". Set the FontFamily="Segoe MDL2 Assets", the FontSize="36", Content="&#xE700;". Finally, add an event handler for Click named HamburgerButton\_Click.

Seventh, set the BackButton’s RelativePanel.RightOf="HamburgerButton". Set the FontFamily="Segoe MDL2 Assets". Set the FontSize="36". Set the Content="&#xE0C4;". Add a click event handler.

Eighth, set the TitleTextBlock’s RelativePanel.RightOf="BackButton". Set its FontSize="28". Set the FontWeight="Bold".

Ninth, set the SearchButton’s to RelativePanel.AlignRightWithPanel="True". Set the FontFamily="Segoe MDL2 Assets". Set the FontSize="36". Set the Content="&#xE1A3;"

Tenth, set the SearchTextBox’s RelativePanel.LeftOf="SearchButton". Set its Height="48". Set the Width="100". Set the FontSize="24". Set the PlaceholderText="Search".

```
10    <Grid>
11        <Grid.RowDefinitions>
12            <RowDefinition Height="Auto" />
13            <RowDefinition Height="*" />
14        </Grid.RowDefinitions>
15
16        <RelativePanel>
17            <Button Name="HamburgerButton"
18                RelativePanel.AlignLeftWithPanel="True"
19                FontFamily="Segoe MDL2 Assets"
20                FontSize="36"
21                Content=""
22                Click="HamburgerButton_Click" />
23
24            <Button Name="BackButton"
25                RelativePanel.RightOf="HamburgerButton"
26                FontFamily="Segoe MDL2 Assets"
27                FontSize="36"
28                Content=""
29                Click="BackButton_Click"
30            />
```

```

-- 
32   <TextBlock Name="TitleTextBlock"
33     RelativePanel.RightOf="BackButton"
34     FontSize="28"
35     FontWeight="Bold"
36     Margin="20,5,0,0"/>
37
38   <Button Name="SearchButton"
39     RelativePanel.AlignRightWithPanel="True"
40     FontFamily="Segoe MDL2 Assets"
41     FontSize="36"
42     Content="" />
43
44   <TextBox Name="SearchTextBox"
45     RelativePanel.LeftOf="SearchButton"
46     Height="48"
47     Width="200"
48     FontSize="24"
49     PlaceholderText="Search" />
50 </RelativePanel>
51
52 </Grid>
53 </Page>

```

Step 7: Add a SplitView control and name it “MySplitView “. Set the SplitView’s Grid.Row=”1”. Inside of the SplitView, create a SplitView.Pane, and then a SplitView.Content.

Set DisplayMode=”CompactOverlay”. Set OpenPanelLength=”200”. Set the CompactPaneLength=”56”

Inside of SplitView.Content add a Frame named “MyFrame”.

```
52     <SplitView Grid.Row="1"  
53         Name="MySplitView"  
54         DisplayMode="CompactOverlay"  
55         OpenPaneLength="200"  
56         CompactPaneLength="56">  
57     <SplitView.Pane>  
58     ~~~  
59     </SplitView.Pane>  
60     <SplitView.Content>  
61         <Frame Name="MyFrame"></Frame>  
62     </SplitView.Content>  
63  
64 </SplitView>
```

Step 8: Inside of the SplitView's Pane add a ListBox with two ListBoxItems. Set the ListBox's SelectionMode="Single".

Step 9: In the first ListBoxItem, add a StackPanel and set the Orientation ="Horizontal". This will allow us to add two TextBlocks next to each other ... the first for the icon, the second for the text.

In the first TextBlock, set the FontFamily="Segoe MDL2 Assets", and set the FontSize="36". I set the Content"=&#xE80F;"

In the second TextBlock set the Text="Financial". Set the FontSize="24".

I copy the entire ListBoxItem and paste it below the first ListBoxItem. I'll change the Content properties for the two TextBlocks. In the first TextBlock set the Content="&#xE1CE;" and in the second TextBlock set the Content="Food".

```

57     <SplitView.Pane>
58         <ListBox SelectionMode="Single"
59             SelectionChanged="ListBox_SelectionChanged">
60             <ListBoxItem Name="Financial">
61                 <StackPanel Orientation="Horizontal">
62                     <TextBlock FontFamily="Segoe MDL2 Assets"
63                         FontSize="36"
64                         Text="" />
65                     <TextBlock FontSize="24"
66                         Margin="20,0,0,0">Financial</TextBlock>
67                 </StackPanel>
68             </ListBoxItem>
69             <ListBoxItem Name="Food">
70                 <StackPanel Orientation="Horizontal">
71                     <TextBlock FontFamily="Segoe MDL2 Assets"
72                         FontSize="36"
73                         Text="" />
74                     <TextBlock FontSize="24"
75                         Margin="20,0,0,0">Food</TextBlock>
76                 </StackPanel>
77             </ListBoxItem>
78         </ListBox>
79     </SplitView.Pane>

```

Step 9: Next we'll implement the click event of the HamburgerButton to open and close the MySplitView's Pane. I put my mouse cursor inside of the event handler name and press hit F12 on my keyboard. Add the following code to the method stub:

```

33     private void HamburgerButton_Click(object sender, RoutedEventArgs e)
34     {
35         MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen;
36     }
37

```

Step 10: Next we'll navigate directly to the Financial page when the app starts. We'll also set the TitleTextBlock to indicate the page we're currently on.

```

24
25     public MainPage()
26     {
27         this.InitializeComponent();
28
29         MyFrame.Navigate(typeof(Financial));
30         TitleTextBlock.Text = "Financial";
31     }
-->

```

Step 11: Next we'll implement the `ListBox_SelectionChanged` event to allow the user to select one of the two items in our `SplitView`'s Pane:

```
47     private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
48     {
49         if (Financial.IsSelected)
50         {
51             BackButton.Visibility = Visibility.Collapsed;
52             MyFrame.Navigate(typeof(Financial));
53             TitleTextBlock.Text = "Financial";
54         }
55         else if (Food.IsSelected)
56         {
57             BackButton.Visibility = Visibility.Visible;
58             MyFrame.Navigate(typeof(Food));
59             TitleTextBlock.Text = "Food";
60         }
61     }
```

Notice in the code that we're not only navigating to the proper page, but also changing the `TitleTextBlock` and the `BackButton`'s visibility depending on the selection.

Step 12: Next I'll implement the `BackButton`'s functionality. We'll check whether the `MyFrame` can go back and if it can, then not only should it navigate, but also select the `Financial` `ListBoxItem`:

```
38     private void BackButton_Click(object sender, RoutedEventArgs e)
39     {
40         if (MyFrame.CanGoBack)
41         {
42             MyFrame.GoBack();
43             Financial.IsSelected = true;
44         }
45     }
```

Step 13: Finally, I'll revisit `MainPage()` to ensure that the `BackButton`'s visibility is collapsed when the app first opens. Furthermore, since we're navigating to the `Financial` page, we want to make sure the `ListBoxItem` is selected.

```
25     public MainPage()
26     {
27         this.InitializeComponent();
28
29         MyFrame.Navigate(typeof(Financial));
30         TitleTextBlock.Text = "Financial";
31
32         BackButton.Visibility = Visibility.Collapsed;
33         Financial.IsSelected = true;
34     }
35
```

## UWP-025 - Common XAML Controls - Part 2

In this lesson we will add more controls to our repertoire taking the approach from our previous examination of XAML controls whereby I copied and pasted XAML defining the control into the XAML editor, talked about the Control itself, explained how to access the features and functions of that Control and so on.

I'll begin by creating a new project named ControlsExample2, then in the MainPage.xaml I add Row and Column Definitions to create a structure for our MainPage.

```
10     <Grid Margin="20,20,0,0">
11         <Grid.RowDefinitions>
12             <RowDefinition Height="Auto" />
13             <RowDefinition Height="Auto" />
14             <RowDefinition Height="Auto" />
15             <RowDefinition Height="Auto" />
16             <RowDefinition Height="Auto" />
17             <RowDefinition Height="Auto" />
18             <RowDefinition Height="Auto" />
19             <RowDefinition Height="Auto" />
20             <RowDefinition Height="Auto" />
21             <RowDefinition Height="*" />
22         </Grid.RowDefinitions>
23         <Grid.ColumnDefinitions>
24             <ColumnDefinition Width="Auto" />
25             <ColumnDefinition Width="*" />
26         </Grid.ColumnDefinitions>
27
28     </Grid>
29 </Page>
~~
```

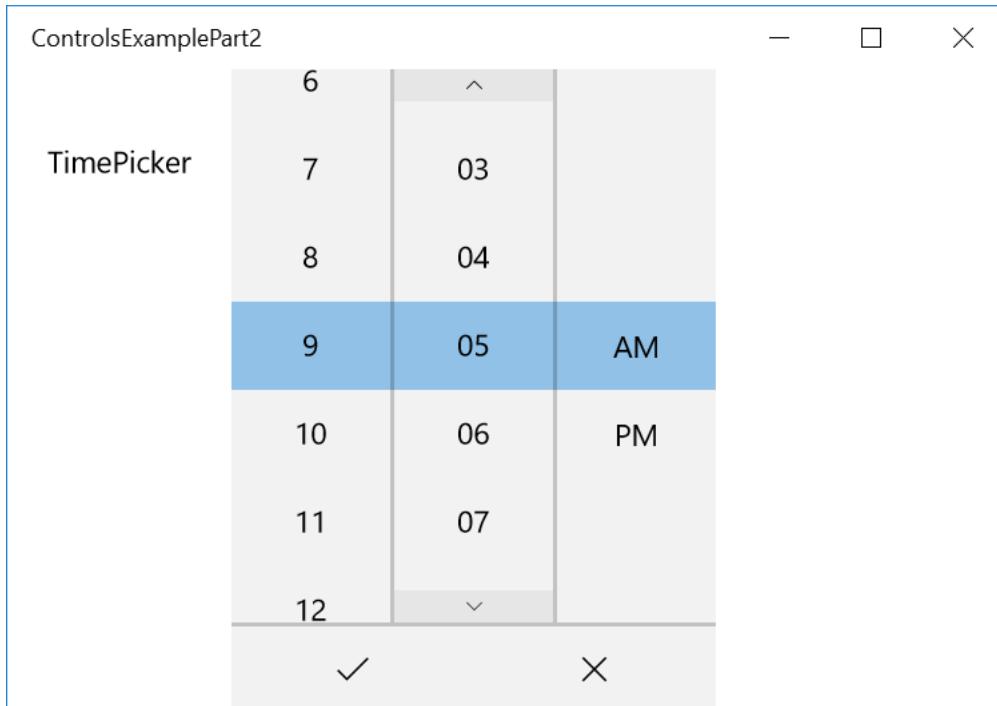
We will start by talking about Controls that allow user input for times and dates. The first Control that I want to call your attention to is the TimePicker. This allows a user to select an hour and a minute as input for your application. Suppose I want to create an app that allows the user to create a reminder at 9:30 PM. The TimePicker control allows your user to easily select that whether with touch or with mouse.

```

28 <TextBlock Grid.Row="0" Text="TimePicker" VerticalAlignment="Center" />
29 <TimePicker Grid.Row="0"
30     Grid.Column="1"
31     ClockIdentifier="12HourClock"
32     Margin="20,0,0,20" />

```

The most interesting property is the `ClockIdentifier` property which allows for two potential values: either a `12HourClock` that will allow the user to select AM/PM, or a `24HourClock`, which in the United States, is known as military time. So I choose the 12-hour clock, and run debug the app:



As I select the Control itself, it shows a nice pop-up that would look good in a mobile device, as well as a desktop. And here I can select different times and choose AM/PM and then click the checkmark for acceptance or the X to cancel.

Once I've made my selections and clicked the checkmark, the pop-up selector disappears and the chosen time is displayed.

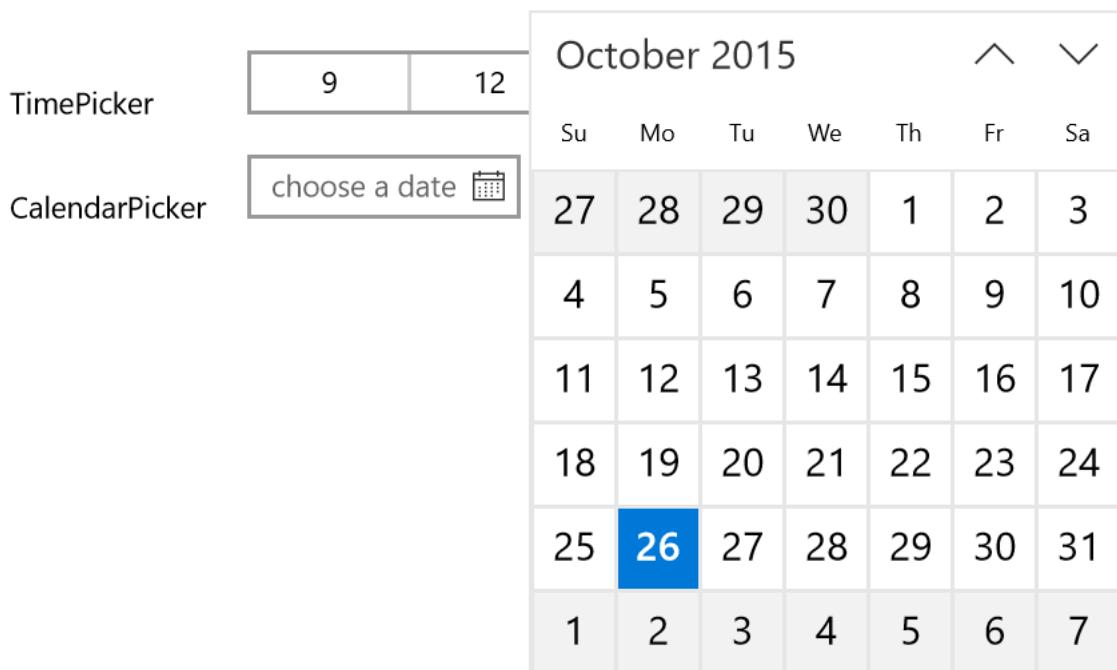
The next two controls relate to dates. The first is called a `CalendarDatePicker`. When you hear the term "Picker", typically you're going to see a fly-out, or rather, a pop-up appear. In other words, you're not going to see the entire body of the Control, you'll just see the default value, but when click on that control it will show you the pop-up with all available selections. In this case, you'll see a Calendar pop up.

```

34 <TextBlock Grid.Row="1"
35     Text="CalendarPicker"
36         VerticalAlignment="Center"/>
37 <CalendarDatePicker
38     Grid.Row="1"
39     Grid.Column="1"
40     Margin="20,0,0,20"
41     PlaceholderText="choose a date" />
42

```

When we run the app you can see the default value displayed. In this case, we set the PlaceHolderText attribute. When you tap the control it displays a calendar fly-out:



In the Properties area, you can see that there are a number of interesting Properties like the type of Calendar that we will use. By default, in the United States, it's set to GregorianCalendar. However, you can see that there are many different types of Calendars for various Asian countries. There is a HebrewCalendar and others as well.

The other Calendar Control is the CalendarView. And when you hear the word "view", typically, in XAML controls, it means that the interactive / selection area will be displayed all of the time instead of hidden in a fly-out.

```
43     <TextBlock Grid.Row="2" Text="CalendarView" VerticalAlignment="Center" />
44     <StackPanel Grid.Row="2"
45         Grid.Column="1"
46         Margin="20,0,0,20"
47         HorizontalAlignment="Left">
48         <CalendarView Name="MyCalendarView"
49             SelectionMode="Multiple"
50             SelectedDatesChanged="MyCalendarView_SelectedDatesChanged" />
51         <TextBlock Name="CalendarViewResultTextBlock" />
52     </StackPanel>
53
```

As you can see, I'm handling the SelectedDatesChanged event. I've also set the SelectionMode to "Multiple". I can retrieve all of the selected dates by retrieving the CalendarView's SelectedDates property. Using a clever LINQ statement I can project out the way I want the date to appear (in this case, as a string in the form Month/Day) and then call ToArray() to work with those selections as an array of strings:

```
34     private void MyCalendarView_SelectedDatesChanged(
35         CalendarView sender,
36         CalendarViewSelectedDatesChangedEventArgs args)
37     {
38         var selectedDates = sender.SelectedDates
39             .Select(p => p.Date.Month.ToString() + "/" + p.Date.Day.ToString())
40             .ToArray();
41
42         var values = string.Join(", ", selectedDates);
43         CalendarViewResultTextBlock.Text = values;
44     }
45
```

To display the values I'll use the string.Join method adding a comma as a separator.

Next we'll talk about the Flyout Control. The Flyout will display a message box. Inside that message box you can add any XAML markup you want to display in the pop-up. The Flyout is displayed as a result of some trigger. Some controls have a Flyout property which produces the necessary trigger to display the flyout. In the following case, we'll use the Button's Flyout property in property element syntax to define the Flyout and the content inside of it:

```

55      <TextBlock Grid.Row="3" Text="Flyout" VerticalAlignment="Center" />
56      <Button Name="MyFlyoutButton"
57          Margin="20,0,0,20"
58          Grid.Row="3"
59          Grid.Column="1"
60          Content="Flyout">
61      <Button.Flyout>
62      <Flyout x:Name="MyFlyout">
63          <StackPanel Margin="20,20,20,20">
64              <TextBlock Text="I just flew out to say I love you."
65                  Margin="0,0,0,10" />
66              <Button Name="InnerFlyoutButton"
67                  HorizontalAlignment="Right"
68                  Content="OK"
69                  Click="InnerFlyoutButton_Click" />
70          </StackPanel>
71      </Flyout>
72  </Button.Flyout>
73 </Button>

```

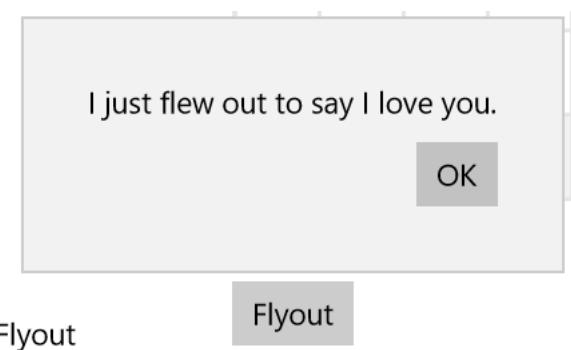
The Flyout can be displayed only using XAML, but you must make provisions for closing the Flyout inside of the Flyout itself. Here we add a Button called InnerFlyoutButton and handle its click event to hide the Flyout:

```

46      private void InnerFlyoutButton_Click(object sender, RoutedEventArgs e)
47      {
48          MyFlyout.Hide();
49      }

```

The result:

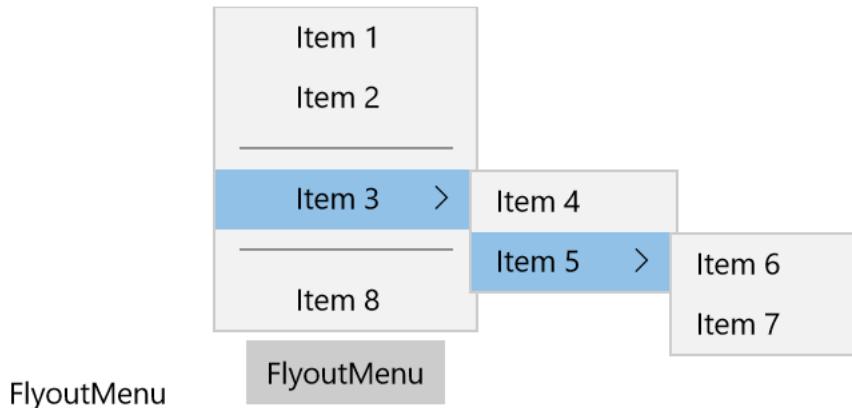


Similarly, we'll examine the Menu Flyout control. You've probably seen a Menu Flyout if you've been using Windows 10. The Menu Flyout is useful whenever you are trying to create a contextual menu for a given control.

Here again I'll use a Button, however you'll see that many different controls have a Flyout property inside of which we can implement a Menu Flyout.

```
/4
75      <TextBlock Grid.Row="4" Text="FlyoutMenu" VerticalAlignment="Center" />
76      <Button Grid.Row="4"
77          Margin="20,0,0,20"
78          Grid.Column="1"
79          Content="FlyoutMenu">
80          <Button.Flyout>
81              <MenuFlyout Placement="Bottom">
82                  <MenuFlyoutItem Text="Item 1" />
83                  <MenuFlyoutItem Text="Item 2" />
84                  <MenuFlyoutSeparator />
85                  <MenuFlyoutSubItem Text="Item 3">
86                      <MenuFlyoutItem Text="Item 4" />
87                      <MenuFlyoutSubItem Text="Item 5">
88                          <MenuFlyoutItem Text="Item 6" />
89                          <MenuFlyoutItem Text="Item 7" />
90                      </MenuFlyoutSubItem>
91                  </MenuFlyoutSubItem>
92                  <MenuFlyoutSeparator />
93                  <ToggleMenuFlyoutItem Text="Item 8" />
94          </MenuFlyout>
95      </Button.Flyout>
96  </Button>
```

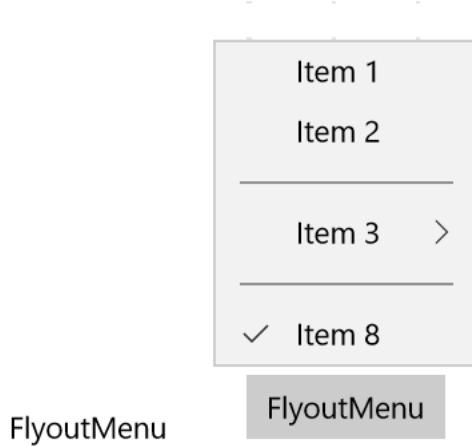
The MenuFlyout consists of one or more MenuFlyoutItems, MenuFlyoutSubItems, ToggleMenuFlyoutItems or MenuFlyoutSeparators. Each of their purposes will become apparent simply by running and examining their XAML code and the result.



In this case, notice that I set the Placement attribute to "Bottom", however the Menu Flyout is displayed above the button. The XAML layout engine attempts to fill our request, however it is bound by the boundaries of the application's overall width and height and when necessary will move the Menu Flyout to an available area.

This example doesn't handle the Click event, however you would handle it for each menu items as you deem fit.

The ToggleMenuItem has an "on" and "off" state perfect for enabling features of your app. You can check the user's select using the IsChecked property:



The next control we'll examine is the AutoSuggestBox which will become very helpful to us whenever we're building real applications that include the search feature like a hamburger-style navigation that we saw previously. This will allow us to utilize any collection or array as a source of potential items that are displayed and filtered as the user types into the text box:

```
101      <TextBlock Grid.Row="5" Text="AutosuggestBox" VerticalAlignment="Center" />
102      <AutoSuggestBox Name="MyAutoSuggestBox"
103          Margin="20,0,0,20"
104          Grid.Row="5"
105          Grid.Column="1"
106          HorizontalAlignment="Left"
107          QueryIcon="Find"
108          PlaceholderText="Find Something"
109          Width="200"
110         TextChanged="MyAutoSuggestBox_TextChanged" />
111
```

To implement this control, you'll need a data source. This could come from a database or some hard-coded list of potential values. In this case I'll create an array of strings containing names called selectionItems. In a more full-featured all, you would retrieve this from a web service, database, flat file, etc.

Inside of the TextChanged event handler I use a clever LINQ statement to retrieve all of the items from the list that start with the letters the user typed in.

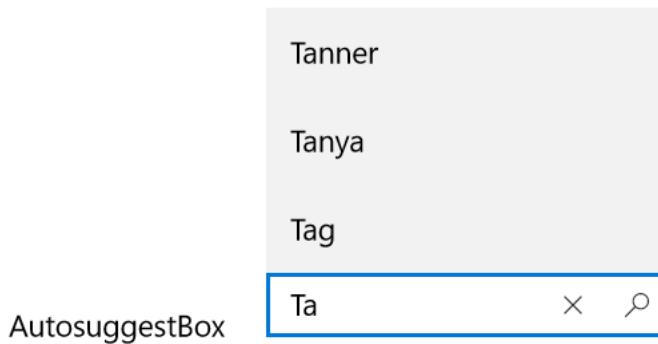
Finally, I set the ItemsSource property of the AutoSuggestBox to the filtered array of strings.

```

51
52     private string[] selectionItems = new string[]
53         { "Ferdinand", "Frank", "Frida", "Nigel", "Tag", "Tanya", "Tanner", "Todd" };
54
55     private void MyAutoSuggestBox_TextChanged(
56         AutoSuggestBox sender,
57         AutoSuggestBoxTextChangedEventArgs args)
58     {
59         var autoSuggestBox = (AutoSuggestBox)sender;
60         var filtered = selectionItems
61             .Where(p => p.StartsWith(autoSuggestBox.Text))
62             .ToArray();
63
64         autoSuggestBox.ItemsSource = filtered;
65     }
66

```

When I run the application I can start to filter all of the possible names by typing a couple of letters. I can then use the keyboard's arrow keys to make the final selection:



Next, the Slider Control which, again, you'll see often used in Windows 10. The Slider Control will allow the user to make a numerical selection between a certain range by dragging the head of the slider to the right or left.

```

115
116     <TextBlock Text="Slider" Grid.Row="6" VerticalAlignment="Center" />
117     <Slider Name="MySlider"
118         Margin="20,0,0,20"
119         Grid.Row="6"
120         Grid.Column="1"
121         HorizontalAlignment="Left"
122         Maximum="100"
123         Minimum="0"
124         Width="200" />

```

Here I set a Minimum and Maximum value. There are other attributes that can affect the “steps” between 0 and 100, and other properties of the Slider.

18

Slider



As the user drags the head on the Slider, the Value attribute is changed. You can see the current Value in a small tooltip above the head.

Next, we'll talked about the ProgressBar which provides feedback to the user for long-running operations.

```
125 <TextBlock Grid.Row="7" Text="ProgressBar" VerticalAlignment="Center" />
126 <ProgressBar Name="MyProgressBar"
127     Margin="20,0,0,20"
128     Grid.Row="7"
129     Grid.Column="1"
130     HorizontalAlignment="Left"
131     Width="200"
132     Maximum="100"
133     Value="{x:Bind MySlider.Value, Mode=OneWay}" />
134
```

In this example, I am using a special syntax to set the Value. Sure, I could set it in C# or XAML by setting the Value property to something like "57". In this case I wanted to show something a bit more advanced as a segue into the topic of data binding later in this series.

I'm using a special binding syntax which will bind the value of the Progress Bar control to the value of the Slider control.

32

Slider



ProgressBar



Admittedly, this isn't a very practical example. Typically you would use the Progress Bar to give the user feedback on some long running operation by setting the Value property, as well as the Minimum and Maximum value to display a percentage complete.

The final XAML control I'll feature is the ProgressRing which is another type of progress control, however it doesn't give the user any specific feedback on exactly how far along they are in a long running process. The only feedback they get is that the app is "still working":

```
135     <TextBlock Grid.Row="8" Text="Progress Ring" VerticalAlignment="Center" />
136     <ProgressRing Name="MyProgressRing"
137         Margin="20,0,0,20"
138         Grid.Row="8"
139         Grid.Column="1"
140         HorizontalAlignment="Left"
141         Width="50"
142         Height="50"
143         IsActive="True" />
144
```

By setting IsActive to "True" the ProgressRing spins displaying the familiar Windows 10 orbiting circles:



## UWP-026 - Working with the ScrollViewer

The ScrollViewer is a Layout allows you to add a scrollable area into your application. This scrollable area could be for the entire viewable area or just for sub-sections. In this lesson I'll demonstrate its use by adding it to specific Cells of a Grid.

I have created a new project called ScrollViewerExample.

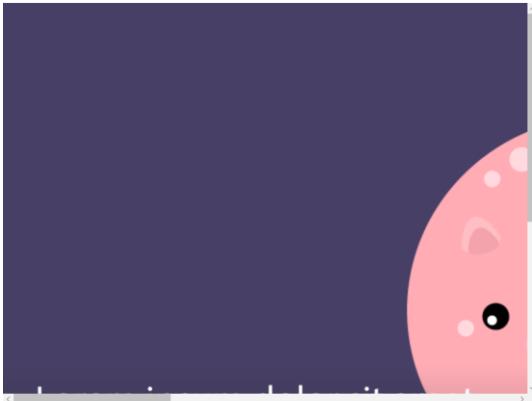
First, I'll create four Grid cells:

```
10  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11      <Grid.RowDefinitions>
12          <RowDefinition Height="*" />
13          <RowDefinition Height="*" />
14      </Grid.RowDefinitions>
15      <Grid.ColumnDefinitions>
16          <ColumnDefinition Width="*" />
17          <ColumnDefinition Width="*" />
18      </Grid.ColumnDefinitions>
19
```

I'll add the ScrollViewer into Row zero, Column zero.

```
20      <ScrollView
21          HorizontalScrollBarVisibility="Auto"
22          VerticalScrollBarVisibility="Auto">
23              <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
24      </ScrollView>
```

This ScrollViewer contains an Image control with its Source property set to a .png image we've worked with previously. It's a large image with a Height="800" therefore I'm setting it's Stretch="None". I've chosen this so that we can see our ScrollViewer at work.



A ScrollViewer is just that -- a viewable area that can be scrolled. By adding more content inside of the ScrollViewer than can be viewed without scrolling, we've created a condition where the ScrollViewer can shine. By setting the HorizontalScrollBarVisibility and VerticalScrollBarVisibility to "Auto" we're allowing the ScrollViewer to determine whether we need scrollbars across the right or bottom sides (respectively). We could programmatically change those Visibility properties to Hidden, Disabled or Visible as well to force the scrollbars to appear and be functional.

I've demonstrated the primary usage of the ScrollViewer. However, I do want to demonstrate one of the little "gotcha's" when using the ScrollViewer: If you enclose a ScrollViewer with a StackPanel you essentially eliminate the ScrollViewer from operating. In this example, I add a StackPanel in the second row, first Column. Inside of that a ScrollViewer I've set the HorizontalScrollBarVisibility and VerticalScrollBarVisibility to "Auto". I am employing the same Image control as the previous example. In fact, everything is same as the previous example – just enclosed in a StackPanel.

```
-->
26      <!-- Stack panel kills a child ScrollViewer -->
27      <StackPanel Grid.Row="1" Grid.Column="0">
28          <ScrollView
29              HorizontalScrollBarVisibility="Auto"
30              VerticalScrollBarVisibility="Auto">
31                  <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
32          </ScrollView>
33      </StackPanel>
```

When I run the application, I get no scroll bars whatsoever.

However, when I use a StackPanel inside of a ScrollViewer, all is well:

```
35      <!-- ScrollViewer can contain the stack panel -->
36      <ScrollView
37          Grid.Row="1"
38          Grid.Column="1"
39          HorizontalScrollBarVisibility="Auto"
40          VerticalScrollBarVisibility="Auto">
41          <StackPanel>
42              <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
43              <Image Source="Assets/Financial.png" Height="800" Stretch="None" />
44          </StackPanel>
45      </ScrollView>
46
```

In this case, I've added two Image controls inside of a StackPanel, inside of a ScrollViewer and the result is that we can scroll around the StackPanel and its contents just fine.

Admittedly, this is a convoluted example for the purpose of experimentation. You would typically use it to display the entire viewable area inside of the “chrome” of the application's window or the entire application itself (in the case of the phone form factor). The ScrollViewer is very important and we'll use it whenever we need to scroll through a lot of data.

## UWP-027 - Canvas and Shapes

In this lesson we're going to talk about the Canvas Layout Control as well as various Shape controls. While they're not inextricably tied together, you'll see that Shapes and the Canvas are complementary in nature.

The Canvas allows for the absolute placement of other controls inside the boundaries of the Canvas. Typically, when creating layouts for Windows apps we do not want absolute positioning because the user resizes the application we would have to either write a lot of display logic and perform a lot of math to determine how to resize and position each item appropriately, or we would essentially limit the form factors that can consume our application.

However, there are some “edge cases” where utilizing the Canvas for absolute positioning would be helpful. So, in the interest of making sure you have a broad knowledge regarding layout I wanted to cover both the Canvas and the various Shape controls.

There are several areas types of application that could benefit from some simple Shape and Canvas usage such as:

- Apps that work with music, musical staffs, musical notation, etc.
- Apps that deal with math requiring special symbols, placement of numbers and symbols
- Apps that have some graphical presentation of data, such as analog clocks, graphs, charts, etc.
- Apps for diagraming, drawing, or perhaps some sort of domain specific language generator which requires the user to drag and drop two symbols on to a design surface and then draw a line between them that has some meaning inside of your particular industry
- Apps that deal with bar readers, or create bars for scanners

First, we'll discuss the Line control. Admittedly, it doesn't need to be hosted by a Canvas control – you can add it to a Grid.

```
10 |<Grid>
11 |    <Line X1="10" X2="200"
12 |        Y1="10" Y2="10"
13 |        Stroke="Black" Fill="Black"
14 |        StrokeThickness="5"
15 |        StrokeEndLineCap="Triangle" />
16 |</Grid>
17 |
```

Here we're putting it inside of the default Cell, we're just specifying the X and Y coordinates for the beginning and the end. So here's the beginning point. X="10" and Y="10", so that gives it this left-most corner here. And then the second point would be X="200", Y="10", so that's going to give us a nice vertical line because the Y's don't change, only the X values.

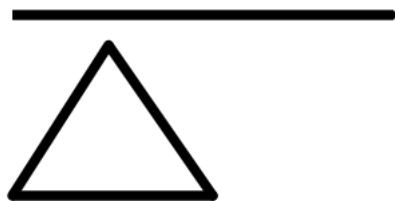
You can see that we've said the `Stroke="Black"` and the `StrokeThickness="5"`, and then we also set the `StrokeEndLineCap="Triangle"` `StrokeEndLineCap="Triangle"`. There are a couple different options here, you can see that we can create a Flat, Round, Square or Triangle. By choosing Triangle we get that little arrowhead on the end.

Next, I created a triangle by creating three more Lines and making sure that they intersect at just the right spot, making sure that their `StrokeStartLineCap` is set to Round, and that the `StrokeEndLineCap` is set to Round so they have a nice rounded appearance on the corners and they come together nicely.

```
--  
17     <Line X1="58" X2="10"  
18         Y1="25" Y2="100"  
19         Stroke="Black" Fill="Black"  
20         StrokeThickness="5"  
21         StrokeEndLineCap="Round" StrokeStartLineCap="Round" />  
22  
23     <Line X1="110" X2="58" Y1="100"  
24         Y2="25" Stroke="Black"  
25         Fill="Black"  
26         StrokeThickness="5"  
27         StrokeLineJoin="Round" StrokeStartLineCap="Round" />  
28  
29     <Line X1="110" X2="10"  
30         Y1="100" Y2="100"  
31         Stroke="Black" Fill="Black"  
32         StrokeThickness="5"  
33         StrokeLineJoin="Round" StrokeStartLineCap="Round" />  
34
```

Produces the following result:

ShapesExample



The point of the previous exercise was to prove that you can use the Line control in a Grid. However now let's change from a Grid to a Canvas:

```
- 10 <Canvas>
11   <Line X1="10" X2="200" ... />
16
17   <Line X1="58" X2="10" ... />
22
23   <Line X1="110" X2="58" Y1="100" ... />
28
```

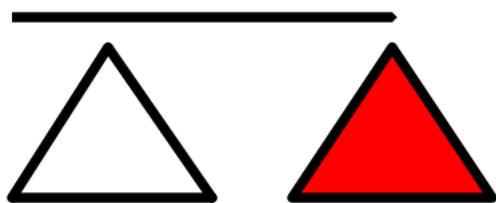
With the Canvas we can use absolute placement by choosing an Left and Top position. Any child element inside the Canvas can be positioned using the attached property syntax: Canvas.Left and Canvas.Top.

I'll also introduce the Polyline object which greatly simplifies the task of creating a triangle:

```
-->
36 <Polyline Canvas.Left="150"
37   Canvas.Top="0"
38   Stroke="Black"
39   StrokeThickness="5"
40   Fill="Red"
41   Points="50,25 0,100 100,100 50,25"
42   StrokeLineJoin="Round" StrokeStartLineCap="Round"
43   StrokeEndLineCap="Round" />
```

Produces:

ShapesExample



In this example I am using attached properties to set the left and the top of the Polyline, and then I start creating a series of Points (X and Y pairs) that define the various points of my Triangle. Then I can even set the Fill color for my Polyline ... if it all joins together.

Notice that I'm placing this entire Polyline here at 150 Left, 0 Top, so all the Points are relative to that position that I defined.

Next, just to emphasize the absolute placement of items in a Canvas, I'm creating a TextBlock and I'm setting it to Canvas.Left and Canvas.Top to 50 and 150, respectively. This demonstrates that I can add any Control (not just Shapes) inside of a Canvas for absolute positioning.

```
44
45      <TextBlock Name="HelloTextBlock"
46          Canvas.Left="50"
47          Canvas.Top="150"
48          FontSize="24"
49          Text="Shapes Example">
50      </TextBlock>
51
```

We've used the Rectangle in the past and for the sake of completeness I'll add it here, too. I define a yellow rectangle. Again, I am setting the Canvas.Top attached to 200 property and the Canvas.Left to zero.

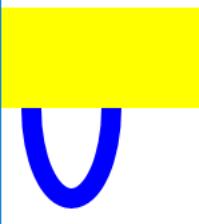
I've also added a Canvas.ZIndex property that will allow me to define the "stack order" of items should they overlap.

Next, I add an Ellipse intentionally overlapping them.

```
-->
53      <Rectangle Canvas.Top="200"
54          Canvas.Left="0"
55          Height="50" Width="100"
56          Fill="Yellow"
57          Canvas.ZIndex="100" />
58
59      <Ellipse Stroke="Blue"
60          Width="50" Height="100"
61          Canvas.Left="10"
62          Canvas.Top="200"
63          StrokeThickness="10"
64          Canvas.ZIndex="15" />
65
66  </Canvas>
```

In this case, I set the ZIndex of the Ellipse to 150 and the ZIndex of the yellow Rectangle to 100. If I change that, you can see that by changing the ZIndex to 15, it puts the yellow Rectangle on top of the Ellipse. So, we are talking in terms of X and Y-Coordinates, but also now adding Z-Coordinate in 3-D space, and so the ZIndex will set that stacking order for you.

## Shapes Example



If you need to draw Shapes in your application.

Finally, most of the shapes have a Click or Tapped event you can handle in the event that the user were to try to interact with that given Shape.

## UWP-028 - XAML Styles

As you already know Windows 10 has a very distinctive look and feel. I'm not just talking about things like the hamburger-style navigation, I'm also talking about the colors, the background/foreground colors, and the typography.

Here's a great resource that deals with the aesthetics of building Universal Windows Platform apps.

<http://dev.windows.com/en-us/design>

It talks about the aesthetics, approaches to navigation, etc. related to the design and user experience. There are probably other great resources available on [www.microsoft.com](http://www.microsoft.com) as well, or [www.windows.com](http://www.windows.com). But even inside of those guidelines, there is room for some creative expression, for example, your company may need its branding, its colors, its font styles to be represented in the application. So, you'll not only want to know what the rules are, but then you'll also want to know why and when to break those rules.

In this lesson, I'll talk about the technical aspects of working with XAML to style your app. Primarily, I'll talk creating reusable styles and resources that can be shared between many different elements across a single Page, multiple Pages in your entire app, or even across multiple applications. Then in the next lesson, we will talk about utilizing the pre-built Themes that are available that will help enforce consistency across all apps on a given user's device.

Suppose that you have a particular color, or any property setting that you know that you're going to want to use throughout the entire page of your application, or the entire app. Perhaps a color or a property setting that you potentially may change occasionally, and you just want to make that change in one place and then have it reflected everywhere that particular style is being utilized, similar to Cascading Style Sheets. In that case, you'll create a resource and then bind to that resource as necessary.

So I've created a new project called XAMLResources and I'll create the simplest possible example of creating and using resources.

In the MainPage.xaml I add a Page.Resources section. Inside of that Page.Resources section I'll add a new Resource – a SolidColorBrush named "brush" and that will simply be used to create a solid color brush with the color brown. Again, an extremely simple example, just to illustrate the concept at first.

Now that I've created a resource in this manner, whenever I want to use that brush that I've defined anywhere in my application, I can reference it using a binding statement. In the following code example, I'll set the Foreground of my TextBlock to the resource using a binding statement.

```

10    <Page.Resources>
11        <ResourceDictionary>
12            <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
13        </ResourceDictionary>
14    </Page.Resources>
15
16    <StackPanel>
17        <TextBlock Text="Hello World"
18            Foreground="{StaticResource MyBrush}" />
19    </StackPanel>
20

```

You can see the open and close curly braces and the word StaticResource, and then I give it the name of the resource that I want to bind to, in this case, MyBrush. The operative part of this example is the binding syntax, which we see often for different purposes in XAML, binding to a resource takes the form of the binding expression syntax of the open and close curly brace and then the word StaticResource and then whatever the resource name is. The curly braces indicate binding, so that first word, StaticResource, defines the type of binding that we're engaging in. We're binding to a resource that's defined in XAML and it's only evaluated once as the application first starts. There are other kinds of binding expressions that allow you to continually evaluate the information that will be bound, or pre-compile the Binding statements that we'll talk about as well, and those will come into play at various points during the remainder of this series of lessons.

In this case, I've only created a SolidColorBrush, however we can use this pattern to create lots of different types of resources. For example, I'll create a Resource that's a string named "greeting" and I use that greeting resource in the Text attribute of my TextBlock.

```

9
10   <Page.Resources>
11       <ResourceDictionary>
12           <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
13           <x:String x:Key="greeting">Hello world</x:String>
14       </ResourceDictionary>
15   </Page.Resources>
16
17   <StackPanel>
18       <TextBlock Text="{StaticResource greeting}"
19           Foreground="{StaticResource MyBrush}" />
20   </StackPanel>
21

```

Beyond simple resources, Styles allow you to collect one or more settings that can be reused across a Page, or across an entire app, for a specific type of Control. Then, whenever you want to reuse that Style, you set a given Control's Style attribute and essentially bind it to the Style that you've defined. So virtually every Control, and even the Page itself has features of its appearance that can be customized, whether it be fonts, or colors or border thickness, or width, or height. These attributes can be set on an individual basis on each Control in your XAML, however that would make your XAML quite wordy and maybe even difficult to parse through visually. The other downside is, obviously, if you miss a setting and you're trying to keep things consistent across all intended uses of those settings, you might forget to set one of the properties.

A better approach would be to define a style once then reusing the style when you need to apply all of those property settings to a given control.

```
10 <Page.Resources>
11   <ResourceDictionary>
12     <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
13     <x:String x:Key="greeting">Hello world</x:String>
14
15   <Style TargetType="Button" x:Key="MyButtonStyle">
16     <Setter Property="Background" Value="Blue" />
17     <Setter Property="FontFamily" Value="Arial Black" />
18     <Setter Property="FontSize" Value="36" />
19   </Style>
20
21 </ResourceDictionary>
22 </Page.Resources>
23
24 <StackPanel>
25   <TextBlock Text="{StaticResource greeting}"
26             Foreground="{StaticResource MyBrush}" />
27   <Button Content="My Button Style Example"
28         Height="100"
29         Style="{StaticResource MyButtonStyle}" />
30 </StackPanel>
31
```

In lines 15 through 19 I create a new style called "MyButtonStyle" that targets the Button control. I have three setter elements that set the values of several properties. Later, in line 27 through 29, I apply that style using the binding syntax and the keyword StaticResource.

These are very simple examples to illustrate the idea, and as a result, it may not be readily apparent why you would actually want to utilize Styles and Resources. What's the value of this approach? As your application grows larger, as you add more Pages or many more Controls to your application, you might find this approach to be quite handy. It would require a lot of effort to reapply even those three Properties (from the previous example) every single time you have a Button Control in your application. Using Styles and Resources will help keep your XAML compact and concise. It will also be easier to

manage. In that regard, if you ever need to make a change, you change it in one spot, in the Style itself, and then it's applied everywhere.

In the previous example I created these as local Resources on the Page where they're being used, meaning that their scoped to just this MainPage.xaml. But what if I wanted to share these Resources across all of the Pages in my application? In that case, I would remove those Resources and the Style from this MainPage.xaml, and I'd put them into an application Resources Element on the App.xaml.

```
App.xaml* Dictionary2.xaml Dictionary1.xaml MainPage.xaml*
Application
1 <Application
2     x:Class="XAMLResources.App"
3     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:local="using:XAMLResources"
6     RequestedTheme="Light">
7     <Application.Resources>
8         <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
9         <x:String x:Key="greeting">Hello world</x:String>
10    <Style TargetType="Button" x:Key="MyButtonStyle">
11        <Setter Property="Background" Value="Blue" />
12        <Setter Property="FontFamily" Value="Arial Black" />
13        <Setter Property="FontSize" Value="36" />
14    </Style>
15    </Application.Resources>
16 </Application>
17
18
```

I've removed the resources and styles from the MainPage.xaml and added them to App.xaml in the Application.Resources section. However at runtime, nothing should change about the application whatsoever.

Before moving on, the Application.Resources section can be used to add Static Resources, Control templates, animations and even more features of XAML and UWP that we've not learned about up until now. Just keep in mind that if you want to reuse something – almost anything -- across the entire application, it may make sense to put it into the Application.Resources section.

Finally, you can also create “merged resource dictionaries” that allow you to define your Resource dictionaries in multiple files and then combine them together at runtime. You may choose to put your style and resource definitions into separate files to help you manage the complexity, to reuse your Dictionary files in other projects, etc.

To add a new Resource Dictionary, I'll go to the Project menu > Add New Item then select the Resource Dictionary template. I'll leave the name as Dictionary1.xaml and click Add.

In the new file, I'll add the following:

```
1 <ResourceDictionary
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4     xmlns:local="using:XAMLResources">
5     <SolidColorBrush x:Key="brush" Color="Red"/>
6 </ResourceDictionary>
7
```

I'm creating a SolidColorBrush resource named "brush", setting its color to "Red".

Now we need to merge this ResourceDictionary into the Page's ResourceDictionary, and to do that, we will go to the MainPage's Page.Resources and create a ResourceDictionary element, and then a ResourceDictionary.MergedDictionaries attribute, and here we will create a ResourceDictionary and set the Source="Dictionary1.xaml".

```
10 <Page.Resources>
11     <ResourceDictionary>
12         <ResourceDictionary.MergedDictionaries>
13             <ResourceDictionary Source="Dictionary1.xaml" />
14         </ResourceDictionary.MergedDictionaries>
15     </ResourceDictionary>
16 </Page.Resources>
17
```

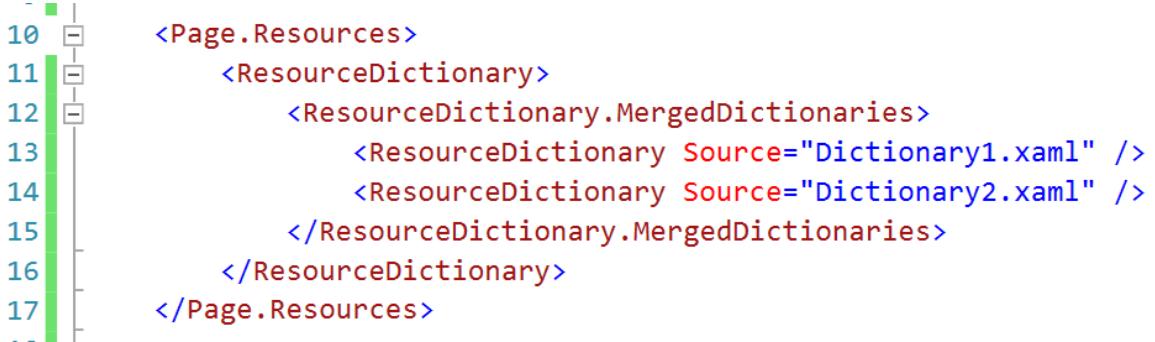
To see whether this works, we can change the resource we're binding to from "MyBrush" to just the "brush" resource:

```
18 <StackPanel>
19     <TextBlock Text="{StaticResource greeting}"
20                 Foreground="{StaticResource brush}" />
21     <Button Content="My Button Style Example"
22                 Height="100"
23                 Style="{StaticResource MyButtonStyle}" />
24 </StackPanel>
25
```

With this change the TextBlock's text should be red.

We can actually merge multiple Dictionaries together. I'll go to the Project menu > Add New Item, select Resource Dictionary, keep the default name Dictionary2.xaml and click Add.

I'll remove the resource named greeting from App.xaml and paste it into Dictionary2, and then merge that into my MainPage.xaml.



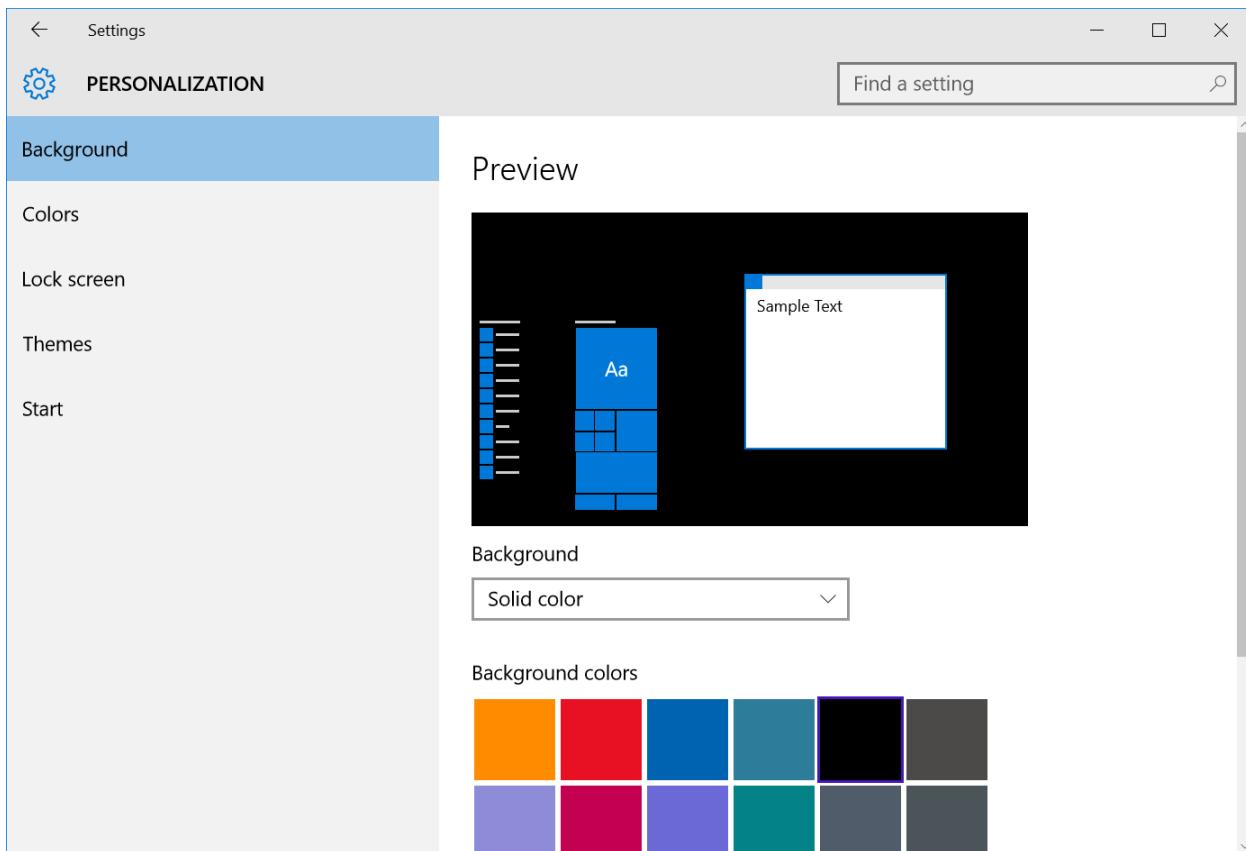
```
10     <Page.Resources>
11         <ResourceDictionary>
12             <ResourceDictionary.MergedDictionaries>
13                 <ResourceDictionary Source="Dictionary1.xaml" />
14                 <ResourceDictionary Source="Dictionary2.xaml" />
15             </ResourceDictionary.MergedDictionaries>
16         </ResourceDictionary>
17     </Page.Resources>
```

After making this change the TextBlock's text should remain "Hello World".

## UWP-029 - XAML Themes

In the previous lesson we created Styles and Resources and used Binding expression to bind them to Controls. We used the keyword `StaticResource` which is a type of binding that only happens once when the app first starts. That's why it's a "StaticResource"; it won't change throughout the life of the app.

However, there are other examples of Binding statements that are not static. For example, there are themed resources which are similar to the `StaticResource`, but the resource lookup is evaluated when the theme changes. What's a theme? A theme is a collection of colors that are selected by the end user at an operating system level. In the Windows 10 Desktop you can pop open the Settings App, go to personalization, and here you can choose a background color and I think also an accent color, alright, and so in this case I've got black as my background and then this blue accent color.



You can also personalize the phone and the Xbox One as well by choosing a background and an accent, now admittedly, each have a different set of options, but in general the end user can personalize their colors in all the Windows 10 flavors.

As a developer you can choose to utilize these color selections in your app so that your app honors the user's choices. You're not required to do this, but you probably should unless you have a particular branding goal in mind. There are a set of styles that allow you to utilize those colors that were selected

by the user. Getting to the ResourceDictionary where those Styles are defined is a bit tricky and I'll show you how to find it in the next lesson.

I've created a new project named "ThemeResources". On the MainPage.xaml, the default page template contains a Grid and that Grid utilizes a ThemeResource called ApplicationPageBackgroundThemeBrush.



```
9
10 <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11
12 </Grid>
13 </Page>
14
```

If you put your mouse cursor just anywhere in that ThemeResource name and press the F12 key on your keyboard, it will open up a preview of a file named generic.xaml. If I hover my mouse cursor over the tab you can see the full path of that file on my copy of Windows is

Program Files (x86)\Windows Kits\10\DesignTime\CommonConfiguration\Neutral\UAP (version number)\Generic\generic.xaml Program Files (x86)\Windows Kits\10\DesignTime\CommonConfiguration\Neutral\UAP (version number)\Generic\generic.xaml

In the file itself you can see here that's it's defined a SolidColorBrush with that Key (ApplicationPageBackgroundThemeBrush) that's automatically being used in our Grid and that sets it to this color #FFFFFF. I recognize this to be the color white.

In this lesson I'm primarily going to refer to colors defined as ThemeResources, however there are many different types of Styles that are defined in this generic.xaml file, including Font Faces and weights and sizes and thicknesses and there are even default Styles, behavior and layout for all the basic XAML Controls. So if you ever wonder why something is styled a certain way or why it behaves in a certain way, this generic.xaml file is where you can look and you can use that technique that I used just a moment ago to get it. Generic.xaml is 14,000 lines of code so you'll probably want to use the search function of Visual Studio to find what you're looking for.

The way that you utilize those Styles is to reference them inside of your own Styles. Back in MainPage.xaml, I'll create Page.Resources element and a ResourceDictionary element. I may create a SolidColorBrush that I call AccentBrush. I set the color equal to a ThemeResource called SystemAccentColor. That SystemAccentColor will give me access to the accent colors that the user selects.

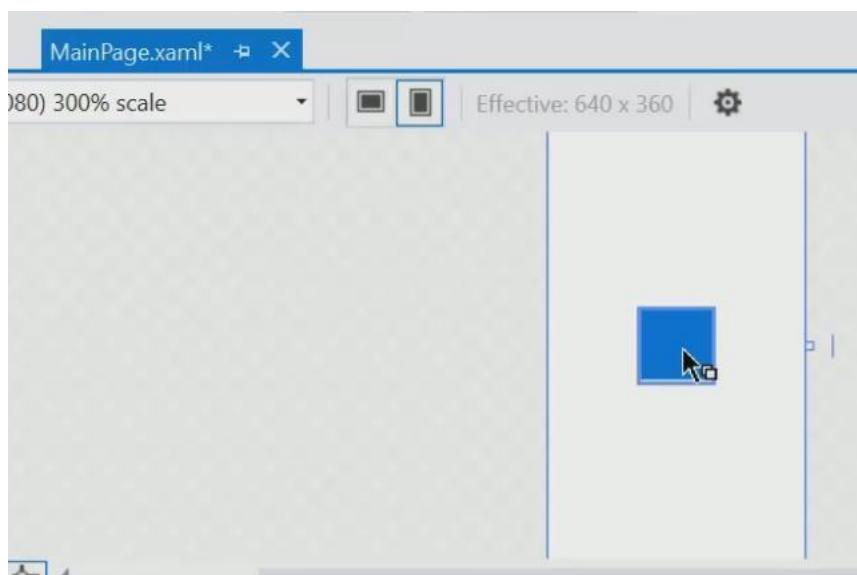
To use this resource, I'll add a Rectangle, set the Width="100", the Height="100" and the Fill equal to that StaticResource that I called AccentBrush.

```

9     <Page.Resources>
10    <SolidColorBrush x:Key="AccentBrush" Color="{ThemeResource SystemAccentColor}" />
11  </Page.Resources>
12
13  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
14    <Rectangle Width="100" Height="100" Fill="{StaticResource AccentBrush}" />
15  </Grid>
16 </Page>
17

```

A peek at the designer and you can see that it chose that same blue color that you see in the tab of Visual Studio because Visual Studio's using that blue color for the selected tab.



I could skip the part where I create a resource first and just use that theme resource directly in the Fill property:

```

13  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
14    <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
15  </Grid>

```



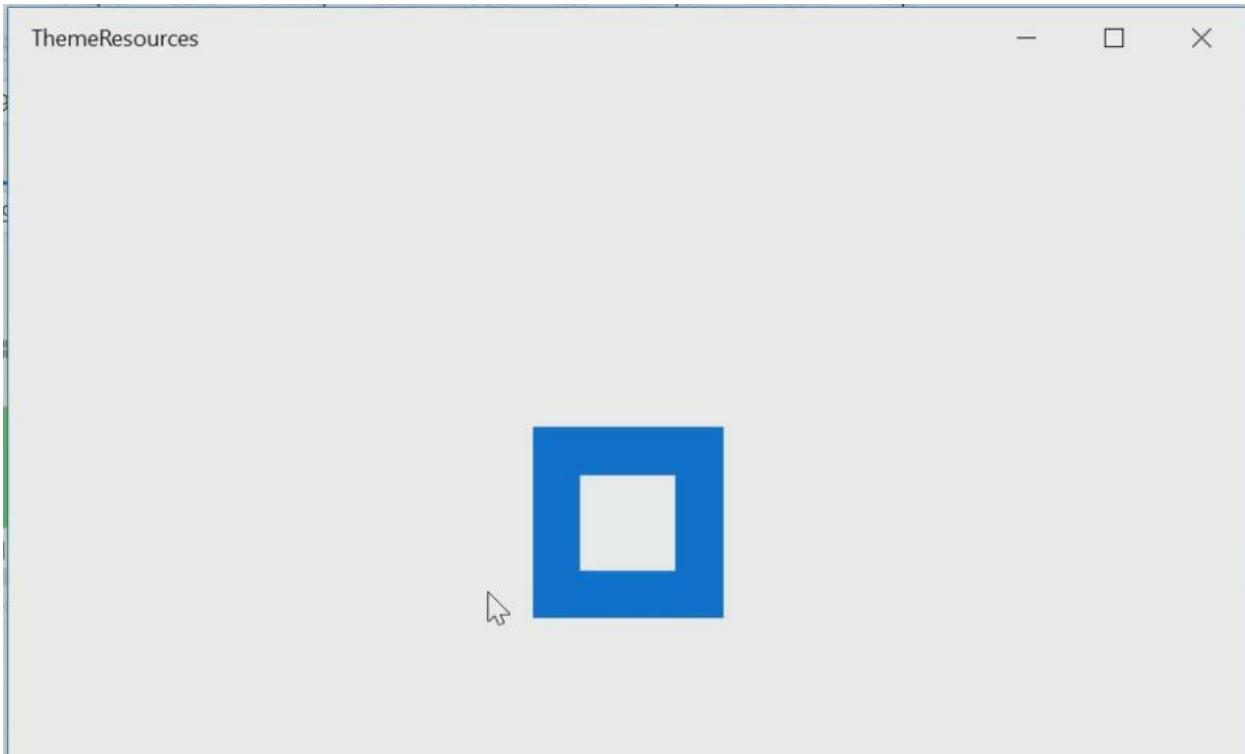
I can also access the color of the window like so:

```

13  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
14    <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
15    <Rectangle Width="50" Height="50" Fill="{ThemeResource SystemColorWindowColor}" />
16  </Grid>
17 </Page>

```

SystemColorWindowColor gives us a light-colored area inside of the blue box.



Again, that matches that light color and the rest of the window. This will become important as we talk about the preferred Theme in just a moment.

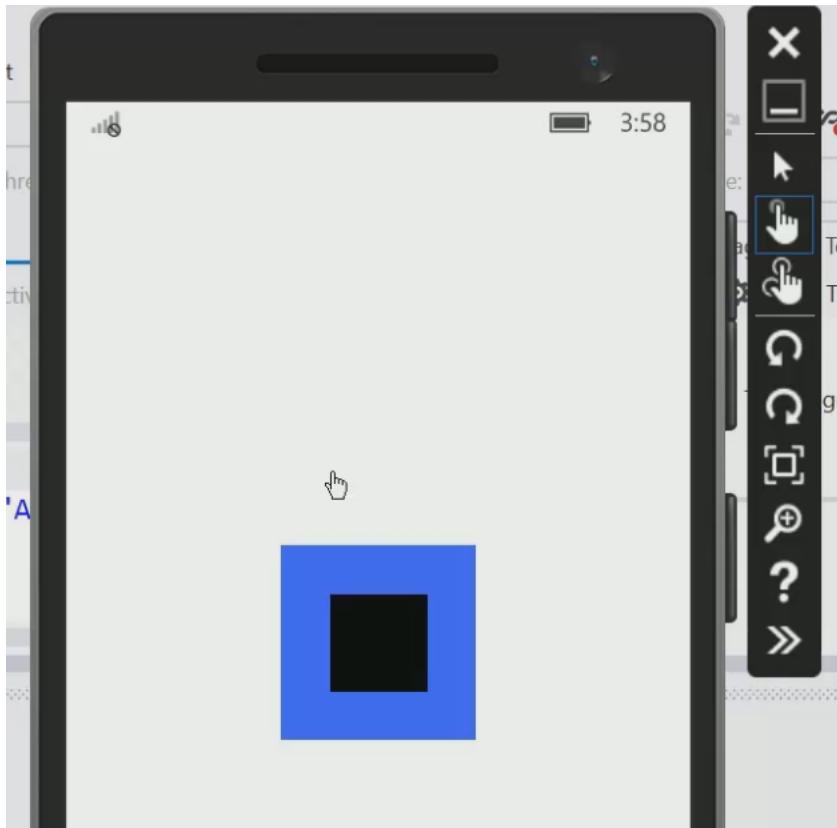
If you want an entire list of all of these Themes, take a look at “XAML ThemeResource” at:

[bit.do/theme-resources](http://bit.do/theme-resources)

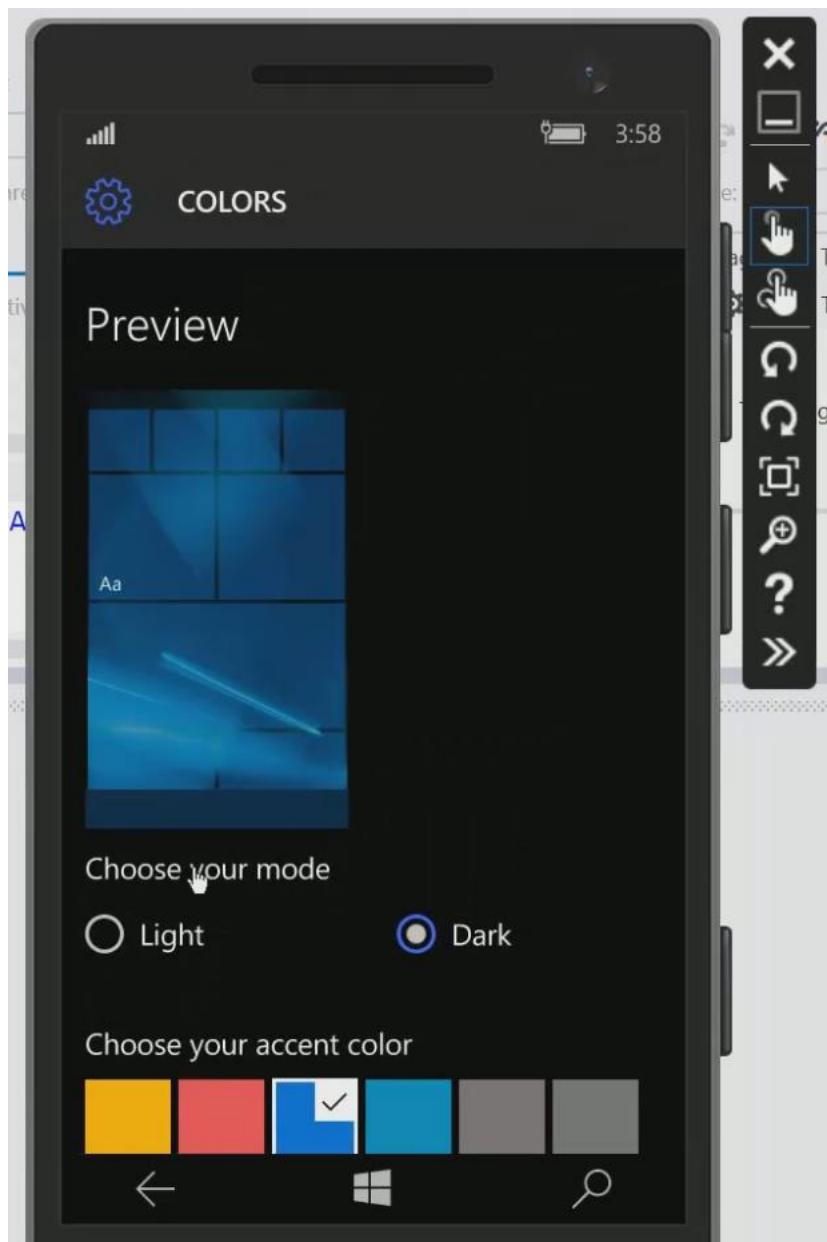
This contains long list of the different theme brushes, colors for buttons, text, etc.

On the Windows Phone this all works a little bit differently.

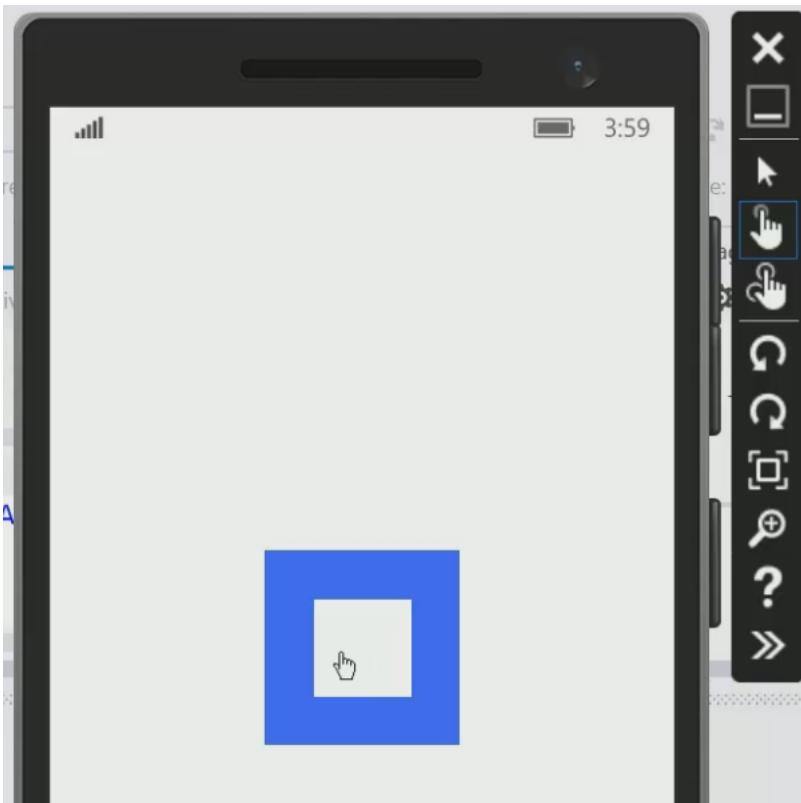
On the Windows 10 Phone, the user can select both a theme and an accent color, just like whenever you're building for the Windows 10 Desktop, you the developer get to decide whether to use those colors or not. On the phone, the user can select a “Dark Theme” or a “Light Theme” and you can see here in this particular case that I've chosen the Dark Theme. How do I know that? Because this SystemColorWindowColor is not white on the phone, but rather black.



In the simulator go down to Settings > Personalization > Colors. Here you can choose your mode, either Light or Dark. I'll change from Dark to Light ...



and go back to my ThemeResources app again and notice that that system color changed from Black to White.

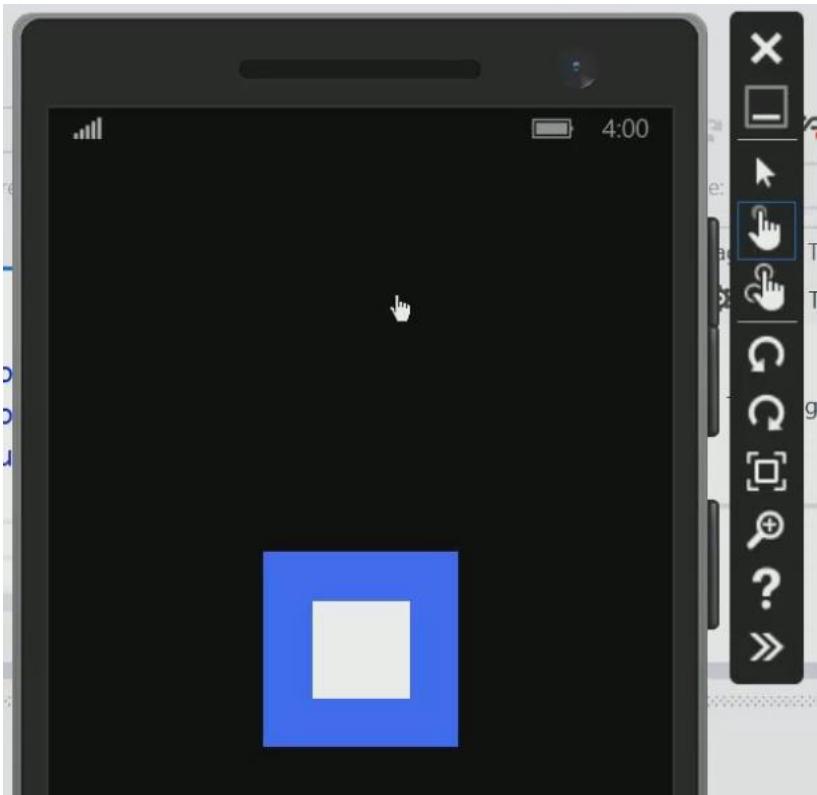


The intent of that Light and Dark Theme was really to change the “shell”, the colors that are used by the operating system, but again, as the developer you can choose to actually utilize those colors and each control is already Themed for both Light and Dark and new phones are typically configured with the black theme.

However, on an application level, you can see by default the RequestedTheme is Light which accounts for the light color window of our app.

```
App.xaml X generic.xaml MainPage.xaml
1 <Application
2     x:Class="ThemeResources.App"
3     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:local="using:ThemeResources"
6     RequestedTheme="Light">
7
8 </Application>
9
```

If I change it to Dark then re-run the app, background color for the window is black



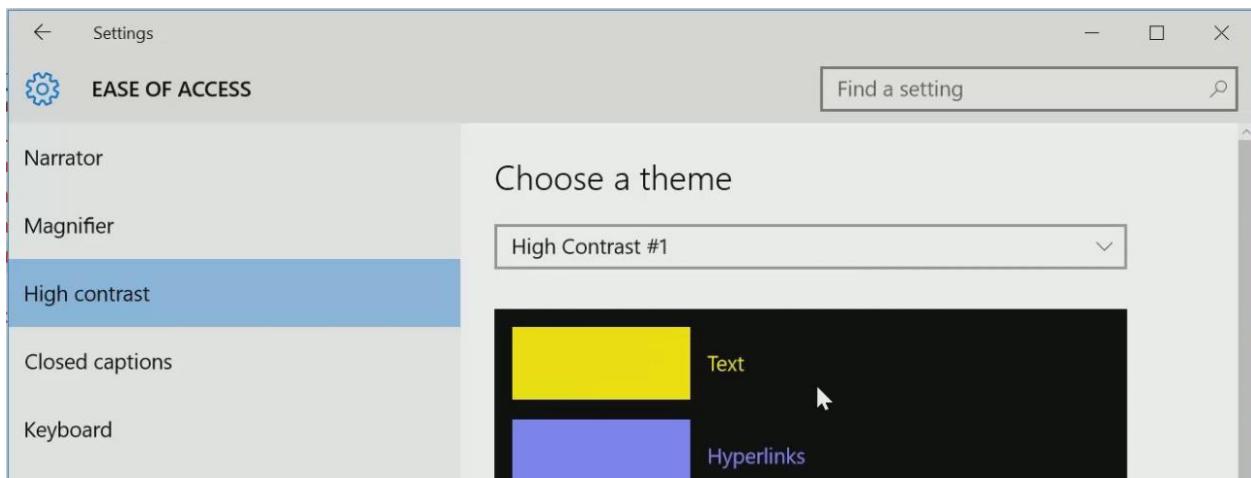
This works the same way inside of our Desktop apps as well.

I'm not going to show you how to override the Styles that are defined in this generic.xaml, but you can read about it in that help article titled "XAML ThemeResources".

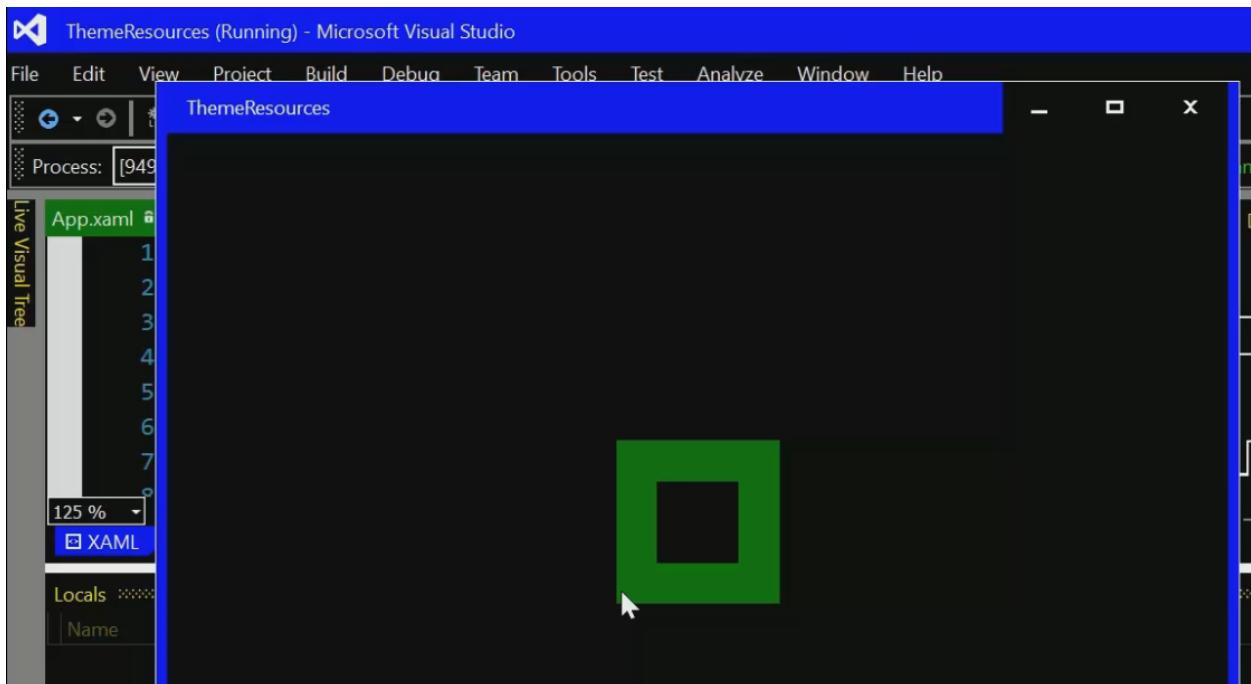
Finally, a moment ago I said that you could request a Theme and I wanted to emphasize the fact that it's merely a request and even the property says RequestedTheme. If the user is using one of the high-contrast Themes on the desktop, it's going to override the RequestedTheme as well as any Styles that you create for your Control.

High-contrast Themes are for accessibility -- for people that are vision impaired -- and that takes precedence over any aesthetic that you might want to use for your app.

In Settings I'll search for "High Contrast". That setting is part of the Ease of Access settings.



In the Ease of Access section I'll choose "High contrast", then choose "High Contrast #1", then select the "Apply" button. The system does a quick log-out / log-in and now you can see that the Settings App looks completely different, Visual Studio looks different, when we run our app, this time we get the high contrast theme colors.



ThemedResources should be leveraged to keep your app looking like they belong on the user's desktop or device, so you should always resist the urge to use custom colors or fonts and things of that nature unless you have a really good reason to do so.

One of the good reasons would be for branding purposes for your company.

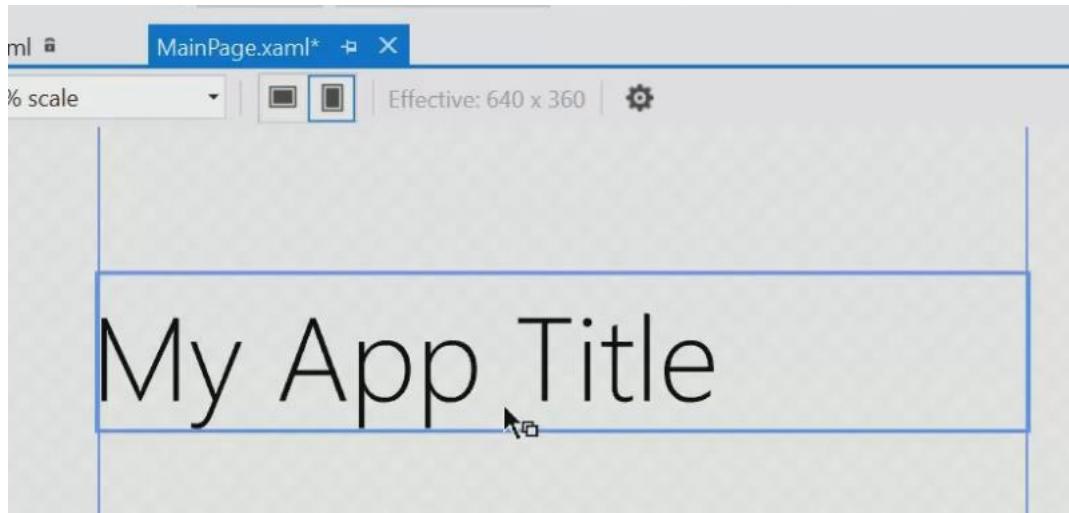
In addition to ThemeResources, there are built in Styles that are available to your app defined in generic.xaml.

First, I'll switch to use a StackPanel for layout, then I'll going to add a TextBlock and set the Text="My App Title". I'll bind the Style to "StaticResource HeaderTextBlockStyle".

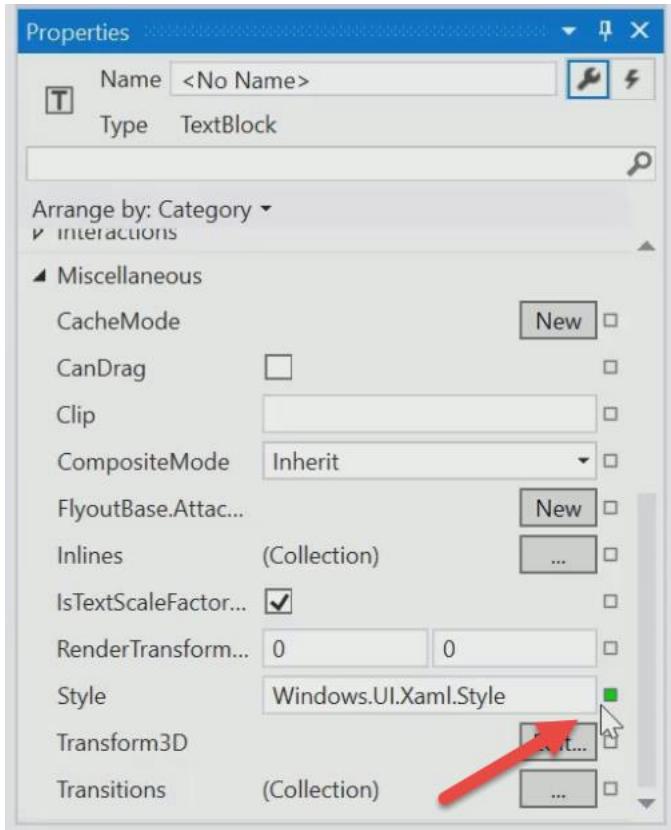
```
14
13     <StackPanel>
14         <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
15         <Rectangle Width="50" Height="50" Fill="{ThemeResource SystemColorWindowColor}" />
16         <TextBlock Text="My App Title" style="{StaticResource HeaderTextBlockStyle}" />
17     </StackPanel>
18 </Page>
19
```



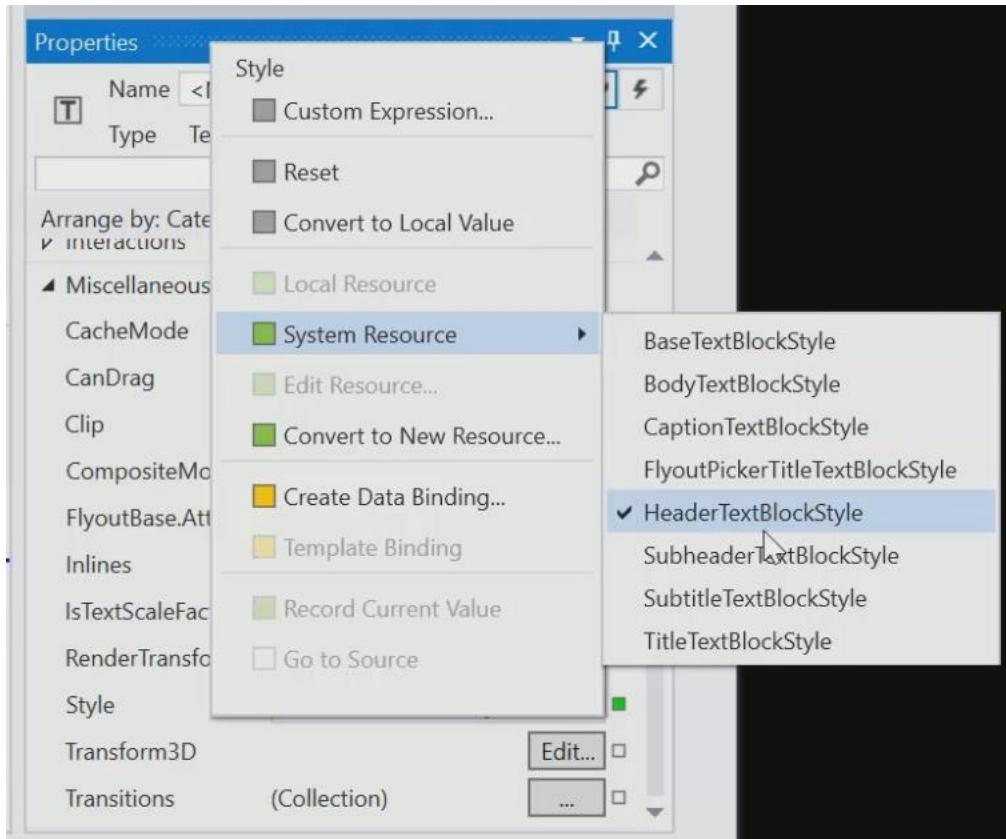
This style utilizes one of the Segoe fonts used often in applications.



If you ever are wondering where you can actually find these Styles, make sure the given control you're styling is selected then go to the Properties window, Miscellaneous section, and focus on the Style property:



The value of Style is set to Windows.UI.Xaml.Style. If I click this the green square, you can see that I can choose from a number of options. For this discussion, we'll expand System Resources which displays all of the available styles defined as System Resources.



I'll choose "CaptionTextBlockStyle" which is defined using a much smaller font.

Back in my XAML, If I put my mouse cursor on the word "CaptionTextBlockStyle" and hit F12, you'll notice that it brings us to line 12,400 in the generic.xaml file.

```

App.xaml      generic.xaml  MainPage.xaml*
12430   <Style x:Key="SubheaderTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">
12431     <Setter Property="FontWeight" Value="Normal"/>
12432     <Setter Property="FontSize" Value="20"/>
12433     <Setter Property="OpticalMarginAlignment" Value="TrimSideBearings"/>
12434   </Style>
12435
12436   <Style x:Key="BodyTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">
12437     <Setter Property="FontWeight" Value="Normal"/>
12438     <Setter Property="FontSize" Value="15"/>
12439   </Style>
12440
12441   <Style x:Key="CaptionTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">
12442     <Setter Property="FontSize" Value="12"/>
12443     <Setter Property="FontWeight" Value="Normal"/>
12444   </Style>
12445

```

If you scroll over to the right, you'll see that this Style has a BasedOn Attribute. This particular Style builds on something called BaseTextBlockStyle. It's based on it, but it adds a couple of modifications to it. This works a lot like Cascading Style Sheets where you take a Style and you keep building on it and adding changes to it to create new styles.

If I search for the “BaseTextBlockStyle” I can find that it's not based on anything else.

```
12400      <!-- FRAMEWORK STYLES -->
12401
12402      <Style x:Key="BaseTextBlockStyle" TargetType="TextBlock">
12403          <Setter Property="FontFamily" Value="Segoe UI"/>
12404          <Setter Property="FontWeight" Value="SemiBold"/>
12405          <Setter Property="FontSize" Value="15"/>
12406          <Setter Property="TextTrimming" Value="None"/>
12407          <Setter Property="TextWrapping" Value="Wrap"/>
12408          <Setter Property="LineStackingStrategy" Value="MaxHeight"/>
12409          <Setter Property="TextLineBounds" Value="Full"/>
12410      </Style>
12411
```

These style properties can be overridden or added to in order to create new styles.

As developers we can utilize the BasedOn attribute in our own Styles, again, to give us that Cascading Style Sheet approach to Styling the app.

## UWP-030 - Cheat Sheet Review: Controls, ScrollViewer, Canvas, Shapes, Styles, Themes

### UWP-025 - Common XAML Controls - Part 2

```
<TimePicker ClockIdentifier="12HourClock" />

<CalendarDatePicker PlaceholderText="choose a date" />

<CalendarView SelectionMode="Multiple"
    SelectedDatesChanged="MyCalendarView_SelectedDatesChanged" />

private void MyCalendarView_SelectedDatesChanged(CalendarView sender,
    CalendarViewSelectedDatesChangedEventArgs args)
{
    var selectedDates = sender.SelectedDates.Select(p => p.Date.Month.ToString() + "/" +
    p.Date.Day.ToString()).ToArray();
    var values = string.Join(", ", selectedDates);
    CalendarViewResultTextBlock.Text = values;
}

<Button Content="Flyout">
    <Button.Flyout>
        <Flyout x:Name="MyFlyout">

            </Flyout>
        </Button.Flyout>
    </Button>

    MyFlyout.Hide();

<Button Content="FlyoutMenu">
    <Button.Flyout>
        <MenuFlyout Placement="Bottom">
            <MenuFlyoutItem Text="Item 1" />
            <MenuFlyoutItem Text="Item 2" />
            <MenuFlyoutSeparator />
```

```

<MenuFlyoutSubItem Text="Item 3">
    <MenuFlyoutItem Text="Item 4" />
    <MenuFlyoutSubItem Text="Item 5">
        <MenuFlyoutItem Text="Item 6" />
        <MenuFlyoutItem Text="Item 7" />
    </MenuFlyoutSubItem>
</MenuFlyoutSubItem>
<MenuFlyoutSeparator />
<ToggleMenuFlyoutItem Text="Item 8" />
</MenuFlyout>
</Button.Flyout>
</Button>

<!-- You can apply this to anything ... ex. Image: --&gt;
<!-- https://msdn.microsoft.com/en-us/library/windows/apps/xaml/dn308516.aspx --&gt;
</pre>

```

```

<AutoSuggestBox Name="MyAutoSuggestBox"
    QueryIcon="Find"
    PlaceholderText="Search"
    TextChanged="MyAutoSuggestBox_TextChanged" />
```

```
private string[] selectionItems = new string[] { "Ferdinand", "Frank", "Frida", "Nigel", "Tag", "Tanya",
    "Tanner", "Todd" };
```

```

private void MyAutoSuggestBox_TextChanged(AutoSuggestBox sender,
    AutoSuggestBoxTextChangedEventArgs args)
{
    var autoSuggestBox = (AutoSuggestBox)sender;
    var filtered = selectionItems.Where(p => p.StartsWith(autoSuggestBox.Text)).ToArray();
    autoSuggestBox.ItemsSource = filtered;
}
```

```
<Slider Maximum="100" Minimum="0" />
```

```
<ProgressBar Maximum="100" Value="{x:Bind MySlider.Value, Mode=OneWay}" />
```

```
<ProgressRing IsActive="True" />
```

## UWP-026 - Working with the ScrollViewer

```
<ScrollViewer  
    HorizontalScrollBarVisibility="Auto"  
    VerticalScrollBarVisibility="Auto">  
</ScrollViewer>
```

You can put anything inside of it, however, don't put it inside of a StackPanel!

## UWP-027 - Canvas and Shapes

Canvas allows you to do absolute positioning via attached properties.

```
<Line X1="10"  
      X2="200"  
      Y1="10"  
      Y2="10"  
      Stroke="Black"  
      Fill="Black"  
      StrokeThickness="5"  
      StrokeEndLineCap="Triangle" />
```

```
<Polyline Canvas.Left="150"  
         Canvas.Top="0"  
         Stroke="Black"  
         StrokeThickness="5"  
         Fill="Red"  
         Points="50,25 0,100 100,100 50,25"  
         StrokeLineJoin="Round"  
         StrokeStartLineCap="Round"  
         StrokeEndLineCap="Round" />
```

```
<Rectangle />
```

```
<Ellipse />
```

```
Canvas.ZIndex="100"
```

The higher the ZIndex, the higher in the stack it appears (covering what is below it).

## UWP-028 - XAML Styles

<https://dev.windows.com/en-us/design>

```
<Page.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Brown" />
    <Style TargetType="Button" x:Key="MyButtonStyle">
        <Setter Property="Background" Value="Blue" />
        <Setter Property="FontFamily" Value="Arial Black" />
        <Setter Property="FontSize" Value="36" />
    </Style>
</Page.Resources>
```

Binding: {StaticResource ResourceName}

```
<Button Content="OK" Style="{StaticResource MyButtonStyle}" />
```

Create Page or Application level resource dictionaries

```
<Application.Resources>
</Application.Resources>
```

Split up your styles into Resource Dictionary files:

```
<!-- Dictionary1.xaml -->
<ResourceDictionary>
    <SolidColorBrush x:Key="brush" Color="Red"/>
</ResourceDictionary>
```

```
<Page>
    <Page.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Dictionary1.xaml"/>
                <ResourceDictionary Source="Dictionary2.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Page.Resources>
    <TextBlock Foreground="{StaticResource SomeStyle}" Text="Hi" />
</Page>
```

<http://bit.do/theme-resources>

Put your mouse on a style, hit F12 to open generic.xaml

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
    <Rectangle Width="50" Height="50" Fill="{ThemeResource SystemColorWindowColor}" />
</Grid>
```

```
<App RequestedTheme="Light">
```

High Contrast themes override styles.

Lots of different styles of system styles defined:

```
<TextBlock Text="page name" Style="{StaticResource HeaderTextBlockStyle}" />
```

Many styles defined in generic.xaml used BasedOn attribute ... and you can too!

## UWP-031 - Stupendous Styles Challenge

This next challenge creates an app for a fictitious company I created called GoNuts, a drive-through donut shop. This app would allow you to create your order and schedule it for a future date and time, then drive up to the take out window where robot delivers it to your car.

We're only going to focus on the layout and styling for this challenge. In other words, there are no real e-commerce functionality.

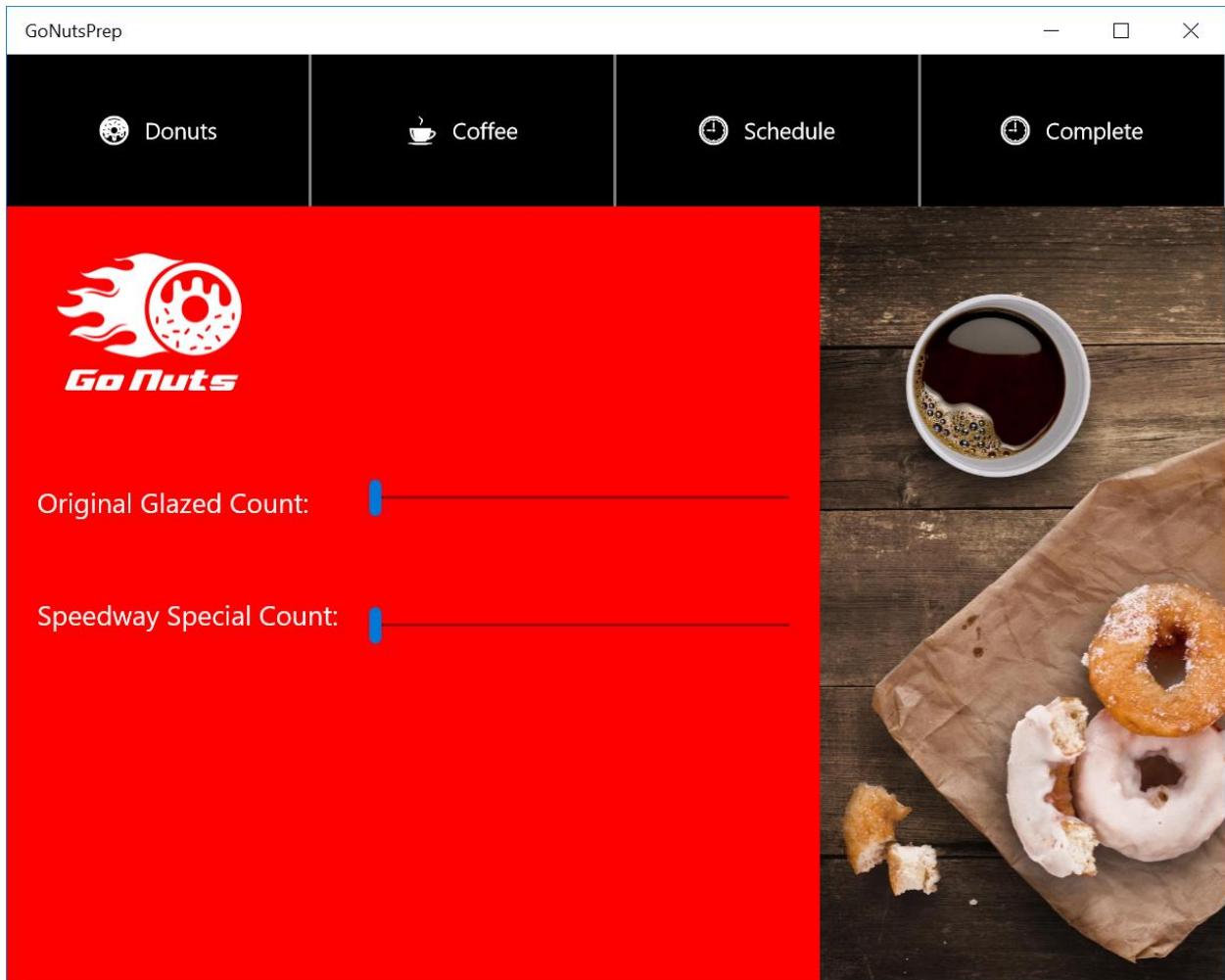
You'll use the resources contained in the zipped folder accompanying this lesson.

Refer to the instructions in the file called UWP-031-Instructions.txt

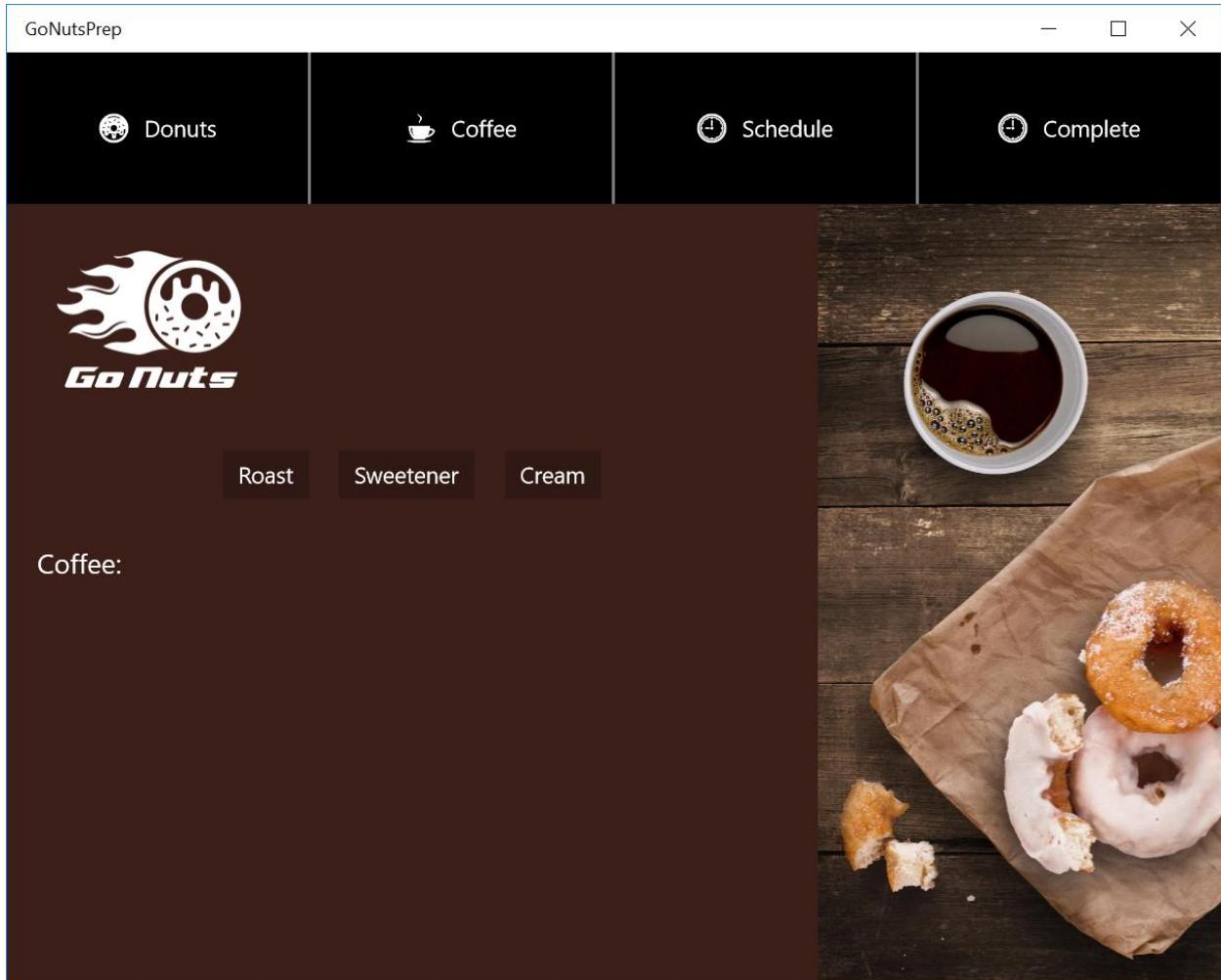
I'll briefly walk through each of the four pages you'll create.

There's a top navigation area with four clickable buttons.

The Donuts button will allow the user to select the number of donuts for each type of donut the company makes:

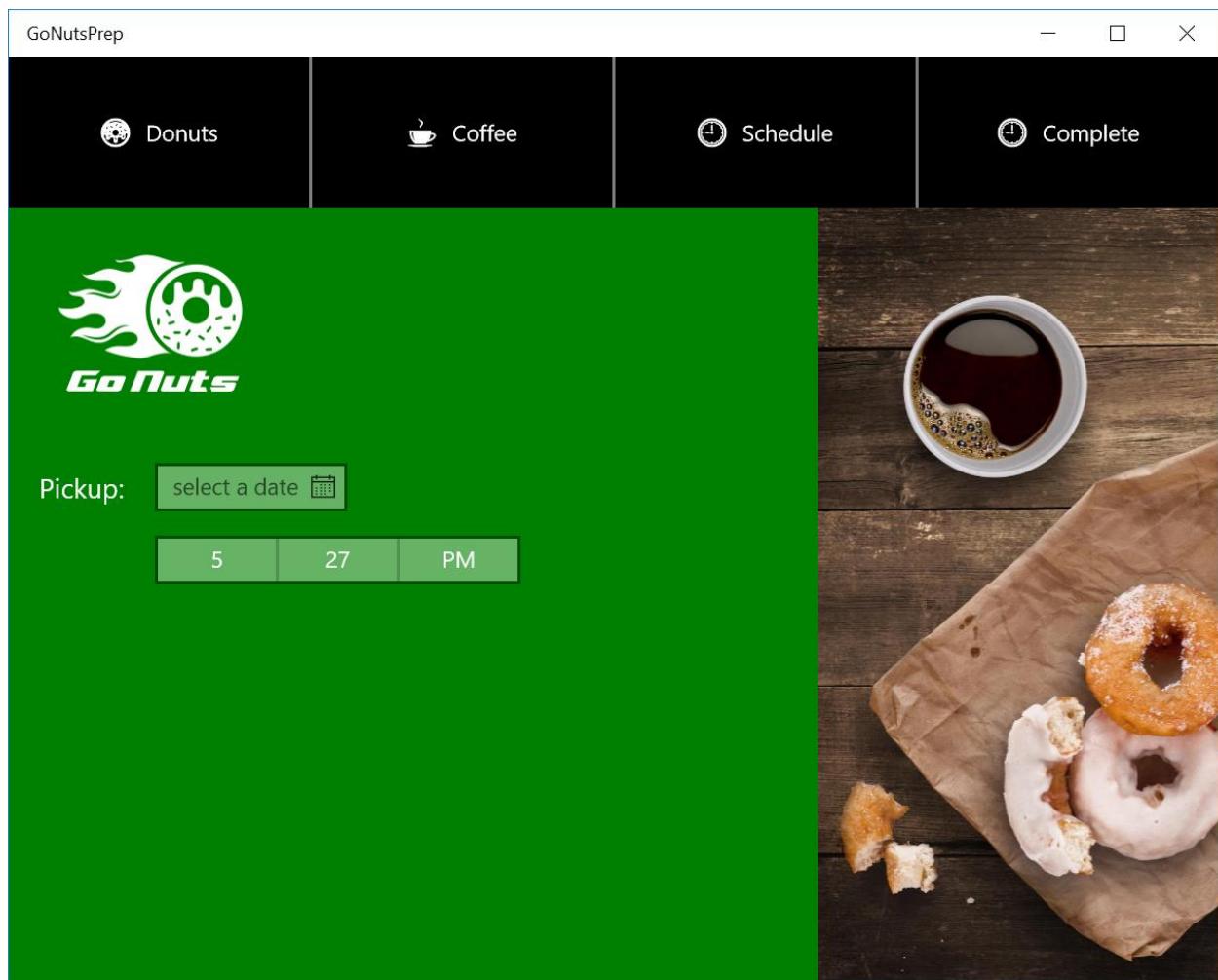


The coffee button allows the user to add on a cup of coffee and customize it with cream and sweetener:

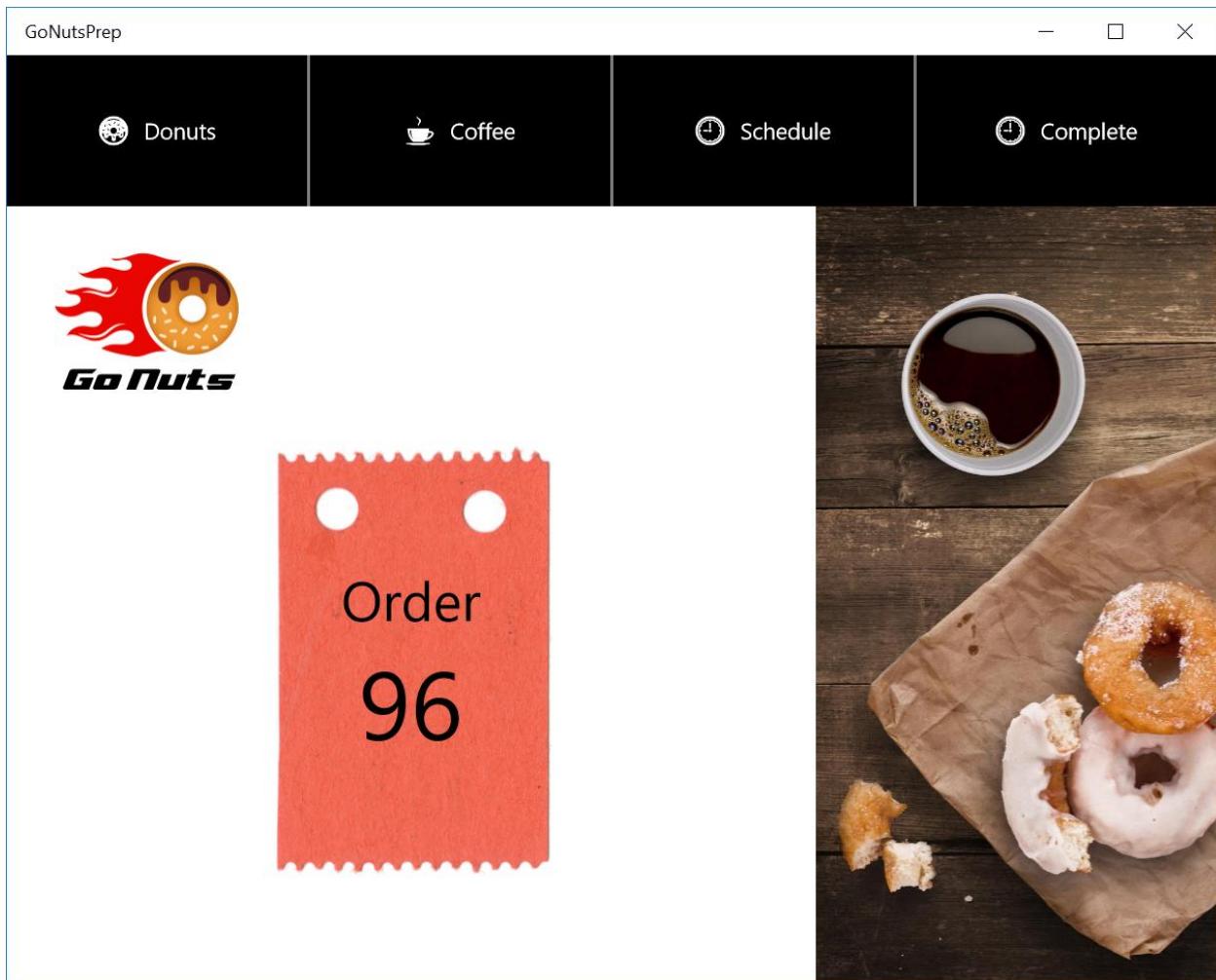


Coffee:

The schedule allows the user to specify a date and time to pick up the order:



The Complete button allows the user to get an order confirmation number styled as a ticket:



Create styles and resources as often as possible. Again, refer to the instructions document in the zipped folder, and utilize all of the graphical assets provided for you, along with the screenshots of the app you're attempting to build.

## UWP-032 - Stupendous Styles Challenge Solution - Part 1: MainPage

I'll split the solution to the challenge into several lessons focusing on just one part each time.

Also, the text version of these lessons are greatly abbreviated from the video format.

Step 1: Create a new Blank App Template project named GoNuts.

Step 2: Create an Assets folder and copy all of the graphic assets from the zipped folder associated with the previous lesson into that folder.

Step 3: Create the other pages named:

- DonutPage.xaml
- CoffeePage.xaml
- SchedulePage.xaml
- CompletePage.xaml

Step 4: On MainPage.xaml create an overall layout for the app. Add column and row definitions for each of the main sections of the app's user interface. In this case, two rows. Each row will contain its own inner Grid.

```
23     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
24         <Grid.RowDefinitions>
25             <RowDefinition Height="100" />
26             <RowDefinition Height="*" />
27         </Grid.RowDefinitions>
~~
```

Step 6: In the first row, I'll create a Grid with four columns – one for each of the buttons.

```
~~
29     <Grid>
30         <Grid.ColumnDefinitions>
31             <ColumnDefinition Width="*" />
32             <ColumnDefinition Width="*" />
33             <ColumnDefinition Width="*" />
34             <ColumnDefinition Width="*" />
35         </Grid.ColumnDefinitions>
~~
```

Step 7: I'll create one of the four buttons in the first column of that grid, then copy / paste the other three modifying what's different about each one. Notice that during the course of building the first button I realize there are two opportunities to dramatically reduce the amount of code that will be required to define the button by creating two styles:

```
37 <Button Grid.Column="0" Style="{StaticResource NavigationButtonStyle}">
38   <StackPanel Orientation="Horizontal">
39     <Image Source="Assets/donut-icon.png"
40       Style="{StaticResource IconImageStyle}" />
41     <TextBlock Text="Donut" Foreground="White" />
42   </StackPanel>
43 </Button>
..
```

Here are the two styles as defined at the top of the Page:

```
9 <Page.Resources>
10   <Style TargetType="Button" x:Key="NavigationButtonStyle">
11     <Setter Property="Background" Value="Black" />
12     <Setter Property="HorizontalAlignment" Value="Stretch" />
13     <Setter Property="VerticalAlignment" Value="Stretch" />
14     <Setter Property="BorderBrush" Value="Gray" />
15     <Setter Property="BorderThickness" Value="0,0,2,0" />
16   </Style>
17   <Style TargetType="Image" x:Key="IconImageStyle">
18     <Setter Property="Height" Value="20" />
19     <Setter Property="Width" Value="20" />
20     <Setter Property="Margin" Value="0,0,10,0" />
21   </Style>
22 </Page.Resources>
```

Step 8: I copy and paste the first button three times, modifying the column, image and text for each:

```
45     <Button Grid.Column="1" Style="{StaticResource NavigationButtonStyle}">
46         <StackPanel Orientation="Horizontal">
47             <Image Source="Assets/coffee-icon.png"
48                 Style="{StaticResource IconImageStyle}" />
49             <TextBlock Text="Coffee" Foreground="White" />
50         </StackPanel>
51     </Button>
52
53     <Button Grid.Column="2" Style="{StaticResource NavigationButtonStyle}">
54         <StackPanel Orientation="Horizontal">
55             <Image Source="Assets/schedule-icon.png"
56                 Style="{StaticResource IconImageStyle}" />
57             <TextBlock Text="Schedule" Foreground="White" />
58         </StackPanel>
59     </Button>
60
61     <Button Grid.Column="3" Style="{StaticResource NavigationButtonStyle}">
62         <StackPanel Orientation="Horizontal">
63             <Image Source="Assets/complete-icon.png"
64                 Style="{StaticResource IconImageStyle}" />
65             <TextBlock Text="Complete" Foreground="White" />
66         </StackPanel>
67     </Button>
--
```

Step 9: Remove the frame counter in the App.xaml.cs file.

## UWP-033 - Stupendous Styles Challenge Solution - Part 2: Navigation and DonutPage

Next I tackle navigation up in the Stupendous Styles Challenge solution and then work on the Donut Page.

Step 1: In the MainPage.xaml add a Grid that will be a child to the top-most grid, filling the second row. Create two column definitions. In the first column add a Frame named “MyFrame”. In the second column, add an Image control whose source is the background.jpg in the Source folder. Set it to fill the entire area without stretching.

```
```
81     <Grid Grid.Row="1">
82         <Grid.ColumnDefinitions>
83             <ColumnDefinition Width="2*" />
84             <ColumnDefinition Width="1*" />
85         </Grid.ColumnDefinitions>
86         <Frame Name="MyFrame"></Frame>
87         <Image Source="Assets/background.jpg" Grid.Column="1" Stretch="UniformToFill" />
88     </Grid>
```

```

Step 2: Give each button a name corresponding to its function then add an event handler to each of the buttons.

```
~~
37   <Button Name="DonutButton"
38     Click="DonutButton_Click"
39     Grid.Column="0"
40     Style="{StaticResource NavigationButtonStyle}">
41     <StackPanel Orientation="Horizontal">
42       <Image Source="Assets/donut-icon.png"
43         Style="{StaticResource IconImageStyle}" />
44       <TextBlock Text="Donut"
45         Foreground="White" />
46     </StackPanel>
47   </Button>
48
49   <Button Name="CoffeeButton"
50     Click="CoffeeButton_Click"
51     Grid.Column="1"
52     Style="{StaticResource NavigationButtonStyle}">
53     <StackPanel Orientation="Horizontal">
54       <Image Source="Assets/coffee-icon.png"
55         Style="{StaticResource IconImageStyle}" />
56       <TextBlock Text="Coffee" Foreground="White" />
57     </StackPanel>
58   </Button>
59
60   <Button Name="ScheduleButton"
61     Click="ScheduleButton_Click"
62     Grid.Column="2"
63     Style="{StaticResource NavigationButtonStyle}">
64     <StackPanel Orientation="Horizontal">
65       <Image Source="Assets/schedule-icon.png"
66         Style="{StaticResource IconImageStyle}" />
67       <TextBlock Text="Schedule" Foreground="White" />
68     </StackPanel>
69   </Button>
70
71   <Button Name="CompleteButton"
72     Click="CompleteButton_Click"
73     Grid.Column="3"
74     Style="{StaticResource NavigationButtonStyle}">
75     <StackPanel Orientation="Horizontal">
76       <Image Source="Assets/complete-icon.png"
77         Style="{StaticResource IconImageStyle}" />
78       <TextBlock Text="Complete" Foreground="White" />
79     </StackPanel>
80   </Button>
```

Put your mouse cursor on each of the click event handler names and press F12 to create the method stub for each.

Step 3: In the MainPage.xaml.cs, add navigation code to each button so that the Frame navigates to each of the XAML pages created in the previous lesson. Also, navigate to the DonutPage when the app first opens:

```
23     public sealed partial class MainPage : Page
24     {
25         public MainPage()
26         {
27             this.InitializeComponent();
28             MyFrame.Navigate(typeof(DonutPage));
29         }
30
31         private void DonutButton_Click(object sender, RoutedEventArgs e)
32         {
33             MyFrame.Navigate(typeof(DonutPage));
34         }
35
36         private void CoffeeButton_Click(object sender, RoutedEventArgs e)
37         {
38             MyFrame.Navigate(typeof(CoffeePage));
39         }
40
41         private void ScheduleButton_Click(object sender, RoutedEventArgs e)
42         {
43             MyFrame.Navigate(typeof(SchedulePage));
44         }
45
46         private void CompleteButton_Click(object sender, RoutedEventArgs e)
47         {
48             MyFrame.Navigate(typeof(CompletePage));
49         }
50     }
```

Step 4: In the DonutPage.xaml, create the layout for the page adding four row and two column definitions. Set the Grid's background to red.

```

10    <Grid Background="Red">
11        <Grid.RowDefinitions>
12            <RowDefinition Height="Auto" />
13            <RowDefinition Height="Auto" />
14            <RowDefinition Height="Auto" />
15            <RowDefinition Height="*" />
16        </Grid.RowDefinitions>
17        <Grid.ColumnDefinitions>
18            <ColumnDefinition Width="Auto" />
19            <ColumnDefinition Width="*" />
20        </Grid.ColumnDefinitions>
21

```

Step 5: Add the white logo into row 1, column 1. Two TextBlocks as labels and two slider controls to allow the user to select the donut count for both types of donuts. Set both slider's Maximum property to 24. Add margins as needed.

```

22        <Image Source="Assets/white-logo.png" Width="150" Margin="20,20,0,0" />
23
24    <TextBlock Grid.Row="1"
25        FontSize="18"
26        Text="Original Glazed Count:"
27        Foreground="White"
28        Margin="20,20,20,0" />
29
30    <Slider Grid.Row="1"
31        Grid.Column="1"
32        Maximum="24"
33        Margin="20,20,20,0" />
34
35    <TextBlock Grid.Row="2"
36        FontSize="18"
37        Text="Speedway Special Count:"
38        Foreground="White"
39        Margin="20,20,20,0" />
40
41    <Slider Grid.Row="2"
42        Grid.Column="1"
43        Maximum="24"
44        Margin="20,20,20,0" />

```

## UWP-034 - Stupendous Styles Challenge Solution - Part 3: CoffeePage

Step 1: In the CoffeePage.xaml create the layout using four row definitions. Set the background to the required brown color:

```
16 <Grid Background="#3C1F19">
17     <Grid.RowDefinitions>
18         <RowDefinition Height="Auto" />
19         <RowDefinition Height="Auto" />
20         <RowDefinition Height="Auto" />
21         <RowDefinition Height="*" />
22     </Grid.RowDefinitions>
23
```

Step 2: Now that we see we'll use the white logo on multiple pages, it's time to extract it's property settings and create a style. In App.xaml, create a new Style targeted at the image control named "WhiteLogoStyle".

```
1 <Application
2     x:Class="GoNuts.App"
3     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:local="using:GoNuts"
6     RequestedTheme="Light">
7     <Application.Resources>
8         <Style TargetType="Image" x:Key="WhiteLogoStyle">
9             <Setter Property="Source" Value="Assets/white-logo.png" />
10            <Setter Property="Width" Value="150" />
11            <Setter Property="Margin" Value="20,20,20,0" />
12            <Setter Property="HorizontalAlignment" Value="Left" />
13        </Style>
14
```

Back on the CoffeePage.xaml, add the image using the "WhiteLogoStyle":

```
23
24     <Image Style="{StaticResource WhiteLogoStyle}" />
25
```

Back on the donut page, do the same.

Step 2: Add a StackPanel for row 2, set the orientation to horizontal and horizontal alignment to center:

```

25
26     <StackPanel Orientation="Horizontal"
27         Grid.Row="1"
28         HorizontalAlignment="Center">
29

```

Step 3: Create a Button with a MenuFlyout, and once styled correctly, copy and paste two times replacing the content of each. The button's properties can be extracted to a local style to reduce the amount of XAML required so create a new style named FlyoutButtonStyle at the top of the page:

```

30     <Button Content="Roast"
31         Style="{StaticResource FlyoutButtonStyle}">
32         <Button.Flyout>
33             <MenuFlyout>
34                 <MenuFlyoutItem Text="None" Click="Roast_Click" />
35                 <MenuFlyoutItem Text="Dark" Click="Roast_Click" />
36                 <MenuFlyoutItem Text="Medium" Click="Roast_Click" />
37             </MenuFlyout>
38         </Button.Flyout>
39     </Button>
40

```

Notice the MenuFlyoutItems' Click event handler names are all the same.

The FlyoutButtonStyle should be defined like this on the CoffeePage.xaml:

```

1  <Page
2      x:Class="GoNuts.CoffeePage"
3      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:local="using:GoNuts"
6      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8      mc:Ignorable="d">
9      <Page.Resources>
10     <Style TargetType="Button" x:Key="FlyoutButtonStyle">
11         <Setter Property="Margin" Value="10" />
12         <Setter Property="Foreground" Value="White" />
13     </Style>
14 </Page.Resources>
15

```

Copy, paste and change the button twice to add the other two buttons, changing the appropriate values:

```

41     <Button Content="Sweetener"
42         Style="{StaticResource FlyoutButtonStyle}">
43         <Button.Flyout>
44             <MenuFlyout>
45                 <MenuItem Text="None" Click="Sweetener_Click" />
46                 <MenuItem Text="Sugar" Click="Sweetener_Click" />
47             </MenuFlyout>
48         </Button.Flyout>
49     </Button>
50
51     <Button Content="Cream"
52         Style="{StaticResource FlyoutButtonStyle}">
53         <Button.Flyout>
54             <MenuFlyout>
55                 <MenuItem Text="None" Click="Cream_Click" />
56                 <MenuItem Text="2% Milk" Click="Cream_Click" />
57                 <MenuItem Text="Whole Milk" Click="Cream_Click" />
58             </MenuFlyout>
59         </Button.Flyout>
60     </Button>
61 </StackPanel>

```

Again, take note of the Click event handler for each of the MenuFlyoutItems.

Step 4: In the third row of the layout Grid, add a StackPanel and two TextBlocks, one for the label and the other to be used to display the current selections from the Button's FlyoutMenus.

```

63     <StackPanel Orientation="Horizontal" Grid.Row="2">
64         <TextBlock Text="Coffee:" Style="{StaticResource LabelTextBlockStyle}" />
65         <TextBlock Name="ResultTextBlock" Style="{StaticResource LabelTextBlockStyle}" />
66     </StackPanel>
67

```

When styling the TextBlocks, it seems like we have a good candidate for an app level Style. Create a new style in App.xaml named: LabelTextBlockStyle:

```

14     <Style TargetType="TextBlock" x:Key="LabelTextBlockStyle">
15         <Setter Property="FontSize" Value="18" />
16         <Setter Property="Foreground" Value="White" />
17         <Setter Property="Margin" Value="20,20,20,0" />
18     </Style>
19     </Application.Resources>
20 </Application>
21

```

Go to the DonutPage.xaml and utilize the new style for the two TextBlocks.

Step 5: Place your mouse cursor on each of the three Click event handler names and press F12 to create method stubs.

Step 6: On CoffeePage.xaml.cs, create three private fields. Inside of each of the Click events, grab the selected MenuFlyoutItem, grab its text and put it into the respective private field.

```

34     private void Roast_Click(object sender, RoutedEventArgs e)
35     {
36         var selected = (MenuItem)sender;
37         _roast = selected.Text;
38         displayResult();
39     }
40
41     private void Sweetener_Click(object sender, RoutedEventArgs e)
42     {
43         var selected = (MenuItem)sender;
44         _sweetener = selected.Text;
45         displayResult();
46     }
47
48
49     private void Cream_Click(object sender, RoutedEventArgs e)
50     {
51         var selected = (MenuItem)sender;
52         _cream = selected.Text;
53         displayResult();
54     }
55 }
```

Notice that I'm calling a private helper method `displayResult()` in each click event handler. This will be delegated presentation logic for what will be displayed in the result label.

Step 7: Create the `displayResult()` helper method. The cream and sweetener should only be displayed when a roast has been selected. Style the entire string of text appropriately:

```

57     private void displayResult()
58     {
59         if (_roast == "None" || String.IsNullOrEmpty(_roast))
60         {
61             ResultTextBlock.Text = "None";
62             return;
63         }
64
65         ResultTextBlock.Text = _roast;
66
67         if (_cream != "None" && !_String.IsNullOrEmpty(_cream))
68             ResultTextBlock.Text += " + " + _cream;
69
70         if (_sweetener != "None" && !_String.IsNullOrEmpty(_sweetener))
71             ResultTextBlock.Text += " + " + _sweetener;
72     }
73 }
```

## UWP-035 - Stupendous Styles Challenge Solution - Part 4: SchedulePage

Next we'll focus on the Schedule Page. It is pretty straight forward now that we have several app level styles defined.

Step 1: Add four row and two column definitions, set the background of the grid to Green:

```
10      <Grid Background="Green">
11          <Grid.RowDefinitions>
12              <RowDefinition Height="Auto" />
13              <RowDefinition Height="Auto" />
14              <RowDefinition Height="Auto" />
15              <RowDefinition Height="*" />
16      </Grid.RowDefinitions>
17      <Grid.ColumnDefinitions>
18          <ColumnDefinition Width="Auto" />
19          <ColumnDefinition Width="*" />
20      </Grid.ColumnDefinitions>
```

Step 2: Add the logo:

```
21
22      <Image Style="{StaticResource WhiteLogoStyle}" />
23
```

Step 3: Create the label and use the LabelTextBLockStyle:

```
24      <TextBlock Grid.Row="1"
25          Style="{StaticResource LabelTextBlockStyle}"
26          Text="Pickup:" />
27
```

Step 4: Add a CalendarDatePicker

```
27      <CalendarDatePicker Grid.Row="1" Grid.Column="1" Foreground="White" />
28
29
```

Step 5: Add a TimePicker

```
29      <TimePicker Grid.Row="2" Grid.Column="1" Foreground="White" />
30
31
```

## UWP-036 - Stupendous Styles Challenge Solution - Part 5: CompletePage

This is the last of the solution lesson for the Stupendous Styles Challenge and will focus on the CompletePage.

Step 1: Create two row definitions.

```
10     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11         <Grid.RowDefinitions>
12             <RowDefinition Height="Auto" />
13             <RowDefinition Height="Auto" />
14         </Grid.RowDefinitions>
15
```

Step 2: The logo on this final page uses a different image file, otherwise it shares all of the same settings as the previous Page's logos. Use the WhiteLogoStyle, but add a Source property to override the white logo with the color logo.

```
16         <Image Style="{StaticResource WhiteLogoStyle}"
17             Source="Assets/color-logo.png" />
18
```

Step 3: In the second row, add the ticket.jpg and set its width to 300. Center it both horizontally and vertically.

```
18
19         <Image Grid.Row="1"
20             Source="Assets/ticket.jpg"
21             Width="300"
22             VerticalAlignment="Center"
23             HorizontalAlignment="Center" />
24
```

Step 4: We'll take advantage of how the Grid works by overlaying two TextBlocks containing the text "Order" and "96", respectively. To ensure they are centered, use a StackPanel and set its vertical and horizontal alignment to center.

```
25 <StackPanel Grid.Row="1"
26     VerticalAlignment="Center"
27     HorizontalAlignment="Center">
28     <TextBlock Text="Order"
29         FontSize="36"
30             HorizontalAlignment="Center" />
31     <TextBlock Text="96"
32         FontSize="64"
33             VerticalAlignment="Center" />
34 </StackPanel>
--
```

## UWP-037 - Utilizing the VisualStateManager to Create Adaptive Triggers

Starting in this lesson we'll focus on the techniques and strategies for creating responsive apps that adapt based on the given device.

In the resources that accompany this lesson, please open and start debugging SimpleVisualStateTriggerExample. Once running, resize the app's window as large and as small as possible. As you can see, a certain screen sizes the color of the background and the size of the font changes.

This is made possible by a class called the VisualStateManager and it does exactly what it sounds like; it manages the visual state of your application, including the position, sizes, colors, fonts, etc. -- virtually any properties on any objects you want to manipulate based on the current size of the window.

To use the VisualStateManager, you define a series of VisualStates with StateTriggers (what should prompt a change) and Setters (the values of target object properties that should change).

Think about how that applies to the Universal Windows Platform. One of the selling points is that you're able to write one code base and then use it across all these different form factors. This allows you to accommodate different screen resolutions for different form factors with the same code base. The VisualStateManager will be leverages to change the entire layout of your application based on the screen size.

Here's one of the three VisualStates defined for the project.

```
10     <Grid Name="ColorGrid" Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11         <VisualStateManager.VisualStateGroups>
12             <VisualStateGroup x:Name="VisualStateGroup">
13                 <VisualState x:Name="VisualStatePhone">
14                     <VisualState.StateTriggers>
15                         <AdaptiveTrigger MinWindowWidth="0"/>
16                     </VisualState.StateTriggers>
17                     <VisualState.Setters>
18                         <Setter Target="ColorGrid.Background" Value="Red" />
19                         <Setter Target="MessageTextBlock.FontSize" Value="18" />
20                     </VisualState.Setters>
21                 </VisualState>

```

First you define one or more VisualStateGroups. Each VisualStateGroup is comprised of a series of VisualStates. This VisualState is called "VisualStatePhone" inferring that this "state" is valid for the smallest Windows 10 form factor. Each VisualState is comprised of one or more StateTriggers and one or more Setters. The StateTrigger is set by using the AdaptiveTrigger element along with a MinWindowWidth or MinWindowHeight. In this case, since the MinWindowWidth="0" the Setters will apply to any screen size. In this case, there are two Setters: the first changes the ColorGrid's Background to red, and the MessageTextBlock's FontSize to 18.

There are two other VisualStates defined below it.

```

22         <VisualState x:Name="VisualStateTablet">
23             <VisualState.StateTriggers>
24                 <AdaptiveTrigger MinWindowWidth="600" />
25             </VisualState.StateTriggers>
26             <VisualState.Setters>
27                 <Setter Target="ColorGrid.Background" Value="Yellow" />
28                 <Setter Target="MessageTextBlock.FontSize" Value="36" />
29             </VisualState.Setters>
30         </VisualState>
31         <VisualState x:Name="VisualStateDesktop">
32             <VisualState.StateTriggers>
33                 <AdaptiveTrigger MinWindowWidth="800" />
34             </VisualState.StateTriggers>
35             <VisualState.Setters>
36                 <Setter Target="ColorGrid.Background" Value="Blue" />
37                 <Setter Target="MessageTextBlock.FontSize" Value="54" />
38             </VisualState.Setters>
39         </VisualState>
40     </VisualStateGroup>
41 </VisualStateManager.VisualStateGroups>

```

The VisualStateTablet will modify those same object properties when the window's width is 600 or greater. The VisualStateDesktop will modify those same object properties when the window's width is 800 or greater.

Conceptually it's very easy to understand. This ability to define visual states allows you to get creative with how you want to make changes to your application to conform to a given screen size. We define the Trigger and then once that Trigger is fired off, we apply the Setters.

While conceptually it is easy to understand, it may be difficult at first to remember exactly what objects and their relationship to each other is necessary to get this to work. Microsoft Blend will help you build VisualStates if you prefer to use a visual editor.

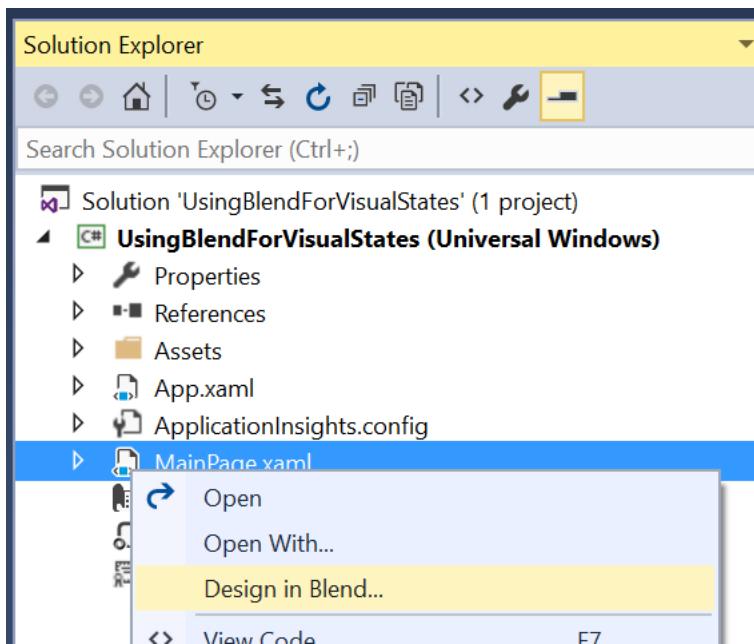
I've created a new project called UsingBlendForVisualStates, and I have the same color Grid and TextBlock, but I haven't added anything else to it just yet.

```

10     <Grid Name="ColorGrid">
11         <TextBlock Name="MessageTextBlock"
12             Text="Hello VisualStateManager" />
13     </Grid>
14 </Page>

```

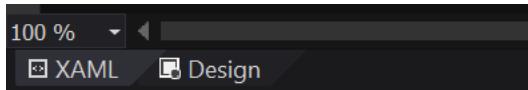
Here in the Solution Explorer right-click on the project and select Design in Blend.



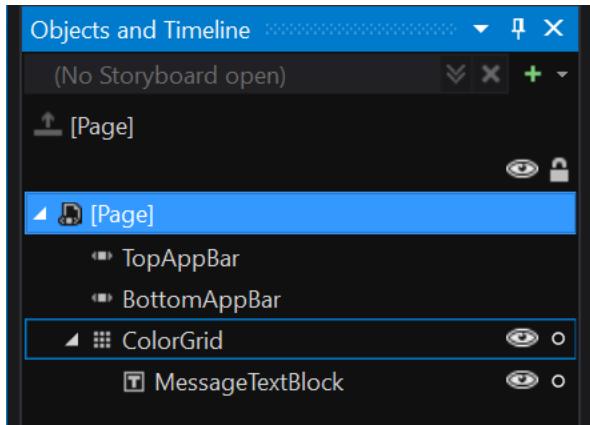
And so Blend is a tool that is usually installed along with Visual Studio (unless you chose custom install and de-selected it for installation). Blend was intended for developers who focus on aesthetic design work. It has many of the same features of Visual Studio. By default, on the right-hand side is the Properties window. A Solution Explorer docked by default on the left. While things are in slightly different positions the overall look and feel should be familiar.

There are two things that Blend does for you that you can't easily do in Visual Studio. First, it gives you tools to creating Visual States in a visual manner. Second, there are tools that allow you to work with animation, which we're not going to talk about in this series.

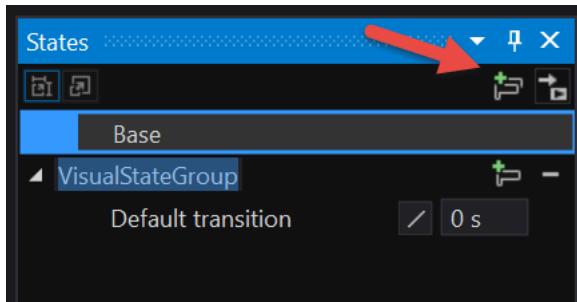
With the MainPage.xaml open in the main area, I'll switch to design view by clicking the tab at the very bottom.



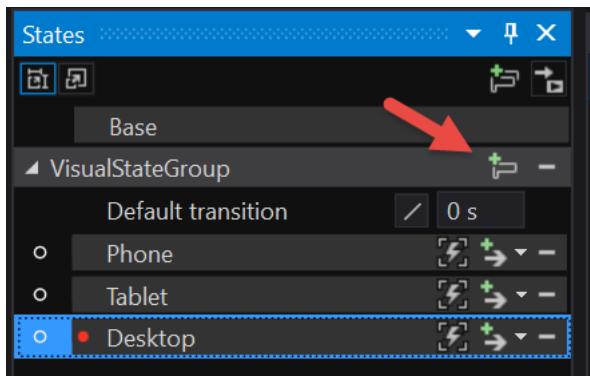
The Objects and Timeline window is typically docked on the left-hand side. I can drill down and see the ColorGrid and whatever is contained inside the ColorGrid, in this case, the MessageTextBlock.



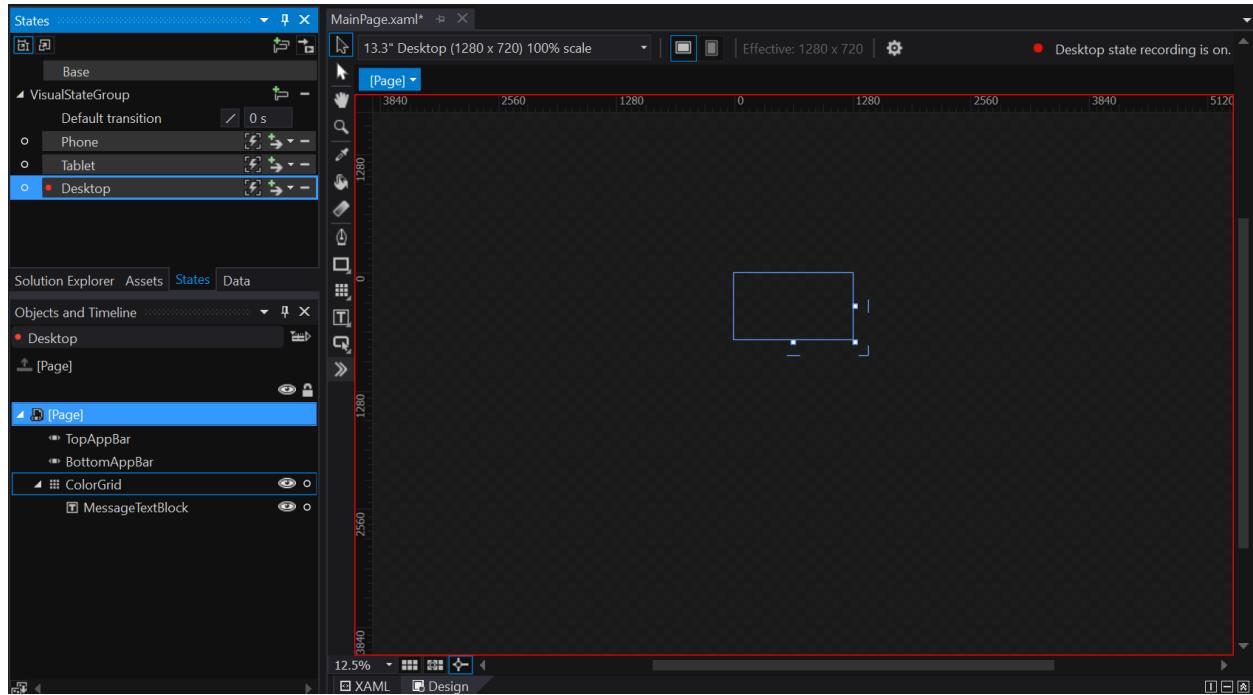
Next, I'll go to the States tab that's usually docked up on the upper left-hand corner. I can add a new "state" by adding a state group, so I'll click the small button in the upper right-hand corner (see red arrow, below). This creates a `VisualStateGroup`.



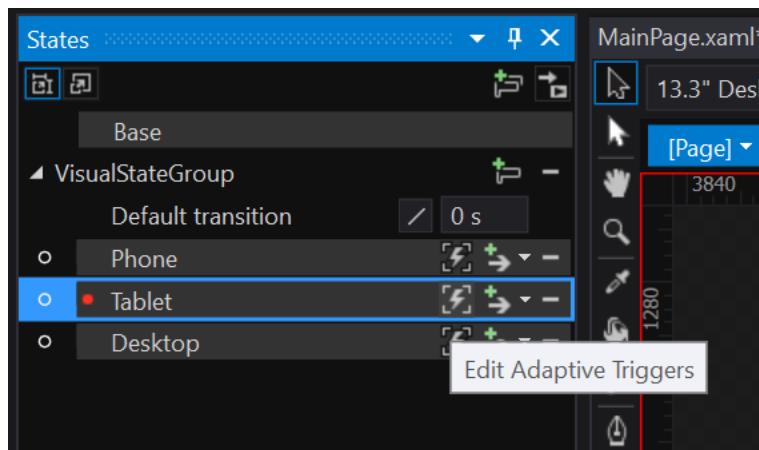
In the default `VisualStateGroup`, I'll click the small button to its right (see red arrow, below) to create a `VisualState`. I will call this state "Phone". I'll add another state called "Tablet". And add another state that I call "Desktop".



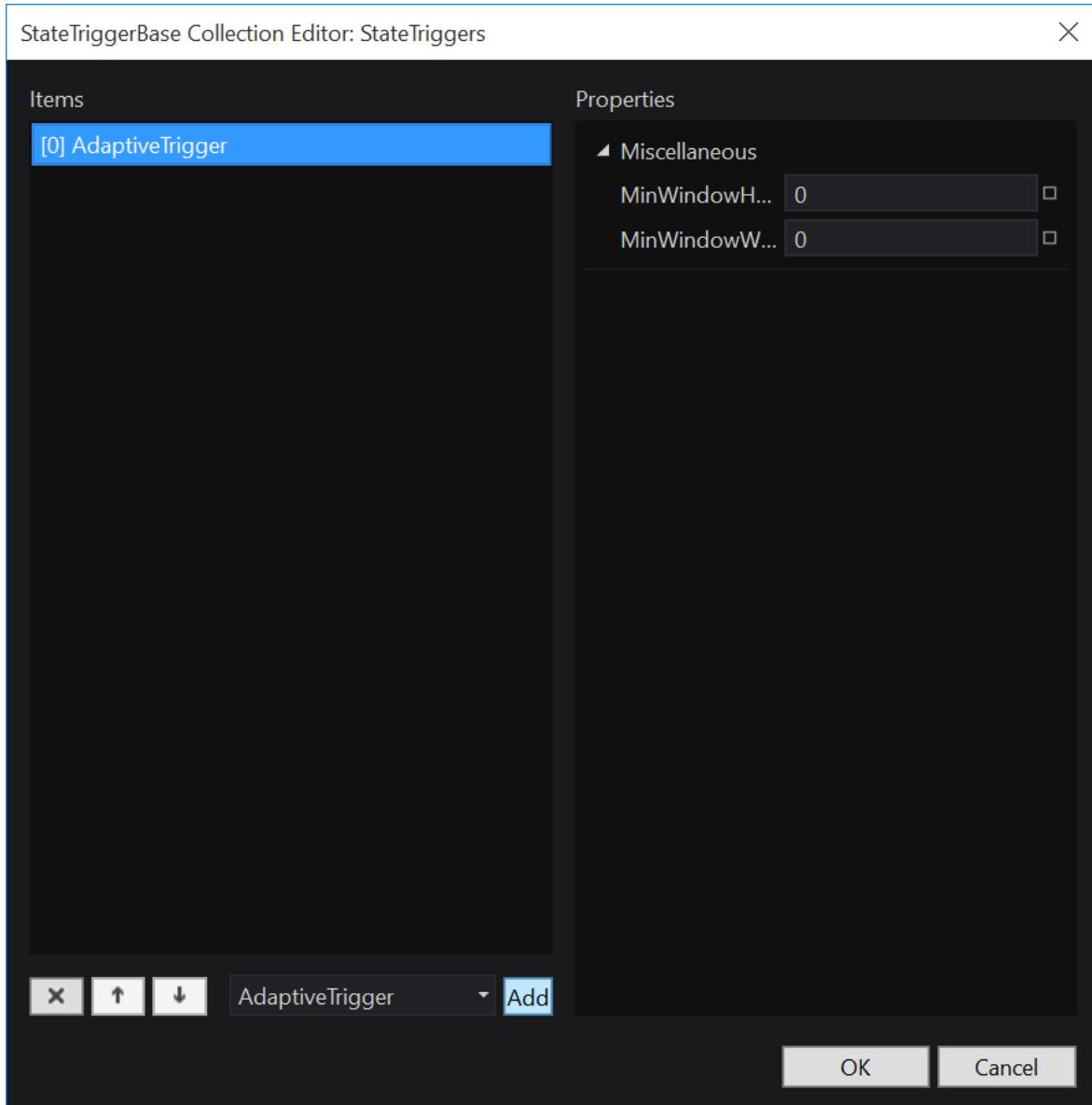
And notice that whenever I select any of these, that a little red light is selected on the left-hand side, and there are a red border all around the designer area. You might also see that Phone state recording is on.



When select the Tablet VisualState, any changes I make in the Design and Properties window will create VisualState Setters. To create the Triggers click the lightning bolt icon to the right:



This will open a dialog that allows you to first add an AdaptiveTrigger (bottom, left), then set its MinWindowHeight and / or MinWindowWidth.

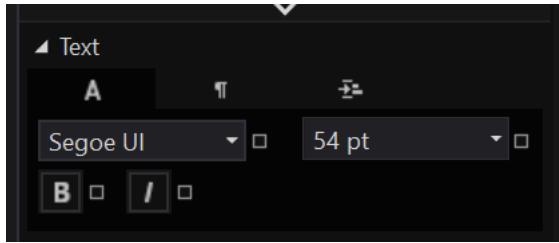


Since we're working with the Tablet state, I'll set the MinWindowWidth to 600, then click OK.

Note: The sizes I'm choosing are arbitrary numbers. You would want to test this for your own app.

I'll repeat the steps to create an AdaptiveTrigger for the Desktop VisualState setting the MinWindowWidth to 800.

When in Desktop mode I want the MessageTextBlock's FontSize to be 54. I select the MessageTextBlock in the Objects and Timeline window, then in the Properties window I locate the Text section and change the font's size to 54.



I'll repeat this process for the Tablet VisualState changing the MessageTextBlock's FontSize to 36.

Finally, I'll repeat this process for the Phone VisualState changing the MessageTextBlock's FontSize to 16.

Running the application, you should see the TextBlock's FontSize change as you resize the window.

If you peek at the XAML designer you can see all the XAML that was generated for you by Blend.

```
9
10    <Grid Name="ColorGrid">
11        <VisualStateManager.VisualStateGroups>
12            <VisualStateGroup x:Name="VisualStateGroup">
13                <VisualState x:Name="Phone">
14                    <VisualState.Setters>
15                        <Setter Target="MessageTextBlock.(TextBlock.FontSize)" Value="21.333"/>
16                    </VisualState.Setters>
17                </VisualState>
18                <VisualState x:Name="Tablet">
19                    <VisualState.Setters>
20                        <Setter Target="MessageTextBlock.(TextBlock.FontSize)" Value="48"/>
21                    </VisualState.Setters>
22                    <VisualState.StateTriggers>
23                        <AdaptiveTrigger MinWindowWidth="600"/>
24                    </VisualState.StateTriggers>
25                </VisualState>
26                <VisualState x:Name="Desktop">
27                    <VisualState.Setters>
28                        <Setter Target="MessageTextBlock.(TextBlock.FontSize)" Value="72"/>
29                    </VisualState.Setters>
30                    <VisualState.StateTriggers>
31                        <AdaptiveTrigger MinWindowWidth="800"/>
32                    </VisualState.StateTriggers>
```

I prefer to just type it in myself. While Blend does create nice clean XAML, but there are some cases like working with colors, and brushes where the generated XAML was overly verbose.

When you close down Blend, you'll return to Visual Studio and you may see that a message asking whether you want to re-load the file because it has changed.

I want to emphasize that this lesson is a foundational concept for building real Universal Windows Platform applications. In the next lesson we'll use the VisualStateManager to create a more realistic app that changes its layout based on the available screen resolution.

## UWP-038 - Working with Adaptive Layout

In this lesson we will cover a bit on adaptive layout. This takes what was covered in the previous lesson, talked about the nuts and bolts of actually using the Visual State Manager and AdaptiveTriggers to change attributes of objects in XAML based on screen size, to a higher level.

Adaptive layout is critical to understanding the Universal Windows Platform story, where we build one code base and we can use it across multiple form factors. Here is a creative example that illustrates this, found on "Wintellect's Blog" by Jeff Prosise, who wrote how 'To Build Adaptive UIs in Windows 10'.

<http://bit.do/adaptive-ui>

This lesson is based on Jeff's "Contoso Cookbook" example.



Pellentesque porta, mauris quis interdum vehicula,  
uma sapien ultrices velit, nec venenatis dui odio in  
augue. Cras posuere, enim a cursus convallis,  
neque turpis malesuada erat, ut adipiscing neque  
tortor ac erat.

80 minutes

And here's how it  
looks on my Windows phone:



Here is an example for how you could construct an adaptive layout in your MainPage.xaml

```

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

    <!-- Huge props to Jeff Prosise at Wintellect for this technique
        http://www.wintellect.com/ -->

    <Grid Name="LayoutRoot" Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="VisualStateGroup">
                <VisualState x:Name="Wide">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="800" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Target="First.(Grid.Row)" Value="0" />
                        <Setter Target="First.(Grid.Column)" Value="0" />
                        <Setter Target="Second.(Grid.Row)" Value="0" />
                        <Setter Target="Second.(Grid.Column)" Value="1" />
                        <Setter Target="Third.(Grid.Row)" Value="0" />
                        <Setter Target="Third.(Grid.Column)" Value="2" />

                        <Setter Target="First.(Grid.ColumnSpan)" Value="1" />
                        <Setter Target="Second.(Grid.ColumnSpan)" Value="1" />
                        <Setter Target="Third.(Grid.ColumnSpan)" Value="1" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="Narrow">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="0" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Target="First.(Grid.Row)" Value="0" />
                        <Setter Target="First.(Grid.Column)" Value="0" />
                        <Setter Target="Second.(Grid.Row)" Value="1" />
                        <Setter Target="Second.(Grid.Column)" Value="0" />
                        <Setter Target="Third.(Grid.Row)" Value="2" />
                        <Setter Target="Third.(Grid.Column)" Value="0" />

                        <Setter Target="First.(Grid.ColumnSpan)" Value="3" />
                        <Setter Target="Second.(Grid.ColumnSpan)" Value="3" />
                        <Setter Target="Third.(Grid.ColumnSpan)" Value="3" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
    </Grid>

```

Notice the different VisualState settings for either a Wide or a Narrow layout. And below all of that is a series of StackPanels

```

<Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
<ScrollViewer Grid.Row="1">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <StackPanel Name="First" Margin="20,20,0,0">
            <Image Source="Assets/Tibbles.jpg" HorizontalAlignment="Left" />
            <TextBlock>Information on my cat, Mr. Tibbles</TextBlock>
        </StackPanel>
        <StackPanel Name="Second" Grid.Row="1" Margin="20,20,0,0">
            <TextBlock TextWrapping="Wrap">
                Lorem ipsum dolor...
            </TextBlock>
        </StackPanel>
        <StackPanel Name="Third" Grid.Row="2" Margin="20,20,0,0">
            <TextBlock TextWrapping="Wrap">
                Nam sollicitudin justo ..
            </TextBlock>
        </StackPanel>
    </Grid>
</ScrollViewer>
</Grid>
</Page>

```

You can see how the image and text is organized within the StackPanels, forming 3 distinct sections (labeled accordingly), and when you run it in a wide viewport it would appear something like this

The screenshot shows a Windows application window titled "AdaptiveLayoutExample". Inside, there's a grid layout. The first column contains a photo of a brown cat sitting on a patterned rug. Below the photo is the caption "This is some information about Mr. Tibb". The second column contains three stacked sections. The top section has an image and some descriptive text. The middle section contains a single paragraph of text. The bottom section contains another paragraph of text. All text is wrapped to fit the width of the columns.

Lorem ipsum dolor sit amet,  
 consectetur adipiscing elit. Cras id orci  
 iaculis, aliquet nibh at, dictum lorem.  
 Vivamus tempus tristique sollicitudin.  
 Etiam interdum et lectus semper  
 molestie. Phasellus lobortis felis quis  
 risus posuere, id molestie mi sagittis.  
 Cras odio leo, dictum vitae euismod et,  
 lacinia non lectus. Integer quis massa

Nam sollicitudin justo quis consequat  
 molestie. Etiam dictum sodales tellus,  
 ut consectetur magna sodales in.  
 Phasellus viverra volutpat porttitor.  
 Pellentesque sed condimentum neque  
 in ultrices ex ac lacus tincidunt, eget  
 euismod urna cursus. Donec tempor  
 mauris leo, ac cursus nisl tempus a.  
 Aliquam dignissim eleifend lorem a

However, when the viewport dramatically narrows (such as on a Phone) it stacks the columns on top of each other, rendering a result that looks something like this



This is some information about Mr. Tibbles.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras id orci iaculis, aliquet nibh at, dictum lorem. Vivamus tempus tristique sollicitudin. Etiam interdum et lect semper molestie. Phasellus lobortis felis qu

Now, this switching between visual states is done up near the top of the code, within the VisualStateManager

```
<VisualStateGroup x:Name="VisualStateGroup">
    <VisualState x:Name="Wide">
        <VisualState.StateTriggers>
            <AdaptiveTrigger MinWindowWidth="800" />
        </VisualState.StateTriggers>
        <VisualState.Setters>
            <Setter Target="First.(Grid.Row)" Value="0" />
            <Setter Target="First.(Grid.Column)" Value="0" />
            <Setter Target="Second.(Grid.Row)" Value="0" />
            <Setter Target="Second.(Grid.Column)" Value="1" />
            <Setter Target="Third.(Grid.Row)" Value="0" />
            <Setter Target="Third.(Grid.Column)" Value="2" />
        
```

So, in the wide states, it takes constructs the StackPanel (here labeled, "First," "Second," "Third") with the row/column arrangement you would expect to find in a wide viewport. And, the column span is also set accordingly below that

```
<Setter Target="First.(Grid.ColumnSpan)" Value="1" />
<Setter Target="Second.(Grid.ColumnSpan)" Value="1" />
<Setter Target="Third.(Grid.ColumnSpan)" Value="1" />
```

And, of course you can understand the narrow state using much the same kind of reasoning

So, essentially the different visual states activate depending on if the width of the device (or the size of the window on the device) is either less than 800 pixels (the narrow state), or greater than 800 pixels (the wide state)

```
<VisualState x:Name="Narrow">
  <VisualState.StateTriggers>
    <AdaptiveTrigger MinWindowWidth="0" />
<VisualState x:Name="Wide">
  <VisualState.StateTriggers>
    <AdaptiveTrigger MinWindowWidth="800" />
```

So, what this demonstrates is that you have the tools to make your application change layout based on the break points you set, and you get to decide how those layout changes need to happen. It's all up to you. So, you will want to choose carefully, and test, thinking each step of the way what would make sense on the phone versus sitting on the couch, scrolling through and working with a larger application. Consider how the layout changes, as well as how the user expects to interact with the UI depending on these different contexts.

## UWP-039 - Adaptive Layout with Device Specific Views

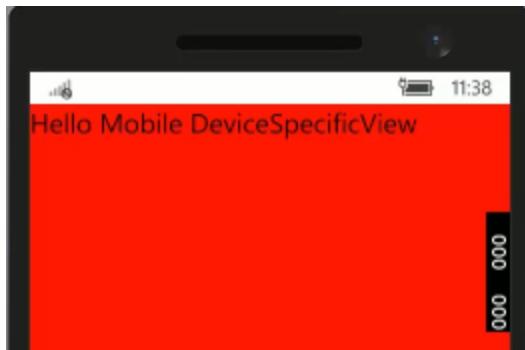
Having now looked at how to use AdaptiveTriggers, and the VisualStateManager, to change the layout of your application based on the current window size, let's now take a look at a second technique that allows you to create a dedicated view for a given device family that your application could potentially run on.

This would work by identifying the device that your app is running on, and then triggering a dedicated view, or what's called the "DeviceSpecificView." It's a very simple technique that Microsoft has made extremely easy to use, however there are a few advanced techniques which you will see in a great article that will be presented to you later. But, first, here is a demo of this technique in action.

Below is an example app running as an ordinary Windows app



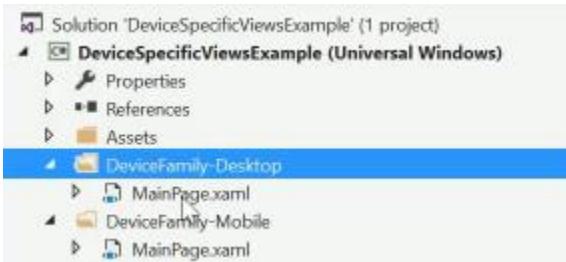
And, then running on Windows Phone (via the Visual Studio emulator)



On the phone you can see a dramatically different background and font shows up. Also, notice that the wording is changed to reflect the platform the app is running on.

Let's go through the steps detailing how you could get this kind of result (bear in mind, this is the simplest possible way of doing this in order to just show the technique used. You can take this as far as you want)

Within your project you would create 2 separate folders for the Desktop/Mobile settings, respectively. Notice how each folder has its own MainPage.xaml



And in each folder's MainPage.xaml are the specific settings for each intended device

```
<Page
    x:Class="DeviceSpecificViewsExample.DeviceFamily_Desktop.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:DeviceSpecificViewsExample.DeviceFamily_Desktop"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    
    <Grid Background="Blue">
        <TextBlock Text="Hello Desktop DeviceSpecificView" FontSize="36" />
    </Grid>
</Page>

<Page
    x:Class="DeviceSpecificViewsExample.DeviceFamily_Mobile.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:DeviceSpecificViewsExample.DeviceFamily_Mobile"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    
    <Grid Background="Red">
        <TextBlock Text="Hello Mobile DeviceSpecificView" FontSize="18" />
    </Grid>
</Page>
```

It may immediately strike you that this is a much cleaner, simpler solution to adaptive layout than using the `VisualStateManager` in the previous lesson. That is definitely one of the benefits of using this approach. However, you may still want to use a `VisualStateManager` or `AdaptiveTriggers` for your application because there are different screen resolutions, even within the context of these different device families.

For more information on these techniques, including advanced uses, refer to this helpful resource

<http://bit.do/device-specific-views>

## UWP-040 - Data Binding to the GridView and ListView Controls

In this lesson we will talk about data binding, and a couple of Controls that will allow us to bind data in a grid/list type fashion. Data binding has been around for a long time, and basically it's the process of taking data from a source, and then associating it with elements of a user interface.

Consider having an object in your code – one that describes a book, or a car, etc – and you then want to be able to take that data and display it onto a user interface. That is essentially what data binding lets you do. It lets you specify a template, of sorts, that each object instance uses in order to be represented within the user interface.

Let's start by looking at a basic example project, for demonstration purposes. Here is a simple grid-like layout which we can use as our starting point

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
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}" Margin="20">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="100" />
    </Grid.RowDefinitions>

    <TextBlock Grid.Row="1"
              Name="ResultTextBlock"
              FontSize="24"
              Foreground="Red"
              FontWeight="Bold"
              Margin="0,20,0,0" />

</Grid>
</Page>
```

Now, let's refer to some of our basic class information, which will just hold basic data that we can populate later

```
namespace xBindDataExample.Models
{
    public class Book
    {
        public int BookId { get; set; }
        public string Title { get; set; }
        public string Author { get; set; }
        public string CoverImage { get; set; }
    }
}
```

And, next we have a BookManager class that is responsible for populating instances of each Book, and then adding them to a List collection

```
public class BookManager
{
    public static List<Book> GetBooks()
    {
        var books = new List<Book>();

        books.Add(new Book { BookId = 1, Title = "Vulpate", Author = "Futurum", CoverImage="Assets/1.png" });
        books.Add(new Book { BookId = 2, Title = "Mazim", Author = "Sequiter Que", CoverImage = "Assets/2.png" });
        books.Add(new Book { BookId = 3, Title = "Elit", Author = "Tempor", CoverImage = "Assets/3.png" });
    }
}
```

Now, in MainPage.xaml, one of the things you will need to do is establish a GridView template, describing how each individual instance of Book will appear on-screen.

```
<Grid Background="{ThemeResource ApplicationPageBackground}"
      <Grid.RowDefinitions>
          <RowDefinition Height="*" />
          <RowDefinition Height="100" />
      </Grid.RowDefinitions>

      <GridView>
          <GridView.ItemTemplate>
              <DataTemplate>
                  <Image Width="150" />
                  <TextBlock FontSize="16" />
                  <TextBlock FontSize="10" />
              </DataTemplate>
          </GridView.ItemTemplate>
      </GridView>

      <TextBlock Grid.Row="1"

```

The next thing is to tell the GridView what it should, ultimately, bind to. In order to do this, set the ItemSource property to what you want it to bind to

```
<GridView ItemsSource="{x:Bind Books}">
    <GridView.ItemTemplate>
        <DataTemplate>
            <Image Width="150" />
```



This, in turn, will refer to a public property called Books in MainPage.xaml. This property will be of type List<Book>, and will store the list of Books returned from the BookManager.GetBooks()

But before that, we have to set a property on the DataTemplate called “x:DataType” and set it to type Book. And, in order to reference this class type in my XAML, you will need to add a XML namespace up at the page level.

```
xmlns:data="using:xBindDataExample.Models"
mc:Ignorable="d">

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="100" />
    </Grid.RowDefinitions>

    <GridView ItemsSource="{x:Bind Books}">
        <GridView.ItemTemplate>
            <DataTemplate x:DataType="Book">
```



This namespace reference just refers to where the Book class is located. In this case, it's in the Models namespace, as it is in the “Models” folder within the project.



With that set up, you should now be able to properly reference that class by changing the x:DataType as follows

```
<DataTemplate x:DataType="data:Book">
```

Now, we need to set properties for each Control, and have them refer to each property of the Book class.

```
<GridView ItemsSource="{x:Bind Books}">
    <GridView.ItemTemplate>
        <DataTemplate x:DataType="data:Book">
            <StackPanel>
                <Image Width="150" Source="{x:Bind CoverImage}" />
                <TextBlock FontSize="16" Text="{x:Bind Title}" />
                <TextBlock FontSize="10" Text="{x:Bind Author}" />
            </StackPanel>
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>
```



With that out of the way, all that's left to do is to populate that property called "Books" – that we're ultimately binding to - within MainPage.xaml.cs

```
<GridView ItemsSource="{x:Bind Books}">
    <GridView.ItemTemplate>
        <DataTemplate>
            <Image Width="150" />
```



In MainPage.xaml.cs make sure you have the necessary using statement relative to your project, to make the binding happen

```
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;
using xBindDataExample.Models;
```

And then populate those instances

```
public sealed partial class MainPage : Page
{
    private List<Book> Books;

    public MainPage()
    {
        this.InitializeComponent();
        Books = BookManager.GetBooks();
    }
}
```

And when the program runs, the XML will now magically bind to that data set

xBindDataExample



To pretty this up a bit, you can center the elements back in the XML

```
<StackPanel HorizontalAlignment="Center">
    <Image Width="150" Source="{x:Bind CoverImage}" />
    <TextBlock FontSize="16" Text="{x:Bind Title}" HorizontalAlignment="Center" />
```

Now, to retrieve information when selecting on each item let's set some attributes to the GridView. First, make it clickable.

```
<GridView ItemsSource="{x:Bind Books}" IsItemClickEnabled="True">
```

And then assign it to an event, add the following

```
    <IsItemClickEnabled="True" ItemClick="GridView_ItemClick">
```

Which you can edit in the MainPage.xaml.cs

```
private void GridView_ItemClick(object sender, ItemClickEventArgs e)
{
    var book = (Book)e.ClickedItem;
}
```

The above code gives you the instance of the item that was clicked on. And then outputs that book title to the screen (as shown below)

```
private void GridView_ItemClick(object sender, ItemClickEventArgs e)
{
    var book = (Book)e.ClickedItem;
    ResultTextBlock.Text = "You selected " + book.Title;
}
```



Now, if you want to reuse the styling information as a template (especially for multiple app sections that refer to it), you can take it out of the `GridView.ItemTemplate` (cut)

```
<GridView.ItemTemplate>
    <DataTemplate x:DataType="data:Book">
        <StackPanel HorizontalAlignment="Center">
            <Image Width="150" Source="{x:Bind CoverImage}" />
            <TextBlock FontSize="16" Text="{x:Bind Title}" Horizontal...
            <TextBlock FontSize="10" Text="{x:Bind Author}" Horizontal...
        </StackPanel>
    </DataTemplate>
```

And insert it into a page resource at the top of your document (paste)

```
mc:Ignorable="d">
<Page.Resources>
    <DataTemplate x:DataType="data:Book">
        <StackPanel HorizontalAlignment="Center">
            <Image Width="150" Source="{x:Bind CoverImage}" />
            <TextBlock FontSize="16" Text="{x:Bind Title}" Horizontal...
            <TextBlock FontSize="10" Text="{x:Bind Author}" Horizontal...
        </StackPanel>
    </DataTemplate>
</Page.Resources>
```

And then give that an `x:Key`, so you can refer to it in the rest of the document

```
<DataTemplate x:DataType="data:Book" x:Key="BookDataTemplate">
```

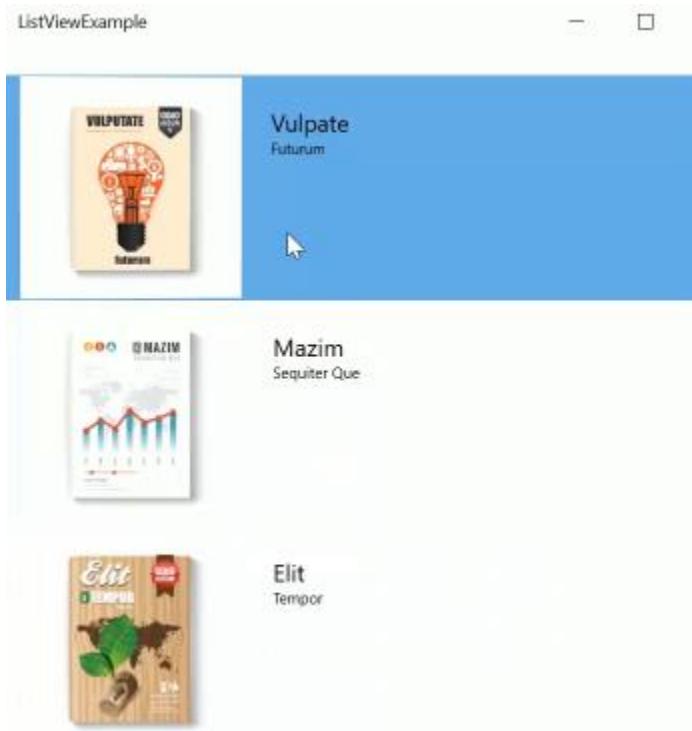
And reference it within the `GridView` as follows

```
<GridView ItemsSource="{x:Bind Books}"
    IsItemClickEnabled="True"
    ItemClick="GridView_ItemClick"
    ItemTemplate="{StaticResource BookDataTemplate}">
</GridView>
```

One other quick thing to demonstrate is how you can take this basic styling structure and, instead of displaying in a grid-like format, have it display in a list-like format (the information becomes stacked, one on top of the other)

```
<DataTemplate x:Key="BookListDataTemplate" x:DataType="data:Book">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Left">
        <Image Name="image" Source="{x:Bind CoverImage}" HorizontalAlignm
        <StackPanel Margin="20,20,0,0">
            <TextBlock Text="{x:Bind Title}" HorizontalAlignment="Left" F
            <TextBlock Text="{x:Bind Author}" HorizontalAlignment="Left"
        </StackPanel>
    </StackPanel>
</DataTemplate>
```

And with such a simple modification you can now display in a list format that looks something like this



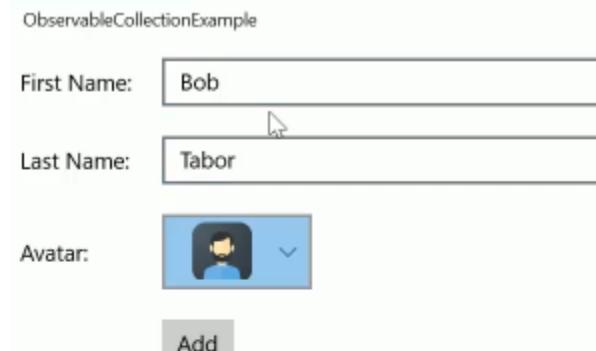
So, now we can take what we learned about the DataTemplate, how to bind to the data, and the difference between the GridView and the ListView, and apply that to the coming lessons.

## UWP-041 - Keeping Data Controls Updated with ObservableCollection

Up to this point, whenever we were binding to data, the only scenario we dealt with was binding to a static set of items, which would bind that data, once, whenever the app first loads. However, we need to discuss another scenario, in which the collection of data gets updated, and we then need to programmatically add/remove new items in the List.

In order for this to work, we would have to tell the GridView that there is some new information to bind to. For that to happen, you have to use a different type of collection than just an ordinary List, and that would be to use an ObservableCollection instead.

To illustrate this point, here is a very simple example application. Imagine an app that takes in contact information and adds it to a contact list, which can then be displayed back in the app once the data binding takes place in real time.



The goal is to have the contact information appear here, once it is added. For that, we need an

**ObservableCollection**

The contact information isn't appearing because, up to this point, the application has been using an ordinary List. Here we make a simple change in MainPage.xaml.cs to step in the right direction

```
public sealed partial class MainPage : Page
{
    private List<Icon> Icons;
    //private List<Contact> Contacts;
    private ObservableCollection<Contact> Contacts;
```



By making the collection an ObservableCollection, your DataBoundControl - whether it's a GridView, ListView, etc - will be watching the collection for changes. And whenever a change happens, the ObservableCollection<Contact>, will tell the the DataBoundControl to re-bind to it.

Now, the x:Bind references in MainPage.xaml have already been configured, in the background, to facilitate this binding process with a List (since, in this example, we started out with a simple List). It is configured to work fine for ordinary Lists which pre-compiles the binding procedure and establishes that right before runtime.

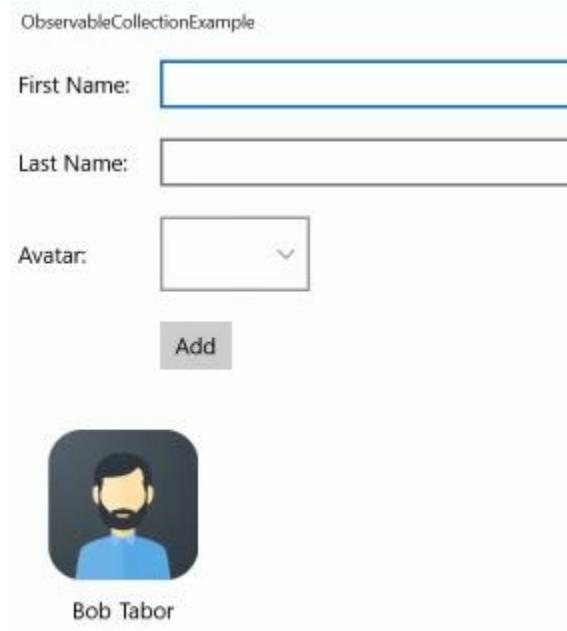
But when changing your binding process from one that is with a List, to one that is an ObservableCollection you are changing this process from pre-compiling, to happening at runtime. So you have to re-initialize this configuration. A simple way to do this is to highlight all of your code that now is supposed to bind to the ObservableCollection

```
<GridView Grid.Row="2" ItemsSource="{x:Bind Contacts}" Margin="20">
    <GridView.ItemTemplate>
        <DataTemplate x:DataType="data:Contact">
            <StackPanel HorizontalAlignment="Center" Margin="10">
                <Image Source="{x:Bind AvatarPath}" Width="100" Height="100" />
                <StackPanel Orientation="Horizontal" Margin="0,10,0,0">
                    <TextBlock Text="{x:Bind FirstName}" Margin="0,0,5,0" />
                    <TextBlock Text="{x:Bind LastName}" />
                </StackPanel>
            </StackPanel>
        </DataTemplate>
    </GridView.ItemTemplate>

</GridView>
```

And simply cut it, then wait a few seconds and paste it back in. What that does it kicks off the background configuration to recognize it now needs to bind with an ObservableCollection.

Now, the application can run as intended, doing the binding at runtime.



When searching around for further explanations related to data binding, you may come across something called “MVVM.” This is related to binding to data with your Universal Windows Platform app, your Windows 8.1, or Windows Phone 8.1 app, or even the Windows Presentation Foundation app; all of these XAML-based platforms.

MVVM stands for Model-View-ViewModel. It is a design pattern for writing code on the user interface that binds to data. What we showed in this example, with the ObservableCollection, is really just a building block/foundational concept towards learning more about MVVM.

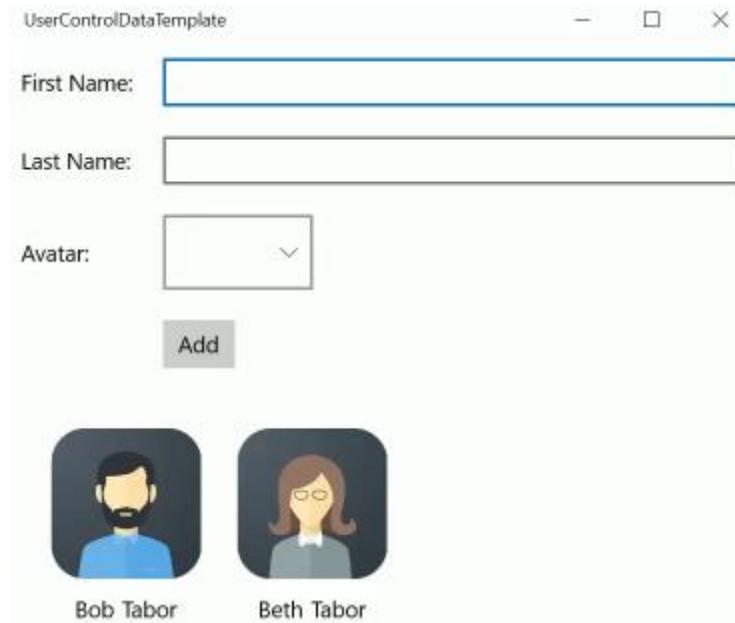
This topic is covered in depth in the previous version of this series, the Windows Phone 8.1 Development for Absolute Beginners series on Channel9. If you want to learn more about MVVM, you'll want to take a look at all lessons that have to do with ObservableCollection, MVVM, or the interface called INotifyPropertyChanged at the following URL

<https://channel9.msdn.com/Series/Windows-Phone-8-1-Development-for-Absolute-Beginners>

## UWP-042 - Utilizing User Controls as Data Templates

This lesson will cover a challenge you're going to run in into if you attempt to use VisualStateManager and AdaptiveTriggers to resize items inside of your DataTemplate for your GridView, List View Control, etc; anything that needs an item template.

In our example Contact app, we could imagine that as the data grows when entries get added, we would want everything to resize and accommodate those changes.

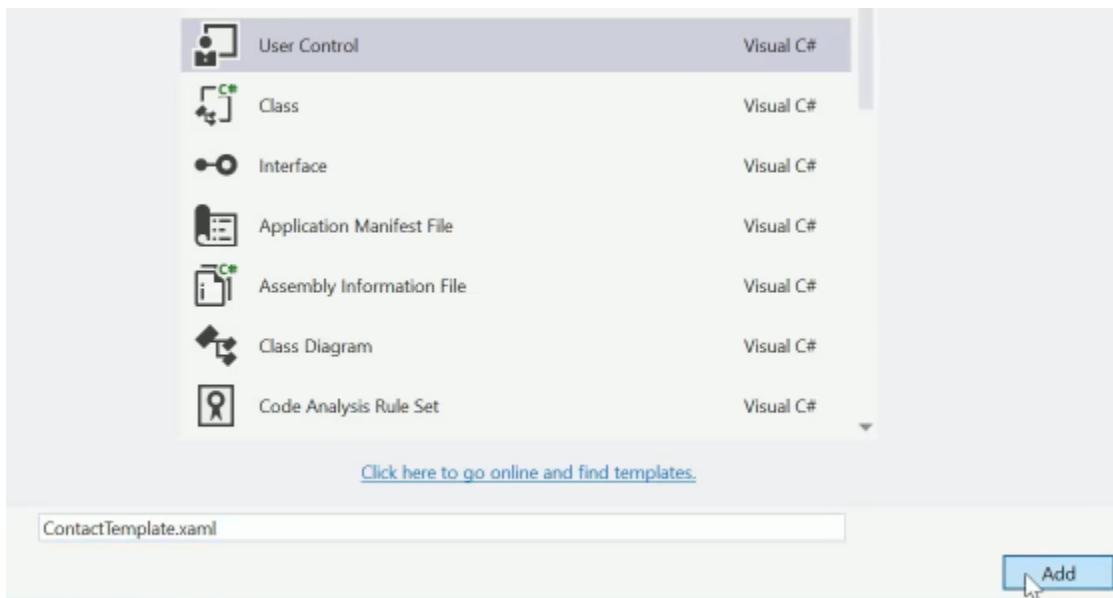


This, unfortunately, won't work by incorporating the data template in-line, the way that we have covered in previous lessons (as shown below).

```
<GridView.ItemTemplate>
    <DataTemplate x:DataType="data:Contact">
        <StackPanel HorizontalAlignment="Center" Margin="10">
            <Image Name="AvatarImage" Source="{x:Bind AvatarPath}" />
            <StackPanel Orientation="Horizontal" Margin="0,10,0,0">
                <TextBlock Text="{x:Bind FirstName}" Margin="0,0,5,0" />
                <TextBlock Text="{x:Bind LastName}" />
            </StackPanel>
        </StackPanel>
    </DataTemplate>
</GridView.ItemTemplate>
```

The solution is to transplant the in-line data - that forms the body of the template - into its own code file. And then inside of that code file, use a VisualStateManager and AdaptiveTriggers, etc, to resize on demand. To do this, we will first have to add a User Control. In Visual Studio's menu, go to

Project > Add New Item...



After that, you will want to take out that StackPanel information we mentioned earlier (shown as removed, below)

```
<GridView.ItemTemplate>
    <DataTemplate x:DataType="data:Contact">
        </DataTemplate>
    </GridView.ItemTemplate>
```

And put it into the User Control file you just created (in this case "ContactTemplate.xaml")

```
<UserControl
    x:Class="UserControlDataTemplate.ContactTemplate"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:UserControlDataTemplate"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300"
    d:DesignWidth="400">

    <StackPanel HorizontalAlignment="Center" Margin="10">
        <Image Name="AvatarImage" Source="{x:Bind AvatarPath}" Width="100" />
        <StackPanel Orientation="Horizontal"
                    Margin="0,10,0,0"
                    HorizontalAlignment="Center">
            <TextBlock Text="{x:Bind FirstName}" Margin="0,0,5,0" />
            <TextBlock Text="{x:Bind LastName}" />
        </StackPanel>
    </StackPanel>
</UserControl>
```

And now to reference this external file back within the MainPage.xaml template, access it as follows

```
<GridView.ItemTemplate>
    <DataTemplate x:DataType="data:Contact">
        <local>ContactTemplate
            HorizontalAlignment="Stretch"
            VerticalAlignment="Stretch" />
    </DataTemplate>
</GridView.ItemTemplate>
```

The next thing to do is actually use the name of the data we will be binding to.

```
<StackPanel HorizontalAlignment="Center" Margin="10">
    <Image Name="AvatarImage" Source="{x:Bind Contact.AvatarPath}" Width="
    <StackPanel Orientation="Horizontal"
        Margin="0,10,0,0"
        HorizontalAlignment="Center">
        <TextBlock Text="{x:Bind Contact.FirstName}" Margin="0,0,5,0" />
        <TextBlock Text="{x:Bind Contact.LastName}" />
    </StackPanel>
</StackPanel>
```

And in the ContactTemplate.xaml.cs, expose a new property called Contact for our UserControl that just returns the object's DataContext

```
namespace UserControlDataTemplate
{
    public sealed partial class ContactTemplate : UserControl
    {
        public Models.Contact Contact { get { return this.DataContext as Models.Contact; } }
```

This DataContext gets passed in through the MainPage.xaml

```
<GridView.ItemTemplate>
    <DataTemplate x:DataType="data:Contact">
        <local>ContactTemplate
            HorizontalAlignment="Stretch"
            VerticalAlignment="Stretch" />
    </DataTemplate>
</GridView.ItemTemplate>
```

That's the technical explanation, however, in practice just consider the Contact property as just being templated code.

One more thing, and that is whenever the DataContext is changed, there will be a DataContextChanged event, and we add to it a lambda expression, which is basically just an in-line method definition. And this lambda calls Bindings.Update() upon event firing. Note that, Bindings.Update() is actually being generated for us because we used the x:Bind statement - which is responsible for updating the data being bound to.

```
public sealed partial class ContactTemplate : UserControl
{
    public Models.Contact Contact { get { return this.DataContext as
        public ContactTemplate()
    {
        this.InitializeComponent();

        this.DataContextChanged += (s, e) => Bindings.Update();
    }
}
```

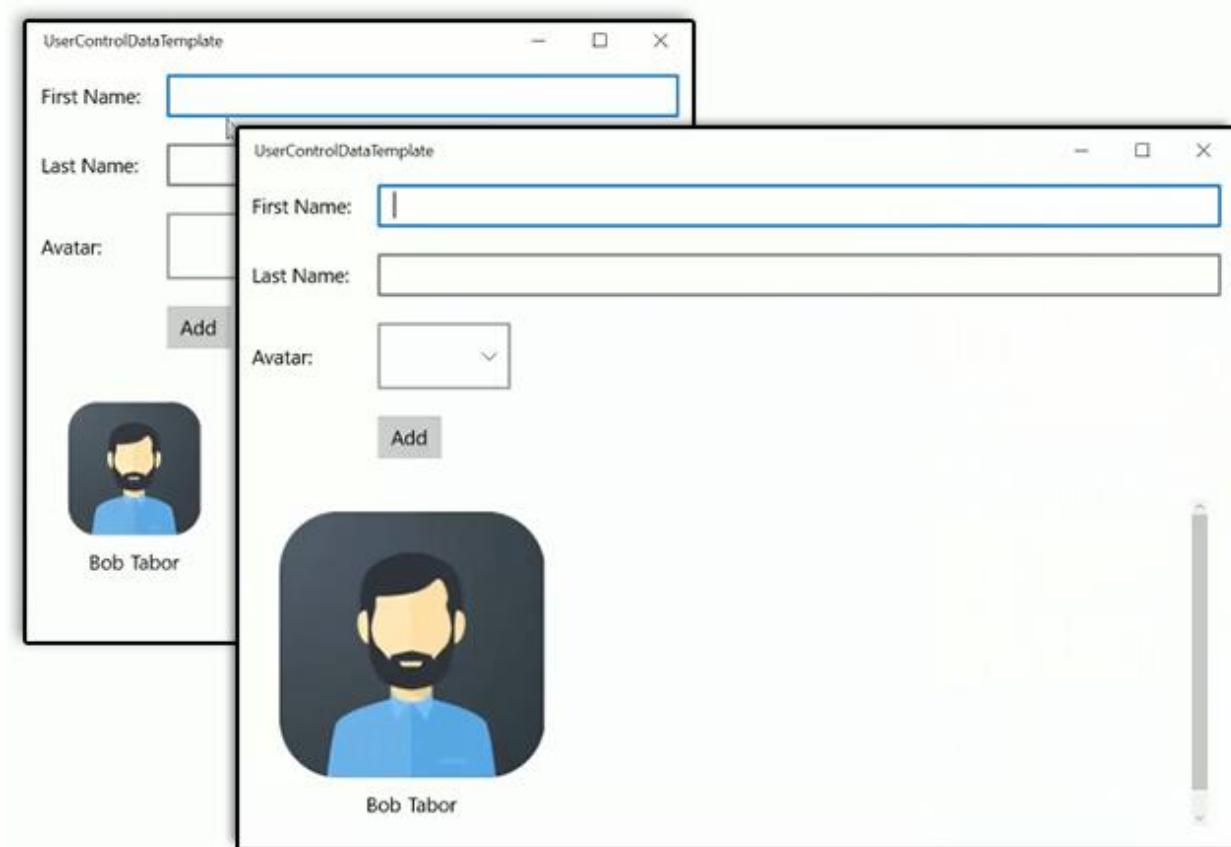
The next step is to define a VisualStateManager, so that whenever we resize the window, the avatar image will resize as well (in this case, from width/height = 100, to width/height = 200). To do that, here we cut the entire existing VisualStateManager from MainPage.xaml, and paste it

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="NarrowLayout">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="0"/>
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="AvatarImage.Width" Value="100" />
                <Setter Target="AvatarImage.Height" Value="100" />
            </VisualState.Setters>
        </VisualState>
        <VisualState x:Name="WideLayout">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="600"/>
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="AvatarImage.Width" Value="200" />
                <Setter Target="AvatarImage.Height" Value="200" />
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

And paste it into ContactTemplate.xaml right underneath the StackPanel (where the red box denotes, below)

```
d:DesignHeight="300"  
d:DesignWidth="400">  
  
<StackPanel HorizontalAlignment="Center" Margin="10">  
|    
  
    <Image Name="AvatarImage" Source="{x:Bind Contact.AvatarPath}" Width="100'  
    <StackPanel Orientation="Horizontal"  
                Margin="0,10,0,0"  
                HorizontalAlignment="Center">  
            <TextBlock Text="{x:Bind Contact.FirstName}" Margin="0,0,5,0" />  
            <TextBlock Text="{x:Bind Contact.LastName}" />  
        </StackPanel>  
    </StackPanel>  
</UserControl>
```

Now, when you run the application and resize the app it should scale the image accordingly



## [UWP-043 - Cheat Sheet Review: Adaptive Layout, Data Binding](#)

### [UWP-037 - Utilizing the VisualStateManager to Create Adaptive Triggers](#)

VisualStateManager manages changes to XAML element attributes based on screen size using Adaptive Triggers (MinWindowWidth, MinWindowHeight) and Setters (to change target property values).

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="NarrowLayout">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0"/>
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="MyImage.Width" Value="200" />
        <Setter Target="LayoutGrid.Background" Value="Blue" />
      </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="WideLayout">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="600"/>
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="MyImage.Width" Value="200" />
        <Setter Target="LayoutGrid.Background" Value="Blue" />
      </VisualState.Setters>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

### [UWP-038 - Working with Adaptive Layout](#)

<http://bit.do/adaptive-ui>

Use adaptive triggers to move StackPanels (filled with content) into different Grid cells.

### [UWP-039 - Adaptive Layout with Device Specific Views](#)

Choose different versions of files to use based on the device running the app.

Example:

/DeviceFamily-Mobile

-- MainPage.xaml

/DeviceFamily-Desktop

-- MainPage.xaml

... or use a different file suffix for different device specific views:

MainPage.DeviceFamily-Mobile.xaml

MainPage.DeviceFamily-Desktop.xaml

<http://bit.do/device-specific-views>

#### UWP-40 - Data Binding to the GridView and ListView Controls

Bind to a List<T> where T is a POCO in your app (I put mine in the Models folder).

```
<Page
...
xmlns:data="using:xBindDataExample.Models">
<Page.Resources>
    <DataTemplate x:DataType="data:Book" x:Key="BookDataTemplate">
        <StackPanel HorizontalAlignment="Center">
            <Image Source="{x:Bind CoverImage}" />
            <TextBlock Text="{x:Bind Title}" />
            <TextBlock Text="{x:Bind Author}" />
        </StackPanel>
    </DataTemplate>
</Page.Resources>
...
<GridView ItemsSource="{x:Bind Books}"
    IsItemClickEnabled="True"
    ItemClick="GridView_ItemClick"
    ItemTemplate="{StaticResource BookDataTemplate}">
</GridView>
...
```

#### Code Behind

```
public sealed partial class MainPage : Page
{
    private List<Book> Books;

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

```

        Books = BookManager.GetBooks();
    }

    private void GridView_ItemClick(object sender, ItemClickEventArgs e)
    {
        var book = (Book)e.ClickedItem;
        ResultTextBlock.Text = "You selected " + book.Title;
    }
}

```

#### UWP-041 - Keeping Data Controls Updated with ObservableCollection

If the contents of List<T> will change, make sure you use ObservableCollection<T> instead!

#### UWP-042 - Utilizing User Controls as Data Templates

If you intend to combine the VisualStateManager with data bound controls, you will need to put your Data Template code inside of a User Control, then create the VisualStateManager code inside of the User Control.

- 1) Create a User Control.
- 2) Cut the Data Template out of the MainPage.xaml and copy it into the User Control.
- 3) Reference the User Control from inside the Data Template:

```
<local:ContactTemplate HorizontalAlignment="Stretch" VerticalAlignment="Stretch" />
```

- 4) Modify the contents of the User Control changing x:Bind statements to utilize object.property notation:

```

<UserControl>
    <StackPanel>
        <Image Source="{x:Bind Contact.AvatarPath}" />
        <TextBlock Text="{x:Bind Contact.FirstName}" />
        <TextBlock Text="{x:Bind Contact.LastName}" />
    </StackPanel>
</UserControl>

```

- 5) Add this in the User Control's Code Behind:

```
public Models.Contact Contact { get { return this.DataContext as Models.Contact; } }
```

```

public ContactTemplate()
{

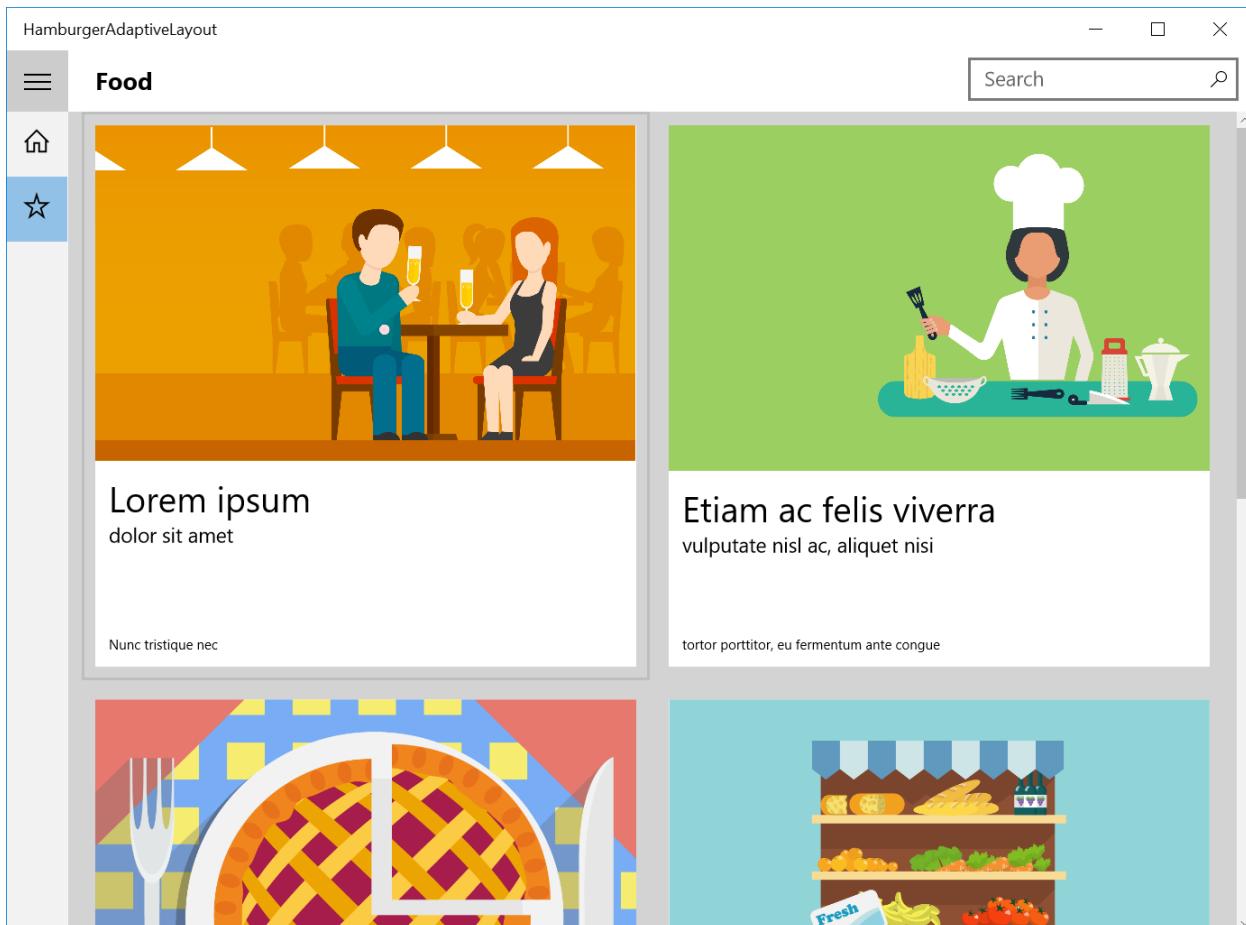
```

```
this.InitializeComponent();
this.DataContextChanged += (s, e) => Bindings.Update();
}
```

## UWP-044 - Adeptly Adaptive Challenge

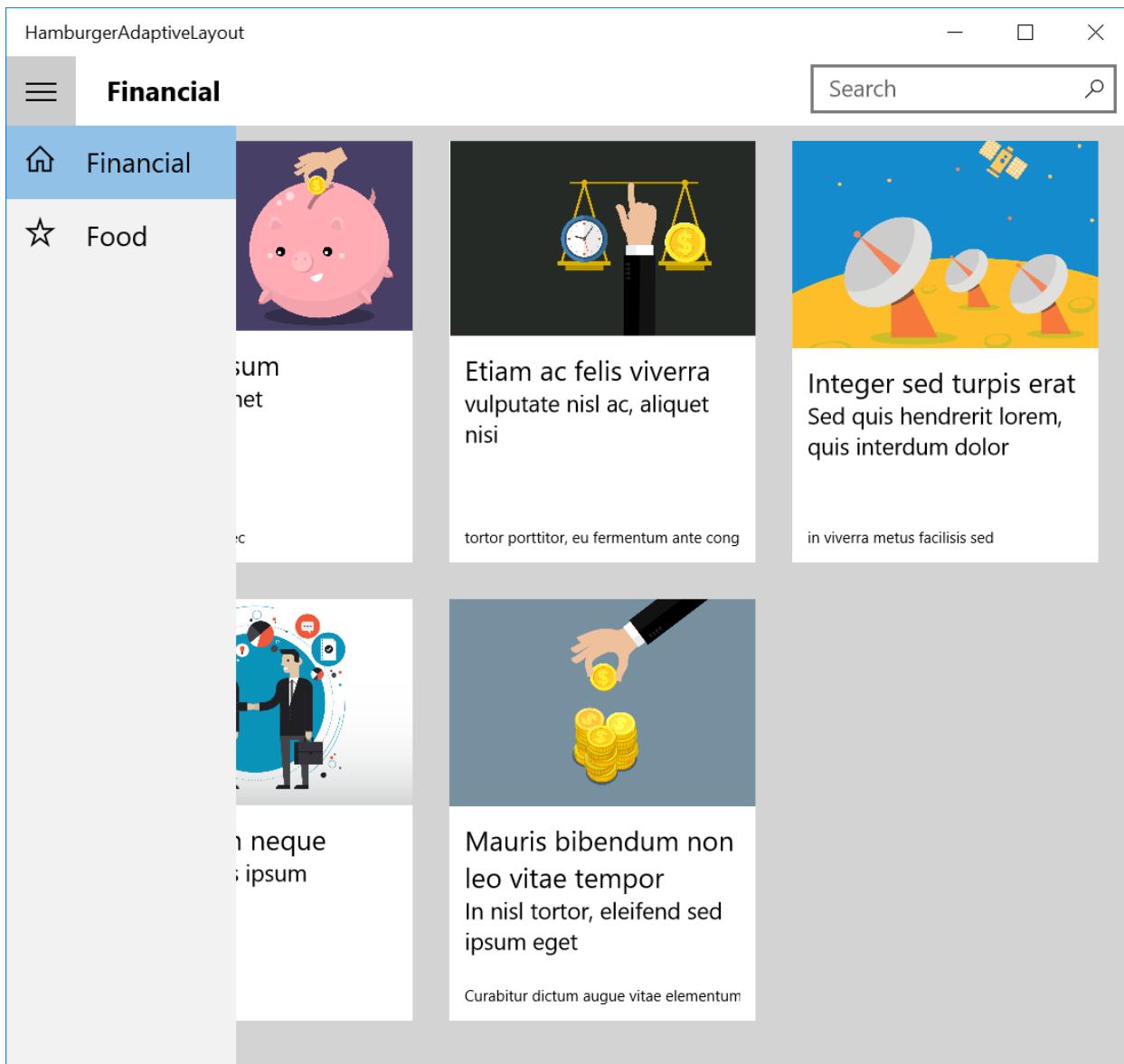
The final solution I'm giving you is the Adeptly Adaptive Challenge. You will be asked to create an adaptive solution for a fake news app. The key to this app is that, as the window size is changed the size and layout of the individual news items, the visibility of the search textbox, and the font sizes used in the app are affected as well.

Here's an example of what the completed application should look like when the window is expanded to the largest form factor:



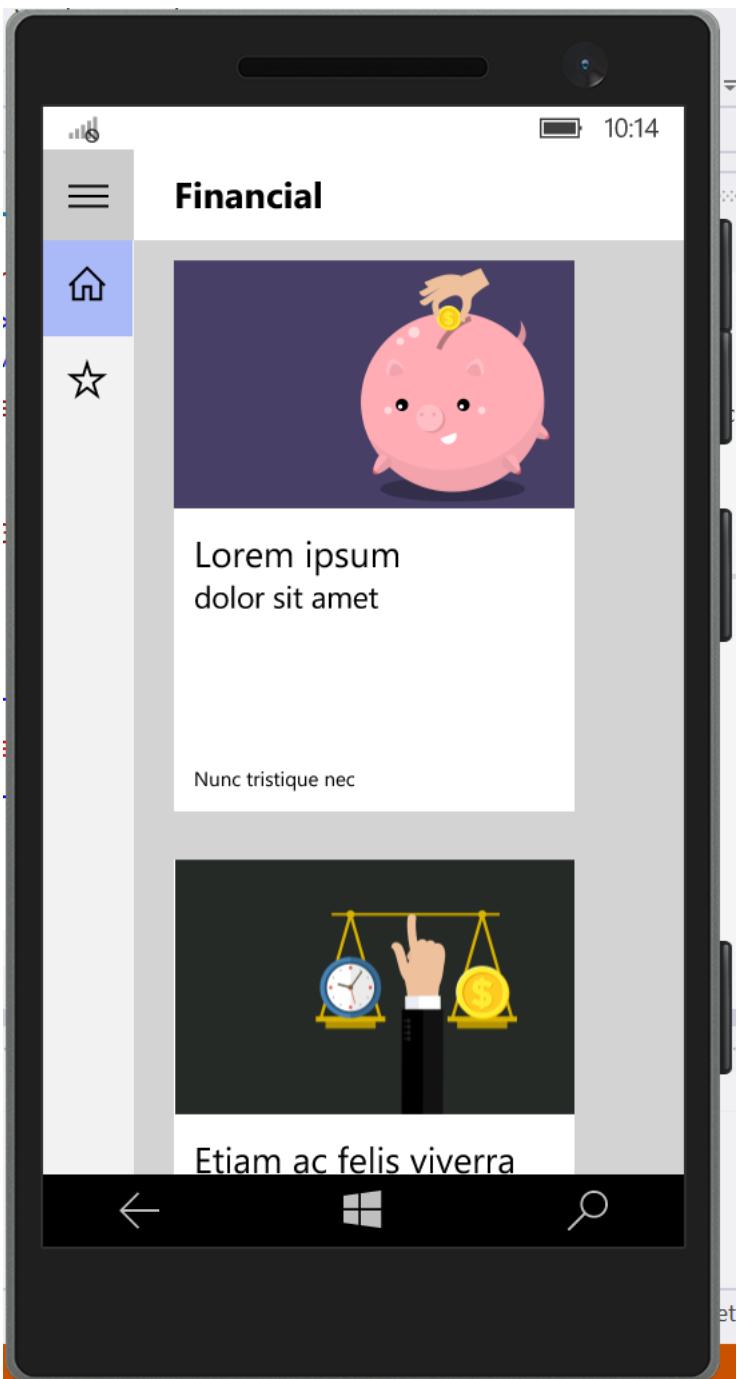
Notice that there's two columns and each of the “cards” representing fake news stories are large as well.

Here's what the completed application should look like when the window is sized a bit smaller:



Notice that there are multiple columns (three in this case) and that each “card” representing a fake news story is smaller.

Finally, here's what the application will look like when running on the smaller form factor of the Phone:



Notice that there's only one column of "cards" and that the search box usually at the top is not visible.

I'm giving you the zipped folder associated with this lesson contains the resources you'll need to complete the challenge. It also includes the instructions and screenshots that you can reference to correctly complete the challenge.

See the document UWP-044.Instructions.txt in the zipped resources folder for this lesson for the complete requirements / instructions for this challenge.

The individual fake news article images are also available in the zipped resource folder, Food1.png through Food5.png, and Financial1.png through Financial5.png.

As far as the implementation goes, each news item will contain an image, a headline, a subhead, and a dateline. So you'll create a NewsItem class, and it's going to have those properties, and you're going to then create a bindable collection of these items and then bind to them using a GridView.

If you look at the very bottom of the instructions txt file, I give you a helper method that will create instances of the NewsItem class (that you will need to build). This just saves you from so much typing and you will be using the exact "dummy data" I used in my solution video.

Use techniques that we discussed previously to create this. Having said that you're only going to need to create a single MainPage.xaml so we not going to utilize the device family-specific view. Instead we will do this all with the VisualStateManager and Adaptive Triggers.

You're going to use a GridView, you're also going to create a User Control that will display instances of the NewsItem class. The User Control techniques will allow you to re-size each of the individual data templates for the NewsItems in the DataTemplate for the GridView.

## UWP-045 - Adeptly Adaptive Challenge Solution - Part 1: Setup and MainPage Layout

This is the first of several solution lessons for the Adeptly Adaptive Challenge.

Step 1: Create a new Universal Windows Blank App named “FakeNews”.

Step 2: In the Solution Explorer, create and Assets folder, then drag and drop the Food and Financial image assets from the zipped resource folder to the new Assets folder.

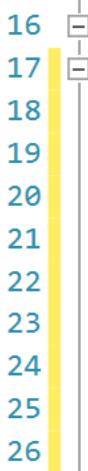
Step 3: In the App.xaml.cs remove the frame counter.

Step 4: In the MainPage.xaml create the overall “hamburger” style navigation layout by adding two row definitions. Add a RelativePanel in the first row and a SplitView in the second row. Name the SplitView “MySplitView”, set its DisplayMode to “CompactOverlay”, set its OpenPaneLength to 150 and its CompactPaneLength to 45. Add the Pane and Content elements as well.

```
10      <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11          <Grid.RowDefinitions>
12              <RowDefinition Height="Auto" />
13              <RowDefinition Height="*" />
14          </Grid.RowDefinitions>
15
16          <RelativePanel>
17
18          </RelativePanel>
19
20          <SplitView Name="MySplitView"
21              Grid.Row="1"
22              DisplayMode="CompactOverlay"
23              OpenPaneLength="150"
24              CompactPaneLength="45" >
25              <SplitView.Pane>
26
27              </SplitView.Pane>
28              <SplitView.Content>
29
30              </SplitView.Content>
31          </SplitView>
32
33      </Grid>
```

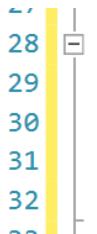
Step 6: Focus on adding the hamburger button, the title and the search textbox to the RelativePanel.

Name the hamburger button “HamburgerButton” and create a click event handler. Furthermore, use the `FontFamily` and `Content` to create the hamburger icon and center it in the button. Resize the button to 45 by 45. Position the button inside of the `RelativePanel` to be aligned left with the panel.



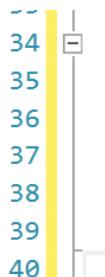
```
16 <RelativePanel>
17     <Button Name="HamburgerButton"
18         RelativePanel.AlignLeftWithPanel="True"
19         FontFamily="Segoe MDL2 Assets"
20         Content=""
21         FontSize="20"
22         Width="45"
23         Height="45"
24         HorizontalAlignment="Center"
25         Click="HamburgerButton_Click"
26     />
```

Create a `TextBlock` named “TitleTextBlock”. Position it to the right of the `HamburgerButton`. Give it a little margin on the left to separate it nicely from the button.



```
28     <TextBlock Name="TitleTextBlock"
29         RelativePanel.RightOf="HamburgerButton"
30         FontSize="18"
31         FontWeight="Bold"
32         Margin="20,0,0,0" />
```

Add an `AutoSuggestBox` named “MyAutoSuggestBox” and position it to be aligned right with the `RelativePanel`. Use the “Find” `QueryIcon` and set the `PlaceholderText` to “Search”. The width should be 200 pixels wide. Add a small 10 pixel margin to space it nicely from the right side of the Window.



```
34     <AutoSuggestBox Name="MyAutoSuggestBox"
35         QueryIcon="Find"
36         PlaceholderText="Search"
37         RelativePanel.AlignRightWithPanel="True"
38         Width="200"
39         Margin="0,0,10,0" />
```

Step 7: Focus on the `SplitView.Pane` by adding a `ListBox` and two `ListBoxItems`.

The ListBox should only allow one selection at a time. We will also want to handle the SelectionChanged event.

Begin by creating the first ListBoxItem, then copy, paste and edit the pertinent values for the second ListBoxItem.

Name the first ListBoxItem “Financial”. Add a StackPanel oriented horizontally. Inside the StackPanel, add two TextBlocks; one for the icon and one for the text. You can use any symbol from the Segoe MDL2 Assets font family for the first TextBlock. The Text of the second TextBlock should be set to “Financial”. Set a small margin to the left to give it room to breathe between the two TextBlocks.

```
49             <SplitView.Pane>
50                 <ListBox SelectionMode="Single"
51                     SelectionChanged="ListBox_SelectionChanged">
52                     <ListBoxItem Name="Financial">
53                         <StackPanel Orientation="Horizontal">
54                             <TextBlock
55                                 Text=""'
56                                 FontFamily="Segoe MDL2 Assets"
57                                 FontSize="20" />
58                             <TextBlock Text="Financial"
59                                 FontSize="18"
60                                 Margin="20,0,0,0" />
61                         </StackPanel>
62                     </ListBoxItem>
```

Copy and paste the first ListBoxItem, change the ListBoxItem’s name (“Food”), the symbol in the first TextBlock and the Text property of the second TextBlock.

```
64             <ListBoxItem Name="Food">
65                 <StackPanel Orientation="Horizontal">
66                     <TextBlock
67                         Text=""'
68                         FontFamily="Segoe MDL2 Assets"
69                         FontSize="20" />
70                     <TextBlock Text="Food"
71                         FontSize="18"
72                         Margin="20,0,0,0" />
73                 </StackPanel>
74             </ListBoxItem>
75         </ListBox>
76     </SplitView.Pane>
```

Step 8: Add a GridView to the SplitView.Content area. Name the GridView "NewsItemGrid", to make it take up the entire available area set the HorizontalAlignment to "stretch". Go ahead and add the markup for the ItemTemplate / DataTemplate.

```
78     <SplitView.Content>
79         <GridView Name="NewsItemGrid" HorizontalAlignment="Stretch"
80             Margin="10,0,0,0">
81             <GridView.ItemTemplate>
82                 <DataTemplate>
83
84                     </DataTemplate>
85                 </GridView.ItemTemplate>
86             </GridView>
87         </SplitView.Content>
88     </SplitView>
```

And I know that create two areas, a top area that will contain the SearchBox and the Title of the current Page that we're on and then in the second Row of the Grid we will add a SplitView with a pane and a content area.

Step 9: Create the event handler method stubs for the HamburgerButton\_Click and the ListBox\_SelectionChanged using the F12 keyboard technique.

Step 10: Add the code to open / close the SplitView when the user clicks the HamburgerButton.

```
23     public sealed partial class MainPage : Page
24     {
25         public MainPage()
26         {
27             this.InitializeComponent();
28         }
29
30         private void HamburgerButton_Click(object sender, RoutedEventArgs e)
31         {
32             MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen; ←
33         }
34
35         private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
36         {
37
38     }
39 }
40 }
```

## UWP-046 - Adeptly Adaptive Challenge Solution - Part 2: Creating the Data Model and Data Binding

Step 1: In the Solution Explorer add a Model folder. Inside of the Model folder add a new class file named NewsItem.cs.

Step 2: Inside of the NewsItem.cs, add a NewsItem POCO containing the properties needed to display all the information about a given fake news story. Extrapolate the property names from the instructions txt file in the zipped resource folder. Look at the bottom of that instructions file for the getNewsItems() method.

```
10     public class NewsItem
11     {
12         public int Id { get; set; }
13         public string Category { get; set; }
14         public string Headline { get; set; }
15         public string Subhead { get; set; }
16         public string DateLine { get; set; }
17         public string Image { get; set; }
18     }
```

Step 3: In the same NewsItem.cs, add another class definition called NewsManager. This will be delegated the responsibility of populating an observable collection of NewsItems, filtering the news items by the Category property.

Copy the getNewsItems() helper method from the instructions file into the NewsManager class.

```
20     public class NewsManager
21     {
22
23         private static List<NewsItem> getNewsItems()
24         {
25             var items = new List<NewsItem>();
26
27             items.Add(new NewsItem() { Id = 1, Category = "Financial", Headline = "Lorem ipsum dolor sit amet, consectetur adipiscing elit." });
28             items.Add(new NewsItem() { Id = 2, Category = "Financial", Headline = "Etiam ac tincidunt nisl. Integer vel massa non nisi lacinia blandit." });
29             items.Add(new NewsItem() { Id = 3, Category = "Financial", Headline = "Integer vel massa non nisi lacinia blandit." });
30             items.Add(new NewsItem() { Id = 4, Category = "Financial", Headline = "Proin semper, nunc id lacinia blandit, nunc." });
31             items.Add(new NewsItem() { Id = 5, Category = "Financial", Headline = "Mauris lacinia blandit, nunc id lacinia blandit, nunc." });
32
33             items.Add(new NewsItem() { Id = 6, Category = "Food", Headline = "Lorem ipsum dolor sit amet, consectetur adipiscing elit." });
34             items.Add(new NewsItem() { Id = 7, Category = "Food", Headline = "Etiam ac tincidunt nisl. Integer vel massa non nisi lacinia blandit." });
35             items.Add(new NewsItem() { Id = 8, Category = "Food", Headline = "Integer vel massa non nisi lacinia blandit." });
36             items.Add(new NewsItem() { Id = 9, Category = "Food", Headline = "Proin semper, nunc id lacinia blandit, nunc." });
37             items.Add(new NewsItem() { Id = 10, Category = "Food", Headline = "Mauris lacinia blandit, nunc id lacinia blandit, nunc." });
38
39             return items;
40         }
41     }
42 }
```

Step 4: Add a public static method called GetNews that will call the private helper method to get the list of news items, then filter that list based on the category. The method should accept the category to filter on as well as an instance of ObservableCollection<NewsItem>.

```
20     public class NewsManager
21     {
22
23         public static void GetNews(
24             string category,
25             ObservableCollection<NewsItem> newsItems)
26         {
27             var allItems = getNewsItems();
28
29             var filteredNewsItems = allItems
30                 .Where(p => p.Category == category)
31                 .ToList();
32
33             newsItems.Clear();
34
35             filteredNewsItems.ForEach(p => newsItems.Add(p));
36         }
37     }
38 }
```

Note the two LINQ statements that greatly compress the amount of code necessary to filter the list of news items by category, then add each of those NewsItem instances to the ObservableCollection<NewsItem>.

Step 5: In the MainPage.xaml, we'll prepare for data binding to the ObservableCollection<NewsItem> in the GridView. First, add the Model namespace for the project to the Page's definition to make classes from this namespace available to our XAML.

```
1 <Page
2   x:Class="FakeNews.MainPage"
3   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5   xmlns:local="using:FakeNews"
6   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8   Loaded="Page_Loaded"
9   xmlns:data="using:FakeNews.Model" ← Red arrow here
10  mc:Ignorable="d">
11
```

Next, in the GridView.ItemTemplate's DataTemplate, add the DataType that we'll bind to:

```
77 <GridView Name="NewsItemGrid"
78   ItemsSource="{x:Bind NewsItems}"
79   HorizontalAlignment="Stretch"
80   Margin="10,0,0,0">
81   <GridView.ItemTemplate>
82     <DataTemplate x:DataType="data:NewsItem"> ← Red arrow here
83   
```

In this lesson, we'll create a temporary solution for displaying each NewsItem just so we can confirm that the data binding is working. Later, we'll remove this implementation and replace it with the UserControl that will allow us to adaptively resize each NewsItem "card" based on the size of the Window.

Add a Grid with two rows. Set its height to 275 and width to 200. Add a margin of 10 pixels around it and set its background to white. The grid should have two row definitions. In the top row add an Image control that binds to the Image property of the NewsItem class. In the second row add a RelativePanel with three TextBlocks ... the TextBlocks should bind to the NewsItem's Headline, Subhead and DateLine properties, respectively.

```

84     <Grid Background="White" Margin="10" Height="275" Width="200">
85         <Grid.RowDefinitions>
86             <RowDefinition Height="Auto" />
87             <RowDefinition Height="*" />
88         </Grid.RowDefinitions>
89         <Image Name="MyImage" Source="{x:Bind Image}" />
90         <RelativePanel Grid.Row="1">
91             <TextBlock Text="{x:Bind Headline}" />
92             <TextBlock Text="{x:Bind Subhead}" />
93             <TextBlock Text="{x:Bind DateLine}" />
94         </RelativePanel>
95
96     </Grid>
97 </DataTemplate>
98 </GridView.ItemTemplate>

```

Step 6: Wire the NewsManager to the MainPage.xaml inside of the MainPage.xaml.cs. First, create a private field to store an instance of the ObservableCollection<NewsItem> we will retrieve from the NewsManager. The data binding we set up in the previous step will look for this private field to bind to:

```

19
20     namespace FakeNews
21     {
22         /// <summary>
23         /// An empty page that can be used on its own or navigated to within a Frame.
24         /// </summary>
25         public sealed partial class MainPage : Page
26         {
27             private ObservableCollection<NewsItem> NewsItems; ←
28
29             public MainPage()
30             {

```

When the MainPage class is initialized, create a new instance of ObservableCollection<NewsItem>:

```

29
30         public MainPage()
31         {
32             this.InitializeComponent();
33             NewsItems = new ObservableCollection<NewsItem>(); ←
34         }

```

Based on the selection of the ListBox, whether the Food or Financial ListBoxItems, retrieve the filtered ObservableCollection<NewsItem> from the NewsManager for the given “category” that the user selected.

```

40     private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
41     {
42         if (Financial.IsSelected)
43         {
44             NewsManager.GetNews("Financial", NewsItems);
45             TitleTextBlock.Text = "Financial";
46         }
47         else if (Food.IsSelected)
48         {
49             NewsManager.GetNews("Food", NewsItems);
50             TitleTextBlock.Text = "Food";
51         }
52     }
--
```

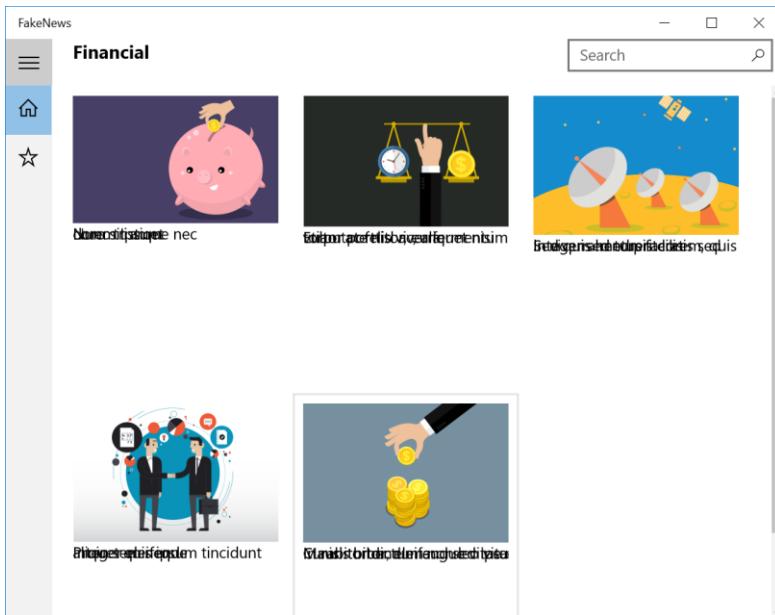
Notice that I also changed the TitleTextBlock's Text property as well.

Finally, as the Page loads, select the Financial ListBoxItem to begin.

```

-- 54     private void Page_Loaded(object sender, RoutedEventArgs e)
55     {
56         Financial.IsSelected = true;
57     }
```

At this point, the app should run and bind to the news items. Even so, there are significant formatting issues that we'll address in the next lessons.



## UWP-047 - Adeptly Adaptive Challenge Solution - Part 3: Creating a User Control as the Data Template

In this lesson we'll create a UserControl as the DataTemplate so that each news item "card" can be resized adaptively.

Step 1: Add a new User Control to the project named "NewsItemControl.xaml". (Remember: Project menu > Add New Item ... > select User Control in the middle).

Step 2: In the MainPage.xaml, cut out everything inside the opening and closing DataTemplate element and paste it into the NewsItemControl.

Step 3: In the NewsItemControl.xaml.cs code behind, add the necessary (templated) code to return the DataContext as an instance of Model.NewsItem, then add handle the DataContextChanged event by updating the bindings in the User Control.

```
18     namespace FakeNews
19     {
20         public sealed partial class NewsItemControl : UserControl
21         {
22             public Model.NewsItem NewsItem { get { return this.DataContext as Model.NewsItem; } }
23
24             public NewsItemControl()
25             {
26                 this.InitializeComponent();
27                 this.DataContextChanged += (s, e) => Bindings.Update();
28             }
29         }
30     }
```

Remember: this code may be a bit complicated, but you can copy from the cheat sheet replacing the model's data type as necessary.

Step 4: Back in the MainPage.xaml, reference the NewsItemControl as the data template for the GridView.

```
82     <GridView.ItemTemplate>
83         <DataTemplate x:DataType="data:NewsItem">
84             <local:NewsItemControl />
85         </DataTemplate>
86     </GridView.ItemTemplate>
```

Step 5: In the NewsItemControl.xaml, update the three TextBlocks in the RelativePanel so that they display properly. Align the first to the top, the second below the first, and align the third one to the bottom. Set the font sizes appropriately and use TextWrapping to ensure all the text is displayed.

```
17     <Image Name="MyImage" Source="{x:Bind NewsItem.Image}" />
18     <RelativePanel Grid.Row="1" Margin="10">
19         <TextBlock Text="{x:Bind NewsItem.Headline}"
20             Name="HeadlineTextBox"
21             RelativePanel.AlignTopWithPanel="True"
22             FontSize="18"
23             TextWrapping="Wrap" />
24         <TextBlock Text="{x:Bind NewsItem.Subhead}"
25             RelativePanel.Below="HeadlineTextBox"
26             TextWrapping="Wrap" />
27         <TextBlock Text="{x:Bind NewsItem.DateLine}"
28             RelativePanel.AlignBottomWithPanel="True"
29             FontSize="10" />
30     </RelativePanel>
```

In this lesson we added no new functionality but merely prepared the project for adaptive resizing.

## UWP-048 - Adeptly Adaptive Challenge Solution - Part 4: Adaptively Resizing

In this final solution lesson we will add adaptive resizing to both the MainPage.xaml and the NewsItemControl.xaml.

Step 1: Add two VisualStates in the MainPage.xaml. This should be added immediately after the row definitions. The first should be named “NarrowLayout” and should have a minimum window width of 0. The second should be named “WideLayout” and should have a minimum window width of 400. When the NarrowLayout is triggered, set the MyAutoSuggestBox’s Visibility to Collapsed. When the WideLayout is triggered, set the MyAutoSuggestBox’s Visibility to Visible.

```
15             <RowDefinition Height="*" />
16         </Grid.RowDefinitions>
17
18     <VisualStateManager.VisualStateGroups>
19         <VisualStateGroup>
20             <VisualState x:Name="NarrowLayout">
21                 <VisualState.StateTriggers>
22                     <AdaptiveTrigger MinWindowWidth="0" />
23                 </VisualState.StateTriggers>
24                 <VisualState.Setters>
25                     <Setter Target="MyAutoSuggestBox.Visibility"
26                           Value="Collapsed" />
27                 </VisualState.Setters>
28             </VisualState>
29             <VisualState x:Name="WideLayout">
30                 <VisualState.StateTriggers>
31                     <AdaptiveTrigger MinWindowWidth="400" />
32                 </VisualState.StateTriggers>
33                 <VisualState.Setters>
34                     <Setter Target="MyAutoSuggestBox.Visibility"
35                           Value="Visible" />
36                 </VisualState.Setters>
37             </VisualState>
38         </VisualStateGroup>
39     </VisualStateManager.VisualStateGroups>
```

Step 2: In NewsItemControl.xaml, give the Grid the name “MainPanel”. It will need a name since we’ll be resizing it adaptively.

Step 3: Add two VisualStates in the NewsItemControl.xaml beneath the Grid's row definitions. The first VisualState should be named "NarrowLayout" and should have a minimum window width of 0. The second should be named "WideLayout" and should have a minimum window width of 900.

When the NarrowLayout is triggered, the MainPanel's width should be 200 and height should be 275. Furthermore, the HeadlineTextBlock's FontSize should be set to 18.

When the WideLayout is triggered, the MainPanel's width should be 400 and the height should be 400. Furthermore, the HeadlineTextBlock's FontSize should be set to 26.

```
19      <VisualStateGroup>
20      <VisualState x:Name="NarrowLayout">
21          <VisualState.StateTriggers>
22              <AdaptiveTrigger MinWindowWidth="0" />
23          </VisualState.StateTriggers>
24          <VisualState.Setters>
25              <Setter Target="MainPanel.Width" Value="200" />
26              <Setter Target="MainPanel.Height" Value="275" />
27              <Setter Target="HeadlineTextBlock.FontSize" Value="18" />
28          </VisualState.Setters>
29      </VisualState>
30      <VisualState x:Name="WideLayout">
31          <VisualState.StateTriggers>
32              <AdaptiveTrigger MinWindowWidth="900" />
33          </VisualState.StateTriggers>
34          <VisualState.Setters>
35              <Setter Target="MainPanel.Width" Value="400" />
36              <Setter Target="MainPanel.Height" Value="400" />
37              <Setter Target="HeadlineTextBlock.FontSize" Value="26" />
38          </VisualState.Setters>
39      </VisualState>
40  </VisualStateGroup>
41 </VisualStateManager.VisualStateGroups>
42
```

Step 4: One final tweak to the MainPage.xaml would be to add a 5 pixel margin to the top of the AutoSuggestBox to push it down from the top a tiny bit.

```
-->
59      <AutoSuggestBox Name="MyAutoSuggestBox"
60          QueryIcon="Find"
61          PlaceholderText="Search"
62          RelativePanel.AlignRightWithPanel="True"
63          Width="200"
64          Margin="0,5,10,0" />
65      </RelativePanel>
```

## UWP-049 - UWP SoundBoard – Introduction

As we reach the midway point of this series of lessons we will be changing-up the format a bit from the one we have been following so far of confronting loosely related topics and then undertaking challenges to reinforce those ideas. What we will do instead is focus on building four full applications from scratch to illustrate the process of how to take an application from the conceptual stage into actual construction.

The first application we will look at is a simple “Sound Board” that basically lists a collection of predefined sounds that can be played when you click on an icon that corresponds to that sound. It will have various features such as sound categories, playing a sound via drag/drop, searching for sounds using auto-suggest, etc. The application will implement things like Hamburger Navigation, State Management, Media Elements and so forth.

At the end we will build the application and walk through the process of submitting it to the Windows Store. As usual, it’s a good idea to follow along by writing the code on your end.

## UWP-050 - UWP SoundBoard - Setup and MainPage Layout

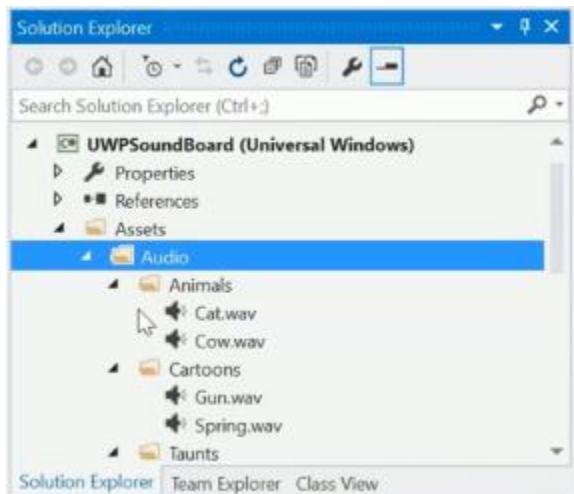
Without any further ado, let's begin creating our SoundBoard App.

Step 1: Create a new project named “UWPSoundBoard”

Step 2: Download the zip file associated with this lesson and add the assets to the project using the following process. Start with audio assets by creating a subfolder for it in your Assets folder, then select the folders containing your audio assets as follows:

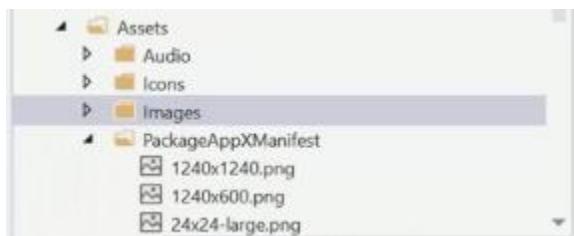
Name	Date modified	Type	Size
Animals	9/2/2015 12:54 PM	File folder	
Cartoons	9/2/2015 12:54 PM	File folder	
Taunts	9/2/2015 12:54 PM	File folder	
Warnings	9/2/2015 12:54 PM	File folder	

Then click and drag all of that into your Audio folder



If you come across a nag screen simply select “Apply to all items.”

Step 3: Repeat this process for the “Icons,” “Images,” and “PackageAppXManifest” folders



Step 4: Open up MainPage.xaml and set up the Panels

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <RelativePanel>
        <Button Name="HamburgerButton"
            RelativePanel.AlignLeftWithPanel="True"
            Click="HamburgerButton_Click"
            FontFamily="Segoe MDL2 Assets"
            Content="" />

        <Button Name="BackButton"
            RelativePanel.RightOf="HamburgerButton"
            Click="BackButton_Click"
            FontFamily="Segoe MDL2 Assets"
            Content="" />
        <AutoSuggestBox Name="SearchAutoSuggestBox"
            PlaceholderText="Search for sounds"
            Width="200"
            QueryIcon="Find"
            TextChanged="SearchAutoSuggestBox_TextChanged"
            QuerySubmitted="SearchAutoSuggestBox_QuerySubmitted"
            RelativePanel.AlignRightWithPanel="True" />
    </RelativePanel>
```

Step 5: Adjust the Height, Width and FontSize for the HamburgerButton and BackButton by adding the following

```
        Click="HamburgerButton_Click"
        FontFamily="Segoe MDL2 Assets"
        FontSize="20"
        Height="45"
        Width="45"
        Content=""/>
```

Step 6: Then for the SplitView add the following. Note that we are adding a ListView instead of a ListBox under the SplitView.Pane

```

<SplitView Name="MySplitView"
    DisplayMode="CompactOverlay"
    CompactPaneLength="45"
    OpenPaneLength="200">
    <SplitView.Pane>
        <ListView Name="MenuItemsListView" IsItemClickEnabled="True" />
    </SplitView.Pane>
    <SplitView.Content>
        <Grid>

            </Grid>
        </SplitView.Content>
    </SplitView>
</Grid>

```

And add this at the end of `<ListView`

```
ItemClick="MenuItemsListView_ItemClick" />
```

Step 7: Add this code in between the `SplitView.Content` Grid

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <TextBlock Name="CategoryTextBlock" Text="All Sounds" FontSize="24" />

    <GridView Grid.Row="1"
        Name="SoundGridView"
        SelectionMode="None"
        IsItemClickEnabled="True"
        ItemClick="SoundGridView_ItemClick">
        <GridView.ItemTemplate>
            <DataTemplate>
                <Image Name="MyImage" Height="112" Width="101" />
            </DataTemplate>
        </GridView.ItemTemplate>
    </GridView>

```

And the `TextBlock` will look like a Header so make its `FontSize` fairly large

```
<TextBlock Name="CategoryTextBlock" Text="All Sounds" FontSize="24" />
```

Step 8: Next we will add a `Media Element`, which allows you to play sounds

```
</Grid.RowDefinitions>

<MediaElement Name="MyMediaElement" AutoPlay="True" />

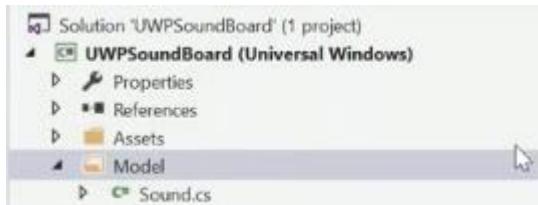
<TextBlock Name="CategoryTextBlock" Text="All Sounds" FontSize="24" />
```

Having completed all of that, you now have something akin to the shell of an App which we will continue to build throughout the next series of lessons.

## UWP-051 - UWP SoundBoard - Creating the Data Model & Data Binding

The next thing we will do is create the Data Model with a class called “Sound,” and Sound, in turn, will have an audio file, an image file, and a category. After that we will create a Sound Manager, which will allow us to either get access to all sounds, or just a sound category.

Step 1: Create a folder called “Model” and add to it a class called “Sound”



Step 2: In the Sound class type in the following properties (we will create the SoundCategory enum in the next step)

```
namespace UWPSoundBoard.Model
{
    public class Sound
    {
        public string Name { get; set; }
        public SoundCategory Category { get; set; }
        public string AudioFile { get; set; }
        public string ImageFile { get; set; }
    }
}
```

Step 3: Create an enum called SoundCategory which correlates to the different types of sounds we have in the Audio folder

```
public enum SoundCategory
{
    Animals,
    Cartoons,
    Taunts,
    Warnings
}
```

Step 4: Next, create a Constructor for Sound that contains references to the asset folders for Audio and Images

```
public Sound(string name, SoundCategory category)
{
    Name = name;
    Category = category;
    AudioFile = String.Format("/Assets/Audio/{0}/{1}.wav", category, name);
    ImageFile = String.Format("/Assets/Images/{0}/{1}.png", category, name);
}
```

Step 5: Now, add a new class to the Model folder and call it “SoundManager.” Within the SoundManager class create a static method called getSounds() that returns a List of type Sound.

```
private static List<Sound> getSounds()
{
    var sounds = new List<Sound>();

    sounds.Add(new Sound("Cow", SoundCategory.Animals));
    sounds.Add(new Sound("Cat", SoundCategory.Animals));

    sounds.Add(new Sound("Gun", SoundCategory.Cartoons));
    sounds.Add(new Sound("Spring", SoundCategory.Cartoons));

    sounds.Add(new Sound("Clock", SoundCategory.Taunts));
    sounds.Add(new Sound("LOL", SoundCategory.Taunts));

    sounds.Add(new Sound("Ship", SoundCategory.Warnings));
    sounds.Add(new Sound("Siren", SoundCategory.Warnings));

    return sounds;
}
```

Step 6: Also add to the SoundManager class a static method as follows

```
public class SoundManager
{
    public static void GetAllSounds(ObservableCollection<Sound> sounds)
    {
        var allSounds = getSounds();
        sounds.Clear();
        allSounds.ForEach(p => sounds.Add(p));
    }
}
```

The objective with this method is to be able to pass in an ObservableCollection of type Sound as that's what will be passing in from MainPage.xaml.cs to this SoundManager. Note that sounds.Clear() is referenced because you will want to get rid of anything that's already in the Collection. Also note how the allSounds.ForEach() method allows us to pass in a lambda expression, which is used to loop through every sound that was pulled from getSounds() and stored into allSounds. This loop is responsible for iterating over each sound, and then add it to the variable called “sounds” of type ObservableCollection<Sound>.

Step 7: Next, create a static method within the class called GetSoundsByCategory(). This will have a very similar task as the GetAllSounds() method, passing in an ObservableCollection<Sound> called sounds, and also a SoundCategory called soundCategory.

```
public static void GetSoundsByCategory(ObservableCollection<Sound> sounds, SoundCategory soundCategory)
{
```

And inside of the GetSoundsByCategory() method write the following code

```
    var allSounds = getSounds();
    var filteredSounds = allSounds.Where(p => p.Category == soundCategory).ToList();
    sounds.Clear();
    filteredSounds.ForEach(p => sounds.Add(p));
```

Step 8: Now, in MainPage.xaml.cs we will want to create a private reference to ObservableCollection<Sound> and then in the Constructor initialize the Component and then call SoundManager.GetAllSounds() to populate the List of sounds.

```
public sealed partial class MainPage : Page
{
    private ObservableCollection<Sound> Sounds;

    public MainPage()
    {
        this.InitializeComponent();
        Sounds = new ObservableCollection<Sound>();
        SoundManager.GetAllSounds(Sounds);
    }
}
```

Step 9: And then in MainPage.xaml create an XML namespace for data

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

And in the same file also add these edits

```

        Name="SoundGridView"
        SelectionMode="None"
        IsItemClickEnabled="True"
        ItemsSource="{x:Bind Sounds}"
        ItemClick="SoundGridView_ItemClick">
    <GridView.ItemTemplate>
        <DataTemplate x:DataType="data:Sound">
            <Image Name="MyImage" Height="112" Width="101" Source="{x:Bind ImageFile}" />
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>

```

Finally, adjust the SplitView by adding Grid.Row="1"

```

<SplitView Grid.Row="1" Name="MySplitView" |
```

At the end of all of that, you should run the program and see something that looks like this



Step 10: Now, to get the HamburgerButton\_Click() working in MainPage.xaml.cs add the following code

```

private void HamburgerButton_Click(object sender, RoutedEventArgs e)
{
    MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen;
}

```

Step 11: Let's add a class called MenuItem, and then inside the class write some code to help filter the sounds based on the selected menu item in the SplitView.Pane



```

namespace UWPSoundBoard.Model
{
    public class MenuItem
    {
        public string IconFile { get; set; }
        public SoundCategory Category { get; set; }
    }
}

```

And then add a private List called MenuItems, and populate it in the Constructor by adding the following code.

```

public sealed partial class MainPage : Page
{
    private ObservableCollection<Sound> Sounds;

    private List<MenuItem> MenuItems;

    public MainPage()
    {
        this.InitializeComponent();
        Sounds = new ObservableCollection<Sound>();
        SoundManager.GetAllSounds(Sounds);
        MenuItems = new List<MenuItem>();
        MenuItems.Add(new MenuItem { IconFile = "Assets/Icons/animals.png", Category = SoundCategory.Animals });
        MenuItems.Add(new MenuItem { IconFile = "Assets/Icons/cartoon.png", Category = SoundCategory.Cartoons });
        MenuItems.Add(new MenuItem { IconFile = "Assets/Icons/taunt.png", Category = SoundCategory.Taunts });
        MenuItems.Add(new MenuItem { IconFile = "Assets/Icons/warning.png", Category = SoundCategory.Warnings });
    }
}

```

Then, make the following changes to the SplitView.Pane

```

<SplitView.Pane>
    <ListView Name="MenuItemsListView"
              IsItemClickEnabled="True"
              ItemsSource="{x:Bind MenuItems}"
              ItemClick="MenuItemsListView_ItemClick">
        <ListView.ItemTemplate>
            <DataTemplate x:DataType="data:MenuItem">
                <StackPanel Orientation="Horizontal">
                    <Image Source="{x:Bind IconFile}" Height="45" Width="45" />
                    <TextBlock Text="{x:Bind Category}" FontSize="18" Margin="10,0,0,0" />
                </StackPanel>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</SplitView.Pane>

```

And, now when you run the program you should now see the icons on the left-hand side.



There will be some minor issues needing to be fixed, but this should be a good starting point for now. In the next lesson we will hook things up to the Media Element in order to start playing sounds.

## UWP-052 - UWP SoundBoard - Playing Sounds with the Media Element

Now comes the fun part of wiring up the Media Element in order to play sounds, and setting up filtering by category.

Step 1: First let's grab the Item when it becomes clicked, which will be referenced by ItemClickEventArgs e in the SoundGridView\_ItemClick() method within MainPage.xaml.cs

Write the following code within that method

```
private void SoundGridView_ItemClick(object sender, ItemClickEventArgs e)
{
    var sound = (Sound)e.ClickedItem;
    MyMediaElement.Source = new Uri(this.BaseUri, sound.AudioFile);
}
```

The first step grabs the clicked item referenced in e.ClickedItem, which then immediately becomes cast to type Sound, and then it stores that in a temporary variable called "sound."

Notice when you hover over the Source property that it accepts a URI (Uniform Resource Indicator), which is just going to be a location for the file. The easiest way to handle this is just to specify the base URI - that will give us the root of the project - and then we can give it the actual audio file that will contain the specific location of that /Assets/Audio/Category/Sound in question.

When you run the program you should notice that clicking on the items now produces the corresponding sound.

Step 2: Next, let's fix some issues with the visual elements by making the following changes to the StackPanel

```
<StackPanel Orientation="Horizontal">
    <Image Source="{x:Bind IconFile}"
        Height="40"
        Width="40"
        Margin="-10,15,0,15"
    />
    <TextBlock
        Text="{x:Bind Category}"
        FontSize="18"
        Margin="10,0,0,0"
        VerticalAlignment="Center" />
</StackPanel>
```

And also adjust the Grid Margin as follows

```
<Grid Margin="20,0,0,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
```

Step 3: It's looking a lot better now but we can make a few minor adjustments – tinkering with the margins, and sizing, until it all starts looking right

```
<StackPanel Orientation="Horizontal">
    <Image Source="{x:Bind IconFile}"
        Height="35"
        Width="35"
        Margin="-10,10,0,10"
    />

    <SplitView.Content>
        <Grid Margin="20,20,0,0">
```

Step 4: Next, let's get the category filtering to work. Go back to MainPage.xaml.cs and this time reference ItemClickEventArgs e within the MenuItemsListView\_ItemClick() method, similar to how we did it before

```
private void MenuItemsListView_ItemClick(object sender, ItemClickEventArgs e)
{
    var menuItem = (MenuItem)e.ClickedItem;

    // Filter on category
    CategoryTextBlock.Text = menuItem.Category.ToString();
    SoundManager.GetSoundsByCategory(Sounds, menuItem.Category);
}
```

Notice how we call SoundManager.GetSoundsByCategory() this time, passing in our ObservableCollection<Sound> Sounds, and then the category as arguments. When you run the program you should now be able to click on the icons to filter by category.

Step 5: Now, let's get the back button to work – which should really only be displayed when we click on one of the items. We do that by setting the visibility for it once an item is clicked, again inside of the SoundGridView\_ItemClick() method

```
    SoundManager.GetSoundsByCategory(Sounds, menuItem.Category);
    BackButton.Visibility = Visibility.Visible;
}
```

And, since we do not want the back button visible otherwise, set it to Visibility.Collapsed in the MainPage() Constructor

```
    BackButton.Visibility = Visibility.Collapsed;
}
```

And, of course, when we hit the back button we will want to navigate back to All Sounds so add this code as well

```
private void BackButton_Click(object sender, RoutedEventArgs e)
{
    SoundManager.GetAllSounds(Sounds);
    CategoryTextBlock.Text = "All Sounds";
    MenuItemsListView.SelectedItem = null;
    BackButton.Visibility = Visibility.Collapsed;
}
```

When you run the program you should now notice the back button appears and works as we intended. In the next lesson we will look into adding drag/drop functionality for adding sounds by, for example, dragging and dropping from your desktop.

## UWP-053 - UWP SoundBoard - Adding Drag and Drop

In this lesson we'll be adding drag/drop functionality, allowing us to audition sounds by placing them onto the SoundBoard app. Let's start off our drag/drop feature by going back to MainPage.xaml and modifying the GridView Control

Step 1: Add AllowDrop, Drop, and DragOver to the GridView to begin referencing the drag/drop events within MainPage.xaml



Step 2: In MainPage.xaml.cs make sure your using statements are declared to import all of the required namespaces

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using UWPSoundBoard.Model;
using Windows.ApplicationModel.DataTransfer;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.Storage;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Imaging;
using Windows.UI.Xaml.Navigation;
```

And now in the SoundGridView\_ItemClick() method insert the following code

```
private async void SoundGridView_Drop(object sender, DragEventArgs e)
{
    if (e.DataView.Contains(StandardDataFormats.StorageItems))
    {
        var items = await e.DataView.GetStorageItemsAsync();

        if (items.Any())
        {
            var storageFile = items[0] as StorageFile;
            var contentType = storageFile.ContentType;

            StorageFolder folder = ApplicationData.Current.LocalFolder;

            if (contentType == "audio/wav" || contentType
            {
                StorageFile newFile = await storageFile.CopyAsync(folder,
                storageFile.Name, NameCollisionOption.GenerateUniqueName);

                MyMediaElement.SetSource(await storageFile.OpenAsync(FileAccessMode.Read),
                contentType);
                MyMediaElement.Play();
            }
        }
    }
}
```

There are a few things worth mentioning about this code. First, whenever you see the await, or async, keyword, it's essentially saying that the method that it's attached to could be long-running in nature. And, instead of allowing that long-running method to block the ongoing operation of all of the other code beneath the current line (where that method is called), instead allow the rest of the code to continue running until the process is finished. And when everything is done it'll all magically work together.

There are two different ways to introduce asynchrony into your applications based on the two different types of blocking (as in “roadblock”). There's blocking due to an operation being mathematically intensive, and then there's blocking because of indeterminate lag time (such as calls made out over a network).

So, without using async/await, a request over the internet could cause an app to wait – halting all other execution - until a response came back from a remote web server. In this case, there's are not a lot of computational time being spent, it's just lag time that will appear to the user as though the application is unresponsive and stalling.

Those are the basics regarding async/await. Now, the rest of the code is basically responsible for pulling off the ContentType of the sound file, getting a reference to that local folder, and then checking the contentType. And if its type is audio/wav tell the media player to go ahead and play that file. But, before

it plays, what happens is the application actually copies that file into your local storage area, and then reads from that location.

Step 3: Write the following code for the SoundGridView\_DragOver() method/event

```
private void SoundGridView_DragOver(object sender, DragEventArgs e)
{
    e.AcceptedOperation = DataPackageOperation.Copy;

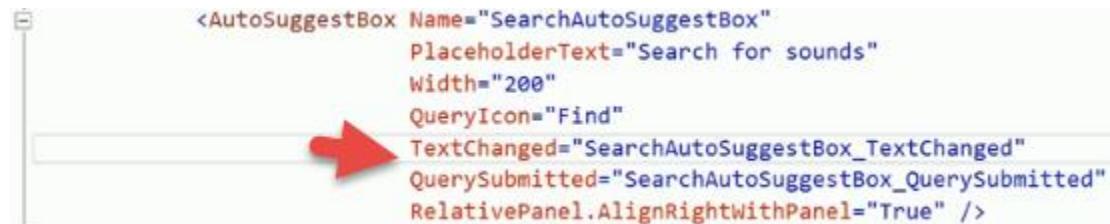
    e.DragUIOverride.Caption = "drop to create a custom sound and tile";
    e.DragUIOverride.IsCaptionVisible = true;
    e.DragUIOverride.IsContentVisible = true;
    e.DragUIOverride.IsGlyphVisible = true;
}
```

Now, when you run the program you should be able to drag and drop a sound from anywhere on your computer, and onto an icon and have the sound play back.

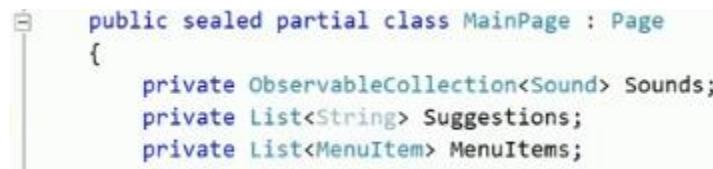
## UWP-054 - UWP SoundBoard - Finishing Touches

The next thing we will want to do is create an auto-suggestion search filter to refine down the items in the results to just those that match the search criteria.

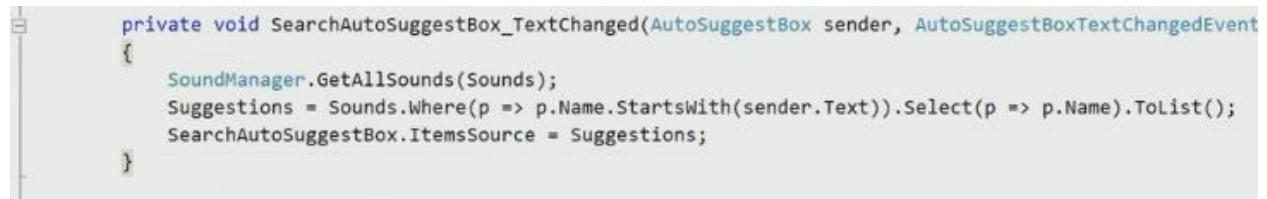
Step 1: To implement this we have two Events which we will reference in MainPage.xaml via TextChanged and QuerySubmitted.



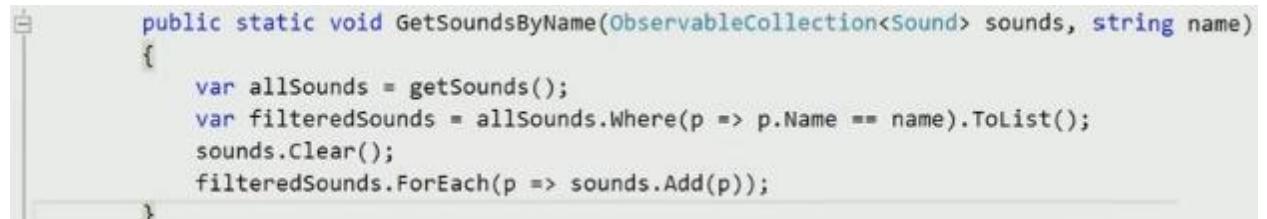
Step 2: Create a List<String> at the class level in MainPage.xaml.cs



Step 3: Populate this list by writing the following in the SearchAutoSuggestBox\_TextChanged() method



Step 4: Create a custom method that will filter a set of sounds



Step 5: Then, inside of the SearchAutoSuggestBox\_QuerySubmitted() we will reference that last method as well as write the following

```
private void SearchAutoSuggestBox_QuerySubmitted(AutoSuggestBox sender)
{
    SoundManager.GetSoundsByName(Sounds, sender.Text);
    CategoryTextBlock.Text = sender.Text;
    MenuItemsListView.SelectedItem = null;
    BackButton.Visibility = Visibility.Visible;
}
```

## UWP-055 - UWP SoundBoard - Add Assets with Package.AppXManifest

In the last lesson we created an auto-suggest search filter, however, there was a small issue with the search box retaining what we last typed in. There is a simple fix to this that you can achieve with the following code:

Step 1: Create a custom method within MainPage.xaml.cs

```
private void goBack()
{
    SoundManager.GetAllSounds(Sounds);
    CategoryTextBlock.Text = "All Sounds";
    MenuItemsListView.SelectedItem = null;
    BackButton.Visibility = Visibility.Collapsed;
}
```

Step 2: Call that method here

```
private void SearchAutoSuggestBox_TextChanged(AutoSuggestBox sender,
{
    if (string.IsNullOrEmpty(sender.Text)) goBack();
```

And here

```
private void BackButton_Click(object sender, RoutedEventArgs e)
{
    goBack();
}
```

So, the GoBack() method contains all of the functionality necessary to put the application in a valid state after we move away from the search functionality, or whenever we hit the back button. It achieves this by grabbing all of the sounds, set the CategoryTextBlock.Text to "All Sounds," and then set the MenuItemsListView.SelectedItem to no selection, and then set the BackButton.Visibility to Collapsed.

So, the next thing is to do is touch upon deployment – covering a bit on how your applications will actually be deployed onto the end-user computer. As a developer, you don't write any installers in your routines to install/uninstall your Windows runtime app (the app that you're building). Instead, you package your app and you submit it to the Windows Store. Then, users will acquire your app from the Store in the form of an "App Package." The operating system uses information that you supply in an app package to install the app and ensure that all traces of the app are gone from the device whenever they choose to uninstall that app. To understand how this works, refer to the Package.appxmanifest file in the Solution Explorer.

Package.appxmanifest

When you double-click that a beautiful designer pops up. But actually what this file is, is just XML - it just has a nice Visual Studio interface on top of it so you don't have to deal directly with the XML. But

essentially that Package.AppXManifest file is just a document that contains the information that the system needs to deploy, to display, or to update a Windows app.

The screenshot shows the 'Application' tab of the Package Manifest Editor. It includes fields for 'Display name' (UWPSoundBoard), 'Entry point' (UWPSoundBoard.App), 'Default language' (en-US), and 'Description' (UWPSoundBoard). Below these, there's a section for 'Supported rotations' with four orientation icons. Underneath, there are dropdowns for 'Lock screen notifications' (set to 'not set') and 'Resource group'.

You will also notice there are tabbed sections that allow you to further configure your package for deployment. Among all of this information are things like the identity of the Package, any dependencies that it has on other DLL's or outside resources, and any capabilities that it requires in order to run. Therefore, every app package must include at least one Package.AppXManifest file.

Note that the package manifest is digitally signed as a part of signing the app package. After it's been signed, you can't modify the manifest without invalidating the package signature. So it contains some security features for the end user to ensure that they're getting the application that you built and that somebody didn't hack in and make some changes to it before it gets installed on a computer.

Focusing first on the Application tab, let's change how the app is displayed in the Store, such as its description that will be displayed

The screenshot shows the 'Application' tab with the 'Description' field updated to 'Plays fun sounds to amuse and amaze friends.'

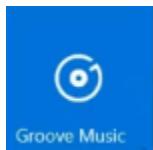
Also select all of the supported rotations

The screenshot shows the 'Application' tab with the 'Supported rotations' section where all four options (Landscape, Portrait, Landscape-flipped, Portrait-flipped) have their checkboxes checked.

Now, under the Visual Assets tab first set the tile attributes, such as the short name for your app, and the applicable logos (which we will setup in a moment).



If you are not familiar with a Windows tile, it is basically a clickable icon from which you can launch the app, and looks something like this



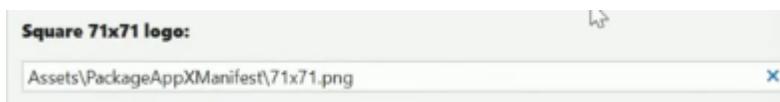
For a guideline on Windows tiles and badges visit the URL below:

<http://bit.do/tiles-badges>

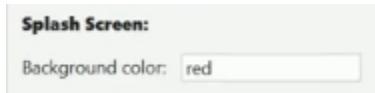
You can set these tiles using the Scaled Assets section, and usually what you can do is set a tile sized at one of the recommended settings and the smaller sizes will be automatically scaled down for you.



For this app you can set the logos by referring to the Assets\PackageAppManifest\ folder files as follows



And, if you wish you can see the background color for the Splash Screen



Now, if you right click on Package.appxmanifest



And select “Open With,” choosing the XML (Text) Editor

XML (Text) Editor

You will see all of the XML settings that correspond to settings we just made in the visual editor, allowing you to make changes here as well. And, now, when you run the program you should see the Splash Screen briefly run on startup. That cover the basics of getting your app ready for publishing. Note that you will probably want to test your app on multiple devices before submitting it to the Windows Store.

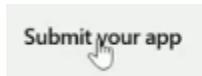
Now, before we move onto the next lesson, let’s make one more minor change to the AutoSuggestBox Margin, to make it a little more visually pleasing

```
<AutoSuggestBox Name="SearchAutoSuggestBox"
    PlaceholderText="Search for sounds"
    Width="200"
    QueryIcon="Find"
    Margin="0,5,10,0"
```

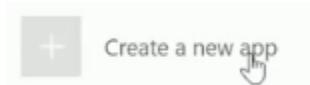
## UWP-056 - UWP SoundBoard - Submitting to the Windows Store

Now we will complete the process of submitting the app to the Windows Store. Begin by navigating to dev.windows.com and it will take you to the correct page relative to the country you are in.

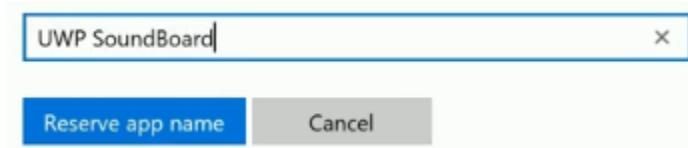
While there, find and click on the link that says "Submit your app."



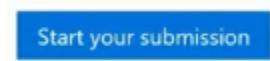
The link, as well as the other steps taken in this lesson, may not look exactly like this because web pages often change. However, you should be able to find it if you look for a link that lets you submit your app. From here, you may have to go through some additional registration steps. After that, select "Create a new app."



Once you sign into your Microsoft account you can reserve the app name (hoping that nobody else has taken it ahead of you).



Then, select "Start your submission"



Which should lead you to a series of steps that track your progress

# Submission 1

Delete

Pricing and availability



Not started

App properties

Not started

Packages

Not started

Descriptions

Not started

You'll be able to edit your descriptions after you upload packages.

Notes for certification

Optional

Begin by clicking on “Pricing and availability.” Here you can set your app’s price, as well as allow a free trial, as well as more detailed sales options.

Base price\*

Free trial

Here you can also change the visibility for the distribution of our app. This might be useful if you’re only distributing it to a small, known clientel – but not just anybody, only our customers or people that we already know. We could also change the device families on which our app can run. Go ahead and let it run on desktop and mobile. I’m also going to let Microsoft decide whether to make it available for future device families.

Desktop

Mobile



Let Microsoft decide whether to make this app available to any future device families

You can set Organizational Licensing if you plan to sell to large corporations, etc, so we can safely ignore that right now. Next, set the publication date for as soon as the app passes certification, and click save to complete that part of the process.

Publish date

Publish this app as soon as it passes certification.

Moving on to the App Properties submission step, select the category (in this case, “Entertainment”).

## Category and subcategory

Pick the category (and subcategory, if applicable) that best describes your app. [Learn more](#)



And set the age rating to 3+



Next, set the App declarations as follows (making sure your app has been fully tested, of course). You can learn more about these options by clicking on “Learn more.” Once these settings are in place, click “save” to move on.

Check any appropriate boxes below. This may affect the way your app is displayed or whether it is offered to certain users. [Learn more](#)

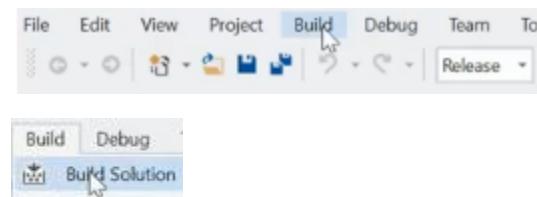
This app allows users to make purchases, but does not use the Windows Store commerce system.

This app has been tested to meet accessibility guidelines. ←

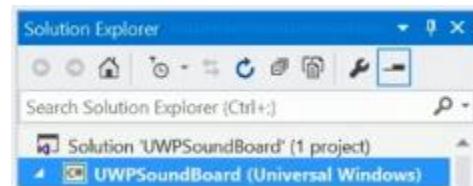
Customers can install this app to removable media such as SD cards.

Windows can include this app's data in automatic backups to OneDrive.

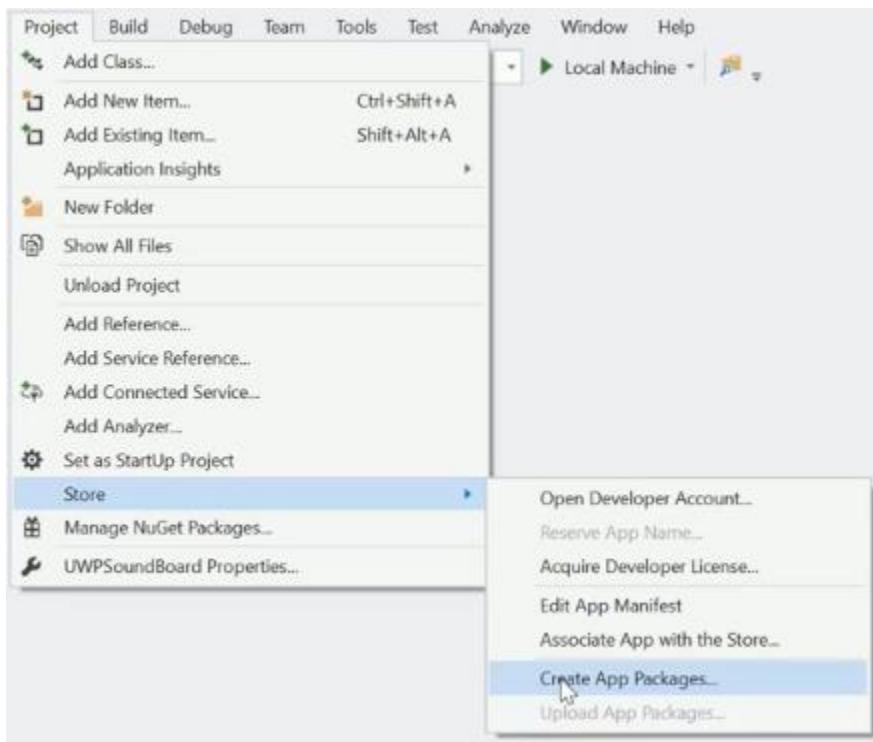
Now, going back to the project in Visual Studio let's change the version to a Release version, and build that solution.



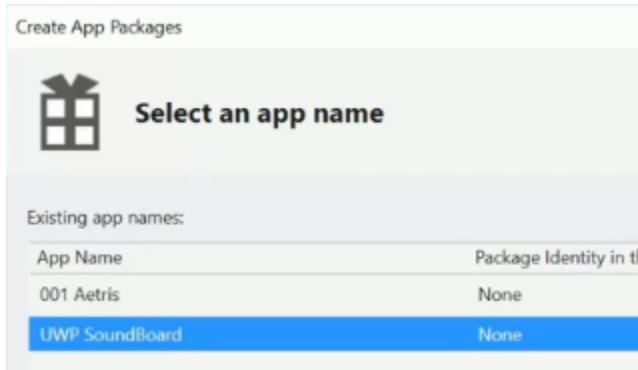
Next, with the project selected in the Solution Explorer



Go to Project > Store > Create App Packages...



Click “Yes” on the next screen to build the app Packages and sign into the Store with your Microsoft account if it asks you to. It will then ask you for an authentication Code which you can have sent to you via a text message or an email. Once you’re signed in select the existing app name you set for your package, and click “Next.”



After that, leave the current settings as they are and click on “Create.” Visual Studio will then go through the requisite steps and, if there are no errors, give you a message that the task was completed successfully.

Going back to the browser where you were in the middle of completing your submission, click on “Packages.”

## Submission 1

Delete

Pricing and availability

Not started

App properties

Not started

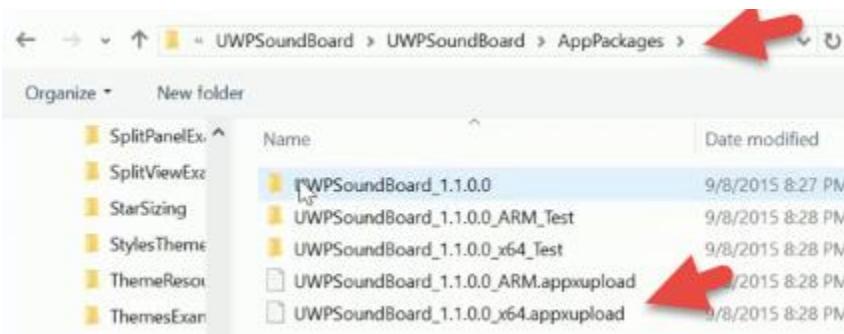
Packages



And then click on “browse your files”



Navigating through your directories to locate your builds, you can upload your app Packages. In this case there are two Packages (ARM, x64) for different devices. Once your Packages are uploaded click on “Save” to complete this step.



	Name	Date modified
SplitPanelEx.	UWPSoundBoard_1.1.0.0	9/8/2015 8:27 PM
SplitViewEx.	UWPSoundBoard_1.1.0.0_ARM_Test	9/8/2015 8:28 PM
StarSizing	UWPSoundBoard_1.1.0.0_x64_Test	9/8/2015 8:28 PM
StylesTheme	UWPSoundBoard_1.1.0.0_ARM.appxupload	9/8/2015 8:28 PM
ThemeResor	UWPSoundBoard_1.1.0.0_x64.appxupload	9/8/2015 8:28 PM
ThemesExan		

Next, create a description for the app and add screenshots if you wish.

Description\*

A fun sound board application to play noises for friends.

App features

Play built in sounds

Drag and drop your custom sounds to play them

Add more

Recommended hardware

Add more

Keywords

sound board

mp3 wav

Once that is all done, continue on to “Notes for certification” at the Submission page. Include here any non-obvious feature you think would be interesting to note for the certification process.

## Notes for certification

Provide any info that helps testers use and understand your app.

Customers won't see this info.

Include the following (if applicable):

- Username and password for a test account
- Steps to access hidden or locked features
- Steps to verify background audio usage

Supports drag and drop ... drag an MP3 or WAV file from the desktop to the SoundBoard.

Save

With all of the steps completed, click on “Submit to the store,” and all that is left to do is wait and see if your app is accepted!

## Pricing and availability

Free and available to customers.

Complete ✓

## App properties

Entertainment, 3+

Complete ✓

## Packages

UWPSoundBoard_1.1.0.0_x64.appxu...	Validated
UWPSoundBoard_1.1.0.0_ARM.appx...	Validated

Complete ✓

## Descriptions

English (United States)	Complete
-------------------------	----------

Complete ✓

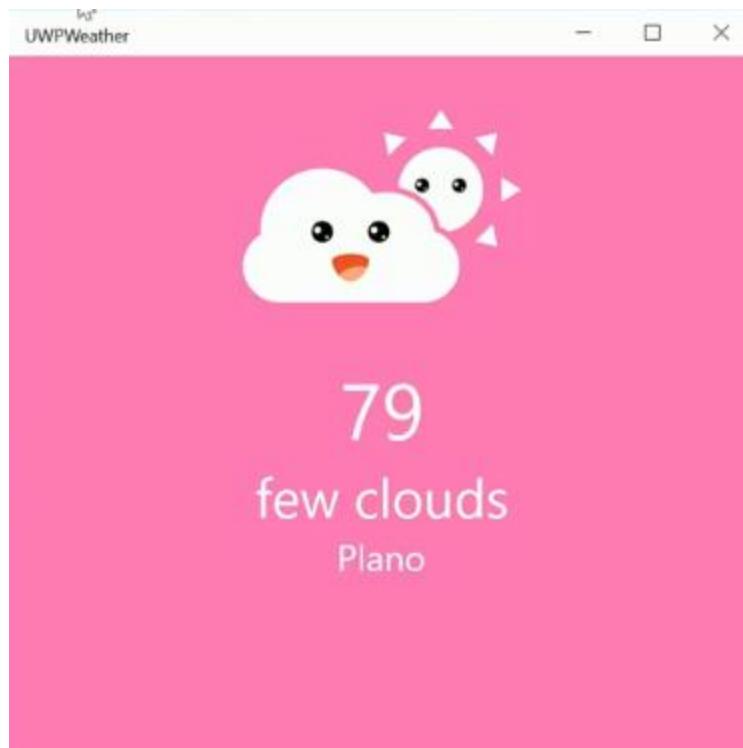
## Notes for certification

Complete ✓

Submit to the Store

## UWP-057 - UWP Weather - Introduction

The next app that we will build together is a very simple weather application that brings together a lot of cool technologies in order to produce a very interactive app. Now, simple weather apps are ubiquitous, however, building one ourselves will demonstrate some pretty cool features of Windows 10 and how to work with external services to get data to update and make your apps come alive. This app will be able to run on, both, mobile and desktop platforms and the way that it will work is - based on your current location - it's going to find the current weather conditions, your exact location, and display an image that relates to the weather condition.



One of the main things that you'll learn with this example is how to use location services for Windows 10 to determine the geo position of where your device, or your computer, is currently at. And once you have that information (latitude and longitude) we can then make calls out to an external web service - like [openweathermap.org](http://openweathermap.org) - which will return the weather for that location, and even future forecasts for the hours and days ahead.

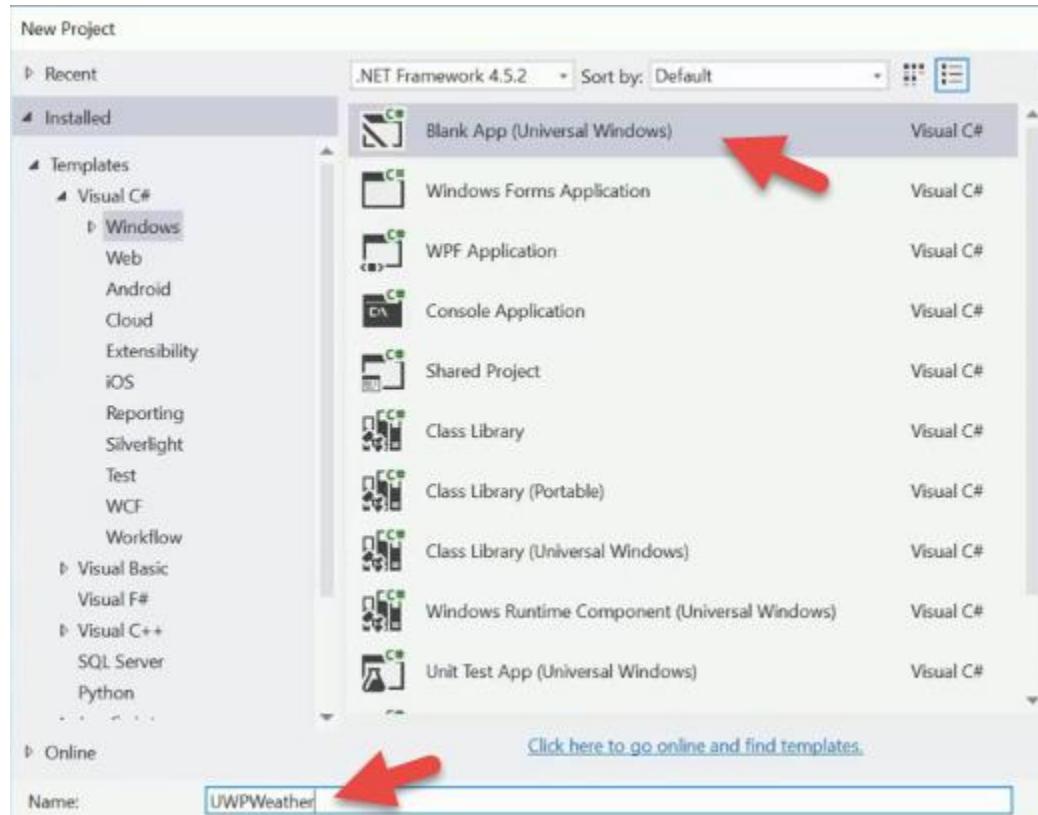
Part of this process will reveal how to retrieve data from a web service through a file format called JSON (JavaScript Object Notation), which in turn lets you deserialize that information into instances of classes that we can work with in C#. This will be helpful not only to the task at hand but also in learning a crucial development skill of serializing/deserializing data to and from formats like XML/JSON and objects in your code.

We will also look at how to work with the Phone Emulator to test our application. This will include testing not only the design and layout, but also how to change the position, and tell the location service, that we're actually coming from a given location.

And, finally, we will also take a look at how to build a live tile for our application that updates periodically with current information. Aside from building live tiles, this process will also teach you how to create a web service using ASP.NET MVC 5 and host it out of your website.

## UWP-058 - UWP Weather - Setup and Working with the Weather API

Let's get started in creating our weather app by opening up a blank UWP project and naming it "UWPWeather."



Before we get started with this project, note that we are going to be pulling weather data from [openweathermap.org](http://openweathermap.org)

If you intend to publish an app using the API found on that site you will likely have to purchase a license depending on how many people intend to use the app. But for the simple purpose of demonstration used in this lesson you will not have to worry about that. With that in mind you should turn your attention next to the API information found at

[openweathermap.org/current](http://openweathermap.org/current)

Notice that there are a variety of ways to make API calls to this resource, depending on the information you provide via the URL. The one we will be using in our application will be the one that uses geographic coordinates

## By geographic coordinates

API call:

[api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}](http://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon})

Parameters:

**lat, lon** coordinates of the location of your interest

Examples of API calls:

[api.openweathermap.org/data/2.5/weather?lat=35&lon=139](http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139)

API respond:

```
{"coord":{"lon":139,"lat":35},  
"sys":{"country":"JP","sunrise":1369769524,"sunset":1369821049},  
"weather":[{"id":804,"main":"clouds","description":"overcast clouds","icon":"04n"}]}
```

With that being the case let's begin by finding the latitude/longitude for the location we have in mind (in this case, we're interested in the coordinates for Dallas, Texas)

bing dallas latitude longitude

Web Images Videos Maps News Explore

1,130,000 RESULTS Any time

Dallas, Texas coordinates

32.7771° N, 96.7962° W

Copy that information somewhere and keep it handy while going back to the openweathermap.org API page – clicking on the example API call link

Examples of API calls:

[api.openweathermap.org/data/2.5/weather?lat=35&lon=139](http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139)

And replace the values for latitude and longitude with the values we obtained in the web search (latitude being roughly 32.77, while longitude is roughly -96.79)

[api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.79](http://api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.79)

What you will get back is a bunch of information arranged in JSON format. Quickly convert that Object Notation into C# code by copying and pasting it into the converter found at

[Json2csharp.com](http://Json2csharp.com)

```
{"coord":{"lon":-96.78,"lat":32.77}, "weather":  
[{"id":500,"main":"Rain","description":"light rain","icon":"10n"}], "base":"cmc  
stations", "main":  
{"temp":299.43,"pressure":1012,"humidity":83,"temp_min":297.59,"temp_max":301.15}, "wind":  
{"speed":2.1,"deg":80}, "rain": {"lh":0.2}, "clouds": {"all":40}, "dt":1441847186, "sys":  
{"type":1, "id":2592, "message":0.0076, "country": "US", "sunrise":1441886849, "sunset":14419  
32012}, "id":4684904, "name": "Dallas County", "cod":200}
```

Generate



The resulting conversion is a deserialized version of this information, represented in C# Classes and Properties. Copy and paste all of that into a new class within your Visual Studio project. Call that class "OpenWeatherMapProxy"

Add New Item - UWPWeather

The screenshot shows the 'Add New Item' dialog in Visual Studio. The left sidebar lists categories: 'Installed' (Visual C#, Code, Data, General, Web, XAML) and 'Online'. The main area shows a list of item templates under 'Visual C#': XAML View, Blank Page, Content Dialog, Resource Dictionary, Templated Control, User Control, Class (selected), Interface, Application Manifest File, Assembly Information File, Class Diagram, and Code Analysis Rule Set. At the bottom, there is a link 'Click here to go online and find templates.' and a 'Name:' input field containing 'OpenWeatherMapProxy.cs'.

Template	Language
XAML View	Visual C#
Blank Page	Visual C#
Content Dialog	Visual C#
Resource Dictionary	Visual C#
Templated Control	Visual C#
User Control	Visual C#
Class	Visual C# (Selected)
Interface	Visual C#
Application Manifest File	Visual C#
Assembly Information File	Visual C#
Class Diagram	Visual C#
Code Analysis Rule Set	Visual C#

Name: OpenWeatherMapProxy.cs

Step 1: alongside that class just paste in everything you copied from the JSON-to-C# conversion (partially represented in the image below)

```
namespace UWPWeather
{
    public class OpenWeatherMapProxy
    {

    }

    public class Coord
    {
        public double lon { get; set; }
        public double lat { get; set; }
    }

    public class Weather
    {
        public int id { get; set; }
        public string main { get; set; }
        public string description { get; set; }
    }
}
```

Step 2: Now in the MainPage.xaml start with a simple StackPanel so we can simply test grabbing the data and displaying it

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

    <StackPanel Background="HotPink">
        <Button Content="Get Weather" Click="Button_Click" />
        <TextBlock Name="ResultTextBlock" />
    </StackPanel>

```

Step 3: create the static method called GetWeather() with the aim of making a call to the outside web service via an HttpClient

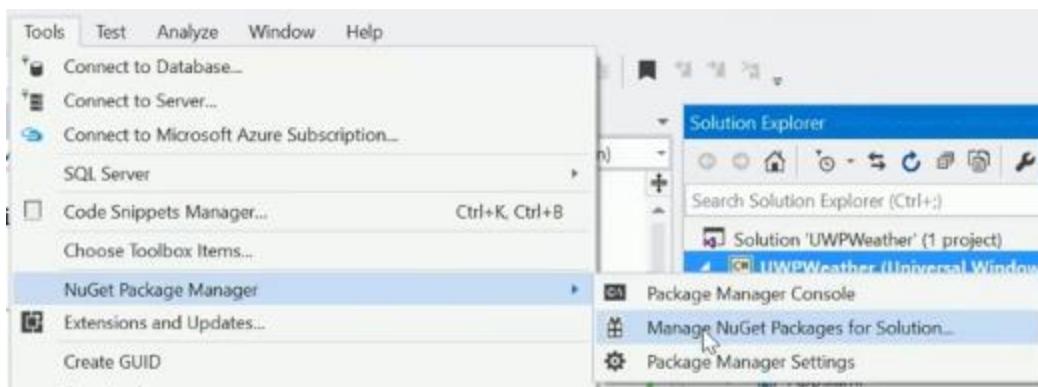
```

public class OpenWeatherMapProxy
{
    public static RootObject GetWeather(double lat, double lon)
    {
        // Your code here
    }
}

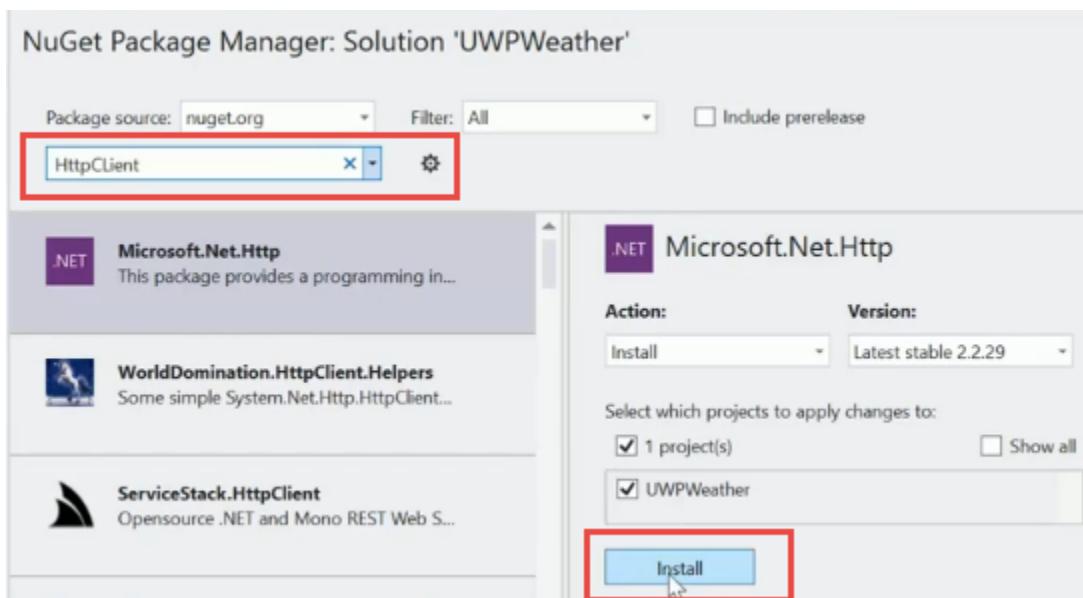
```

Step 4: add the HttpClient to this code block by going to

Tools > NuGet Package Manager > Manage NuGet Packages for Solution



Step 5: search for the HttpClient and install it (click and accept the resulting nag screens)



And back in your code add the using statement and code as follows

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http; ←
using System.Text;
using System.Threading.Tasks;

namespace UWPWeather
{
    public class OpenWeatherMapProxy
    {
        public async static RootObject GetWeather(double lat, double lon)
        {
            var http = new HttpClient();
            var response = await http.GetAsync("");
        }
    }
}

```

Step 6: Now, for testing purposes, simply copy and paste that previous URL we had before in between the double quotes of the http.GetAsync() method. That URL again is

<http://api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.79>

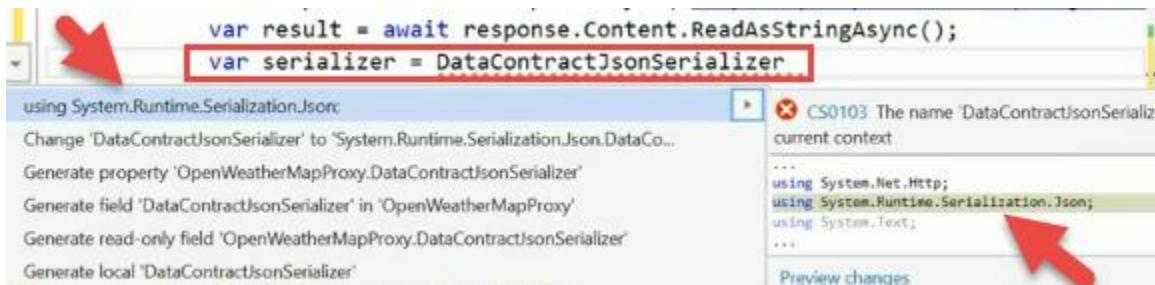
Now, take all of that data pulled from the web server and stored locally in a variable called “response,” and store the content as a string in another variable called “result”

```

var response = await http.GetAsync("http://api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.79");
var result = await response.Content.ReadAsStringAsync();

```

Step 6: Now let’s use a serializer to handle the serialization/deserialization of JSON/Object data by typing in this reference (hit the Ctrl key + the period key while DataContractJsonSerializer is selected to import the namespace, else add the reference to “using System.Runtime.Serialization.Json;” manually at the top of your script)



And complete this line of code as follows

```
var serializer = new DataContractJsonSerializer(typeof(RootObject));
```

Step 7: Add an attribute to each class in order to allow serialization. Type in “[DataContract]” above the class and import the requisite namespace (using System.Runtime.Serialization) as follows

The screenshot shows a code editor with a tooltip for the [DataContract] attribute. The tooltip contains the following text:

```
CS0246 The type or namespace name 'System.Runtime.Serialization' could not be found (are you missing a using directive or an assembly reference?)
```

The code in the editor is:

```
[DataContract]
using System.Runtime.Serialization;
public class Coord
{
    public double lon { get; set; }
    public double lat { get; set; }
}
```

That's for serializing the class, now to serialize the properties within the class add an attribute to the top of each property (do this for every class and property that we copied and pasted from the JSON-to-C# conversion)

The screenshot shows the same code editor with the [DataMember] attributes added to the properties:

```
[DataContract]
public class Coord
{
    [DataMember]
    public double lon { get; set; }

    [DataMember]
    public double lat { get; set; }
}
```

Step 8: Go ahead now and comment out some code that might be invalid or not important to our app

The screenshot shows the Rain class with several lines of code commented out using /\* \*/:

```
/*
[DataContract]
public class Rain
{
    [DataMember]
    public double __invalid_name__1h { get; set; }
}

/*
[DataMember]
public Rain rain { get; set; }
*/
```

Step 9: Add “using System.IO;” to your using declarations at the top of the document and add a MemoryStream in your code and properly encode JSON using UTF8

```
var serializer = new DataContractJsonSerializer(typeof(RootObject));  
  
var ms = new MemoryStream(Encoding.UTF8.GetBytes(result));
```

A MemoryStream allows us to just have data come in and go out - whenever you have something sending data at a different rate of speed than something accepting data, you use a Stream. By using a MemoryStream here we are just keeping everything in memory until it all gets sorted out. Data will go in one end, and then it'll be taken out as it's needed on the other end.

Step 10: Now, let's get data out of the serializer and return it at the end of our method

```
var ms = new MemoryStream(Encoding.UTF8.GetBytes(result));  
var data = (RootObject)serializer.ReadObject(ms);  
  
return data;
```

And to make this work properly as an async method, we need to modify the return type in the method signature as follows

```
public async static Task<RootObject> GetWeather(double lat, double lon)  
{
```

Step 11: Let's make use of this code by referencing it in MainPage.xaml.cs

```
private async void Button_Click(object sender, RoutedEventArgs e)  
{  
    RootObject myWeather = await OpenWeatherMapProxy.GetWeather(20.0, 30.0);  
  
    ResultTextBlock.Text = myWeather.name + " - " + myWeather.
```

And to get the temperature and weather description complete this line of code with

```
myWeather.main.temp + " - " + myWeather.weather[0].description;
```

The result when you run the program should look something like this



Step 12: Get the temperature reading to look right by adding the following modifications to your code

```
http://api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.7&units=imperial
```

Now cast the temperature to int, and after that convert it to a string to get rid of the decimal with this modification

```
((int)myWeather.main.temp).ToString()
```

Step 13: You may have noticed that the OpenWeatherMap API gives us access to icons, with specific names for each (you can find them at [openweathermap.org/weather-conditions](http://openweathermap.org/weather-conditions)). Let's implement an icon in our app by adding the following



```
<StackPanel Background="HotPink">
    <Button Content="Get Weather" Click="Button_Click" />
    <TextBlock Name="ResultTextBlock" />
    <Image Name="ResultImage" Width="200" Height="200" />
</StackPanel>
```

And in MainPage.xaml.cs start by adding the namespace

```
Using Windows.UI.Xaml.Media.Imaging;
```

And, finally, add the code to the Button\_Click method

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    RootObject myWeather = await OpenWeatherMapProxy.GetWeather(20.0, 30.0);
    string icon = String.Format("http://openweathermap.org/img/w/{0}.png", myWeather.weather[0].icon);
    ResultImage.Source = new BitmapImage(new Uri(icon, UriKind.Absolute));
    ResultTextBlock.Text = myWeather.name + " - " + ((int)myWeather.main.temp).ToS
}
```

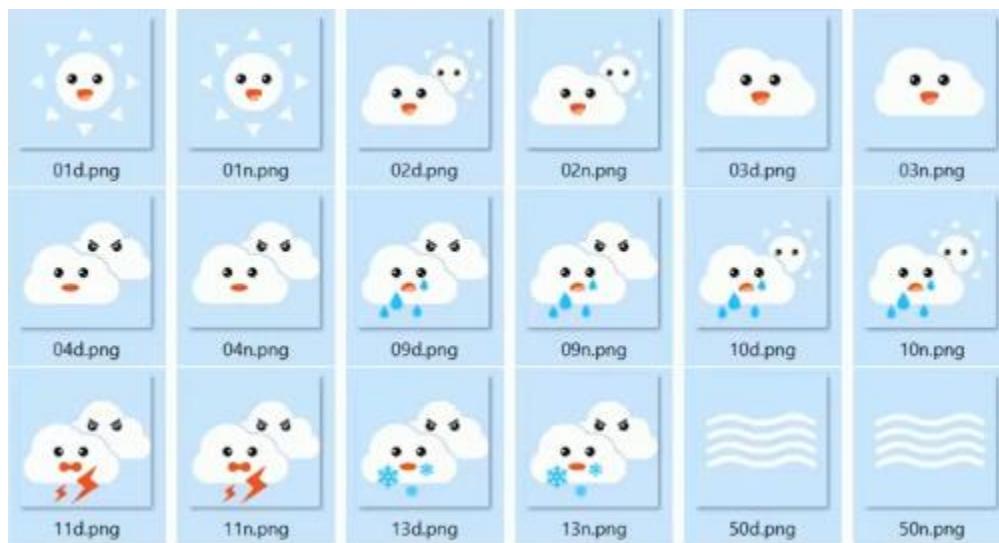
When you run the program you should now see an image in the application, however, it's a bit generic so in the next lesson we will look at replacing it with our own local asset.

## UWP-059 - UWP Weather - Accessing the GPS Location

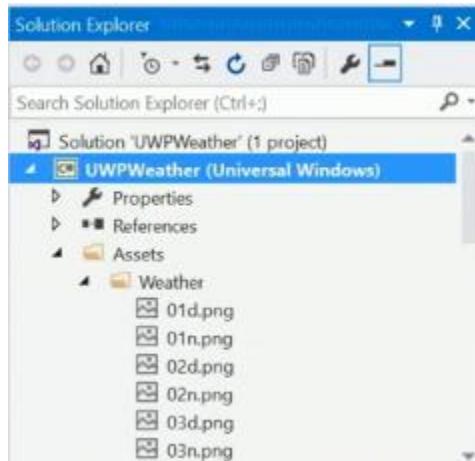
For this lesson you can download the icons from

<http://openweathermap.org/weather-conditions>

Or, you can create your own icons with the same dimensions and names used in this lesson. The icons used in this lesson look like this



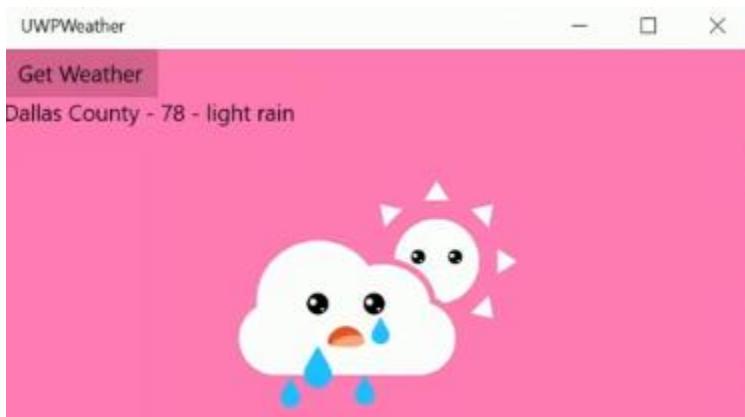
Step 1: Create a folder called “Weather” and drag/drop your icons into that folder



Step 2: Modify the code from the previous lesson to refer to your local directory of icons, rather than the web server

```
ng icon = String.Format("ms-appx:///Assets/Weather/{0}.png", myWeather.weather[0].icon);
```

Your application should now show the icon when you run it



Step 3: In order to utilize a sensor on the platform you're running on, you have to declare a capability in the Package.appxmanifest

The screenshot shows the Windows Manifest Designer interface for a UWP app. The title bar says "Package.appxmanifest". Below it, there are tabs for "MainPage.xaml.cs", "OpenWeatherMapProxy.cs", and "MainPage.xaml". A red box highlights the "Capabilities" tab in the navigation bar. The main content area has tabs for "Declarations", "Content URIs", "Packaging", and "Visual Assets". Under "Visual Assets", there is a link to "Manifest Designer Help". The "Capabilities" tab is selected and contains a table:

Capabilities:	Description:
<input type="checkbox"/> All Joyn	Provides access to the current location, which is obtained from dedicated hardware or derived from available network information.
<input type="checkbox"/> Blocked Chat Messages	
<input type="checkbox"/> Chat Message Access	
<input type="checkbox"/> Code Generation	
<input type="checkbox"/> Enterprise Authentication	
<input checked="" type="checkbox"/> Internet (Client)	
<input type="checkbox"/> Internet (Client & Server)	
<input checked="" type="checkbox"/> Location	

A large red arrow points to the "Location" checkbox in the list.

For more information, and code, that allows for more sophisticated location services in your app visit the following URL

[Msdn.microsoft.com/en-us/library/windows/xaml/mt219698.aspx](http://Msdn.microsoft.com/en-us/library/windows/xaml/mt219698.aspx)

Or, use a search engine and look for “Windows Dev Center Get current location.” However, for this demonstration we will just keep things simple.

Step 4: Add a new class to the project and call it “LocationManager.” Add the using declaration and method shown here

```

using Windows.Devices.Geolocation;

namespace UWPWeather
{
    public class LocationManager
    {
        public async static Task<Geoposition> GetPosition()
        {
            var accessStatus = await Geolocator.RequestAccessAsync();

            if (accessStatus != GeolocationAccessStatus.Allowed) throw new Exception();

            var geolocator = new Geolocator { DesiredAccuracyInMeters = 0 };

            var position = await geolocator.GetGeopositionAsync();

            return position;
        }
    }
}

```

And make a reference to GetPosition() within the Button\_Click() method

```

private async void Button_Click(object sender, RoutedEventArgs e)
{
    var position = await LocationManager.GetPosition();
}

```

And in that same method modify the GetWeather() call with the following code

```

RootObject myWeather =
    await OpenWeatherMapProxy.GetWeather(
        position.Coordinate.Latitude,
        position.Coordinate.Longitude);

```

Note the green squiggly lines means this code is considered deprecated, so there may come a time in the future when it is no longer supported. You can now test this line of code by setting a breakpoint and checking the value for these variables (with the green squiggly lines)

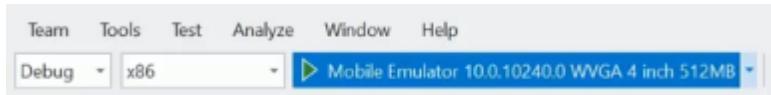
35        RootObject myWeather =  
 36        await OpenWeatherMapProxy.GetWeather(  
 37              position.Coordinate.Latitude,  
 38              position.Coordinate.Longitude);  
 39  
 40

The line of code `position.Coordinate.Longitude` is highlighted with a yellow background and has green squiggly lines underneath it, indicating it is deprecated.

In the next lesson we will use the phone emulator, simulating changing the geo position within the country, and see what it would look like if the app was running from Seattle, or from New York, or from Chicago, etc.

## UWP-060 - UWP Weather - Testing Location in the Phone Emulator

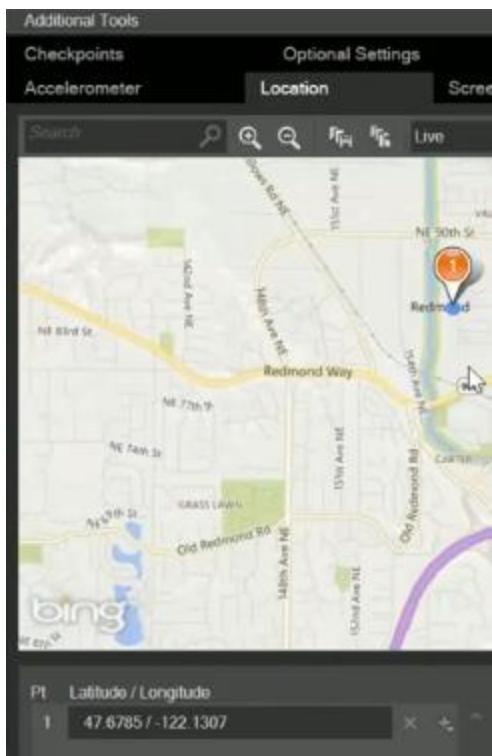
In this lesson, we will learn a little bit more about how to use some of the features of the Phone Emulator. The first thing you should do is change the environment in Visual Studio to a Mobile Emulator and the run the application



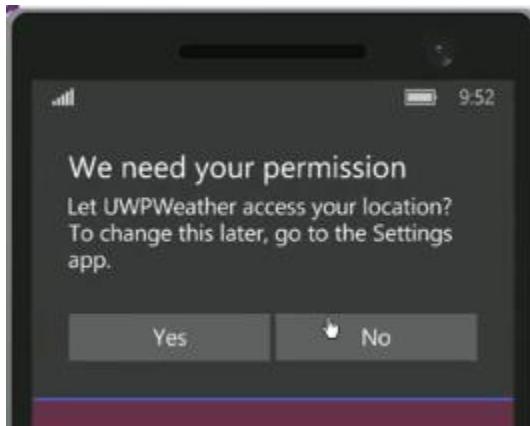
What we will want to do is test our app to see if it's grabbing coordinates correctly - getting geo coordinates from different places in the world, and so forth. So, once the emulator is running, double click the chevron icon to open up the Tools pane for the emulator and go to the location tab.



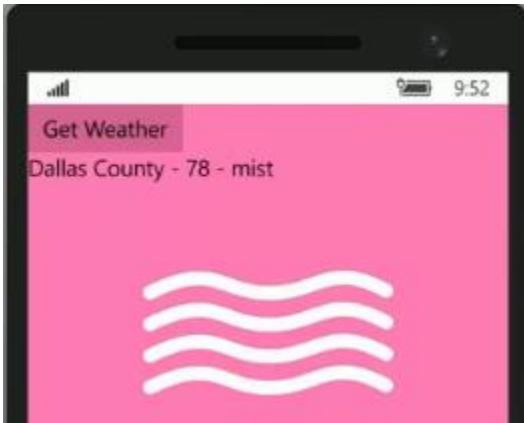
And on the map that pops up, zoom in and click on a location to tell the emulator that that is your location (in this case we are choosing Redmond, Washington)



Step 1: In your running app click on the “Get Weather” button and it will ask you to allow location access for the UWPWeather app



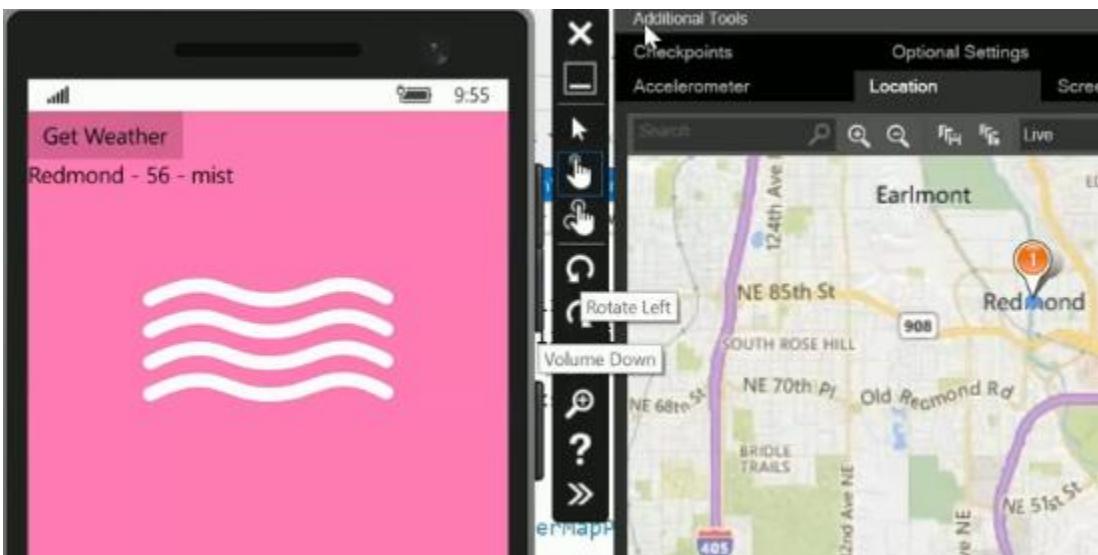
You may notice that even though the location is different from before, it still reports the previous location's weather that was hard-coded in the app previously



Step 2: So, in the code we need to modify the references for the latitude and longitude we supplied previously. To do so, make all of the changes to the GetWeather() method shown here, paying particular attention to the formatting used in the URL (using {0} and {1} as placeholders for the lat, and lon arguments)

```
public async static Task<RootObject> GetWeather(double lat, double lon)
{
    var http = new HttpClient();
    var url = String.Format("http://api.openweathermap.org/data/2.5/weather?lat={0}&lon={1}&units=imperial", lat, lon);
    var response = await http.GetAsync(url);
```

Now run the application, again choosing a different location on the map and you should see it reflected in the app itself.



As you can see the emulator can be used to troubleshoot and test out functionality in your code that is otherwise difficult to pin down.

## UWP-061 - UWP Weather - Updating the Tile with Periodic Notifications

Our application is now feature complete, for the most part. However, we still need to address some layout issues and make it look a little nicer but before we get to that point, let's add one more cool feature. And that feature is adding Live Tiles to your application, and have those tiles update based on content from your application.

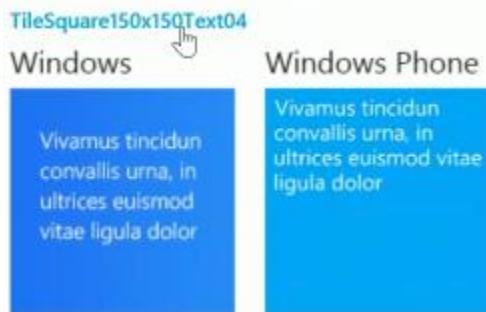
First, we will change the content of the tile on demand, which means the program is going to actively push data into that tile. And then second, we will schedule periodic notification - and give it some source online to pull data from - to present data to the user.

For more information on tiles, refer to the tile template catalog

<http://bit.do/template-catalog>



Take note of the slightly different ways a tile may appear depending on the platform it will be displayed on. When you find a tile that fits the scenario you have in mind for your app, click on the link to that tile in order to get more descriptive information on it, including its template.



And you can pull, from that resource, the XML template information for use within your code

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text01" fallback="TileSquareText01">
      <text id="1">Text Field 1 (larger text)</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4</text>
    </binding>
  </visual>
</tile>
```

Below you will find some example code which shows this relationship between the XML template and your code

```
private void ChangeTileContentButton_Click(object sender, RoutedEventArgs e)
{
    var tileXml = TileUpdateManager.GetTemplateContent(TileTemplateType.TileSquare150x150Text01);

    var tileAttributes = tileXml.GetElementsByTagName("text");
    tileAttributes[0].AppendChild(tileXml.CreateTextNode(MyTextBlock.Text));
    var tileNotification = new TileNotification(tileXml);
    TileUpdateManager.CreateTileUpdaterForApplication().Update(tileNotification);
}
```

Let's break this code block down a bit, and describe what it's doing. Here, we will find an element with the name "text" and we will add a child to that first element with whatever we typed into that TextBlock.

```
var tileAttributes = tileXml.GetElementsByTagName("text");
tileAttributes[0].AppendChild(tileXml.CreateTextNode(MyTextBlock.Text));
```

So, in the XML, it will find this text element, and then add a child to the element (highlighted in blue)

```
<binding template="TileSquare150x150Text01" fallback="TileSquareText01">
<text id="1">Text Field 1 (larger text)</text>
```

And it will then fill in the actual text between that element's text tags, with whatever we typed in.

```
<text id="1">Text Field 1 (larger text)</text>
```

And then after it's done that, it will add that new XML to a new tile notification object.

```
tileAttributes[0].AppendChild(tileXml.CreateTextNode(MyTextBlock.Text));
var tileNotification = new TileNotification(tileXml);
```

And then tell the TileUpdateManager to go update our application's tile with this other tile template that we've implemented.

```
var tileNotification = new TileNotification(tileXml);
TileUpdateManager.CreateTileUpdaterForApplication().Update(tileNotification);
```

Okay, so that's the basic course of events here in five lines of code. Next, we have a ScheduleNotificationButton\_Click() event asks the TileUpdateManager to just update every half hour with this URL that's given to it.

```
private void ScheduleNotificationButton_Click(object sender, RoutedEventArgs e)
{
    var tileContent = new Uri("http://periodic.azurewebsites.net/");
    var requestedInterval = PeriodicUpdateRecurrence.HalfHour;

    var updater = TileUpdateManager.CreateTileUpdaterForApplication();
    updater.StartPeriodicUpdate(tileContent, requestedInterval);
}
```

So, now we would need to just create some sort of web service, at that URL, that will return XML sufficient for updating our tile. Below is part of an example web service, running on Azure, that simply returns the string formatted current DateTime (and, as you can see, it's using a particular tile template to display that information).

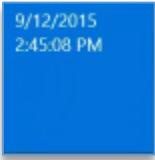
```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text04" fallback="TileSquareText04">
      <text id="1">@DateTime.Now.ToString()</text>
    </binding>
  </visual>
</tile>
```

So, again, this bit of code reaches out to that web service (roughly every half hour) and pushes that remote information to update a local tile

```
private void ScheduleNotificationButton_Click(object sender, RoutedEventArgs e)
{
    var tileContent = new Uri("http://periodic.azurewebsites.net/");
    var requestedInterval = PeriodicUpdateRecurrence.HalfHour;

    var updater = TileUpdateManager.CreateTileUpdaterForApplication();
    updater.StartPeriodicUpdate(tileContent, requestedInterval);
}
```

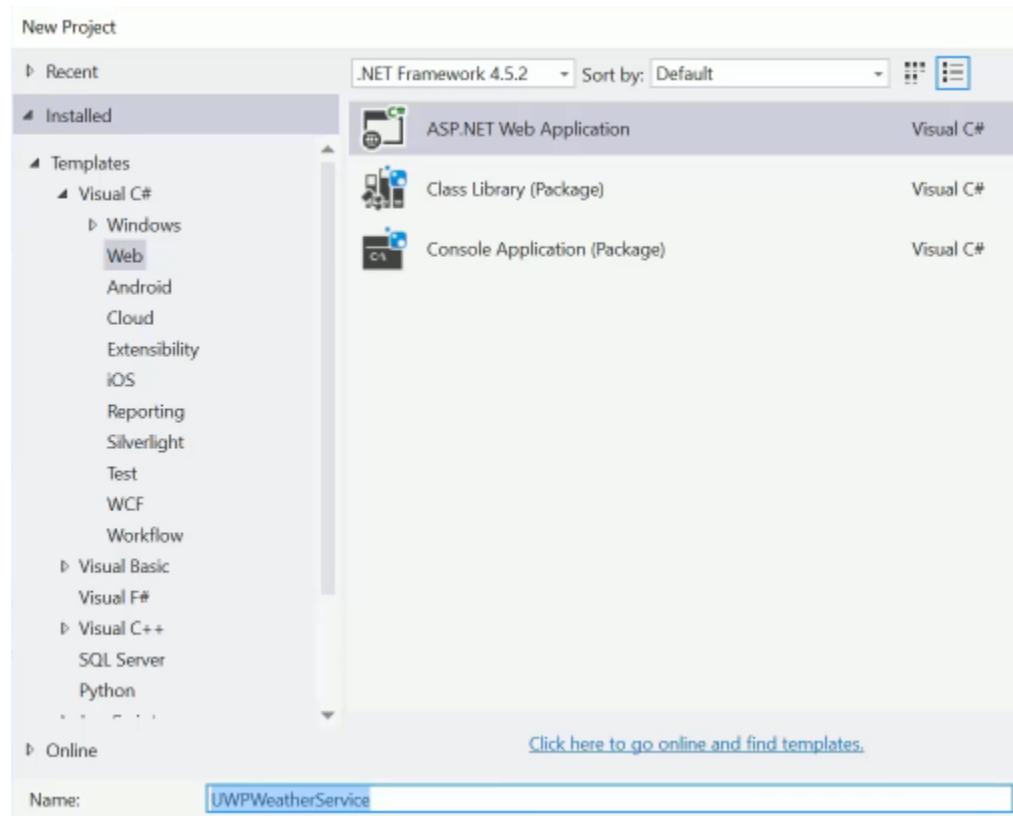
And, within your device you will see the tile automatically animate and display updated information in real time (similar to this)



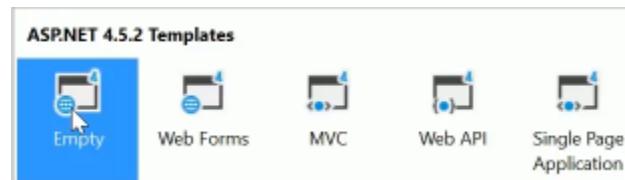
So, that's the basics for how this process is supposed to work. For more info on periodic updating (polling sequence, etc), take a look at this URL

<http://bit.do/periodic>

Now, let's implement these ideas into our current UWPWeather project. To do that, let's first create a web service through an ASP.NET Web Application project called UWPWeatherService (Note that this is an example only and will be a unique web service at a unique URL, you will want to create your own unique web service to implement this on your end. Also note that this web service will not be available when you are reading this)



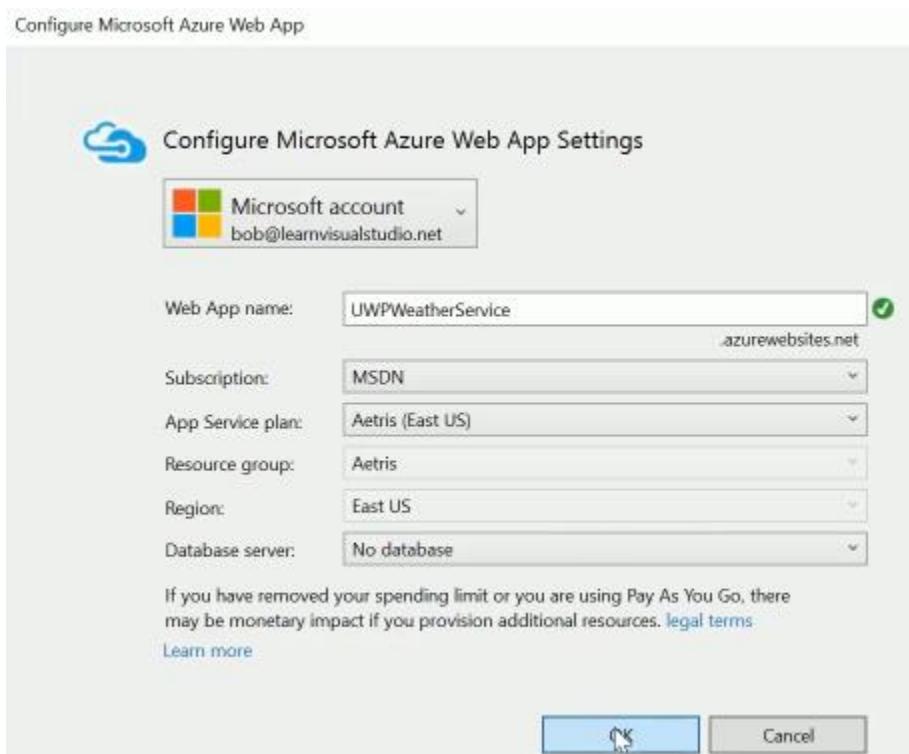
And select an Empty template for this project



Also, it will be an MVC Web App, hosted in the cloud

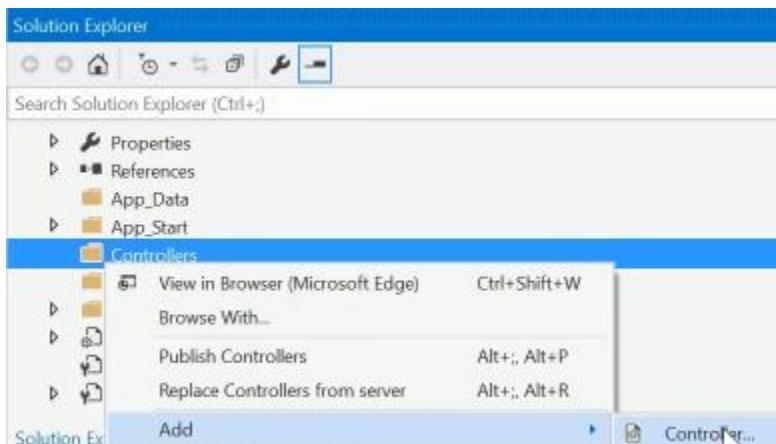


After that login with your Microsoft Account, and set up the web service using the same name as the project (this will be hosted, in this case, at <http://UWPWeatherService.azurewebsites.net>)



This should setup the publishing credentials, as well as the project itself for the MVC application. The first thing to do is right click on “Controllers” in the Solution Explorer and select

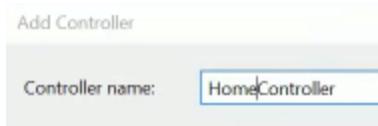
Add > Controller...



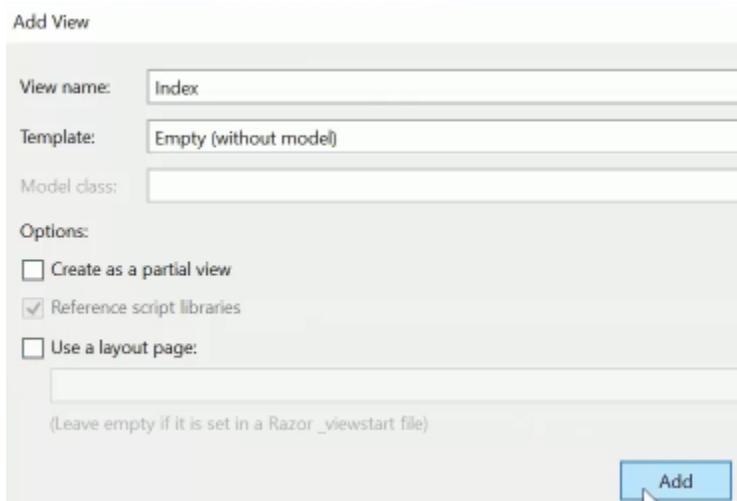
And select “MVC 5 Controller – Empty”



And then in the next dialog add a Controller called “HomeController”



Once the project is open, add a View by right clicking empty space in HomeController.cs and selecting from the menu “Add View,” and then in that dialog add it as follows



Now, remove everything from Index.cshtml and from the online template catalog we referred to earlier copy and paste, into Index.cshtml, the following XML from TileSquareText01/TileSquare150x150Text01

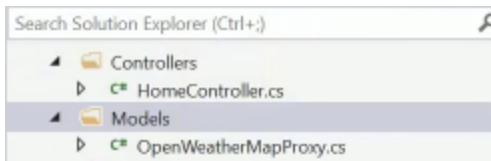
#### TileSquareText01/TileSquare150x150Text01

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text01" fallback="TileSquareText01">
      <text id="1">Text Field 1 (larger text)</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4</text>
    </binding>
  </visual>
</tile>
```

And in Index.cshtml edit the XML as follows

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text01" fallback="TileSquareText01">
      <text id="1">@ViewBag.Temp</text>
      <text id="2">@ViewBag.Description</text>
      <text id="3">@ViewBag.Name</text>
      <text id="4">@DateTime.Now.ToShortTimeString()</text>
    </binding>
  </visual>
</tile>
```

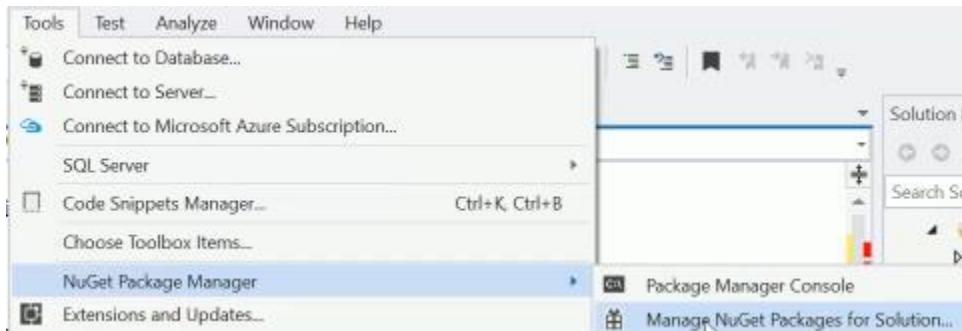
And then in the Solution Explorer, add a class to the Models folder called “OpenWeatherMapProxy”



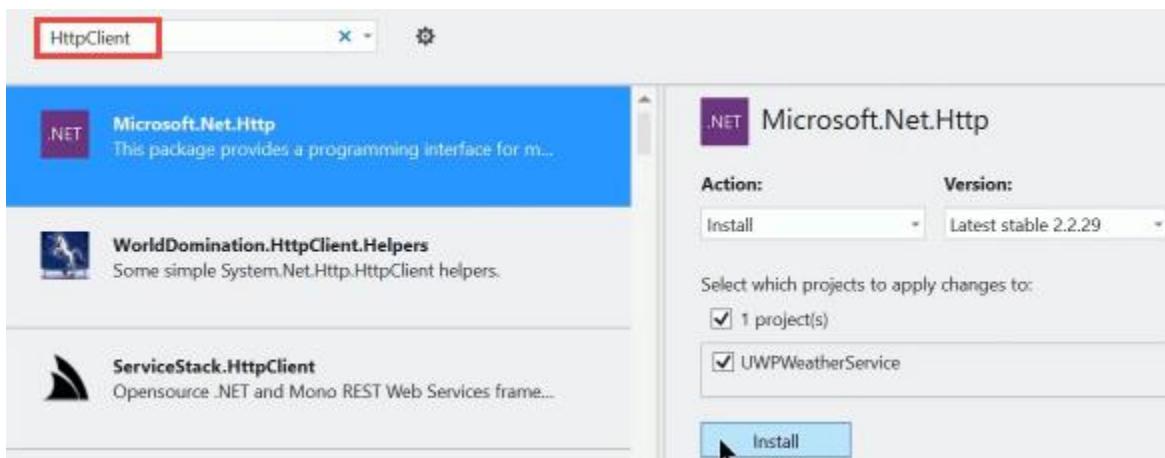
And this will be identical to the file of the same name in the UWPWeather application, so simply copy and paste the contents from that file into this new one within UWPWeatherService.

Now, add an HttpClient by going to

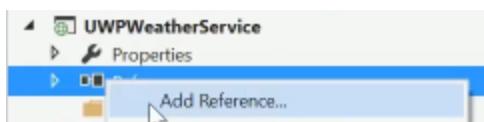
Tools > NuGet Package Manager > Manage NuGet Packages for Solution...



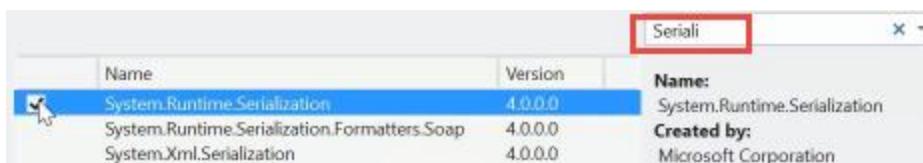
And add an HttpClient (accept and click OK to the following prompts)



In the Solution Explorer, right-click on “References” and select “Add Reference...”



And add a reference to System.Runtime.Serialization



Also add these namespaces to the top of OpenWeatherMapProxy.cs

```
using System.Threading.Tasks;
using System.Net.Http;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;
using System.IO;
using System.Text;
```

Now, inside of HomeController.cs, write the following method that populates those values from Index.cshtml and also takes in a given location

```
// GET: Home
public async Task<ActionResult> Index(string lat, string lon)
{
    // Validation / trimming
    var latitude = double.Parse(lat.Substring(0, 5));
    var longitude = double.Parse(lon.Substring(0, 5));

    var weather = await Models.OpenWeatherMapProxy.GetWeather(latitude, longitude);

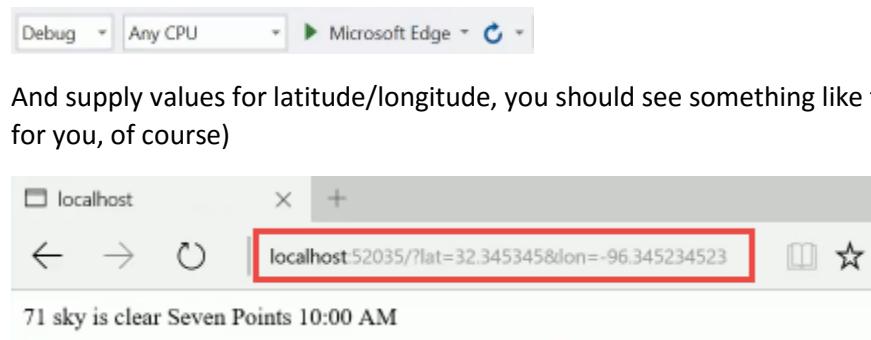
    ViewBag.Name = weather.name;
    ViewBag.Temp = ((int)weather.main.temp).ToString();
    ViewBag.Description = weather.weather[0].description;

    return View();
}
```

And add to the top of HomeController.cs the using statement

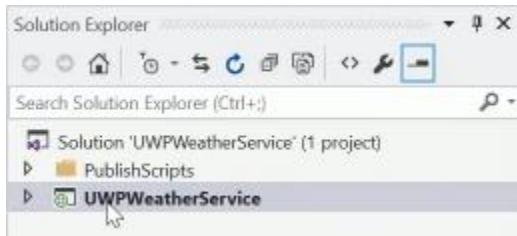
```
using System.Threading.Tasks;
```

If you now run this locally

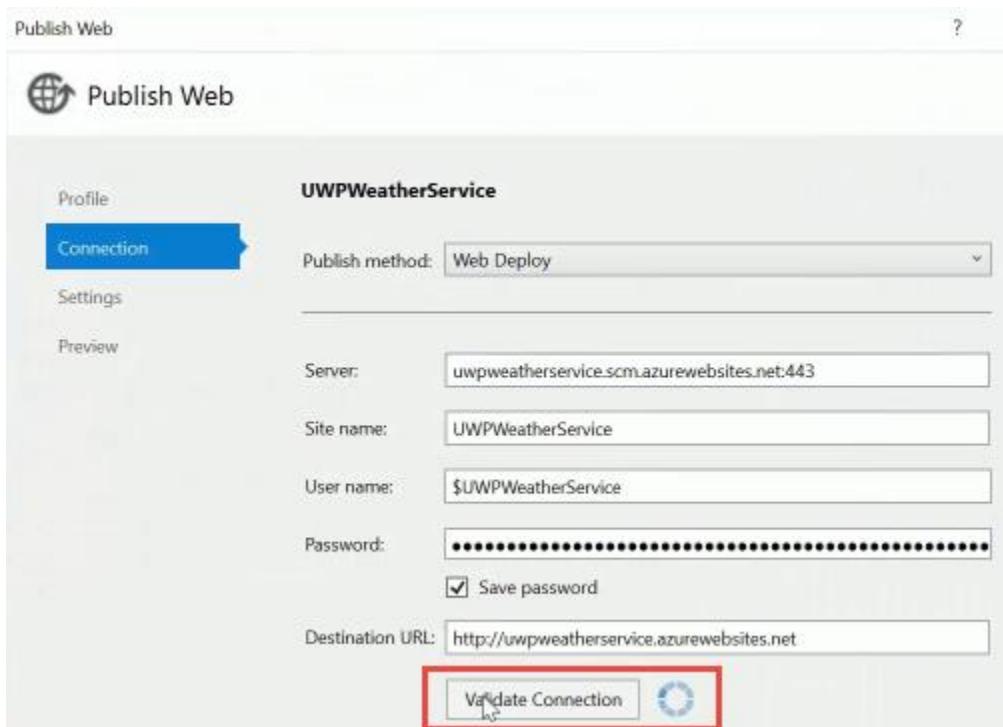


And supply values for latitude/longitude, you should see something like this (the values will be unique for you, of course)

Now, you can deploy it to your remote service by right-clicking on the project name in the Solution Explorer, and selecting “Publish”



Click on “Validate Connection”



If that succeeds, click on “Publish,” and once publishing is complete, go back to the MainPage.xaml.cs for the original UWPWeather app and modify the Button\_Click() method

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    var position = await LocationManager.GetPosition();

    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);

    // Schedule update
    var uri = String.Format(
        "http://uwpweatherservice.azurewebsites.net/?lat={0}&lon={1}", lat, lon);
```



Note that the URL will be unique to the one you happen to have deployed to. Leave the code as-is below this point.

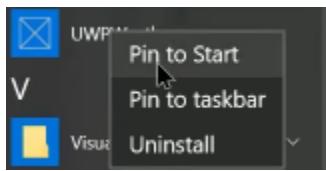
And then reference the code we mentioned at the outset into this method as follows

```
// Schedule update
var uri = String.Format("http://uwpweatherservice.azurewebsites.net/?lat{0}&lon={1}

var tileContent = new Uri(uri);
var requestedInterval = PeriodicUpdateRecurrence.HalfHour;
[redacted]
var updater = TileUpdateManager.CreateTileUpdaterForApplication();
updater.StartPeriodicUpdate(tileContent, requestedInterval);

string icon = String.Format("ms-appx:///Assets/Weather/{0}.png", myWeather.weathe
```

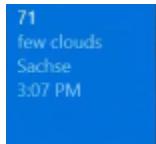
Now, to see the efforts pay off, find and pin the UWPWeatherApp



And run the program locally, clicking on “Get Weather”



And after a short period of time the remote web service should update your local tile with the weather information you input



## UWP-062 - UWP Weather - Finishing Touches

Let's finalize this app by putting on some finishing touches such as removing the "Get weather" button, centering elements and adding some exception handling.

Step 1: Modify MainPage.xaml as follows

```
<Page
    x:Class="UWPWeather.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:UWPWeather"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Loaded="Page_Loaded">
    <StackPanel Background="HotPink" >
        <StackPanel VerticalAlignment="Center">
            <Image Name="ResultImage" Width="200" Height="200" HorizontalAlignment="Center" />
            <TextBlock Name="ResultTextBlock" FontSize="36" Foreground="White" HorizontalAlignment="Center" />
        </StackPanel>
    </StackPanel>
</Page>
```

Step 2: In MainPage.xaml.cs take the entire code block within the Button\_Click() method and move it to

the Page\_Loaded() method (partially displayed below)

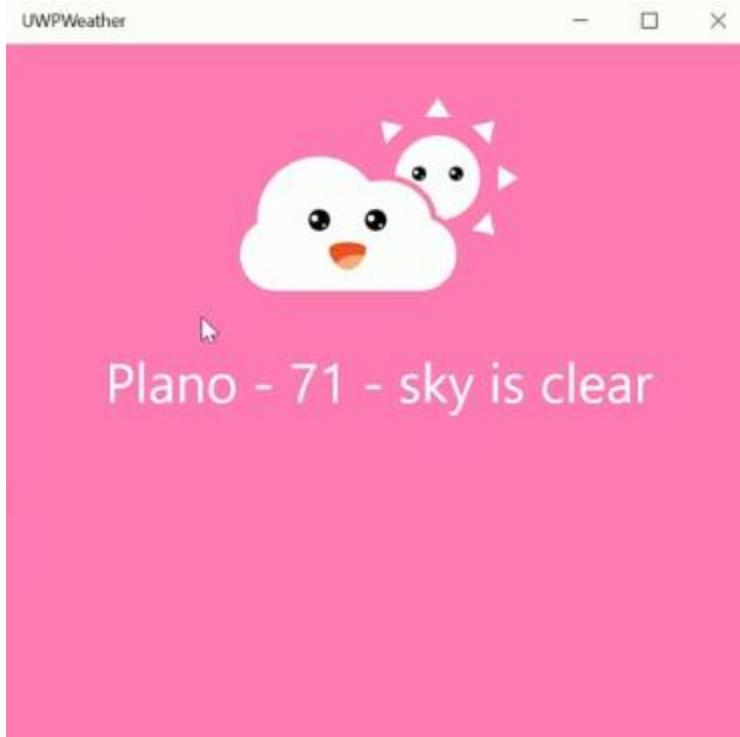
```
public MainPage()
{
    this.InitializeComponent();

}
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    var position = await LocationManager.GetPosition();

    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);
```

And when you run the program it should look like this



Step 3: Ideally, the information for each item displayed should be on its own line, so make the following changes to the StackPanel

```
<StackPanel Background="HotPink" >
    <StackPanel VerticalAlignment="Center">
        <Image Name="ResultImage" Width="200" Height="200" HorizontalAlignment="Center" />
        <TextBlock Name="TempTextBlock" FontSize="52" Foreground="White" HorizontalAlignment="Center" />
        <TextBlock Name="DescriptionTextBlock" FontSize="36" Foreground="White" HorizontalAlignment="Center" />
        <TextBlock Name="LocationTextBlock" FontSize="24" Foreground="White" HorizontalAlignment="Center" />
    </StackPanel>
</StackPanel>
```

And in MainPage.xaml.cs reference those changes by modifying the original text being displayed

```
string icon = String.Format("ms-appx:///Assets/Weather/{0}.png",
ResultImage.Source = new BitmapImage(new Uri(icon, UriKind.Absolute));

TempTextBlock.Text = ((int)myWeather.main.temp).ToString();
DescriptionTextBlock.Text = myWeather.weather[0].description;
LocationTextBlock.Text = myWeather.name;
```

Step 4: Now, to include structured exception handling let's handle a try/catch block of code. This is important whenever you have a state in your code that you can't control (such as, an unresponsive server).

```

private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    try
    {

    }
    catch
    {

    }

    var position = await LocationManager.GetPosition();
    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);

    // Schedule update
    var uri = String.Format("http://uwpweatherservice.azurewebsites.ne

```

Normally you would only wrap try/catch around the block of code that might fail, but in this case if any of it fails it will result in a crash of the application, so let's simply put the entire code block in Page\_Loaded() within a try statement (partially represented below)

```

try
{
    var position = await LocationManager.GetPosition();

    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);

    // Schedule update

```

And if an exception occurs, simply output the error that was caught like so

```

catch
{
    LocationTextBlock.Text = "Unable to get weather at this time";
}

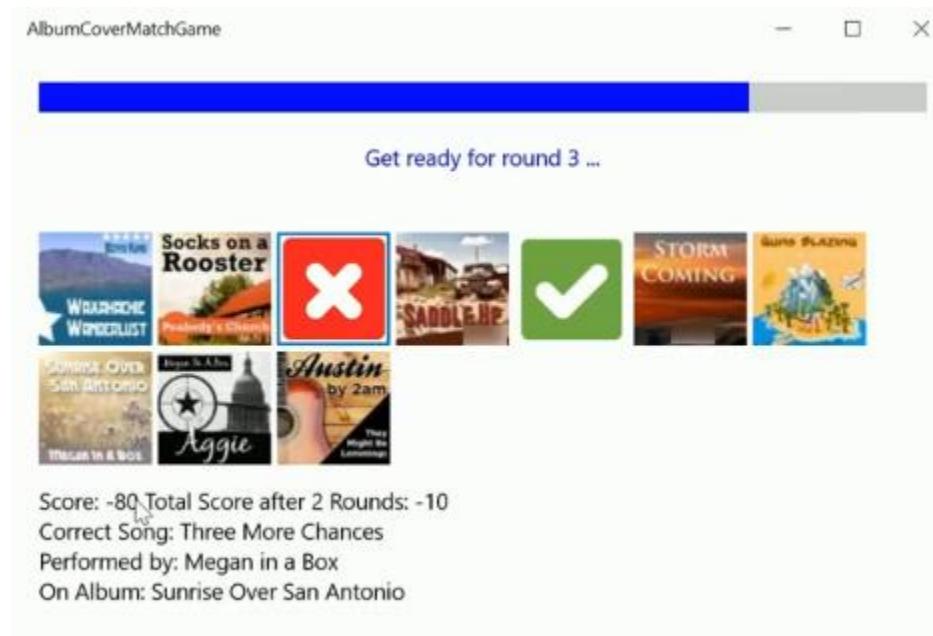
```

That pretty much completes the entirety of this weather app. You can definitely make a few tweaks to improve it if you wish. Here's an optional challenge you can take on if you so choose: have it so that along with the current weather conditions, you were able to see the forecast for the next seven days (hint: refer to the OpenWeatherMap API for a way to accomplish this).

## UWP-063 - Album Cover Match Game – Introduction

Most people have music collections that can literally have hundreds of albums and sometimes it's hard to remember what you have. In this lesson you will learn how to create an app that will allow you to rediscover music that you have in your music collection - within your music library on your computer - and make a little game out of it.

The app will look at the metadata for your albums/songs on your hard drive and when you run it, it will play a song at random and you have to guess which album it belongs to by clicking on the image of the album cover. The game will span several round and you will gain and lose points depending on how well you play. It will look something like this



You will also learn how to monetize the app by placing an advertisement within the app itself in order to serve ads through the Microsoft Advertisement Network. And then we will show you how to allow the user to remove the ad by making an in-app purchase.

## UWP-064 - Album Match Game – Setup, Working with Files & Folders

For this project, we'll start off by doing some nominal work on the user interface - just enough to facilitate this process of searching the Music Library and grabbing all the songs out - and then figure out how to retrieve 10 random songs, and their associated albums to access the various metadata.

Step 1: Create a new blank project and call it "AlbumCoverMatchGame." And in the MainPage.xaml create a simple grid to start off with

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Button Click="Button_Click" Content="Click Me" />
</Grid>
</Page>
```

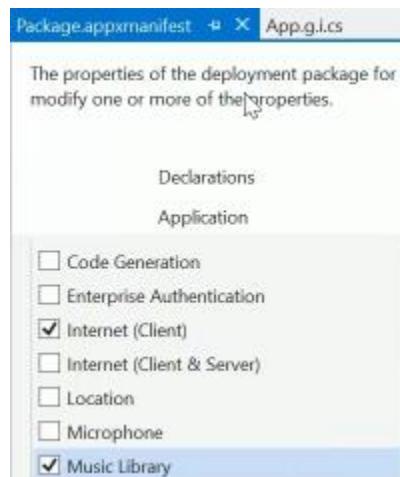
Step 2: Add this namespace to the top of MainPage.xaml.cs

```
using Windows.Storage;
```

And now write some code that will locate known Music Library folders/files to retrieve information about them

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // 1. Get access to Music library
    StorageFolder folder = KnownFolders.MusicLibrary;
    var allSongs = new ObservableCollection<StorageFile>();
```

You will also need to add a capability via the Package.appxmanifest to locate the Music Library



Step 3: Now, to get all of the files in all of the folders we will need a recursive method. A recursive method is a one that can call itself as many times as it needs to in order to traverse an entire folder

structure. We will create a custom method to handle this task. It will be asynchronous, so add this namespace

```
using System.Threading.Tasks
```

And write the custom method called RetrieveFolders() as follows

```
private async Task RetrieveFilesInFolders(
    ObservableCollection<StorageFile> list,
    StorageFolder parent)
{
    foreach (var item in await parent.GetFilesAsync())
    {
        if (item.FileType == ".mp3")
            list.Add(item);
    }

    foreach (var item in await parent.GetFoldersAsync())
    {
        await RetrieveFilesInFolders(list, item);
    }
}
```

And now call that method in the following code block, remember to make the calling method async in order to accommodate the asynchronous RetrieveFilesInFolders() method

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    // 1. Get access to Music library
    StorageFolder folder = KnownFolders.MusicLibrary;
    var allSongs = new ObservableCollection<StorageFile>();
    await RetrieveFilesInFolders(allSongs, folder);
```

Step 4: Now we will need to choose random songs from the library by writing the following, and rather large, custom method

```

private async Task<List<StorageFile>> PickRandomSongs(ObservableCollection<StorageFile> allSongs)
{
    Random random = new Random();
    var songCount = allSongs.Count;

    var randomSongs = new List<StorageFile>();

    while(randomSongs.Count < 10)
    {
        var randomNumber = random.Next(songCount);
        var randomSong = allSongs[randomNumber];

        // Find random songs BUT:
        // 1) Don't pick the same song twice!
        // 2) Don't pick a song from an album that I've already picked.

        MusicProperties randomSongMusicProperties =
            await randomSong.Properties.GetMusicPropertiesAsync();

        bool isDuplicate = false;
        foreach (var song in randomSongs)
        {
            MusicProperties songMusicProperties = await song.Properties.GetMusicPropertiesAsync();
            if (String.IsNullOrEmpty(randomSongMusicProperties.Album)
                || randomSongMusicProperties.Album == songMusicProperties.Album)
                isDuplicate = true;
        }

        if (!isDuplicate)
            randomSongs.Add(randomSong);
    }

    return randomSongs;
}

```

And in Button\_Click() reference that method as follows

```

// 2. Choose random songs from library
var randomSongs = await PickRandomSongs(allSongs);

```

Step 5: Now, to handle the metadata for each file, create a class called “Song” and put it in a new folder in the Solution Explorer called “Models”



And then create the basic properties and using statements for that class as follows

```
using Windows.Storage;
using Windows.UI.Xaml.Media.Imaging;

namespace AlbumCoverMatchGame.Models
{
    public class Song
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Artist { get; set; }
        public string Album { get; set; }
        public StorageFile SongFile { get; set; }
        public bool Selected { get; set; }

        public BitmapImage AlbumCover;
    }
}
```

Step 6: Add this namespace to MainPage.xaml.cs

```
Using AlbumCoverMatchGame.Models;
```

And create an ObservableCollection of type Song in the MainPage class

```
public sealed partial class MainPage : Page
{
    private ObservableCollection<Song> Songs;

    public MainPage()
    {
        this.InitializeComponent();

        Songs = new ObservableCollection<Song>();
    }
}
```

Step 7: Create another private helper method in the class

```
private async Task PopulateSongList(List<StorageFile> files)
{
    int id = 0;

    foreach (var file in files)
    {
        MusicProperties songProperties = await file.Properties.GetMusicPropertiesAsync();

        StorageItemThumbnail currentThumb = await file.GetThumbnailAsync(
            ThumbnailMode.MusicView,
            200,
            ThumbnailOptions.UseCurrentScale);

        var albumCover = new BitmapImage();
        albumCover.SetSource(currentThumb);

        var song = new Song();
        song.Id = id;
        song.Title = songProperties.Title;
        song.Artist = songProperties.Artist;
        song.Album = songProperties.Album;
        song.AlbumCover = albumCover;
        song.SongFile = file;

        Songs.Add(song);
        id++;
    }
}
```

And, finally, in the Button\_Click() method call PopulateSongList() as follows

```
// 3. Pluck off meta data from selected songs
await PopulateSongList(randomSongs);
```

Note that if you run the program, it will not display any data at the moment. The next lesson will cover displaying some of the data that is currently being processed in the background.

## UWP-065 - Album Match Game - Layout, Data Binding & Game Setup

In this lesson we will layout the user interface for the game and set up the data binding. We've already collected a lot of data, so now it's time to connect it to a GridView, and the DataTemplate and make it so that each individual item is clickable. The game won't be playable by the end of this lesson, but will be in really good shape to actually start the game logic, like scoring and countdown, in the next lesson.

Step 1: Setup MainPage.xaml as follows, beginning with adding the reference to Loaded="Page\_Loaded"

```
    Loaded="Page_Loaded"  
    mc:Ignorable="d">
```

And below that write the following

```

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <StackPanel Grid.Row="0">
        <ProgressBar Name="MyProgressBar" Maximum="100" Minimum="0" Value="100" Height="20" Foreground="Blue" />
        <TextBlock Name="InstructionTextBlock" Text="" Foreground="Blue" HorizontalAlignment="Center" />
        <MediaElement Name="MyMediaElement" AutoPlay="True" />
    </StackPanel>

    <StackPanel Grid.Row="1" Orientation="Vertical">
        <GridView Name="SongGridView"
            ItemsSource="{x:Bind Songs}"
            IsItemClickEnabled="True"
            ItemClick="SongGridView_ItemClick">
            <GridView.ItemTemplate>
                <DataTemplate>
                    <Grid>
                        <Image Name="AlbumArtImage"
                            Height="75"
                            Width="75"
                            Source="{x:Bind AlbumCover}" />
                    </Grid>
                </DataTemplate>
            </GridView.ItemTemplate>
        </GridView>

        <TextBlock Name="ResultTextBlock" />
        <TextBlock Name="TitleTextBlock" />
        <TextBlock Name="ArtistTextBlock" />
        <TextBlock Name="AlbumTextBlock" />
        <Button Name="PlayAgainButton"
            Content="Play Again"
            Background="Red"
            HorizontalAlignment="Center"
            Visibility="Collapsed"
            Click="PlayAgainButton_Click" />
    </StackPanel>

    <Grid Grid.Row="1">
        <ProgressRing Name="StartupProgressRing"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Width="100"
            Height="100"
            Foreground="Gray" />
    </Grid>

```

## Step 2: Add a private property to the class

```

public sealed partial class MainPage : Page
{
    private ObservableCollection<Song> Songs;
    private ObservableCollection<StorageFile> AllSongs;
}

```



And then move some of the contents of Page\_Loaded() into their own methods

```
private async Task<ObservableCollection<StorageFile>> SetupMusicList()
{
    // 1. Get access to Music library
    StorageFolder folder = KnownFolders.MusicLibrary;
    var allSongs = new ObservableCollection<StorageFile>();
    await RetrieveFilesInFolders(allSongs, folder);
    return allSongs;
}

private async Task PrepareNewGame()
{
    Songs.Clear();

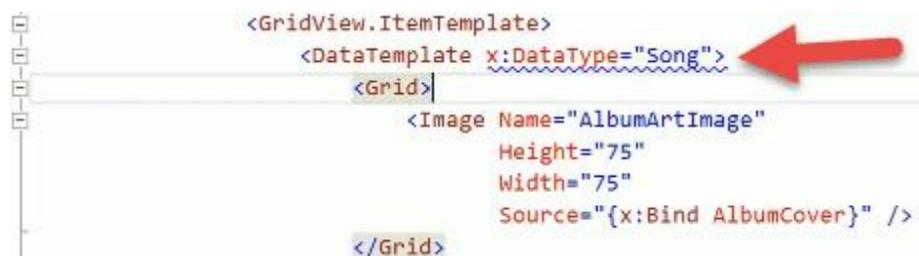
    // Choose random songs from library
    var randomSongs = await PickRandomSongs(AllSongs);

    // Pluck off meta data from selected songs
    await PopulateSongList(randomSongs);
}
```

Remove the previous contents of Page\_Loaded()

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
}
```

Step 3: Modify the MainPage.xaml



```
<GridView.ItemTemplate>
    <DataTemplate x:DataType="Song"> ←
        <Grid>
            <Image Name="AlbumArtImage"
                Height="75"
                Width="75"
                Source="{x:Bind AlbumCover}" />
        </Grid>
    </DataTemplate>

```

And at the top of this file, add the following

```
Loaded="Page_Loaded"
xmlns:data="using:AlbumCoverMatchGame.Models"
mc:Ignorable="d">
```

Step 4: Change The Grid in MainPage.xaml as follows

```
mc:Ignorable="d">
<Grid Loaded="Grid_Loaded">
```

And back in MainPage.xaml.cs add the following to the Grid\_Loaded() method, containing references to those custom methods we recently made

```
private async void Grid_Loaded(object sender, RoutedEventArgs e)
{
    StartupProgressRing.IsActive = true;

    AllSongs = await SetupMusicList();
    await PrepareNewGame();

    StartupProgressRing.IsActive = false;
}
```

When you run the program now, you should see the result that looks something like this



You can improve the margin by modifying the ProgressBar in MainPage.xaml, sticking the following code at the end

```
Margin="20,20,20,20" />
```

And also by adding a margin to the StackPanel

```
<StackPanel Grid.Row="1" Orientation="Vertical" Margin="20">
```

## UWP-066 - Album Cover Match Game - Employing Game Logic

The next thing we need to focus on is the animation and the events surrounding the countdown of the ProgressBar, and what happens when we get all the way to the end of the countdown. For that, we will need to fire off an event that we can handle in our code, and then decide what should happen when that event fires. There are really two scenarios where we need to worry about the countdown. First, there will be a cooling-off period between rounds, and then within the round itself we'll be counting down to determine the score for the player if he makes a selection at a given moment in time.

So, to get the animation to work we will first have to add a Storyboard to the Page's resources (we will delve deeper into StoryBoards, Animations and Keyframes in the coming lessons). Get started with the StoryBoard by adding this to MainPage.xaml

```
mc:Ignorable="d">

<Page.Resources>
    <Storyboard x:Name="CountDown" Completed="CountDown_Completed">
        ...
    </Storyboard>
</Page.Resources>
```

The part that say "x:Name" allows you to attach a name to a Storyboard (in this case we're calling it the "CountDown" storyboard). This will allow us to programmatically access the StoryBoard, by referencing it by name, and allow us to handle the completed event.

Step 1: There are different types of animations that you can add. In this particular case, you will want a DoubleAnimationUsingKeyframes

```
<Page.Resources>
    <Storyboard x:Name="CountDown" Completed="CountDown_Completed">
        <DoubleAnimationUsingKeyFrames>
            ...
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</Page.Resources>
```

To preface here a bit, “double” is the data type, and using numeric keyframes allows you to identify key moments in the life of the animation where key values of given properties are changed.

Step 2: Give the StoryBoard a TargetName and EnableDependentAnimation. And typically, when you have a ProgressBar you have a maximum and minimum in your progress range, so set all of that with the following modifications

```

<DoubleAnimationUsingKeyFrames EnableDependentAnimation="True"
    Storyboard.TargetName="MyProgressBar"
    Storyboard.TargetProperty="(RangeBase.Value)">

    </DoubleAnimationUsingKeyFrames>

```

And now add a DiscreteDoubleKeyframe with the key time, to start, at zero seconds while the value of the ProgressBar will be 100 (keep in mind the pattern as the KeyTime increases)

```

<Page.Resources>
    <Storyboard x:Name="CountDown" Completed="CountDown_Completed">
        <DoubleAnimationUsingKeyFrames EnableDependentAnimation="True"
            Storyboard.TargetName="MyProgressBar"
            Storyboard.TargetProperty="(RangeBase.Value)">
            <DiscreteDoubleKeyFrame KeyTime="0" Value="100" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:1" Value="100" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:2" Value="90" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:3" Value="80" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="70" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:5" Value="60" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:6" Value="50" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:7" Value="40" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:8" Value="30" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:9" Value="20" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:10" Value="10" />
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</Page.Resources>

```

Step 3: Now create a helper method in MainPage.xaml.cs

```

private void StartCooldown()
{
    CountDown.Begin();
}

```

And reference it within Grid\_Loaded()

```

private async void Grid_Loaded(object sender, RoutedEventArgs e)
{
    StartupProgressRing.IsActive = true;

    AllSongs = await SetupMusicList();
    await PrepareNewGame();

    StartupProgressRing.IsActive = false;
    StartCooldown();
}

```

Now, run the program and you should see a countdown bar when it first loads.

Step 4: Now, we need to determine what should happen once the countdown is completed. For that, we turn to the CountDown\_Completed() method. But first, remember that there are two states that the program can be in when counting down: before a round starts, and during a round. Let's handle that state by adding a bool at the class level

```
private ObservableCollection<StorageFile> AllSongs;  
  
bool _playingMusic = false;
```

And in the CountDown\_Completed() method

```
private void CountDown_Completed(object sender, object e)  
{  
    if (!_playingMusic)  
    {  
        // Start playing music  
  
        // Start countdown  
    }  
}
```

Then, we will need to keep track of what round it is, so add an int to the class level

```
int _round = 0;
```

And reference it, along with adding bunch of code to StartCoolDown()

```
private void StartCooldown()  
{  
    _playingMusic = false;  
    SolidColorBrush brush = new SolidColorBrush(Colors.Blue);  
    MyProgressBar.Foreground = brush;  
    InstructionTextBlock.Text = string.Format("Get ready for round {0} ...", _round + 1);  
    InstructionTextBlock.Foreground = brush;  
    CountDown.Begin();  
}
```

Call StartCountDown() within CountDown\_Completed()

```
// Start countdown  
StartCooldown();  
}
```

And we will want to also call StartCoolDown() whenever an item is clicked, and move to the next round as well

```
private void SongGridView_ItemClick(object sender, ItemEventArgs e)  
{  
    // Evaluate the user's selection  
  
    _round++;  
    StartCooldown();  
}
```

Step 5: We would like to have a list of unused songs, songs that we've not touched yet that are still available on that list of songs we initially picked out. So of all the songs, we want all the songs where the "used" property is "false" so we need to add another property to the Song class called "Used."

```
public class Song
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Artist { get; set; }
    public string Album { get; set; }
    public StorageFile SongFile { get; set; }
    public bool Selected { get; set; }
    public bool Used { get; set; }

    public BitmapImage AlbumCover;
}
```



We then write a new helper method called PickSong() in which this bool is then referenced

```
private Song PickSong()
{
    Random random = new Random();
    var unusedSongs = Songs.Where(p => p.Used == false);
    var randomNumber = random.Next(unusedSongs.Count());
    var randomSong = unusedSongs.ElementAt(randomNumber);
    randomSong.Selected = true;
    return randomSong;
}
```

And in CountDown\_Completed make the following changes

```
private void CountDown_Completed(object sender, object e)
{
    if (!_playingMusic)
    {
        // Start playing music
        var song = PickSong();



        MyMediaElement.SetSource(
            await song.SongFile.OpenAsync(FileAccessMode.Read),
            song.SongFile.ContentType);

        // Start countdown
        StartCountdown();
    }
}
```

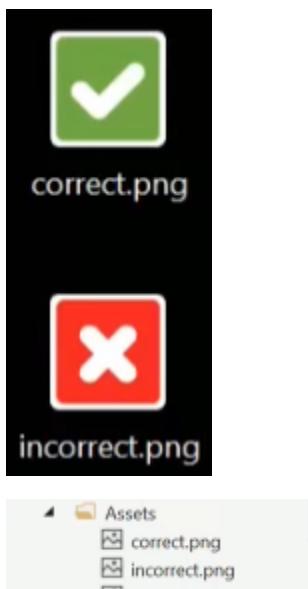
With MyMediaElement.SetSource() we want to set it to the song, while the Song class contains the song file, of type StorageFile. And in order to call this SetSource() method we need to give it two things. First of all, we need to give it the stream from the file itself, and then we need to give it the content (MIME) type.

When you run the program it should now play a song after the indicator counts down all the way. Now we will need to determine the behavior after an item is clicked.

## UWP-067 - Album Match Game - User Input & Tracking Progress

The next thing do is evaluate what the user clicks on, which will be handled in the SongGridView\_ItemClick() method. When the song is playing, they're going to choose an album cover. That click then needs to be evaluated to determine whether or the correct song was selected. If correct, then award the user some points, and display a "correct" symbol in place of that album art. However, if it's not correct, deduct points from the user score and display the "incorrect" icon.

The icons used for the correct/incorrect designation are available in the source code, or you can simply use your own. Be sure to copy the icons to the assets folder and correctly label them.



Step 1: Write the logic in the SongGridView\_ItemClick()

```

private void SongGridView_ItemClick(object sender, ItemClickEventArgs e)
{
    // Ignore clicks when we are in cooldown
    if (!_playingMusic) return;

    CountDown.Pause();
    MyMediaElement.Stop();

    var clickedSong = (Song)e.ClickedItem;

    // Find the correct song
    //var correctSong = Songs.FirstOrDefault(p => p.Selected == true);

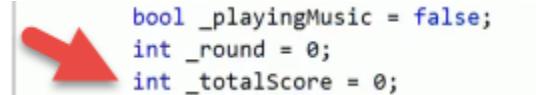
    // Evaluate the user's selection

    Uri uri;
    if (clickedSong.Selected)
    {
        uri = new Uri("ms-appx:///Assets/correct.png");
    }
    else
    {
        uri = new Uri("ms-appx:///Assets/incorrect.png");
    }
    StorageFile file = await StorageFile.GetFileFromApplicationUriAsync(uri);
    var fileStream = await file.OpenAsync(FileAccessMode.Read);
    await clickedSong.AlbumCover.SetSourceAsync(fileStream);

    _round++;
    StartCooldown();
}

```

Step 2: Add `_totalScore` to the class level



```

bool _playingMusic = false;
int _round = 0;
int _totalScore = 0;

```

And either subtract or add to the score, depending on if the selection was correct

```
Uri uri;
int score;
if (clickedSong.Selected)
{
    uri = new Uri("ms-appx:///Assets/correct.png");
    score = (int)MyProgressBar.Value;
}
else
{
    uri = new Uri("ms-appx:///Assets/incorrect.png");
    score = ((int)MyProgressBar.Value) * -1;
}
```

And add that to the \_totalScore as follows

```
_totalScore += score;
_round++;
```

Step 3: Sort through to find the correct song for this round

```
var clickedSong = (Song)e.ClickedItem;
var correctSong = Songs.FirstOrDefault(p => p.Selected == true);
```

And report the results with the following addition to SongGridView\_ItemClick()

```
_round++;

ResultTextBlock.Text = string.Format("Score: {0} Total Score after {1} Rounds: {2}", score, _round, _totalScore);
TitleTextBlock.Text = String.Format("Correct Song: {0}", correctSong.Title);
ArtistTextBlock.Text = string.Format("Performed by: {0}", correctSong.Artist);
AlbumTextBlock.Text = string.Format("On Album: {0}", correctSong.Album);

StartCooldown();
```

Step 4: And then we finalize this bit of logic with resetting the states for the next round, making sure the selected song doesn't play again

```
clickedSong.Used = true;

correctSong.Selected = false;
correctSong.Used = true;

StartCooldown();
```

And finally, we'll make sure that the rounds only run up to five, and if the round is greater than or equal to five, then have some end-game cleanup happen. Otherwise, we'll want to start the cooldown process all over again for the next round

```
correctSong.Used = true;

if (_round >= 5)
{
    InstructionTextBlock.Text = string.Format("Game over ... You scored: {0}", _totalScore);
    PlayAgainButton.Visibility = Visibility.Visible;
} else
{
    StartCooldown();
}
```

## UWP-068 - Album Cover Match Game - Enabling the Play Again Feature

Now with the core functionality out of the way let's focus next on what happens when we get to the end of the game. We could, for instance, present the user with a "play again" button, and when the users clicks it, we can essentially reset everything back to the beginning state. Start by writing the following within the PlayAgainButton\_Click() method

```
private async void PlayAgainButton_Click(object sender, RoutedEventArgs e)
{
    await PrepareNewGame();

    PlayAgainButton.Visibility = Visibility.Collapsed;
}
```

And turn next to adding to the state management logic in PrepareNewGame()

```
// Choose random songs from library
var randomSongs = await PickRandomSongs(AllSongs);

// Pluck off meta data from selected songs
await PopulateSongList(randomSongs);

StartCooldown();

// State management
InstructionTextBlock.Text = "Get ready ...";
ResultTextBlock.Text = "";
TitleTextBlock.Text = "";
ArtistTextBlock.Text = "";
AlbumTextBlock.Text = "";

_totalScore = 0;
_round = 0;
```

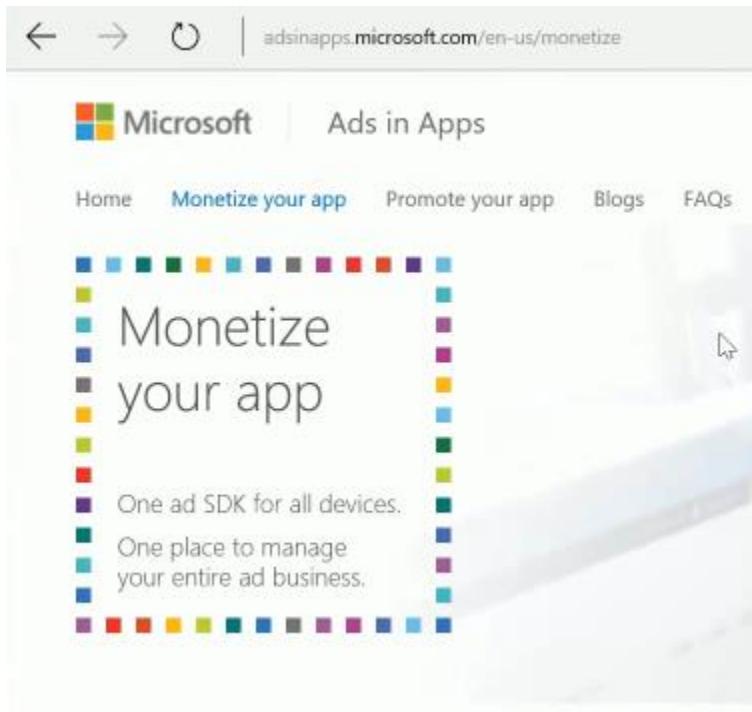
The next step left for us now is to add some new functionality that allows us to monetize this game.

## UWP-069 - Album Cover Match Game - Monetizing with Ads

The next step towards completing this app is to find ways to monetize it. There are two obvious ways to go about doing this. One is with advertisements and the other - for those people that really hate advertisements – will be an upgrade to a "pro" version that removes the ads.

Have a look at the ad SDK found at

<http://bit.do/ad-sdk>



And install the SDK by clicking on the link



And download this extension for Visual Studio (note: make sure Visual Studio is not running during the install)

Extensions > Controls > Windows ad mediation

## Windows ad mediation

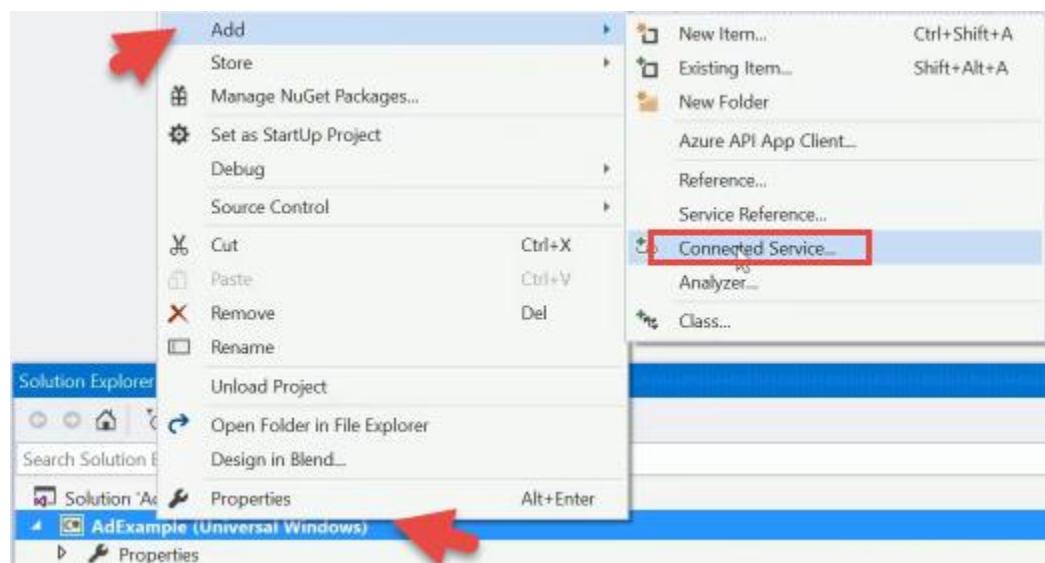
Microsoft

The ad mediator aims to maximize your ad monetization by making sure you are always showing a live ad. This extension enables you to plug and play 3rd party ad network solutions into your project.

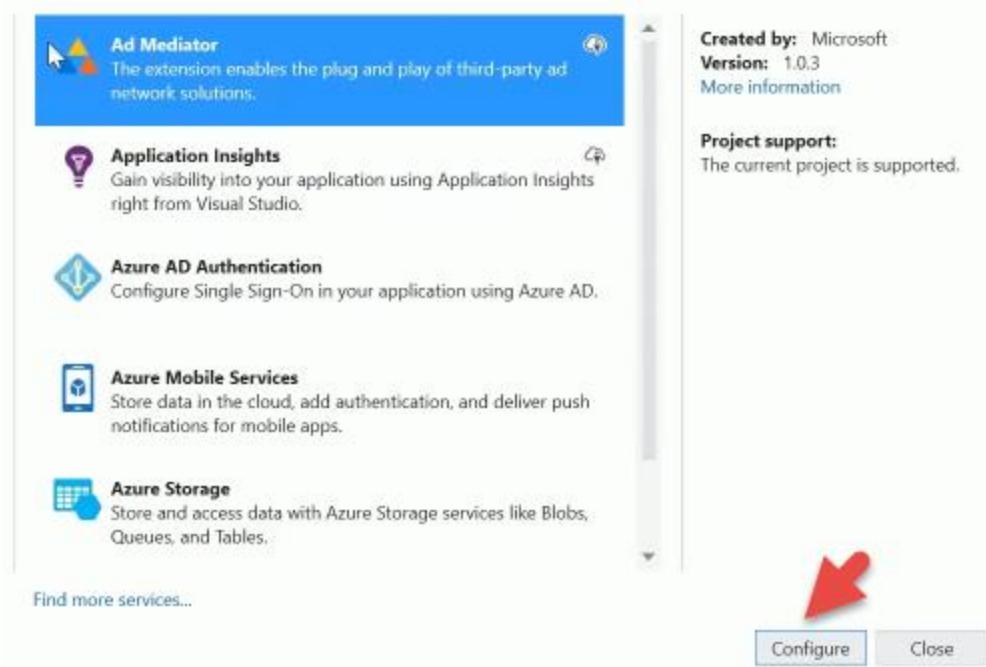
CREATED BY	Microsoft	UPDATED	8/13/2015
REVIEWS	★ ★ ★ ★ (48) <a href="#">Review</a>	VERSION	2.0.0
SUPPORTS	Visual Studio 2015, 2013	SHARE	<a href="#">Email</a> <a href="#">Twitter</a> <a href="#">Facebook</a> <a href="#">LinkedIn</a>
DOWNLOADS	<a href="#">Download</a> (15,329)	FAVORITES	<a href="#">Add to favorites</a>
TAGS	windows 8.1, Windows Phone Ad Control, WP8 Ads, WP8.1 Ads, Windows Phone 8 SDK, WP8 Ads SDK, Windows Phone 8.1 SDK, WP8.1 Ads SDK, Ad Mediator, Windows 10		

Step 1: To demonstrate how the ad mediation extension works, let's create a new blank app and call it "AdExample." When that is open, right click on the project in the Solution Explorer and select

Add > Connected Service...



And then select "Ad Mediator," and click on "Configure"



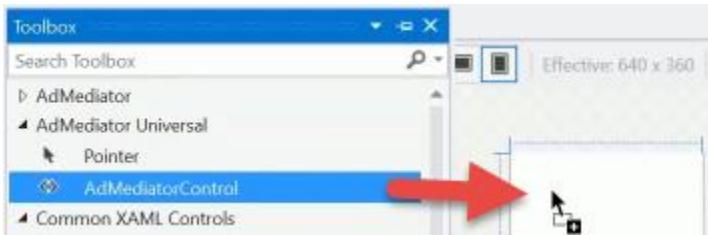
And then in the Ad Mediator dialogue select “Microsoft Advertising” and click “Add”

This screenshot shows the "Ad Mediator" configuration dialog. At the top, there are three columns: "Name", "References fetched", and "Required Capabilities". A single row is listed: "Microsoft Advertising" (highlighted with a blue background), "Not Fetched", and "Internet (Client)". Below this is a "Refresh" button. At the bottom is a control bar with "Add" (highlighted with a blue background) and "Cancel" buttons.

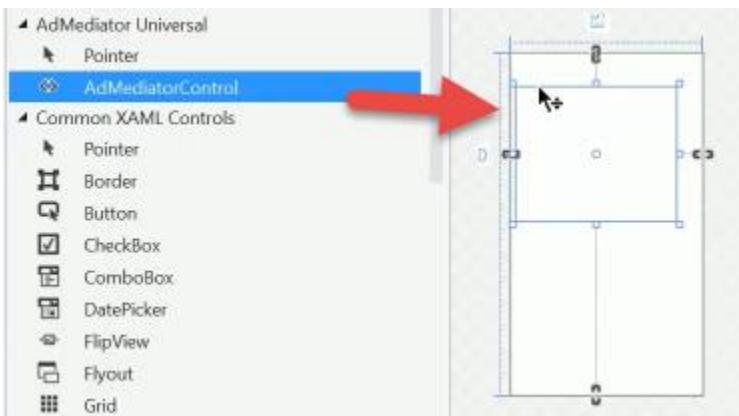
Step 2: In the Toolbox you will have an option to select

AdMediator Universal > AdMediatorControl

Click and drag that onto your MainPage.xaml Designer and click “OK” when the message displays notifying that the reference has been added to the project.



To make sure there are no errors, you can build the solution now, and then simply repeat the process to see the Control added to the Designer



Notice in MainPage.xaml how this adds the following namespace and Control (note that the x:Name and Id for your control will have your own unique label)

```


    xmlns:Universal="using:Microsoft.AdMediator.Universal"
    x:Class="AdExample.MainPage"
    mc:Ignorable="d">


```

```

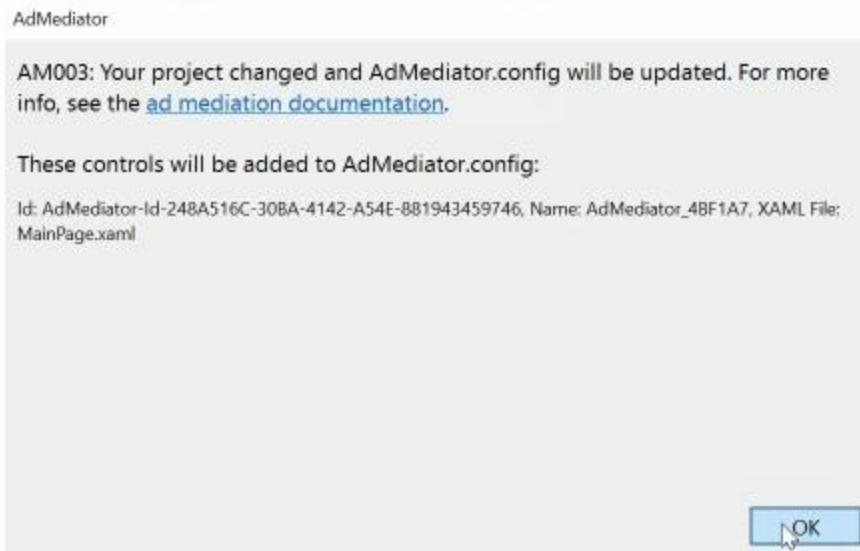

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Universal:AdMediatorControl
            x:Name="AdMediator_4BF1A7"
            HorizontalAlignment="Left"
            Height="250"
            Id="AdMediator-Id-248A516C-30BA-4142-A54E-881943459746"
            Margin="10,63,0,0"
            VerticalAlignment="Top"
            Width="300"/>
    </Grid>


```

Step 3: Once again go to

Add > Connected Service...

And click "OK" to add the following to your project



And now you shoud see “Fetched” under “References fetched.” Click “Add” for that as well and build the solution to make sure everything is set up correctly.

Name	References fetched	Required Capabilities
Microsoft Advertising	Fetched	Internet (Client)

Step 4: Now, go to your dashboard at dev.windows.com, create a new entry for your app and click on “Submissions” and click on your Submission in progress

Ad Example

## App overview

App overview

Analytics ▾

Submissions

IAPs

Monetization ▾

### Submissions

Submission	Status
Submission 1	In progress

And now make a change to the App properties and in the next page select the category/sub category for your app (it doesn't matter what you select, so long as you select something) and save that change.

## Pricing and availability

### App properties

And now click on “App overview” from the menu on the left



### Ad Example

#### App overview

And now you should be able to start ad configuration

### Ad mediation

[See full report](#)

Ad mediation makes it easy to include multiple ad networks in your app and specify when and how each ad network should be called. Mediation performance reports help you improve the ad network configuration for your app and increase your ad revenue.

[Start configuring now](#)



Now go back to “App overview,” and if you scroll down you’ll see something for Ad mediation, and you should be able to start configuring ad mediation now. You can choose what ad networks that you’re going to serve up, and you can even create Microsoft Advertising Ad Units, which is something that you’re going to need to do whenever you go to submit your application. Below you can see an example of an already created ad unit name, which then receives an application ID (which in this case will be a banner ad for the PC/Tablet)

## Microsoft advertising ad units

[Hide options](#)

Fill in the below fields to generate a new ad unit. Your existing ad units are displayed below.

Ad unit name

Ad unit type

▼

Device family

▼

[Create ad unit](#)

Ad unit name

Application ID

Ad unit ID

Ad unit type

Device family

AdExampleUnit

d10775a0-9649-42e3-b886-3ca1f9f84c73

244179

Banner

PC/Tablet

Step 5: Now go to the Windows Dev Center and find the link that says “Select and manage your ad networks”

<http://bit.do/ad-networks>

- ▷ Windows apps
  - ▷ Develop
  - ▷ How-to guides for Windows 10 apps
  - ▷ Monetize your app
    - ☛ Use ad mediation to maximize revenue
- [Select and manage your ad networks](#)

And you will see helpful information for Microsoft Advertising, including a list of currently supported Ad Networks (this list will be updated as platforms become newly supported). Note that, at this time, the Universal Windows Platform only supports Microsoft Advertising.

## Supported ad networks

Currently, the following ad networks are supported for each platform:

	Universal Windows Platform (UWP)	Windows 8.1 XAML	Windows Phone 8.1 XAML	Windows Phone 8.1 Silverlight	Windows Phone 8 Silverlight
<a href="#">Microsoft Advertising</a>	<b>Supported</b>	<b>Supported</b>	<b>Supported</b>	<b>Supported</b>	<b>Supported</b>
<a href="#">AdDuplex</a>	Not supported	<b>Supported</b>	<b>Supported</b>	<b>Supported</b>	<b>Supported</b>
<a href="#">Smaato</a>	Not supported	<b>Supported</b>	<b>Supported</b>	<b>Supported</b>	<b>Supported</b>
<a href="#">AdMob (Google)</a>	Not supported	Not supported	Not supported	<b>Supported</b>	<b>Supported</b>
<a href="#">MobFox</a>	Not supported	Not supported	Not supported	<b>Supported</b>	<b>Supported</b>
<a href="#">InMobi</a>	Not supported	Not supported	Not supported	<b>Supported</b>	<b>Supported</b>
<a href="#">Inneractive</a>	Not supported	Not supported	Not supported	<b>Supported</b>	<b>Supported</b>

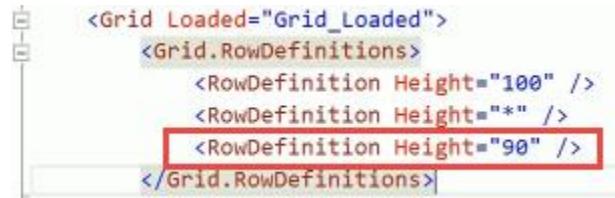
Now, find the link that says “Add and use the ad mediation control”

- Windows apps
- Develop
- How-to guides for Windows 10 apps
- Monetize your app
  - ☛ Use ad mediation to maximize revenue
    - Select and manage your ad networks
    - [Add and use the ad mediation control](#)

And take note of the supported ad sizes shown here, with the one we will be using selected (728 x 90)

- UWP and Windows 8.1 XAML:
  - 250 x 250
  - 300 x 250
  - **728 x 90**
  - 160 x 600
  - 300 x 600

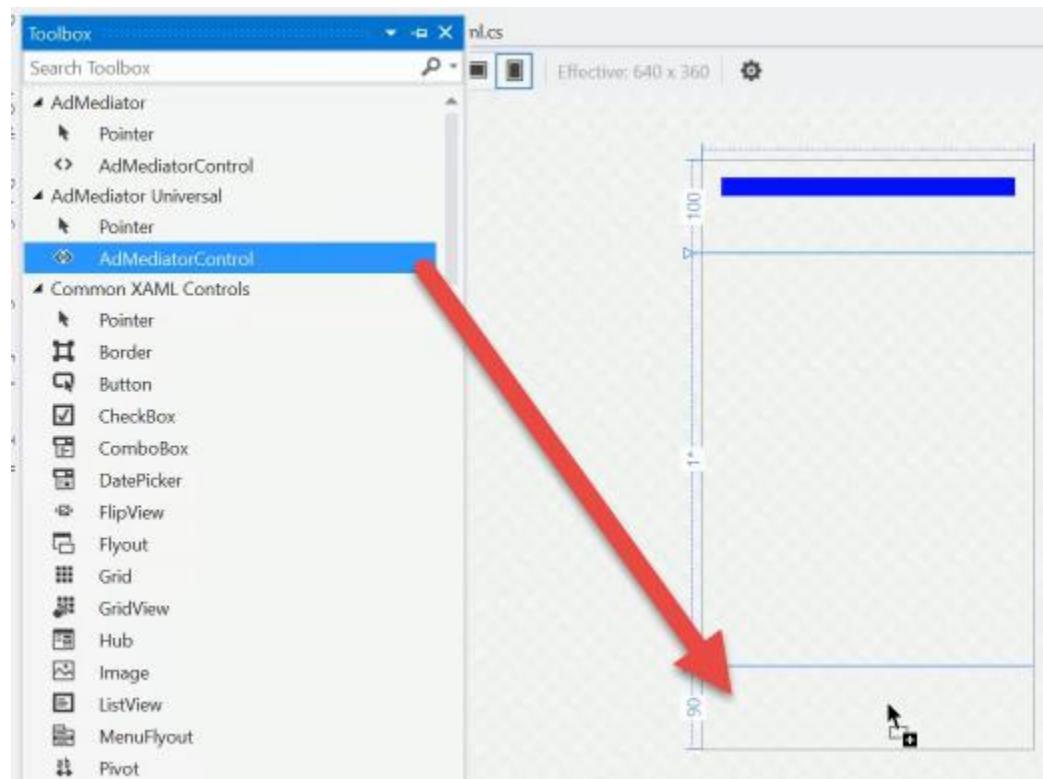
So, now that we know how to get ads going, let's apply this to the "Album Cover Match Game" project and open up MainPage.xaml, making these changes for where we intend to place the ad in our app



And repeat process of Connecting the Ad Mediator service found in Step 1: starting with right clicking on the project in the Solution Explorer and selecting

Add > Connected Service...

And then also go through the same process detailed in Step 2. During this step, be sure to drag and drop the AdMediatorControl to the area in our app that we want the ad to appear



Also revisit Step 3 and complete that process for this project (updating the AdMediator.config)

Going back now to MainPage.xaml, make the following edits to the Universal:AdMediatorControl (get rid of the margins and change the height 90 and the width to 728). Remember also to ignore the x:Name and Id shown here (yours will be unique)

```
<Universal:AdMediatorControl x:Name="AdMediator_FE5AFA"
    HorizontalAlignment="Left"
    Height="90"
    Id="AdMediator-Id-0C6B2726-
    Grid.Row="2"
    VerticalAlignment="Top"
    Width="728"/>
```

And now if you run the program you should see a blue box at the bottom of the app, and when you click it, it should take you to msn.com

Note that you may experience a bug which deletes some of your data bindings, so just make sure that they are still present as shown below

```
<StackPanel Grid.Row="1" Orientation="Vertical" Margin="20">
    <GridView Name="SongGridView"
        ItemsSource="{x:Bind Songs}"
        IsItemClickEnabled="True"
        ItemClick="SongGridView_ItemClick">
        <GridView.ItemTemplate>
            <DataTemplate x:DataType="data:Song">
                <Grid>
                    <Image Name="AlbumArtImage"
                        Height="75"
                        Width="75"
                        Source="{x:Bind AlbumCover}" />
                </Grid>
            </DataTemplate>
        </GridView.ItemTemplate>
    </GridView>
```

## UWP-070 - Album Cover Match Game - In App Purchase for Ad Removal

Next we will look at implementing in-app purchases (in this case, it will be one that allows the user to remove ads). For more information on this, turn your attention to the site detailing this, found at

<http://bit.do/in-app-purchase>

You should be forewarned that there are a few things that you're going to have to do during development time that you're going to have to change right before you submit your application to the Microsoft Store. One of the things that we're going to do is work with something called a "Current App Simulator." Once we have finished debugging our application - and our application is feature complete - then we will change all references to the "Current App Simulator" to simply "Current App."

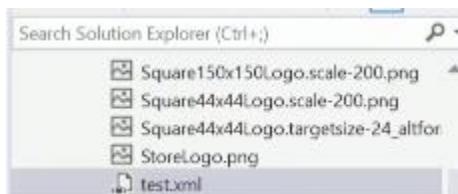
From the above URL:

**Note** When you code and test new in-app products for the first time, you must use the **CurrentAppSimulator** object instead of the **CurrentApp** object. This way you can verify your license logic using simulated calls to the license server instead of calling the live server. To do this, you need to customize the file named "WindowsStoreProxy.xml" in %userprofile%\AppData\local\packages\<package name>\LocalState\Microsoft\Windows Store\ApiData. The Microsoft Visual Studio simulator creates this file when you run your app for the first time—or you can also load a custom one at runtime. For more info, see the **CurrentAppSimulator** docs.

Note the file referenced above called "WindowsStoreProxy.xml" which is a local file that keeps track of your in-app purchases for this specific application. This file is created by the Visual Studio simulator and you will have to customize this file, so first locate it at:

`%userprofile%\AppData\local\packages\<package name>\LocalState\Microsoft\Windows Store\ApiData`

Step 1: Copy and paste the contents of that file to an XML file stored in your projects Assets folder for further editing. Call that XML file "test.xml"



Step 2: In MainPage.xaml add this Visibility reference to the AdMediatorControl and add a Button as well



Step 3: Modify the text.xml, with the following changes



Step 4: Add some using statements to the top of MainPage.xaml.cs

```
using Windows.ApplicationModel;  
using Windows.ApplicationModel.Store;
```

And add this property to MainPage

```
public sealed partial class MainPage : Page
{
    public LicenseInformation AppLicenseInformation { get; set; }
```

Step 5: Add the following code to Page\_Loaded(), noting the references to the app license information (and remember to change all references to “CurrentAppSimulator” to “CurrentApp” when you are ready to publish with the build)

```
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    //+++++
    // Remove these lines of code before publishing!
    // The actual CurrentApp will create a WindowsStoreProxy.xml
    // in the package's \LocalState\Microsoft\Windows Store\ApiData
    // folder where it stores the actual purchases.
    // Here we're just giving it a fake version of that file
    // for testing.
    StorageFolder proxyDataFolder = await Package.Current.InstalledLocation.GetFolderAsync("Assets");
    StorageFile proxyFile = await proxyDataFolder.GetFileAsync("test.xml");
    await CurrentAppSimulator.ReloadSimulatorAsync(proxyFile);
    //++++

    // You may want to put this at the App level
    AppLicenseInformation = CurrentAppSimulator.LicenseInformation;

    if (AppLicenseInformation.ProductLicenses["RemoveAdsOffer"].IsActive)
    {
        // Customer can access this feature. 
    }
    else
    {
        // Customer can NOT access this feature. 
    }
}
```

Step 6: Grab your AdMediatorControl x:Name (yours will be different)

```
<Universal:AdMediatorControl
    x:Name="AdMediator_40F141"
```

And reference it in the following conditional block, along with specifying the visibility of the purchase button depending on if the purchase has been made

```

if (AppLicenseInformation.ProductLicenses["RemoveAdsOffer"].IsActive)
{
    // Customer can access this feature.
    AdMediator_40F141.Visibility = Visibility.Collapsed;
    PurchaseButton.Visibility = Visibility.Collapsed;
}
else
{
    // Customer can NOT access this feature.
    AdMediator_40F141.Visibility = Visibility.Visible;
    PurchaseButton.Visibility = Visibility.Visible;
}

```



Step 7: Modify PurchaseButton\_Click() as follows

```

private async void PurchaseButton_Click(object sender, RoutedEventArgs e)
{
    if (!AppLicenseInformation.ProductLicenses["MyInAppOfferToken"].IsActive)
    {
        try
        {
            // The customer doesn't own this feature, so
            // show the purchase dialog.

            PurchaseResults results = await CurrentAppSimulator.RequestProductPurchaseAsync("RemoveAdsOffer");

            //Check the license state to determine if the in-app purchase was successful.

            if (results.Status == ProductPurchaseStatus.Succeeded)
            {
                AdMediator_40F141.Visibility = Visibility.Collapsed;
                PurchaseButton.Visibility = Visibility.Visible;
            }
        }
        catch (Exception ex)
        {
            // The in-app purchase was not completed because
            // an error occurred.
            throw ex;
        }
    }
    else
    {
        // The customer already owns this feature.
    }
}

```

Note, once again, that the AdMediator code will be unique to you

```

if (results.Status == ProductPurchaseStatus.Succeeded)
{
    AdMediator_40F141.Visibility = Visibility.Collapsed;
    PurchaseButton.Visibility = Visibility.Collapsed;
}

```



Step 8: Fix up some margins and other issues, starting with removing this margin for the AdMediatorControl (again, making sure not to modify your own unique x:Name, and Id)

```
<Universal:AdMediatorControl  
    x:Name="AdMediator_40F141"  
    HorizontalAlignment="Left"  
    Height="90"  
    Id="AdMediator-Id-610899D2-8A2D-422A-9F28-0B1412FD6EA4"  
    Grid.Row="2"  
    VerticalAlignment="Top"  
    Visibility="Collapsed"  
    Width="728"  
    Margin="10" />
```

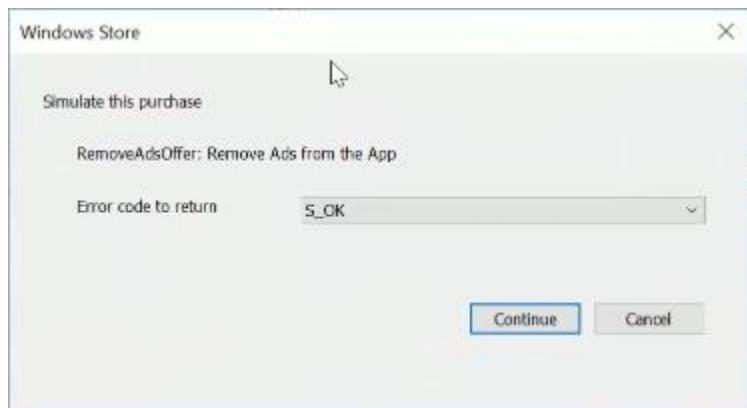
Grid Loaded

```
<Grid Loaded="Grid_Loaded">  
    <Grid.RowDefinitions>  
        <RowDefinition Height="100" />  
        <RowDefinition Height="*" />  
        <RowDefinition Height="110" />  
    </Grid.RowDefinitions>
```

And the Button Grid

```
<Button Grid.Row="2"  
        Name="PurchaseButton"  
        Content="Remove Ads"  
        Click="PurchaseButton_Click"  
        VerticalAlignment="Top"  
        HorizontalAlignment="Right" Margin="10" />
```

Now, when you run the program you will be able to simulate different conditions in your Simulated App environment (below, we're simulating that the submission occurred with no errors). You should also now be able to simulate an in-app purchase to remove the advertisement box.



## UWP-071 - Hero Explorer - Introduction

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

The next, and final, project is probably the coolest and the most fun to work on. We're building an application called the "Hero Explorer," and with the kind blessing of Marvel we'll have access to their API, allowing you to pull data on characters from the Marvel database of over 1500 characters.

With this app we're going to be able to select one of the characters by clicking on a master list on the left-hand side and view the details about that character on the right-hand side.

While producing this app we'll learn how to place calls to a public web API, interpret the documentation, create classes based on JSON examples in the documentation, and then deserialize live data that we request into those classes.

We'll learn how to create and utilize some of the adaptive layout aspects, and design patterns, that we've been using earlier in this series of lessons. There will also be a lot on data binding and working through real world issues that you're likely to experience when building a similar style application.

Also, when we are working with the Marvel API, we will address some security issues. In specific, we'll learn how to create a cache using our (public and private) API key, along with a timestamp value, to ensure the request authenticates, and employ special cryptography classes in the Universal Windows Platform accordingly. And, if that wasn't enough, at the end we will also integrate our application with Cortana so that we can learn how to control the app with voice recognition.

## UWP-072 - Hero Explorer - Accessing the Marvel Web API

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

The first thing that we will run into is we will need to make sure that we know how to make calls out to the Marvel API and get information back. Once that's out of the way we'll figure out how to deserialize anything we get back into instances of classes, that we can then work with inside of our application.

But first, before we begin, you're going to need to sign up for a developer account at the URL

**[developer.marvel.com](https://developer.marvel.com)**

Once there click on the "Get Started" icon, which will walk you through the simple process of creating your account. (Note: although we're using this API for demonstration purposes, if you at any point decide on using the API in a commercial project be sure to first look over Marvel's monetizing restrictions detailed on the site)

Also take a look at the available documentation for the API found at

**[developer.marvel.com/docs](https://developer.marvel.com/docs)**

This will give you a lot of helpful information on how to retrieve, for instance, a list of characters, a single character, or get all the comics for a single character, or all the events, or series, or stories for a single character, etc. If you click on one of the documentation links it will contain implementation notes, the data that is returned (organized into a class-like structure), and there is even a test area that lets you filter information based on various conditions. We will be delving into this later.

Also take note when you go to your developer account, you're going to be given a public key and a private key. And any time that you make a call to the API, you're going to need to pass along either just the public key, as you saw we did a moment ago, or, some combination of public and private key hashed together.

Now if you're using this in a web application, you need to list the URLs that you plan on calling the API from.

In this case, "developer.marvel.com" is one place we are able to use our public key to make a call into and grab data from the Marvel API.

However, if you go to the menu on the site and choose

**How-Tos > Authorization**

You'll learn that this only works for client side applications. If you were to create what's called a Server-Side Application, you're going to need to supply a little bit more information.

In fact, what you're going to have to do is provide a hash. A hash is basically an algorithmic computation based on a public key and a private key that's known to only two parties. So, basically, there are many different hash algorithms that are available (md5 will be used in this case) but the stronger the algorithm, the more computational power it requires to actually calculate the hash.

But essentially what this is going to do is ensure that nobody has tampered with the request from the client to the server. It will be appended at the end of every call that we will make into the Marvel API, and we just have to figure out how to do that in C# (fortunately, there are classes in the Universal Windows Platform that will help us to accomplish that).

Step 1 : Use the filtering tool get some preliminary data that we can incorporate into a project

To start we will just want to select from a range of characters retrieved at an offset in the API. You can test this out by going to the "docs" section of the site and filtering as follows (this is basically asking to return 2 entries starting at entry #350 in the database, with the name "Spider-man")

The resulting data returned can then be fed into the JSON-to-C# converter as we had used previously. Go ahead and paste the data into the form, and clicking generate, at the URL

[Json2charp.com](http://Json2charp.com)

It will look something like this (partially represented below)

```
public class Thumbnail
{
    public string path { get; set; }
    public string extension { get; set; }
}

public class Item
{
    public object resourceURI { get; set; }
    public string name { get; set; }
}

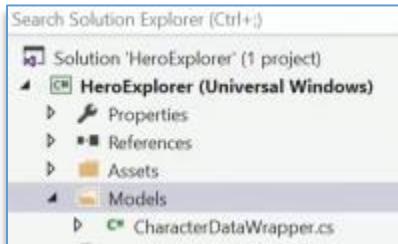
public class Comics
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public List<Item> items { get; set; }
    public int returned { get; set; }
}

public class Item2
{
    public string resourceURI { get; set; }
    public string name { get; set; }
}
```

Be aware that some of the data originally returned from the Marvel API may be missing or omitted, and result in a JSON-to-C# conversion that may have some meaningless class information. It might just be a generic Object reference, for example, so be on the look-out for that.

So, with that converted class information in your clipboard go ahead and open a new UWP project in Visual Studio, calling it “HeroExplorer”

Step 2: Add a new folder to the Solution Explorer called “Models” and in that folder create a new class called “CharacterDataWrapper”



And inside of that class paste in the JSON-to-C# data from your clipboard (partially represented below)

```
namespace HeroExplorer.Models
{
    public class Thumbnail
    {
        public string path { get; set; }
        public string extension { get; set; }
    }

    public class Item
    {
        public object resourceURI { get; set; }
        public string name { get; set; }
    }

    public class Comics
    {
        public int available { get; set; }
        public string collectionURI { get; set; }
    }
}
```

Step 3: Now, remember how we should be on the lookout for generic information, let's make some changes to anything that may seem to fit this pattern

```
public class Item
{
    public object resourceURI { get; set; }
    public string name { get; set; }
}

public class Comics
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public List<Item> items { get; set; }
    public int returned { get; set; }
}
```

Here you can see that "Item" has a single instance in the Comics class. So, let's rename this to "Comic" so the Comics class will have a List of items of type Comic.

```
public class Comic
{
    public object resourceURI { get; set; }
    public string name { get; set; }
}

public class Comics
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public List<Comic> items { get; set; }
    public int returned { get; set; }
}
```

Using the same reasoning, modify the "Item2" and "Series" classes as follows (change "Item2" to "Series," and to original class named "Series" to "SeriesList")

```
public class Series
{
    public string resourceURI { get; set; }
    public string name { get; set; }
}

public class SeriesList
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public List<Series> items { get; set; }
    public int returned { get; set; }
}
```

And, change “Item3” to “Story”

```
public class Story
{
    public string resourceURI { get; set; }
    public string name { get; set; }
    public string type { get; set; }
}

public class Stories
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public List<Story> items { get; set; }
    public int returned { get; set; }
}
```

And, change “Item4” to “Event”

```
public class Event
{
    public string resourceURI { get; set; }
    public string name { get; set; }
}

public class Events
{
    public int available { get; set; }
    public string collectionURI { get; set; }
    public List<Event> items { get; set; }
    public int returned { get; set; }
}
```

Also change the “RootObject” to “CharacterDataWrapper”

```
public class CharacterDataWrapper
{
    public int code { get; set; }
    public string status { get; set; }
    public string copyright { get; set; }
    public string attributionText { get; set; }
    public string attributionHTML { get; set; }
    public string etag { get; set; }
    public Data data { get; set; }
}
```

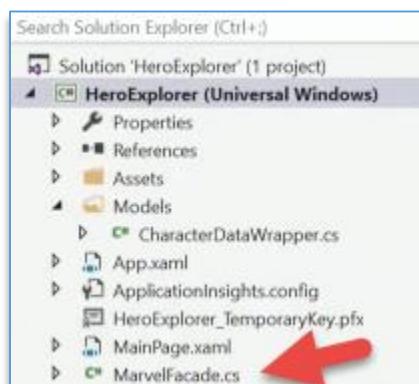
And, change “Data” to “CharacterDataContainer,” along with all references to this class

```
public class CharacterDataContainer
{
    public int offset { get; set; }
    public int limit { get; set; }
    public int total { get; set; }
    public int count { get; set; }
    public List<Result> results { get; set; }
}
```

Also change “Result,” and all references to it, to “Character”

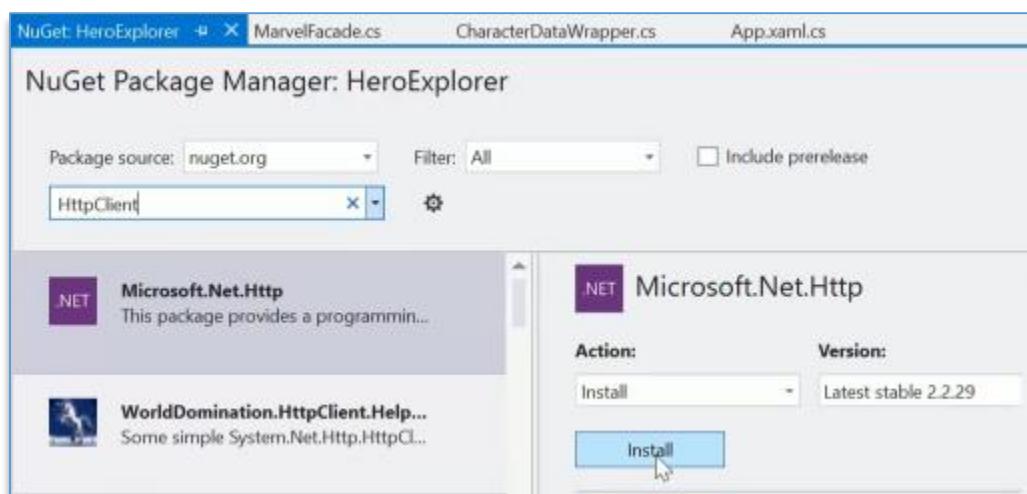
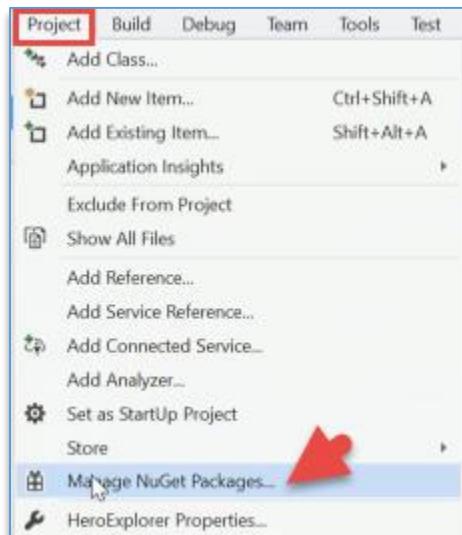
```
public class Character
{
    public int id { get; set; }
    public string name { get; set; }
    public string description { get; set; }
    public string modified { get; set; }
    public Thumbnail thumbnail { get; set; }
    public string resourceURI { get; set; }
    public Comics comics { get; set; }
    public Series series { get; set; }
    public Stories stories { get; set; }
    public Events events { get; set; }
    public List<Url> urls { get; set; }
}
```

Step 4: Create a class called “MarvelFacade” that will handle access to the Marvel API programmatically. It will be delegated the responsibility of going out and retrieving results, parsing through them and putting them into our Model class.



And, before we write any code in the class, let's add the HttpClient. Go to

**Project > Manage NuGet Packages...**



And in the MarvelFacade class, set it up as follows. This method will use the HttpClient to retrieve data from the Marvel API and will also contain the MD5 hash needed to pass along with the public/private keys, along with the time-stamp.

```
namespace HeroExplorer
{
    public class MarvelFacade
    {
        public static void GetCharacterList()
        {

        }
    }
}
```

## UWP-073 - Hero Explorer - Creating an MD5 Hash and Calling the API

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

Next, we need to get the data and deserialize it into our classes in our model. However, to do that, we will need to make a call out to Marvel, which requires us to first make an MD5 Hash. Let's first write some comments detailing the steps we will need to take in our code to retrieve this information.

```
namespace HeroExplorer
{
    public class MarvelFacade
    {
        public static void GetCharacterList()
        {
            // Assemble the URL

            // Get the MD5 Hash

            // Call out to Marvel

            // Response -> string / json -> deserialize
        }
    }
}
```

Step 1: Under the first comment create a formatted string based on the url formatting found on Marvel's API. I go back to the interactive API documentation and click the Try It Out button to see the URL required to get a list of characters and copy that URL.

Note that the formatted string will contain indexes for the variable arguments we will supply later:

```
offset={0}
apikey={1}
```

Also note how the returned results limit is hard coded here as 10

```
limit=10
```

```
string url = String.Format("http://gateway.marvel.com:80/v1/public/characters?limit=10&offset={0}&apikey={1}");
```

Step 2: Set up some private fields and local variables we will need to fill out the URL (establishing the API keys and random offset, in particular)

```

namespace HeroExplorer
{
    public class MarvelFacade
    {
        private const string PrivateKey = "";
        private const string PublicKey = "";
        private const int MaxCharacters = 1500;

        public static void GetCharacterList()
        {
            // Assemble the URL
            Random random = new Random();
            var offset = random.Next(MaxCharacters);

            string url = String.Format("http://gateway.marvel.com:80/

```

And now reference the offset and PublicKey at those indexes we set in the formatted string

```
offset={0}&apikey={1}, offset, PublicKey);
```

Step 3: Copy your public/private keys from your Marvel account (note that your keys will be unique to you. The keys I've hardcoded in this lesson.)

And simply insert them into the private fields:

```

public class MarvelFacade
{
    private const string PrivateKey = "13af975425abdaf29a013f26ef12789ed536c06";
    private const string PublicKey = "4b76a06259953851972ad8977efec731";

```

Step 4: Now, we consulted stackoverflow.com to get the actual Hash algorithm in this case so we will just copy and paste it into a helper method called ComputeMD5(), and add the namespaces to the top of your script as follows:

```

using.Windows.Security.Cryptography;
using.Windows.Security.Cryptography.Core;

```

And put the algorithm into its own method:

```

// From:
// http://stackoverflow.com/questions/8299142/how-to-generate-md5-hash-code-for-my-winrt-app-using-c
private static string ComputeMD5(string str)
{
    var alg = HashAlgorithmProvider.OpenAlgorithm(HashAlgorithmNames.Md5);
    IBuffer buff = CryptographicBuffer.ConvertStringToBinary(str, BinaryStringEncoding.Utf8);
    var hashed = alg.HashData(buff);
    var res = CryptographicBuffer.EncodeToHexString(hashed);
    return res;
}

```

Step 5: Now, to get the MD5 Hash create a private helper method to handle this

```

private static string CreateHash(string timeStamp)
{
    var toBeHashed = timeStamp + PrivateKey + PublicKey;
    var hashedMessage = ComputeMD5(toBeHashed);
    return hashedMessage;
}

```

And reference this in the GetCharacterList() method

```

// Get the MD5 Hash
var timeStamp = DateTime.Now.Ticks.ToString();
var hash = CreateHash(timeStamp);

```

Note that time timestamp is required, by the Marvel API, to be “any long string which can change on a request by request basis.” So, for that we simply created a timestamp based on the current time (when the app runs), converted to a string.

And finally, modify the formatted url string to accommodate the timestamp and hash

```
offset={0}&apikey={1}&ts={2}&hash={3}", offset, PublicKey, timeStamp);
```

Step 6: Next, we will need an HttpClient, so first add this namespace and then the following code to GetCharacterList()

```
using System.Net.Http;
```

```
public async static void GetCharacterList()
{
    // Assemble the URL
    Random random = new Random();
    var offset = random.Next(MaxCharacters);

    // Get the MD5 Hash
    var timeStamp = DateTime.Now.Ticks.ToString();
    var hash = CreateHash(timeStamp);

    string url = String.Format("http://gateway.marvel.com");

    // Call out to Marvel
    HttpClient http = new HttpClient();
    var response = await http.GetAsync(url);
```

Step 7: And then to get the response deserialized, we start with taking in the JSON response

```
var response = await http.GetAsync(url);
var jsonMessage = await response.Content.ReadAsStringAsync();
```

Then, add the necessary namespaces

```
using System.Runtime.Serialization.Json;
using HeroExplorer.Models;
using System.IO;
```

And complete the serialization by adding the following

```
// Response -> string / json -> deserialize
var serializer = new DataContractJsonSerializer(typeof(CharacterDataWrapper));
var ms = new MemoryStream(Encoding.UTF8.GetBytes(jsonMessage));

var result = (CharacterDataWrapper)serializer.ReadObject(ms);
return result;
}
```

And, finally, edit the method signature to specify that return, and make it static because we don't need an instance in this case

```
public static async Task<CharacterDataWrapper> GetCharacterList()
{
```

Step 8: Now, to test and make sure we are actually retrieving data as we would expect, let's add a button to run all of this in MainPage.xaml

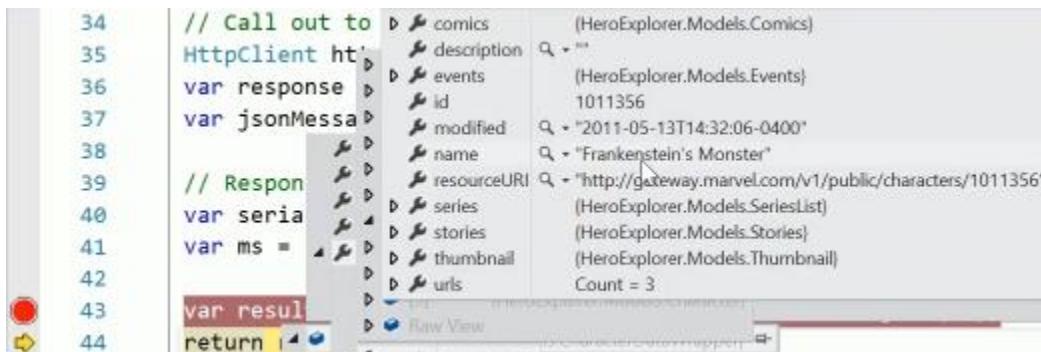
```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}>
    <Button Click="Button_Click" />
</Grid>
```



And in MainPage.xaml.cs have Button\_Click() run GetCharacterList()

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var data = MarvelFacade.GetCharacterList();
}
```

When you run the program you should now get data back from the Marvel API (you will have to run with a breakpoint in Debug mode in order to see the results at this stage).



In the next lesson we will bind to that data in order to start showing it within the app.

## UWP-074 - Hero Explorer - DataBinding & Navigating the Object Graph

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

Before we move on and attempt to display our data as a viewable list, let's fix some issues from the previous lesson. First, the name for GetCharacterList() doesn't really describe what it does. It was supposed to be returning a list of characters but, really, what it's doing is returning back the RootObject for the entire object graph (the CharacterDataWrapper).

What's needed first is to find the characters that are associated with the CharacterDataContainer, and filter through that, as some of the characters that will be brought back won't have image data.

And, from there, we want to be able to pass in an instance of an ObservableCollection<Character> into a method within MarvelFacade, adding characters to it and growing as more and more is added, finally displaying all of that within the app.

Step 1: Change the GetCharacterList() method to GetCharacterDataWrapper() and make it a private helper method

```
private static async Task<CharacterDataWrapper> GetCharacterDataWrapper()
```

Then call that method from the following method, but before that add the using statement

```
using System.Collections.ObjectModel;
```

```

public static async Task PopulateMarvelCharacters(ObservableCollection<Character> marvelCharacters)
{
    var characterDataWrapper = await GetCharacterDataWrapper();

    var characters = characterDataWrapper.data.results;

    foreach (var character in characters)
    {
        // Filter characters that are missing thumbnail images

        if (character.thumbnail != null
            && character.thumbnail.path != ""
            && character.thumbnail.path != ImageNotAvailablePath)
        {

            marvelCharacters.Add(character);
        }
    }
}

```

And reate a class-level field for ImageNotAvailablePath and set it to the following string

```
private const string ImageNotAvailablePath = "http://i.annihil.us/u/prod/marvel/i/mg/b/40/image_not_available";
```

Step 2: Go to developer.marvel.com and click on “How-Tos” and then “Images”

And refer to the two image types we will be using for this application (the smaller one for the list view, and the larger for the main display view)

Now, in the Models folder open up the CharacterDatawrapper and look at the thumbnail class.

```

public class Thumbnail
{
    public string path { get; set; }
    public string extension { get; set; }
}

```

What we will want to do is add two additional properties that will be ignored while we're deserializing into the object graph but will be useful now as we're populating an ObservableCollection of Characters that we intend to bind to.

```

public class Thumbnail
{
    public string path { get; set; }
    public string extension { get; set; }
    public string small { get; set; }
    public string large { get; set; }
}

```

And now populate those properties inside of the previous if statement

```

if (character.thumbnail != null
    && character.thumbnail.path != ""
    && character.thumbnail.path != ImageNotAvailablePath)
{
    character.thumbnail.small = String.Format("{0}/standard_small.{1}",
        character.thumbnail.path,
        character.thumbnail.extension);

    character.thumbnail.large = String.Format("{0}/portrait_xlarge.{1}",
        character.thumbnail.path,
        character.thumbnail.extension);

    marvelCharacters.Add(character);
}

```

Step 3: Go to MainPage.xaml.cs and add the following using statement

```
using System.Collections.ObjectModel;
```

And add the following property, instantiate it in MainPage(), and remove the Button\_Click() method

```

public sealed partial class MainPage : Page
{
    public ObservableCollection<Character> MarvelCharacters { get; set; }

    public MainPage()
    {
        this.InitializeComponent();

        MarvelCharacters = new ObservableCollection<Character>();
    }
}

```

Step 4: In MainPage.xaml remove the previous Grid and replace it with this one (this follows the design pattern called “Master-Detail” where on the left hand side we will have the master ListView, and on the right we will have a Grid with the details)

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="30" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <ListView Name="MasterListView">
    </ListView>
    <Grid Name="DetailGrid">
    </Grid>
</Grid>
</Page>

```

And also add a TextBlock and ProgressRing

```

<Grid Name="DetailGrid">
</Grid>

<TextBlock Text="Attribution"
    VerticalAlignment="Center"
    HorizontalAlignment="Center" />

<ProgressRing Name="MyProgressRing"
    Width="100"
    Height="100"
    Foreground="Gray"
    Grid.ColumnSpan="2"
    Grid.RowSpan="3"
    VerticalAlignment="Center"
    HorizontalAlignment="Center" />

</Grid>
</Page>

```

Step 5: Now create a Page\_Loaded event starting with adding to the top of MainPage.xaml

```

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
Loaded="Page_Loaded"
mc:Ignorable="d">

```

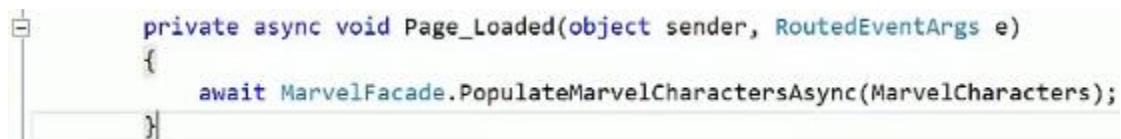
Then change the name for GetCharacterDataWrapper by suffixing "Async" at the end, and then with the name selected hit **ctrl+period** on your keyboard to quickly refactor/rename all references to this method



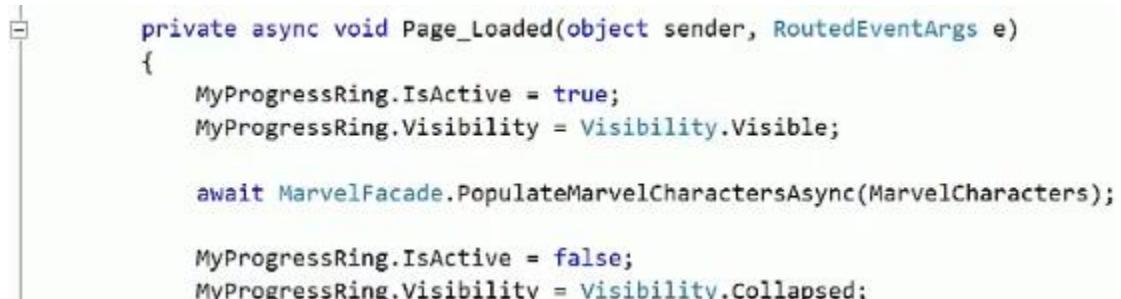
Do the same thing with PopulateMarvelCharacters()



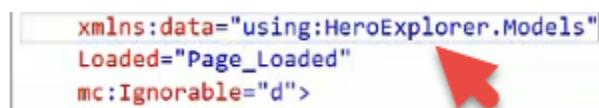
And in the method for the `Page_Loaded` event call into `MarvelFacade.PopulateMarvelCharactersAsync()`



Then, establish the how the ProgressRing is handled



Step 5: Now in `MainPage.xaml` let's data bind the `ListView` to `MarvelCharacters`, but first add the namespace at the top of the document



And then edit the `ListView` as follows

```
<ListView Name="MasterListView" ItemsSource="{x:Bind MarvelCharacters}">
    <ListView.ItemTemplate>
        <DataTemplate x:DataType="data:Character">
            <StackPanel Orientation="Horizontal">
                <Image Source="{x:Bind thumbnail.small}" />
                <TextBlock Text="{x:Bind name}" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Make some additional edits to the Grid and TextBlock

```
<Grid Name="DetailGrid" Grid.Column="1" Grid.Row="0">
</Grid>

<TextBlock Text="Attribution"
    VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Grid.ColumnSpan="2"
    Grid.Row="2" />

<ProgressRing Name="MyProgressRing"
```

If you run the program now you should be able to see the data represented, albeit needing some adjustments

Step 6: Refer the Marvel API's attribution requirements found on the site.

To accommodate the Attribution this, we will just display it on the screen for the entire app by modifying the MainPage.xaml TextBlock as follows (this is being hardcoded for simplicity, however, you will want a programmatic solution to pull this attribution text from the API if you were looking to publish an app such as this)

```
<TextBlock Text="Data provided by Marvel. © 2015 Marvel"
```

Step 7: in the ListView StackPanel format the images with an Ellipse to make them more presentable, and add a few other modifications

```
<ListView Name="MasterListView" ItemsSource="{x:Bind MarvelCharacters}">
    <ListView.ItemTemplate>
        <DataTemplate x:DataType="data:Character">
            <StackPanel Orientation="Horizontal" Margin="10,5,0,5">
                <Ellipse Width="45" Height="45">
                    <Ellipse.Fill>
                        <ImageBrush ImageSource="{x:Bind thumbnail.small}" />
                    </Ellipse.Fill>
                </Ellipse>
                <TextBlock Text="{x:Bind name}"
                           VerticalAlignment="Center"
                           Margin="10,0,0,0"
                           FontSize="18" />
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

And if you run the program it should notice this formatting in place, along with the attribution text.

And, for good measure, remove the frame counter from being displayed.

```
#if DEBUG
    if (System.Diagnostics.Debugger.IsAttached)
    {
        this.DebugSettings.EnableFrameRateCounter = true;
    }
#endif
```

## UWP-075 - Hero Explorer – Displaying Character Details

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

The goal of this lesson is to display details about the character that was selected in the master ListView, such as the thumbnail image, the larger image, the character's name and description. However, we should mention a few potential issues before we get started. First, the conditions we set for grabbing character data may not always be met (some characters have missing thumbnails, for instance), and also there may be server issues outside of our control that we would have to otherwise anticipate in code with some form of error handling.

So, if you have been running into errors when running your program, they may have been only intermittent errors that we will hopefully be able to fix in this lesson. But before all of that, let's first flesh out the existing Grid details.

Step 1: Modify the MainPage.xaml as follows

```

</ListView>

<Grid Name="DetailGrid" Grid.Column="1" Grid.Row="0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid Grid.Row="0" HorizontalAlignment="Left" VerticalAlignment="Top">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Image Name="DetailImage"
            Grid.Column="0"
            Grid.RowSpan="2" />
        <TextBlock Name="DetailNameTextBlock"
            FontSize="32"
            Grid.Column="1" />
        <TextBlock Name="DetailDescriptionTextBlock"
            Grid.Row="1"
            Grid.Column="1"
            TextWrapping="Wrap" />
    </Grid>
</Grid>

```

Step 2: Modify the ListView so that the items become clickable



```

<ListView Name="MasterListView"
    IsItemClickEnabled="True"
    ItemClick="MasterListView_ItemClick"
    ItemsSource="{x:Bind MarvelCharacters}">

```

And go into MasterListView\_ItemClick() in MainPage.xaml.cs and get the selected character through the ItemClickEventArgs e argument

```

private void MasterListView_ItemClick(object sender, ItemClickEventArgs e)
{
    var selectedCharacter = (Character)e.ClickedItem;

    DetailNameTextBlock.Text = selectedCharacter.name;
    DetailDescriptionTextBlock.Text = selectedCharacter.description;
}

```

Step 3: And now to get the image to display we have to use the following process. First, add this namespace

```
using Windows.UI.Xaml.Media.Imaging
```

And in MasterListView\_ItemClick() write

```
DetailDescriptionTextBlock.Text = selectedCharacter.description;  
  
var largeImage = new BitmapImage();  
Uri uri = new Uri(selectedCharacter.thumbnail.large, UriKind.Absolute);  
largeImage.UriSource = uri;  
DetailImage.Source = largeImage;
```

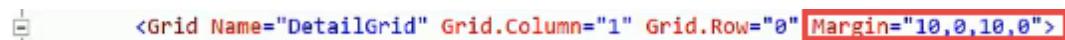
And now when you run the program you should be able to click on the name in the list and see some details on the right.

As you can see if you're following along in the videos, there are some alignment issues, so let's fix those with some edits to MainPage.xaml



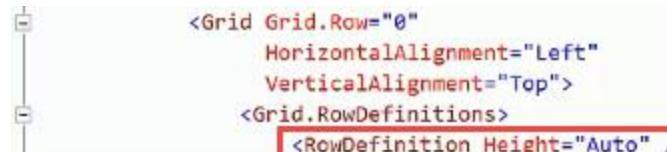
```
<Image Name="DetailImage"  
       Grid.Column="0"  
       Grid.RowSpan="2"  
       VerticalAlignment="Top" />
```

Add margin details to DetailGrid



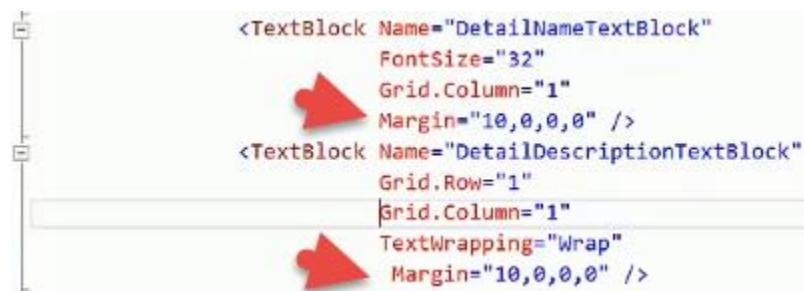
```
<Grid Name="DetailGrid" Grid.Column="1" Grid.Row="0" Margin="10,0,10,0">
```

Change this RowDefinition Height to "Auto"



```
<Grid Grid.Row="0"  
      HorizontalAlignment="Left"  
      VerticalAlignment="Top">  
  <Grid.RowDefinitions>  
    <RowDefinition Height="Auto" />
```

And add margins to the following TextBlocks



```
<TextBlock Name="DetailNameTextBlock"  
          FontSize="32"  
          Grid.Column="1"  
          Margin="10,0,0,0" />  
<TextBlock Name="DetailDescriptionTextBlock"  
          Grid.Row="1"  
          Grid.Column="1"  
          TextWrapping="Wrap"  
          Margin="10,0,0,0" />
```

Now when you run the program it should display better

Step 4: Now, let's modify some code in order to better handle potential issues. Start by adding the following namespace to the top of MainPage.xaml.cs

```
using System.Threading.Tasks;
```

And then in the Page\_Loaded() method, change it as follows

```
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    MyProgressRing.IsActive = true;
    MyProgressRing.Visibility = Visibility.Visible;

    while (MarvelCharacters.Count < 10)
    {
        Task t = MarvelFacade.PopulateMarvelCharactersAsync(MarvelCharacters);
        await t;
    }

    MyProgressRing.IsActive = false;
    MyProgressRing.Visibility = Visibility.Collapsed;
}
```

Here the while loop is used to iterate through MarvelCharacters, and since we want at least 10 records returned it will continue to iterate until that many are given. Now the challenge here is that an async method might take a while to return so what we're doing is checking on the PopulateMarvelCharactersAsync() Task being returned and wait for the Task before we continue on with another run of the while statement.

Step 5: The next thing is dealing with potential server issues outside of our control, so now let's handle that possibility by wrapping try/catch error handling around any of these dependencies we cannot control.

Let's start with the first call out to Marvel's API. In PopulateMarvelCharactersAsync() wrap the entire existing code block into a try statement (partially represented below)

```
public static async Task PopulateMarvelCharactersAsync(ObservableCollection<Character> characters)
{
    try
    {
        var characterDataWrapper = await GetCharacterDataWrapperAsync();

        var characters = characterDataWrapper.data.results;

        foreach (var character in characters)
        {
            // Filter characters that are missing thumbnail images

            if (character.thumbnail != null
                && character.thumbnail.path != ""
                && character.thumbnail.path != ImageNotAvailablePath)
            {

                character.thumbnail.small = String.Format("{0}/standard_small.{1}",
                    character.thumbnail.path,
```

And below that add the catch statement

```
        catch (Exception)
        {
            return;
        }
```

So, this try/catch in tandem with the while loop for the call of this method should mean that the program won't progress unless we have received at least 10 records without error. You could probably add a counter so that the program eventually stops trying after a certain amount of time has passed, but for now this at least avoids the issue of running the program and ending up without any data.

## UWP-076 - Hero Explorer - Displaying Comic Books for a Character

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

The goal in this lesson is to display a list of about 10 comics for the currently selected character. The good news is that we've already got a template for how it should work, we're just going to need to make a slightly different API call.

But before that let's quickly go over the API call Rate Limits defined at developer.marvel.com

*"You will only be able to call most services a certain number of times within a particular time period (calculated on a rolling basis). **Most services default to 1000 calls per day, but some services may have different rate limits.** You can view your particular rate limits on your developer account page. If you attempt to make more API calls to a service than your allotted amount, the call will be rejected by the Marvel Comics API."*

This could be a problem if you intend to create a commercial app, only to learn that the API calls its making (identified by its unique key, used in your code) are severely limited across ever user of that app. There are a few ways to think about solving this issue.

One is to have each user of your app create their own unique API key, and divert the rate limit onto each individual user rather than have it shared across all users. However, that approach can be a complex procedure.

Another possible solution is to create a web service similar to what we did in the UWP Weather app. In this scenario you would make a call from the local app to a web service (in Azure for instance), which then goes out to the Marvel API, making the necessary call to grab the data, and then cache it locally.

Now you can (as per the Marvel usage restrictions) cache results, and save them off locally, for a period of time (24 hours at most). However, the data changes on a regular basis, so you can't just store it all in a local database until the end of time. If you do, you're breaking the agreement that you have with Marvel. By the same token, you can't just go off and grab the entire database in one shot considering that the usage terms stipulates that you can "only make the calls that you need." You can interpret that to mean that you can make a call for one client, and then cache off the data that you get for him, and hold onto it for no more than 24 hours. In this case, each client would have her own cache relating to her particular data requests and there would be no central cache.

The way the cache would be handled is when you run into a case where the user makes too many calls and runs into an HTTP status error code 429 (rate limit exceeded), you will then serve that users requests through the local cache.

Having said all of this, we're not going to actually implement these solutions in this app, but you should know that those are problems you should look to solve if you were to go forward and develop a commercial app from this starting point.

Now, back to the lesson at hand, we will want to look into grabbing the comics that each character has appeared in. So, let's refer back to the tool available on developer.marvel.com. I'll look at the API: /v1/public/comics.

And, for demonstration purposes, let's grab the comics for Spider-Man (he is 1009610 in the Marvel database). I put that number in the "characters" textbox of the interactive API explorer.

Furthermore, I limit the amount of comics returned to 2 (we don't want to give more info than necessary for JSON-to-C# in the next step, and yet we need at least 2 records returned to deserialize into a collection)

As before, copy and paste the data returned, in the Response Body, to the JSON-to-C# converter (partially shown below) and copy the converted class information.



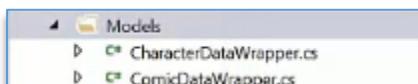
The screenshot shows the json2csharp website interface. At the top, it says "json2csharp" and "generate c# classes from json". Below that is a text area containing a JSON response body:

```
    "available": 0,
    "collectionURI": "http://gateway.marvel.com/...",
    "items": [],
    "returned": 0
```

Below the JSON area is a blue "Generate" button with a cursor icon pointing to it. To the right of the button is a preview of the generated C# code:

```
public class TextObject
{
    public string type { get; set; }
    public string language { get; set; }
    public string text { get; set; }
}
```

Step 1: In the Hero Explorer project, add a class within the “Models” folder and call that class “ComicDataWrapper”



And then paste into that document all of the class data returned from the JSON-to-C# conversion. Then, rename the RootObject class to ComicDataWrapper

```
public class ComicDataWrapper
{
    public int code { get; set; }
    public string status { get; set; }
    public string copyright { get; set; }
    public string attributionText { get; set; }
    public string attributionHTML { get; set; }
    public string etag { get; set; }
    public Data data { get; set; }
}
```

A screenshot of a code editor showing the 'ComicDataWrapper' class. A red arrow points to the class name 'ComicDataWrapper' at the top of the code block.

Step 2: Note that there are a few classes that are the named the same as current classes in the same namespace (these will have red squiggly underlines, indicating the error). Simply remove all of these, as we will rely on the previous classes with these names to supply the data. Also, delete the Item3 class while you are at it.

And, rename the Result class to ComicBook

```
public class ComicBook
{
    public int id { get; set; }
    public int digitalId { get; set; }
    public string title { get; set; }
}
```

A screenshot of a code editor showing the 'ComicBook' class. A red arrow points to the class name 'ComicBook' at the top of the code block.

And also change the above Data class to ComicDataContainer, and fix the reference to <Comic> results

```
public class ComicDataContainer
{
    public int offset { get; set; }
    public int limit { get; set; }
    public int total { get; set; }
    public int count { get; set; }
    public List<ComicBook> results { get; set; }
}
```

A screenshot of a code editor showing the 'ComicDataContainer' class. A red arrow points to the class name 'ComicDataContainer' at the top of the code block.

And in the ComicDataWrapper class fix the reference to the Data class as follows

```
    public ComicDataContainer data { get; set; }
```

Step 3: Now, go to the MarvelFacade.cs and copy all of the GetCharacterDataWrapperASync() method into a new method called GetComicDataWrapperASync(), as the process for each is much the same.

We will also have to refactor the commonalities between these methods in order to trim down the duplicated code. Take the following code out of GetCharacterDataWrapperASync() and GetComicDataWrapperASync(), and transplant it into a new helper method called CallMarvel()

```
private async static Task<string> CallMarvelAsync(string url)
{
    // Get the MD5 Hash
    var timeStamp = DateTime.Now.Ticks.ToString();
    var hash = CreateHash(timeStamp);

    string completeUrl = String.Format("{0}&apikey={1}&ts={2}&hash={3}",
    // Call out to Marvel
    HttpClient http = new HttpClient();
    var response = await http.GetAsync(completeUrl);
    return await response.Content.ReadAsStringAsync();
}
```

And in that method, notice how we now pass in the first part of the URL as a string argument

```
String.Format("{0}&apikey={1}&ts={2}&hash={3}", url, PublicKey, timeStamp)
```

And the resulting methods for GetCharacterDataWrapperAsync() and GetComicDataWrapperAsync() will be refactored as follows

```
private static async Task<CharacterDataWrapper> GetCharacterDataWrapperAsync()
{
    // Assemble the URL
    Random random = new Random();
    var offset = random.Next(MaxCharacters);

    string url = String.Format("http://gateway.marvel.com:80/v1/public/characters?limit=10&offset={0}",
        offset);

    var jsonMessage = await CallMarvelAsync(url);

    // Response -> string / json -> deserialize
    var serializer = new DataContractJsonSerializer(typeof(CharacterDataWrapper));
    var ms = new MemoryStream(Encoding.UTF8.GetBytes(jsonMessage));

    var result = (CharacterDataWrapper)serializer.ReadObject(ms);
    return result;
}
```

...

```
private static async Task<ComicDataWrapper> GetComicDataWrapperAsync(int characterId)
{
    var url = String.Format("http://gateway.marvel.com:80/v1/public/comics?characters={0}&limit=10",
        characterId);

    var jsonMessage = await CallMarvelAsync(url);

    // Response -> string / json -> deserialize
    var serializer = new DataContractJsonSerializer(typeof(ComicDataWrapper));
    var ms = new MemoryStream(Encoding.UTF8.GetBytes(jsonMessage));

    var result = (ComicDataWrapper)serializer.ReadObject(ms);
    return result;
}
```

Step 4: Now we will want to apply the same process in populating thing comics, reusing code from PopulateMarvelCharactersAsync() into a new method called PopulateMarvelComicsAsync() which will look like this

```
public static async Task PopulateMarvelComicsAsync(int characterId,
observableCollection<ComicBook> marvelComics)
{
    try
    {
        var comicDataWrapper = await GetComicDataWrapperAsync(characterId);

        var comics = comicDataWrapper.data.results;

        foreach (var comic in comics)
        {
            // Filter characters that are missing thumbnail images

            if (comic.thumbnail != null
                && comic.thumbnail.path != ""
                && comic.thumbnail.path != ImageNotAvailablePath)
            {

                comic.thumbnail.small = String.Format("{0}/standard_small.{1}",
                    comic.thumbnail.path,
                    comic.thumbnail.extension);

                comic.thumbnail.large = String.Format("{0}/portrait_xlarge.{1}",
                    comic.thumbnail.path,
                    comic.thumbnail.extension);

                marvelComics.Add(comic);
            }
        }
    }
    catch (Exception)
    {
        return;
    }
}
```

Step 5: Now to populate the comics when an given item is selected, first add a property to MainPage and initialize it in the Constructor

```
{
    public ObservableCollection<Character> MarvelCharacters { get; set; }
    public ObservableCollection<ComicBook> MarvelComics { get; set; }

    public MainPage()
    {
        this.InitializeComponent();

        MarvelCharacters = new ObservableCollection<Character>();
        MarvelComics = new ObservableCollection<ComicBook>();
    }
}
```

And in MasterListView\_ItemClick() insert the following (note how we also added the ProgressRing code)

```
private async void MasterListView_ItemClick(object sender, ItemClickEventArgs e)
{
    MyProgressRing.IsActive = true;
    MyProgressRing.Visibility = Visibility.Visible; ←

    var selectedCharacter = (Character)e.ClickedItem;

    DetailNameTextBlock.Text = selectedCharacter.name;
    DetailDescriptionTextBlock.Text = selectedCharacter.description;

    var largeImage = new BitmapImage();
    Uri uri = new Uri(selectedCharacter.thumbnail.large, UriKind.Absolute);
    largeImage.UriSource = uri;
    DetailImage.Source = largeImage;

    MarvelComics.Clear();

    await MarvelFacade.PopulateMarvelComicsAsync(
        selectedCharacter.id,
        MarvelComics); ←

    MyProgressRing.IsActive = false;
    MyProgressRing.Visibility = Visibility.Collapsed;
}
```

Step 6: Now, in MainPage.xaml we need to bind the data to MarvelComicBooks



The screenshot shows the XAML code for the MainPage.xaml page. A red box highlights the section where the GridView is defined. The code includes a TextBlock for the character's name and description, followed by a Grid containing a TextBlock and a GridView. The GridView is bound to the MarvelComics collection and has its ItemClick event set to call the GridView\_ItemClick method.

```
<TextBlock Name="DetailNameTextBlock"
           Grid.Row="1"
           Grid.Column="1"
           TextWrapping="Wrap"
           Margin="10,0,0,0" />

</Grid>

<!-- List of Comics --&gt;
&lt;GridView Grid.Row="1"
          ItemsSource="{x:Bind MarvelComics}"
          IsItemClickEnabled="True"
          ItemClick="GridView_ItemClick"&gt;

&lt;/GridView&gt;</pre>
```

And complete that with the following GridView.ItemTemplate

```
<!-- List of Comics -->
<GridView Grid.Row="1"
    ItemsSource="{x:Bind MarvelComics}"
    IsItemClickEnabled="True"
    ItemClick="GridView_ItemClick">
    <GridView.ItemTemplate>
        <DataTemplate x:DataType="data:ComicBook">
            <Image Source="{x:Bind thumbnail.small}" Width="100" Height="150" />
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>
```

And back in PopulateMarvelComicsAsync() display a medium sized portrait (rather than small)

```
comic.thumbnail.small = String.Format("{0}/portrait_medium.{1}",
    comic.thumbnail.path,
    comic.thumbnail.extension);
```



```
comic.thumbnail.large = String.Format("{0}/portrait_xlarge.{1}",
```

And now when you run the program, you should see a list of comics for each character selected. Again, to see this in action, you should watch the video associated with this lesson.

## UWP-077 - Hero Explorer – Displaying Comic Book Details

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

The goal in this lesson is to display the details for the comic book that was clicked on. But before that let's create some comments so that it will be easier to find where to add edits as we move along.

```
<!-- List of Comics -->
<GridView Grid.Row="1"
    ItemsSource="{x:Bind MarvelComics}"
    IsItemClickEnabled="True"
    ItemClick="GridView_ItemClick">
    <GridView.ItemTemplate>
        <DataTemplate x:DataType="data:ComicBook">
            <Image Source="{x:Bind thumbnail.small}" Width="100" Height="150" />
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>

<!-- Comic Details -->

<!-- Detail Grid --> 
<Grid Name="DetailGrid" Grid.Column="1" Grid.Row="0" Margin="10,0,10,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <!-- Character Detail --> 
    <Grid Grid.Row="0"
        HorizontalAlignment="Left"
        VerticalAlignment="Top">
        <Grid.RowDefinitions>
```

Step 1: And now let's fill in those edits

```
<!-- Comic Details -->
<Grid Grid.Row="2"
      HorizontalAlignment="Left"
      VerticalAlignment="Top">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Image Name="ComicDetailImage"
          Grid.Column="0"
          Grid.RowSpan="2"
          VerticalAlignment="Top" />
    <TextBlock Name="ComicDetailNameTextBlock"
              FontSize="32"
              Grid.Column="1"
              Margin="10,0,0,0" />
    <TextBlock Name="ComicDetailDescriptionTextBlock"
              Grid.Row="1"
              Grid.Column="1"
              TextWrapping="Wrap"
              Margin="10,0,0,0" />
</Grid>

<!-- List of Comics -->
<GridView Name="ComicsGridView"
          Grid.Row="1"
          ItemsSource="{x:Bind MarvelComics}"
          IsItemClickEnabled="True"
          ItemClick="ComicsGridView_ItemClick">
```

And in MainPage.xaml.cs rename GridView\_Click() to ComicsGridView\_Click() and we will just copy some of the same code for populating elements that we had for MasterListView\_Click() and paste it in this method, changing a few lines accordingly

```

private void ComicsGridView_ItemClick(object sender, ItemClickEventArgs e)
{
    var selectedComic = (ComicBook)e.ClickedItem;

    ComicDetailNameTextBlock.Text = selectedComic.title;
    ComicDetailDescriptionTextBlock.Text = selectedComic.description;

    var largeImage = new BitmapImage();
    Uri uri = new Uri(selectedComic.thumbnail.large, UriKind.Absolute);
    largeImage.UriSource = uri;
    ComicDetailImage.Source = largeImage;
}

```

Step 2: Now, when you run the program you should be able to click and retrieve comic listings, and see the description as well. However, you will have to resize the window to see the information so let's fix that as well by adding a Grid ScrollViewer as follows

```

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:data="using:HeroExplorer.Models"
Loaded="Page_Loaded"
mc:Ignorable="d"

<Grid>
    <ScrollView>
        </ScrollView>
    </Grid>

```

And then simply take the rest of the XML below that Grid ScrollViewer that we had previously and cut/paste it in between the ScrollViewer tags (partially represented below)

```
<Page  
    x:Class="HeroExplorer.MainPage"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:local="using:HeroExplorer"  
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
    xmlns:data="using:HeroExplorer.Models"  
    Loaded="Page_Loaded"  
    mc:Ignorable="d">  
  
    <Grid>  
        <ScrollViewer>  
            <Grid>  
                <Grid.RowDefinitions>  
                    <RowDefinition Height="*" />  
                    <RowDefinition Height="Auto" />  
                    <RowDefinition Height="30" />  
                </Grid.RowDefinitions>  
                <Grid.ColumnDefinitions>  
                    <ColumnDefinition Width="Auto" />  
                    <ColumnDefinition Width="*" />  
                </Grid.ColumnDefinitions>  
  
                ...  
  
                <TextBlock Text="Data provided by Marvel. © 2015 Marvel"  
                    VerticalAlignment="Center"  
                    HorizontalAlignment="Center"  
                    Grid.ColumnSpan="2"  
                    Grid.Row="2" />  
  
                <ProgressRing Name="MyProgressRing"  
                    Width="100"  
                    Height="100"  
                    Foreground="Gray"  
                    Grid.ColumnSpan="2"  
                    Grid.RowSpan="3"  
                    VerticalAlignment="Center"  
                    HorizontalAlignment="Center" />  
  
            </Grid>  
        </ScrollViewer>  
    </Grid>  
</Page>
```

Step 3: When running the program you may notice that some descriptions are retrieved as “null” which can cause an error, so simply fix it with this conditional check

```
private void ComicsGridView_ItemClick(object sender, ItemClickEventArgs e)
{
    var selectedComic = (ComicBook)e.ClickedItem;

    ComicDetailNameTextBlock.Text = selectedComic.title;

    if (selectedComic.description != null) ← Red arrow
        ComicDetailDescriptionTextBlock.Text = selectedComic.description;
}
```

Step 4: Another bug you may have noticed is the item details are retained even when you click onto another comic character. To fix that add the following code

```
private async void MasterListView_ItemClick(object sender, ItemClickEventArgs e)
{
    MyProgressRing.IsActive = true;
    MyProgressRing.Visibility = Visibility.Visible;

    ← Red arrow
    ComicDetailNameTextBlock.Text = "";
    ComicDetailDescriptionTextBlock.Text = "";
    ComicDetailImage.Source = null;
}
```

Step 5: In this method put the await keyword before the PopulateMarvelComicsAsync() call

```
await MarvelFacade.PopulateMarvelComicsAsync(
    selectedCharacter.id,
    MarvelComics);

MyProgressRing.IsActive = false;
MyProgressRing.Visibility = Visibility.Collapsed;

}
```

Step 6 (optional): You may want to consider the same kind of error prevention we employed in the previous lessons to retrieving comic book information in this method. However, be forewarned that this won't really work if the character retrieved from the API doesn't appear in at least 10 comic books - which would complicate this check.

If you wish to experiment with how to go about implementing this, here is the starting point detailed below. How would you go about filtering for these edge cases?

```
Uri uri = new Uri(selectedCharacter.thumbnail.large, UriKind.Absolute);
largeImage.UriSource = uri;
DetailImage.Source = largeImage;

MarvelComics.Clear();

while (MarvelComics.Count < 10)
{
    Task t = MarvelFacade.PopulateMarvelComicsAsync(
        selectedCharacter.id,
        MarvelComics);
    await t;
}

MyProgressRing.IsActive = false;
MyProgressRing.Visibility = Visibility.Collapsed;

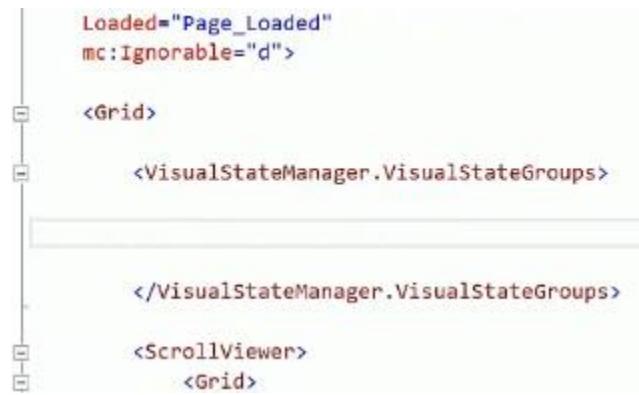
}
```

In the next lesson we will want to look at adding an adaptive layout so that the visual elements fluidly adapt to resizing and different viewports.

## UWP-078 - Hero Explorer – Displaying Comic Book Details

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

In this lesson we will want to create an adaptive layout to handle, both, wide and narrow viewports. So to do that we'll add a VisualStateManager and some VisualStates, starting with the following in MainPage.xaml



Step 1: Set up the conditions for a wide states and narrow states, which triggers under the condition of a minimum width of 800 (wide) and 0 (narrow) respectively

```

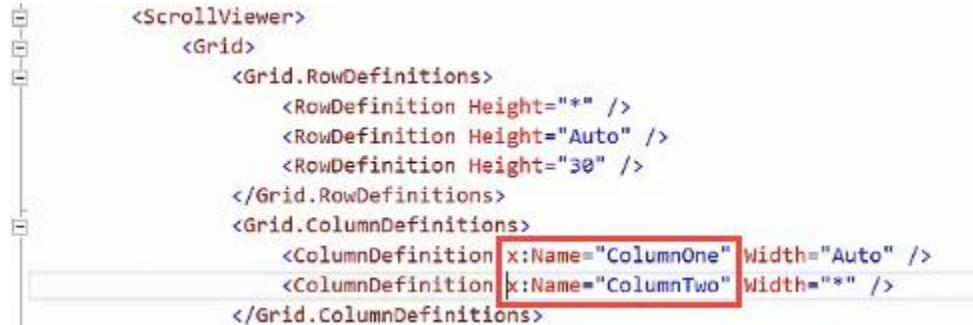
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="VisualStateGroup">
        <VisualState x:Name="Wide">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="800" />
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="DetailGrid.(Grid.Row)" Value="0" />
                <Setter Target="DetailGrid.(Grid.Column)" Value="1" />
            </VisualState.Setters>
        </VisualState>
        <VisualState x:Name="Narrow">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="0" />
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="DetailGrid.(Grid.Row)" Value="1" />
                <Setter Target="DetailGrid.(Grid.Column)" Value="0" />
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>

```

Note the different settings depending on the wide/narrow VisualState.StateTriggers

Now, when you run the program and resize the windows (after selecting a character), you should notice that the visual organization changes from appearing on the right to appearing below.

Step 2: When the information appears below, in the running program, it doesn't appear correctly so follow that with changing the Grid.ColumnDefinition as follows

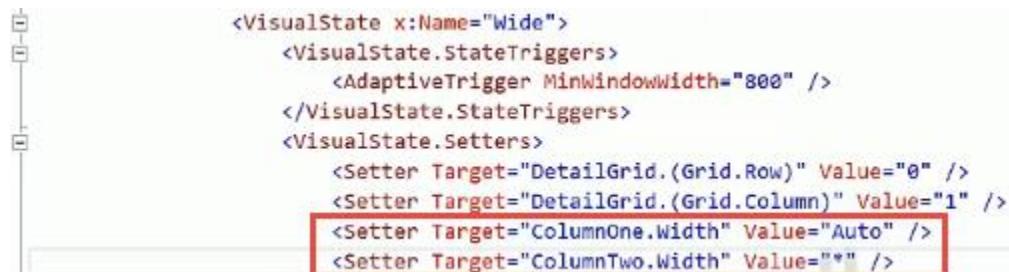


```

<ScrollView>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="30" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition x:Name="ColumnOne" Width="Auto" />
            <ColumnDefinition x:Name="ColumnTwo" Width="*" />
        </Grid.ColumnDefinitions>
    </Grid>

```

And reference this in the VisualState.Setters as follows



```

<VisualState x:Name="Wide">
    <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="800" />
    </VisualState.StateTriggers>
    <VisualState.Setters>
        <Setter Target="DetailGrid.(Grid.Row)" Value="0" />
        <Setter Target="DetailGrid.(Grid.Column)" Value="1" />
        <Setter Target="ColumnOne.Width" Value="Auto" />
        <Setter Target="ColumnTwo.Width" Value="*" />
    </VisualState.Setters>

```

...

```
<VisualState x:Name="Narrow">
    <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0" />
    </VisualState.StateTriggers>
    <VisualState.Setters>
        <Setter Target="DetailGrid.(Grid.Row)" Value="1" />
        <Setter Target="DetailGrid.(Grid.Column)" Value="0" />
        <Setter Target="ColumnOne.Width" Value="*" />
        <Setter Target="ColumnTwo.Width" Value="Auto" />
    </VisualState.Setters>
</VisualState>
```

Step 3: Next we will have to do some text wrapping to the ComicDetailNameTextBlock

```
<Image Name="ComicDetailImage"
    Grid.Column="0"
    Grid.RowSpan="2"
    VerticalAlignment="Top" />
<TextBlock Name="ComicDetailNameTextBlock"
    FontSize="32"
    Grid.Column="1"
    TextWrapping="Wrap"
    Margin="10,0,0,0" />
```

And, to the DetailNameTextBlock

```
VerticalAlignment="Top" />
<TextBlock Name="DetailNameTextBlock"
    FontSize="32"
    TextWrapping="Wrap"
    Grid.Column="1" />
```

And also add 10 to the right-hand margin for the following

```
<TextBlock Name="ComicDetailNameTextBlock"
    FontSize="32"
    Grid.Column="1"
    TextWrapping="Wrap"
    Margin="10,0,10,0" />
<TextBlock Name="ComicDetailDescriptionTextBlock"
    Grid.Row="1"
    Grid.Column="1"
    TextWrapping="Wrap"
    Margin="10,0,10,0" />
```

When you run the program now the margins and item arrangements should look much better for both the wide and narrow viewports. It's probably a good idea now to also check out how it looks in the Phone Emulator.

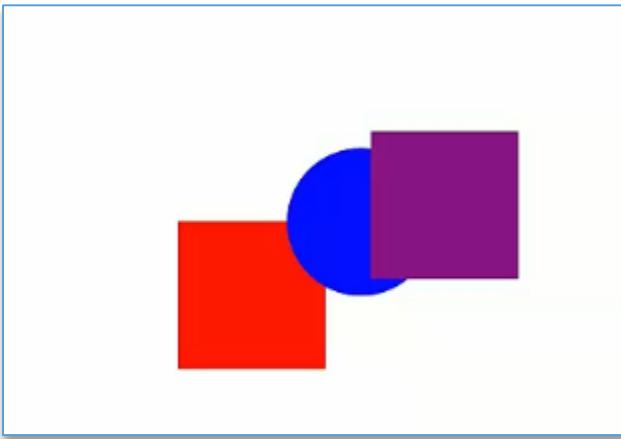


You may notice some areas with room for improvement, but most of the arrangement layout issues have been solved. In the next lesson we will look into Cortana integration for operation via voice recognition.

## UWP-079 – Hero Explorer – Cortana Integration

*Note: We respect Marvel's intellectual property and appreciate their willingness to feature their API in this series of lessons. However, there are rules they put in place for the API that do not extend to electronic or printed documentation and therefore we are not displaying screen shots containing images that belong to them. I apologize for this inconvenience, but I would recommend you watch the videos associated with these lessons if you want to see that app we're building as I progress through these steps.*

The goal in this final lesson is to demonstrate how to integrate Cortana into our application. But before we integrate Cortana into our app, let's take a glimpse at a simple, standalone, application that responds to voice commands such as, "Hey Cortana. In 'Example App', add a red rectangle, add a blue circle and add a rectangle," producing the resulting display within the app.



This functionality all starts with an XML file called a Voice Command Dictionary which allows you create a series of voice commands that Cortana responds to in your application. This file follows a very specific schema, as laid out here in the XML namespace.

<http://schemas.microsoft.com/voicecommands/1.1>

And then you create a set of commands called a Command Set for every language and region that your application will support (in this case, this is the English US version - note that Cortana's not available in every region and every language just yet)

```
<?xml version="1.0" encoding="utf-8"?>
<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.1">
    <CommandSet xml:lang="en-us" Name="ExampleAppCommandSet_en-us">
```

The next thing to do is create a Command Prefix (which, in our example above would be the voice command saying "...in Example App...")

```
<CommandSet xml:lang="en-us" Name="ExampleAppCommandSet_en-us">
  <CommandPrefix> Example App, </CommandPrefix>
```

This is an important cue for Cortana to identify your application, and send the command to your application as opposed to other applications that might be running at that time.

And below that there are two commands for addRectangle and addCircle, which we will reference by name in code later. Also here there are ListenFor elements that allow for variations on a command that Cortana can still respond to. So in this first case, Cortana's listening for "add a rectangle." In the second case, Cortana's listening for "add a color rectangle."

```
<CommandPrefix> Example App, </CommandPrefix>
<Example> Add a rectangle </Example>

<Command Name="addRectangle">
    <Example> add a rectangle </Example>
    <ListenFor> add [a] rectangle </ListenFor>
    <ListenFor> add a {color} rectangle </ListenFor>
    <Feedback> Adding a rectangle </Feedback>
    <Navigate/>
</Command>

<Command Name="addCircle">
    <Example> add a circle </Example>
    <ListenFor> add [a] circle </ListenFor>
    <ListenFor> add a {color} circle </ListenFor>
    <Feedback> Adding a circle </Feedback>
    <Navigate/>
</Command>
```

Now the part in brackets, [a] in this case, designates optional words. So you could say, "add rectangle" or "add a rectangle." Furthermore, if you were to use the color you would have to say, "add a color rectangle."

As far as the color is concerned you can see that it's bound inside of curly braces, which then refers Cortana to a phrase list that goes along with that label, "color." Here we can listen to colors red, blue, yellow, and green.

```
<Command Name="addCircle">
    <Example> add a circle </Example>
    <ListenFor> add [a] circle </ListenFor>
    <ListenFor> add a {color} circle </ListenFor>
    <Feedback> Adding a circle </Feedback>
    <Navigate/>
</Command>

<PhraseList Label="color">
    <Item>Red</Item>
    <Item>Blue</Item>
    <Item>Yellow</Item>
    <Item>Green</Item>
</PhraseList>
```

You will also notice that there is a Feedback command, which is what Cortana would say back to you, upon recognizing your command (in this case, Cortana would say “Adding a rectangle”).

```
<Feedback> Adding a rectangle </Feedback>
```

There are also some unused commands in this example, such as the Navigate command, which might be used to navigate to a different page in the application.

```
<Feedback> Adding a circle </Feedback>
<Navigate/>
</Command>
```

If you want to learn more about commands you can use with Cortana, refer to the following url

[dev.windows.com/Cortana](http://dev.windows.com/Cortana)

And go to the link that says “Cortana Interactions”

### Cortana interactions

Using voice commands with Cortana, you can incorporate functionality from your app directly into Cortana’s UI, launch your app, or deep link into your app to initiate an action, execute a command, or open a page.

And at that page you will find example XML files that you can copy and paste into your applications

- › Custom user interactions
  - Cortana interactions
    - [Launch a foreground app with voice commands in Cortana](#)

Now to actually code this up, you will have to override a method called OnActivated() in your application's App.xaml.cs

```
protected override void OnActivated(IActivatedEventArgs e)
{
    // Was the app activated by a voice command?
    if (e.Kind != Windows.ApplicationModel.Activation.ActivationKind.VoiceCommand)
    {
        return;
    }

    var commandArgs = e as Windows.ApplicationModel.Activation.VoiceCommandActivatedEventArgs;

    var speechRecognitionResult = commandArgs.Result;
    string voiceCommandName = speechRecognitionResult.RulePath[0];
    string textSpoken = speechRecognitionResult.Text;

    string spokenColor = "";
```

Here we're asking if the app was activated as a result of a voice command. If it wasn't, then simply return back to the calling code block, otherwise, grab the command arguments that were passed in as IActivatedEventArgs e (and later cast to VoiceCommandActivatedEventArgs) and store the commandArgs.Result property into a variable called speechRecognitionResult.

From here, we can determine which command was spoken and/or parse through the text, and then attempt to match it with a color defined in the Voice Command Dictionary (the first index [0] is referenced here because there may be multiple commands/phrase lists passed in the arguments)

```
string textSpoken = speechRecognitionResult.Text;

string spokenColor = "";
try
{
    spokenColor = speechRecognitionResult.SemanticInterpretation.Properties["color"][0];
}
catch
{
    //
}
```

And then, there is a simple switch statement to handle the actual color selected

```
        catch
        {
            //
        }

        Windows.UI.Color color;

        switch (spokenColor)
        {
            case "Red":
                color = Colors.Red;
                break;
            case "Blue":
                color = Colors.Blue;
                break;
            case "Yellow":
                color = Colors.Yellow;
                break;
            case "Green":
                color = Colors.Green;
                break;
        }
```

And then get a reference to the MainPage

```
Frame rootFrame = Window.Current.Content as Frame;
MainPage page = rootFrame.Content as MainPage;
if (page == null)
{
    return;
}

switch (voiceCommandName)
{
```

And determine, by putting the VoiceCommandName into a switch statement, which method should be called, either page.CreateRectange() or page.CreateCircle()

```
switch (voiceCommandName)
{
    case "addRectangle":
        page.CreateRectangle(color);
        break;

    case "addCircle":
        page.CreateCircle(color);
        break;

    default:
        // There is no match for the voice command name.
        break;
}
```

These methods are located in MainPage.xaml.cs

```
public void CreateRectangle(Color color)
{
    Random random = new Random();
    var left = random.Next(0, 300);
    var top = random.Next(0, 300);

    var rect = new Windows.UI.Xaml.Shapes.Rectangle();
    rect.Height = 100;
    rect.Width = 100;
    rect.Margin = new Thickness(left, top, 0, 0);

    rect.Fill = new SolidColorBrush(color);

    LayoutGrid.Children.Add(rect);
}
```

```

public void CreateCircle(Color color)
{
    Random random = new Random();
    var left = random.Next(300);
    var top = random.Next(300);

    var circle = new Windows.UI.Xaml.Shapes.Ellipse();
    circle.Height = 100;
    circle.Width = 100;
    circle.Margin = new Thickness(left, top, 0, 0);
    circle.Fill = new SolidColorBrush(color);

    LayoutGrid.Children.Add(circle);
}

```

Also in MainPage.xaml.cs is a reference to the XML Voice Command Dictionary upon page load

```

private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    var storageFile =
        await Windows.Storage.StorageFile.GetFileFromApplicationUriAsync(
            new Uri("ms-appx:///CortanaExampleCommands.xml"));
    await Windows.ApplicationModel.VoiceCommands.VoiceCommandDefinitionManager
        .InstallCommandDefinitionsFromStorageFileAsync(storageFile);
}

```

Now, let's port this example over into our Hero Explorer application in order to integrate it with Cortana

Step 1: Load the Hero Explorer project, and in the Solution Explorer add an XML file called "VoiceCommandDictionary" and write

```

<?xml version="1.0" encoding="utf-8"?>
<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.1">
    <CommandSet xml:lang="en-us" Name="ExampleAppCommandSet_en-us">
        <CommandPrefix> Hero Explorer, </CommandPrefix>
        <Example> Refresh the character list </Example>

        <Command Name="refresh">
            <Example> refresh the character list </Example>
            <ListenFor> refresh [the] [character] list </ListenFor>
            <ListenFor> refresh [the] characters </ListenFor>
            <Feedback> Refreshing the character list </Feedback>
            <Navigate/>
        </Command>

    </CommandSet>
    <!-- Other CommandSets for other languages -->
</VoiceCommands>

```

Step 2: Go to the App.xaml.cs and add in the overridden method for OnActivated()

```
protected override void OnActivated(IActivatedEventArgs e)
{
    // Was the app activated by a voice command?
    if (e.Kind != Windows.ApplicationModel.Activation.ActivationKind.VoiceCommand)
    {
        return;
    }

    var commandArgs = e as Windows.ApplicationModel.Activation.VoiceCommandActivatedEventArgs;

    var speechRecognitionResult = commandArgs.Result;
    string voiceCommandName = speechRecognitionResult.RulePath[0];

    Frame rootFrame = Window.Current.Content as Frame;
    MainPage page = rootFrame.Content as MainPage;
    if (page == null)
    {
        return;
    }

    if (voiceCommandName == "refresh")
    {
        // Call public method on main page
    }
}
```

Step 3: In MainPage.xaml.cs take out the previous contents for Page\_Loaded() and put it in its own method called Refresh()

```
public async void Refresh()
{
    MyProgressRing.IsActive = true;
    MyProgressRing.Visibility = Visibility.Visible;

    while (MarvelCharacters.Count < 10)
    {
        Task t = MarvelFacade.PopulateMarvelCharactersAsync(MarvelCharacters);
        await t;
    }

    MyProgressRing.IsActive = false;
    MyProgressRing.Visibility = Visibility.Collapsed;
}
```

Step 4: Place this in the Page\_Loaded() method, including the reference to Refresh()

```
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    var storageFile =
        await Windows.Storage.StorageFile.GetFileFromApplicationUriAsync(
            new Uri("ms-appx:///VoiceCommandDictionary.xml"));
    await Windows.ApplicationModel.VoiceCommands.VoiceCommandDefinitionManager
        .InstallCommandDefinitionsFromStorageFileAsync(storageFile);

    Refresh();
}
```

And in App.xaml.cs add the reference to Refresh() as well

```
if (voiceCommandName == "refresh")
{
    page.Refresh();
}
```

And in MainPage.xaml.cs add MarvelCharacters.Clear()

```
MarvelCharacters.Clear();
while (MarvelCharacters.Count < 10)
{
    Task t = MarvelFacade.PopulateMarvelCharactersAsync(MarvelCharacters);
    await t;
}
```

When you run the program you should now have Cortana voice recognition in your app by saying “Hey Cortana. In Hero Explorer, refresh the character list.”

## UWP-080 - Wrap Up

Well, it looks like that's a wrap! The fact that you got to this point means that you're probably pretty good at sticking with things, and that quality will serve you well, as it takes time and patience to create a successful app. Constructing apps such as the ones you learned how to make here might take you a couple of hours once you become more experienced. However, you will likely spend a couple of days (or weeks) testing, tweaking, asking beta testers for suggestions, and making sure it's polished and not going to throw an exception error in any circumstance.

And that's exactly what you'll want to do in order to make sure that you get a great first impression, and good reviews, in the Microsoft Store. So if you stuck with me through all of these lessons, congratulate yourself for learning how to create apps using the Universal Windows Platform.

If you enjoyed this training series please make sure you let Microsoft know. There are Microsoft/Channel9 forums and Twitter accounts, and plenty of other ways to tell Microsoft what you think about this series. I'd also recommend that you check out the other series that I've created. Search for my name (Bob Tabor) on either of these Microsoft resources:

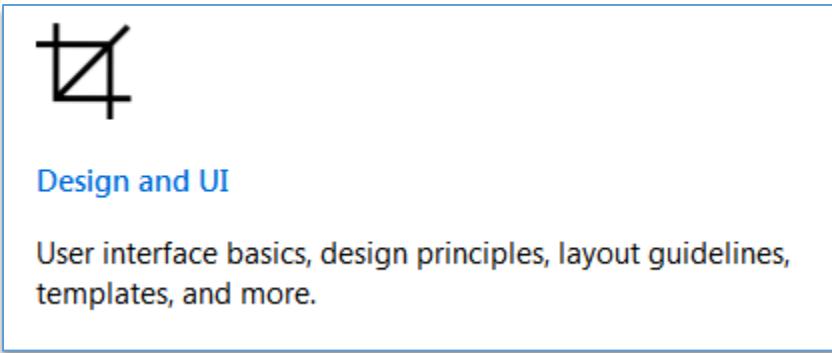
<http://www.microsoftvirtualacademy.com/>  
<http://channel9.msdn.com/>

If you search you will find several free training series - covering different technologies - that I've created and you might like, including "HTML5 and CSS3 for Absolute Beginners," "Javascript and jQuery for Absolute Beginners," and around 100 Azure videos across four different series that I've created, and a bunch more.

If you're looking for additional resources on top of that, then I direct you to a series by Andy Wiggy and Shen Chauhan, called "The Developer's Guide to Windows 10." Also, one resource that I didn't tell you about (and perhaps one of the best) is available from the Windows Dev Center. So go out to

[developer.windows.com](http://developer.windows.com)

And then choose the link called "Design and UI"



The image shows a screenshot of the Windows Dev Center website. At the top left is a large, stylized number '4'. Below it, the text 'Design and UI' is displayed in blue, underlined font. Underneath that, a description reads: 'User interface basics, design principles, layout guidelines, templates, and more.' The entire image is enclosed in a light blue rectangular border.

Becoming familiar with this section on the site really helped me to start thinking like a Windows 10 developer, so I highly recommend it.

I hope you already have an app, or two, in mind that you want to build, and hopefully some of the things that you learn in this series will help you get started. If you're looking for meaty code examples that go way beyond what I was able to demonstrate in this series, then you'll definitely want to check out this link here:

<http://bit.do/Windows-universal-samples>

That will take you to Github, where there are several dozen samples that were created by Microsoft. It's an exhaustive list of examples with just about any feature that you can possibly think of. And, within those projects, there are multiple examples of very advanced concepts, so it's an outstanding resource.

I should also mention that in perusing the Microsoft Store I have come to notice there are a lot of "low-hanging fruit" possibilities that somebody needs to pluck it off the opportunity tree. In other words, I think that if you look around - and think hard about the apps that are successful on other platforms - then you take a look at the Windows store and ask yourself, "of the apps that are available to do so-and-so, are there any features that are missing?" Think about how you can introduce apps that work great on any Windows 10 device. And think of the benefits to having the user of those apps work across an ecosystem of Windows 10 devices. Not just one specific device, but either desktop or phone app.

Think about how you could add Cortana integration, in order to enhance the experience. Think about live tiles - and how you can use those - periodically updating the content on those tiles, in order to inform the user without them having to explode the application open. And just think about how you can improve the user's experience. In the end, don't shoehorn features in that you don't need, advocate for features you want to see as a user yourself. Make it simple, and then give them the tools that they need to achieve the end result that they're after.

And keep in mind that, essentially, you'll be running a company (at least, that's how it's going to look to those who are actually using your apps). So, you'll need to grow thick skin (I tell you this from personal experience). There are lots of nice people out there, who will give you great feedback and will thank you for your work, while occasionally you're going to run into someone who's a little grumpy. And so, you'll want to respond politely to everybody, even the people who are a little bit mean. You want to fix the issues in your app, and upload that to the store so that the users can update on their end, and get the benefit of those updates, as quickly as possible.

From my experience, whenever I treat a grumpy customer kindly, they usually become my greatest advocate and cheerleader. So, I'd recommend that you do that, too. As a famous motivational speaker was known to say:

"You can get what you want out of life, if you help enough other people get what they want."

I truly believe that, and I know that there are great opportunities for you to find your niche, to find your customers, and then deliver solutions that they're looking for.

## Closing Thoughts and Acknowledgements

Before I close, I want to put in a little plug for my own website

<http://www.LearnVisualStudio.NET>

LearnVisualStudio.NET is where I teach beginners the skills that they need to get their first software development job building Windows and Web Apps, for the world's best companies, as quickly as possible. Here you're going to find more comprehensive series than what I've done on Channel9 or Microsoft Virtual Academy, on topics like C# and ASP.NET, ASP.NET MVC, application architecture, data access, and just about everything you're going to need to get an entry level software development job at a major company.

Once again, I want to sincerely thank Andy Wiggly, he really helped me fit about six months to a year's worth of effort into about six weeks. And Clint Rutkas, thank you, sir. Thanks also to Larry Lieberman, and others I've worked with in the past, especially Dan Fernandez, to whom I am eternally grateful.

Furthermore, I want to express a great deal of appreciation to Steven Nikolic for helping me edit this PDF. It would have taken my twice as long to finish this if he were not able or willing to take this project on. Thank you!

And finally, my biggest thanks goes out to you. I hope that, if nothing else, you've gained the confidence to feel like you can do this, as I certainly believe that it's not out of your reach. Don't get discouraged if success doesn't come easy. If you hit roadblocks, just take your time and work through them methodically, and be patient. I try to make all of this look easy, and for the most part, it really is. But not everything is simple at first glance. I have the benefit of editing out my mistakes, whereas real life software development is a bit more *bumpy*, and there's more struggle to it, no matter who you are.

Everybody has to go through the struggle at some point. The only difference between you and I is that I've been doing all of this for 20 years. I just took tiny baby steps on a daily basis. Frankly, you already know more about software development - by watching these videos / reading the PDF or watching the Absolute Beginner videos on C# - than I knew in the first couple years that I was writing code, so you've already got a great head start!

Just know that you CAN do it. And when you do it, go ahead and send me a tweet and let me know what you came up with. I'd love to encourage you, and take a look at what you came up with. It would be exciting for me too, and I'd love to see it.

I sincerely wish you the best as you begin your career writing apps for Windows 10. Good luck, and thank you.

Bob Tabor (Twitter handle: @bobtabor)

