

CHANNEL9'S WINDOWS PHONE 8.1 DEVELOPMENT FOR ABSOLUTE BEGINNERS

Full Text Version of the Video Series

Published April, 2014

Bob Tabor
<http://www.LearnVisualStudio.net>

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

Microsoft makes no warranties, express or implied, in this document.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2014 Microsoft Corporation. All rights reserved.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Microsoft, list Microsoft trademarks used in your white paper alphabetically are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Introduction

This is the third attempt at a Windows Phone Development for Absolute Beginners' series. Each time I've created it, I changed the content dramatically and this time is no exception. No "third attempt" would be possible without the enthusiastic support of those who enjoyed watching it and participated by following along. So THANK YOU for watching (or reading) and supporting what I do. I hope this edition lives up to the previous ones in terms of quality, content and your support.

This document is not intended to be a word-for-word transcription of the videos. Instead, it is formatted and edited for reading. I've found that the way I speak and the way I write are vastly different and without contextual / visual cues, were you to read a transcript of my speech you would think I were illiterate.

Furthermore, I hastily put this together so I'm confident that it will contain mistakes. Many mistakes. Please forgive me. I did the best I could given the tight time frame and limited budget to give you at least *something* you could read.

This document is provided to help those for whom English is not their primary language, for those who have difficulty in hearing, and for those who want a quick reference of the content we cover in this course without having to re-watch large portions to find the content you want to review.

This series is a major departure from the previous series, Windows Phone 8 Development for Absolute Beginners for several reasons. First, in the previous version we had a hard dependency on the Coding4Fun Windows Phone Toolkit that Clint Rutkas developed. Since he moved on to other roles at Microsoft, to my knowledge it doesn't have a new owner, or at the very least, it wouldn't be ready by the time we needed to prepare this series. Second, I decided I wanted to greatly simplify the examples and slow down the training. I emphasized to all involved that the intent of this is to be watched immediately after the C# Fundamentals for Absolute Beginners series, and as such, we cannot expect beginner developers to go from crawling yesterday to running a marathon today. Many people were able to follow along at first, but we quickly lost them as the material became more complicated and demanding. Therefore, I slow down the pace of this series and focus more attention on fewer concepts in hopes that we "leave no developer behind", so to speak. Third, I wanted to create more exercise apps, but keep them smaller and "self-contained". If you got lost on one video, I didn't want you to quit!

While I take a moment at the end of this series to thank these people, I wanted to do it up front as well. This series would not be possible without the support of the Phone Developer Platform team, not the least of which is Larry Lieberman who "green lighted" these three series. I thank Matthias Shapiro for his management and technical advice throughout this series. I probably

asked him 100 questions and he patiently explained everything and cleared many hurdles out of the way for me. Thank you!

I know that the Channel9 folks have a large hand in the production and publishing of the video content, so thanks to Golnaz Alibeigi and others who have helped me with the technical aspects of publishing the content. I appreciate Clint Rutkas who helped me on the previous version of this series. Furthermore, I continue to thank Dan Fernandez for trusting me and working with me for so many years. And a special thanks to former *Microsoftie* Daryll McDade who introduced us. I owe you both a debt of gratitude.

Warm Regards,

A handwritten signature in black ink, appearing to read "Bob Tabor".

Bob Tabor

Contents

Introduction	2
Lesson 1: Series Introduction.....	5
Lesson 2: Exercise: Writing your First Windows Phone 8.1 App	7
Lesson 3: Introduction to XAML	20
Lesson 4: Understanding XAML Layout and Events	32
Lesson 5: Overview of the Common XAML Controls.....	44
Lesson 6: Themes and Styles XAML.....	67
Lesson 7: Understanding the Navigation Model	73
Lesson 8: Working with the package.appxmanifest	79
Lesson 9: Exercise: Tip Calculator	88
Lesson 10: Exercise: Tip Calculator as a Universal App	103
Lesson 11: Working with the Windows Phone 8.1 Emulator	113
Lesson 12: Understanding the App's Lifecycle and Managing State	129
Lesson 13: Working with the Web View App Template	143
Lesson 14: Exercise: Whack-a-Bob App.....	152
Lesson 15: Understanding the Hub App Template Overview and Navigation	177
Lesson 16: Understanding the Hub App Template's Sample Data Model	184
Lesson 17: Understanding Data Binding, Data Sources and Data Contexts	189
Lesson 18: Understanding MVVM: ObservableCollection<T> and INotifyPropertyChanged	197
Lesson 19: Understanding async and Awaitable Tasks	202
Lesson 20: Playing Video and Audio in a MediaElement Control.....	206
Lesson 21: Exercise: I Love Cupcakes App.....	210
Lesson 22: Storing and Retrieving Serialized Data	233
Lesson 23: Working with the Command Bar	242
Lesson 24: Binding to Commands and CommandParameters	248
Lesson 25: Advanced Binding with Value Converters	257
Lesson 26: Exercise: The Daily Rituals App	267
Lesson 27: Working with the Map Control and the Geolocation and GeoPosition Classes	304
Lesson 28: Working with Animations in XAML	320
Lesson 29: Exercise: The Map Notes App	329
Lesson 30: Series Conclusion	359

Lesson 1: Series Introduction

Hello, and welcome to this series on Windows Phone 8.1 development for Absolute Beginners. My name is Bob Tabor, and for the past 12 years I've created screen cast training like this for C#, .NET, Visual Studio and more. This is the third time I've been invited by Channel9 and the Windows Phone Developer Platform team to create this series.

Based on what we've learned from the previous two editions of this series, we've put together lessons focused on a gentle introduction for beginners - so if you're already an experienced developer, you may find that I go a bit slower. There are probably better resources available for you on Channel9 and on MSDN.

If you're just getting started, this series is for you. In this series, we'll start by learning about XAML, the declarative language used for laying out the user interface of Phone apps, as well as the Phone's API, the specific classes and methods you can call to interact with the phone's hardware capabilities and the tools in Visual Studio that make your job easier.

This time around, I'm focusing especially on the pre-built page and project templates available in Visual Studio to help you become productive quickly. The better you understand these templates and what they do, not only will you see basic implementations of important phone app features, but you'll also be equipped to leverage these templates to build your own cool apps.

We'll be building several apps in this series — my aim is to show you how to combine several ideas we discuss into a full-fledged app. I'll even demonstrate how to build Universal apps. In fact, almost everything you learn in this series will apply to building BOTH Phone and Windows Store apps. You can develop one code base and with a few tweaks to the user interface to account for the differences in the form factors and hardware capabilities, you can sell your app in two stores instead of just one.

My goal for this series is to give you the definitive head start — this will not be the only resource, the only instructional video series you'll ever need to build Phone and Windows apps, however, it will help you get you up to speed as quickly as possible so that you can move on to even more advanced ideas that dive deeper into the Phone's hardware.

Before you begin, there's really only requirements prior to beginning this series: First, that you already know C#. In fact, this series assumes that you finished my Absolute Beginners for C# Fundamentals series on Microsoft Virtual Academy. If you're not familiar with C# already, please put this series on the back burner for a day or two and watch that series first.

The second requirement is that you have the tooling installed in Visual Studio for the Phone 8.1. In other words, you should already be able to go to File > New Project > Templates > Visual C# > Store Apps > Windows Phone Apps and see all of the project templates. If you do

not have this installed, that is your first step. Unfortunately, I can't help you with this. There are too many versions and editions of Visual Studio and too many potential issues that arise during installation. If you need help, I would recommend that you visit:

[Http://msdn.Microsoft.com/forums](http://msdn.Microsoft.com/forums) and make sure you ask your question in Windows Phone Development: Tools for Windows Phone development.

If you have any questions specifically about what I talk about in this series, I'd be glad to answer your questions or help you if I can. There's a comments area beneath every video on Channel9 and I try to keep up with the comments. Also, you can tweet to me on Twitter @bobtabor and I'll help if I can. Please keep in mind ... you may be asking months or even years after I recorded a given video, so please try to be as descriptive as possible. Include the video title and the exact time marker in the video to help me help you.

Developing apps for the phone is a lot of fun and pretty easy once you get the hang of it. Make sure you pause and rewind the videos. Most importantly: follow along writing the code that I write to make sure these ideas get cemented in your mind. Following along and writing code is the best way to learn. I also will make the source code and the assets I use - icons, pictures, videos, audio files - so that you can check your work against mine.

Ok, so let's get started. We'll see you in the next lesson.

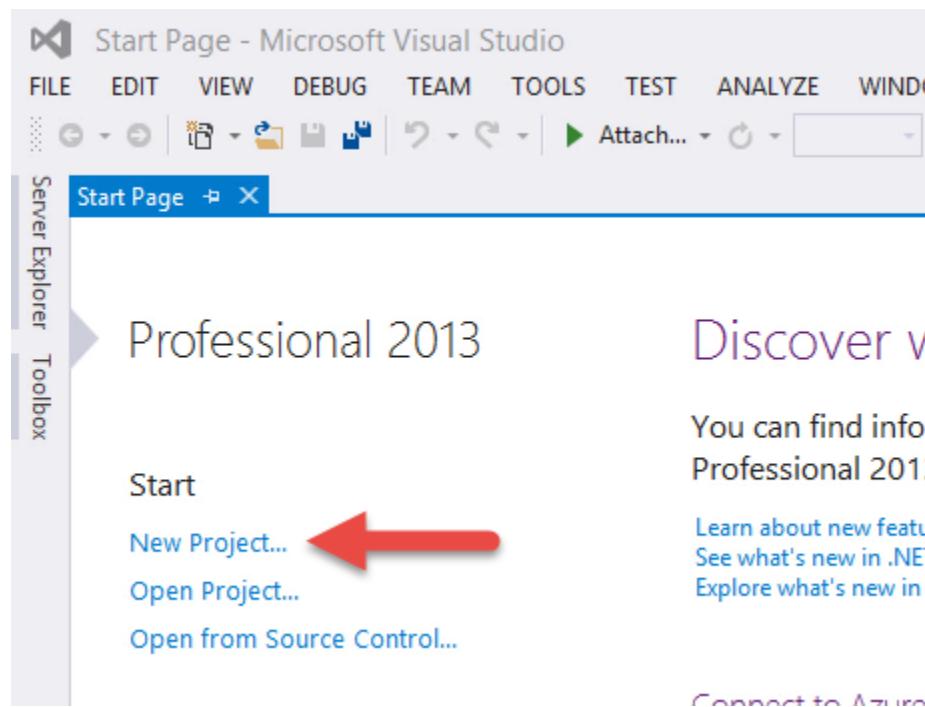
Lesson 2: Exercise: Writing your First Windows Phone 8.1 App

If you recall from the C# Fundamentals For Absolute Beginners Series on Microsoft Virtual Academy and Channel 9, near the end of that series I demonstrated how event work by creating two apps: an ASP.NET Web Forms app and a WPF, or rather, Windows Presentation Foundation app. I took the same basic steps to create both of those apps and they essentially perform the same operation; a button, you click on it, and it displayed a simple “Hello World” message in a label.

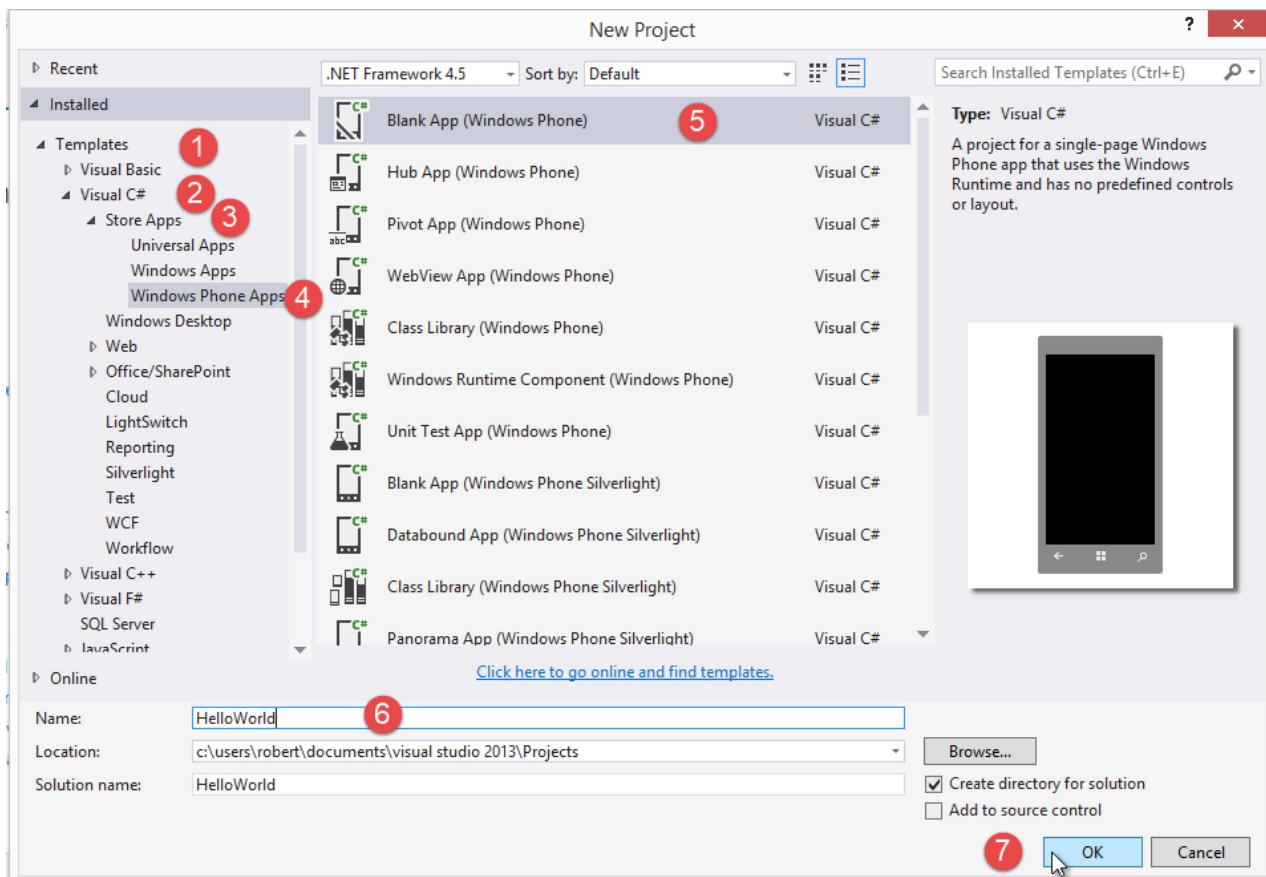
In this lesson, I want to re-create that example, except this time creating a simple Windows Phone app.

Note: Before we get started, make sure that you already have the Visual Studio 2013 Update 2 installed, which will include the Windows Phone 8.1 Tooling, like we talked about in the previous lesson. If you already have that installed, then we're ready to move forward.

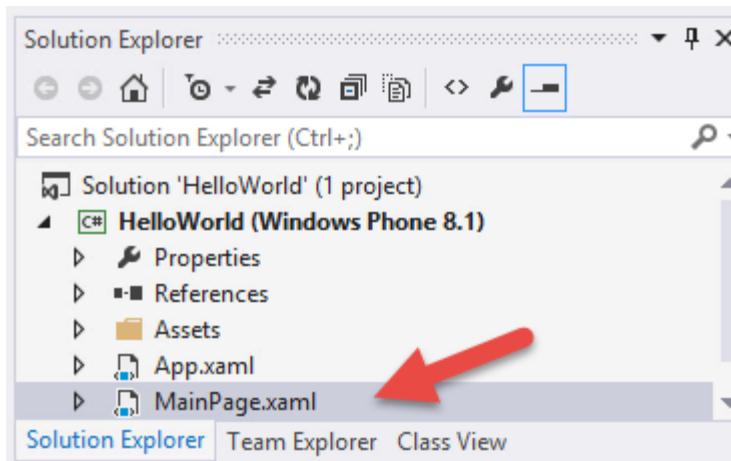
I'll begin by creating a new project:



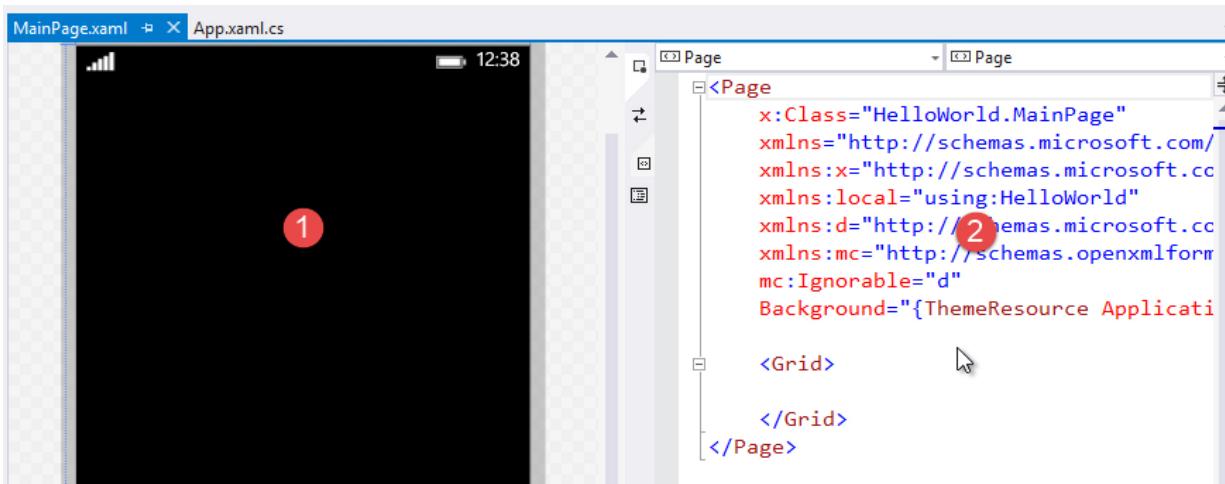
In the New Project Dialog, on the left select (1) Templates (2) Visual C# (3) Store Apps (4) Windows Phone Apps. In the center select the (5) Blank App (Windows Phone) project template. (6) Change the project name to: HelloWorld, then (7) Click OK:



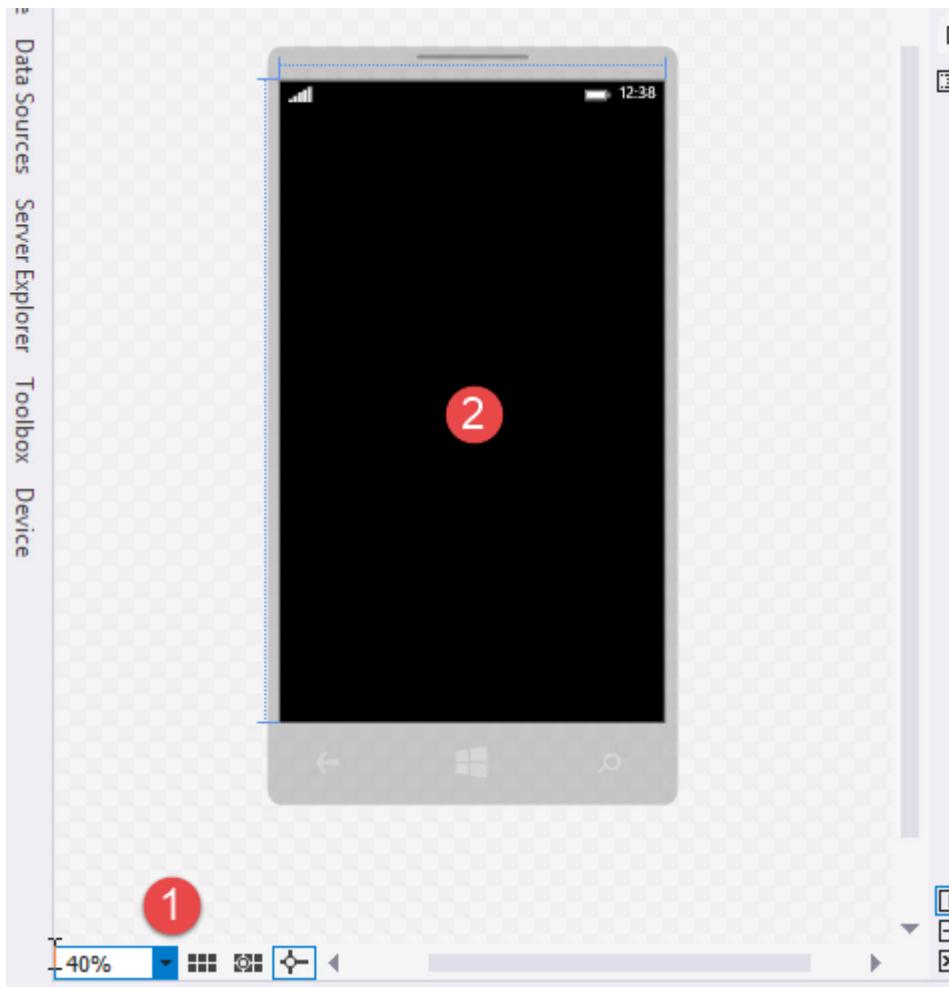
In the Solution Explorer, double-click the MainPage.xaml file:



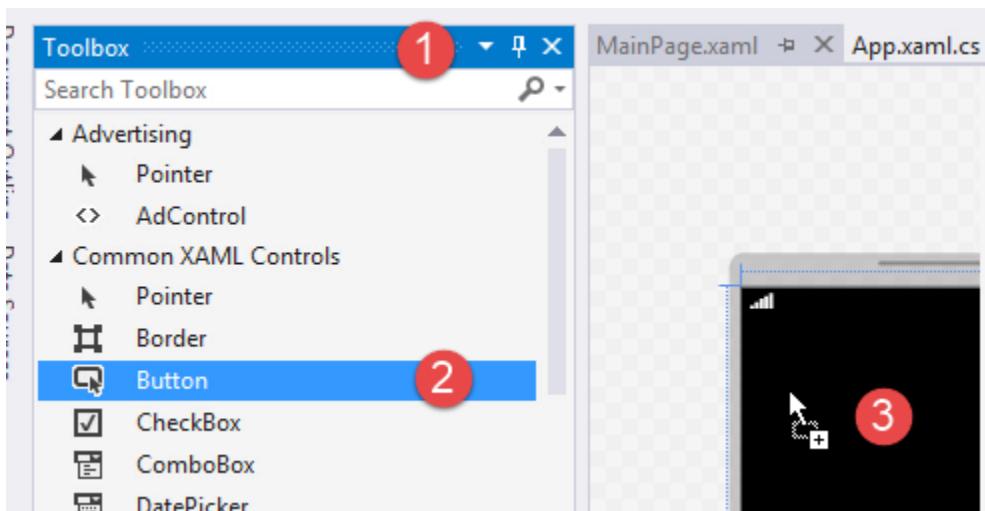
When MainPage.xaml loads into the main area of Visual Studio, you'll notice that it has a (1) preview pane on the left and the (2) XAML code editor view on the right:



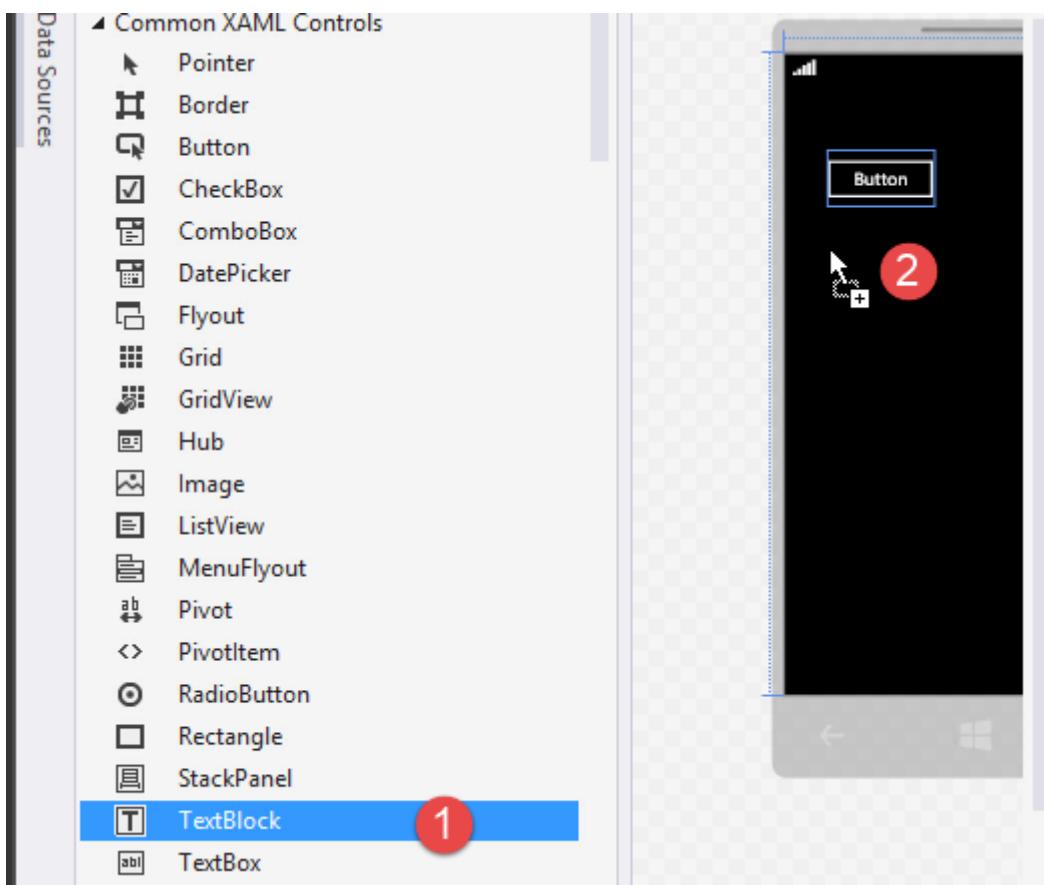
The preview pane displays the “chrome” of a Windows Phone” to help us visualize the changes we make to the user interface. Often in these lessons I’ll resize the default preview so that it can fit in the space available (so that we don’t have to scroll around to see our changes). To do this, (1) I’ll change the percentage to 40%, and (2) that should make the entire phone preview visible:



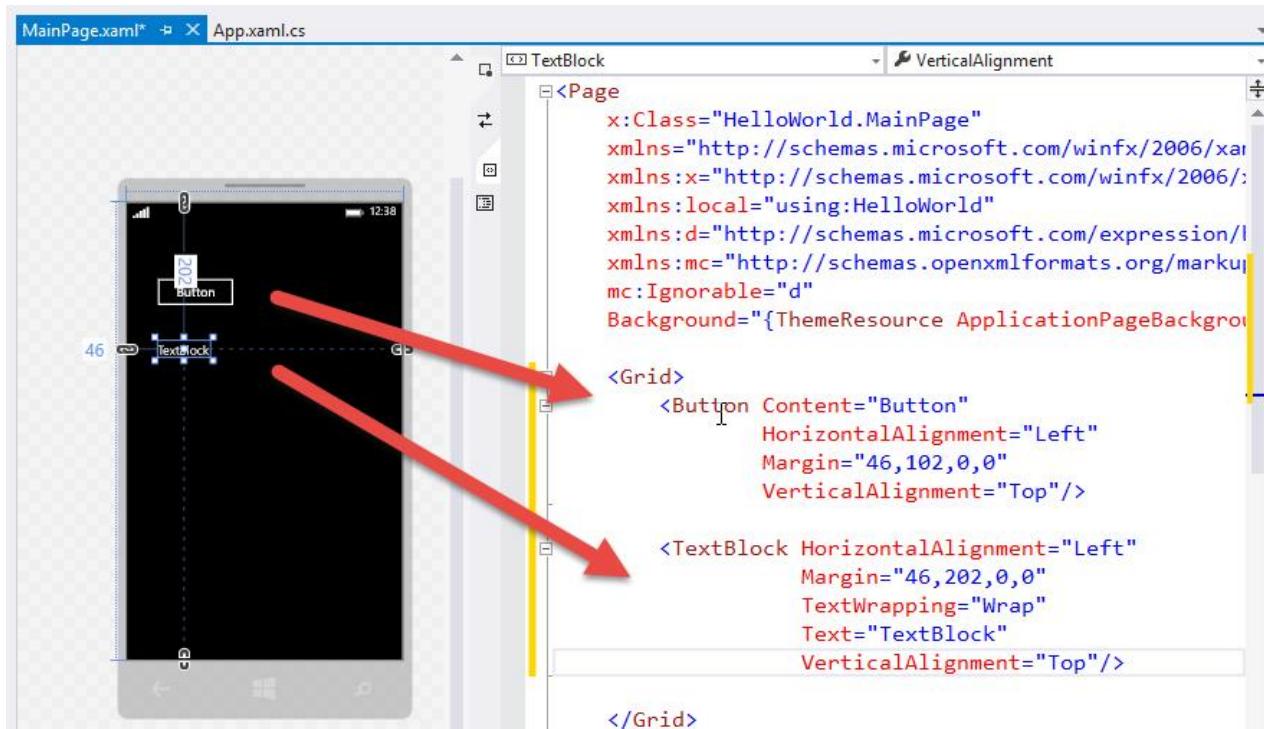
Next, I'll hover my mouse cursor over the Toolbox tab docked to the right. This will display the Toolbox. (1) I'll pin down (or rather, I'll disable auto-hide), then (2) drag a Button control (3) and drop it on the design surface:



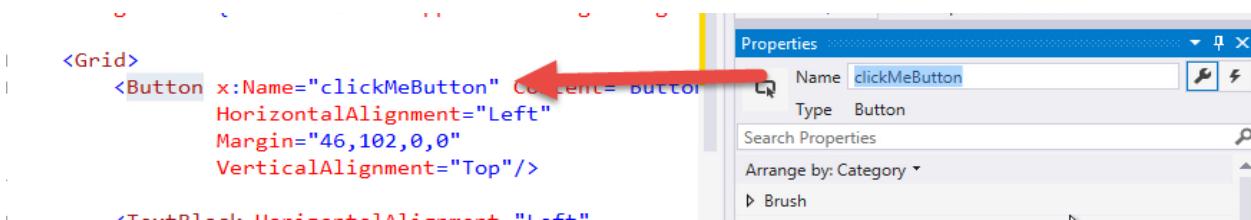
I'll repeat by (1) dragging a TextBlock control and (2) dropping it on the designer surface:



Notice that by dragging and dropping controls from the toolbox on to the designer surface, XAML is produced in the code editor along with some properties set such as default content or text, some sizing and positioning set, and so on:

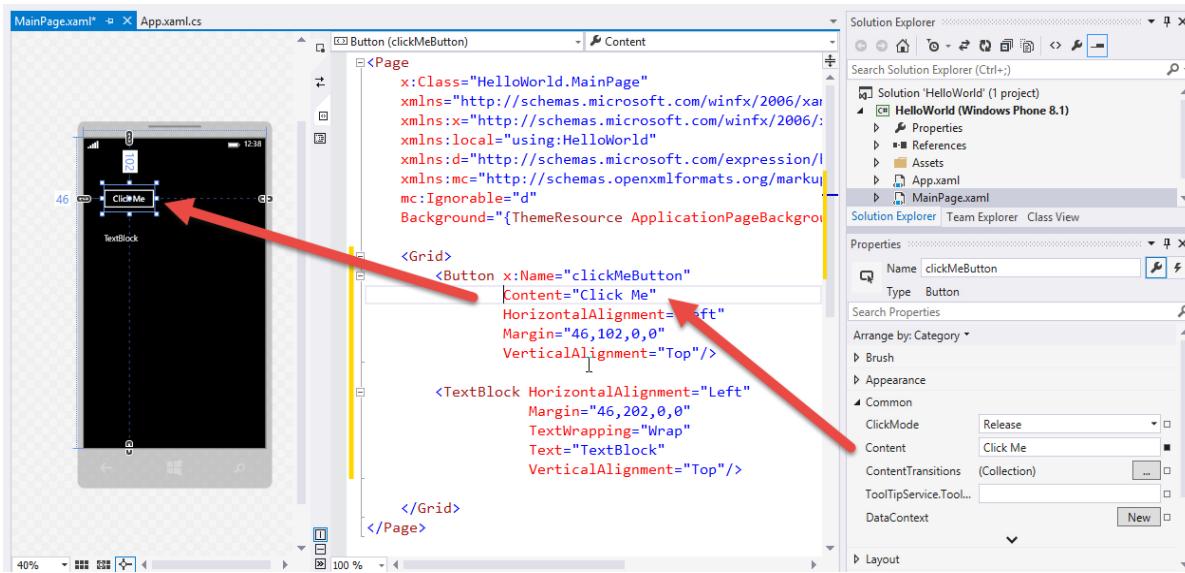


I can also affect the XAML that is generated by Visual Studio using the Properties window. For example, I'll put my mouse cursor into the Button control's XAML definition OR I'll select the button control on the design surface, then give the Button a name: clickMeButton:



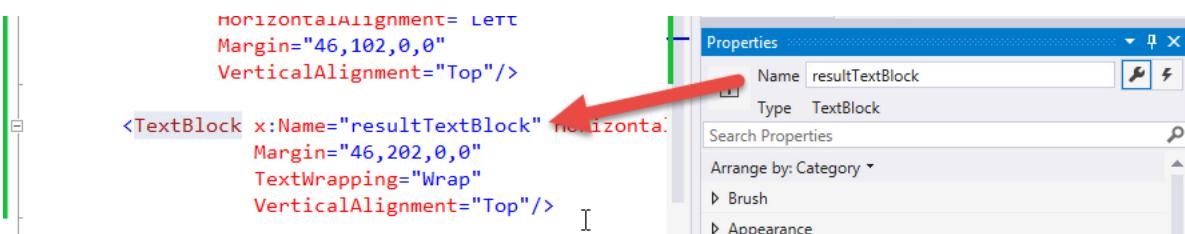
You'll notice that by giving the Button control a name in the Properties window, it generated a `x:Name` property in XAML and set that Name to "clickMeButton".

Likewise, when I modify the Content property of the Button in the Properties window setting the property to "Click Me", there's a change to both the XAML and to the preview pane:



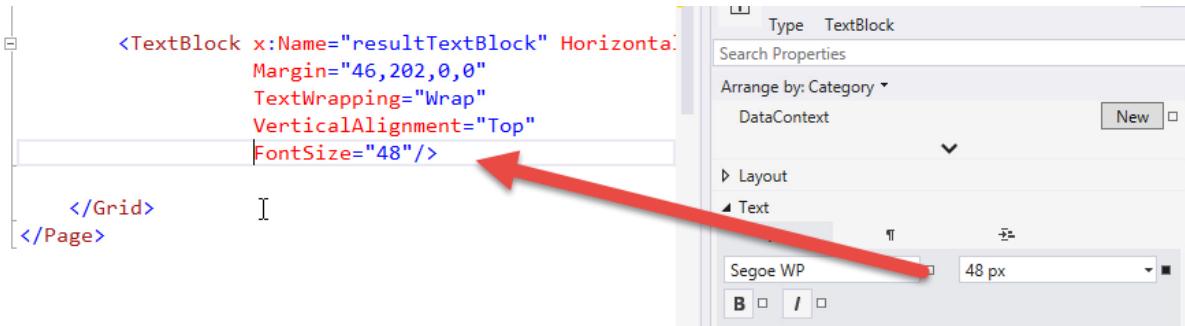
The key idea here is that these are three perspectives of the same fundamental object: a Button, a TextBlock, etc. so changes you make in any of the panels affect the others. Having said that, my personal belief is that you will become much more productive using the XAML text editor than the other panes. It will give you a full fidelity control over the properties that control appearance and positioning and as a result, I'll almost exclusively edit the XAML by hand in this series of lessons.

However, for now I'll make continue to use Properties window as a gentle introduction to building a user interface for our Phone app. I'll select text TextBlock in the XAML or preview pane, then change its Name in the Properties window to: resultTextBlock.

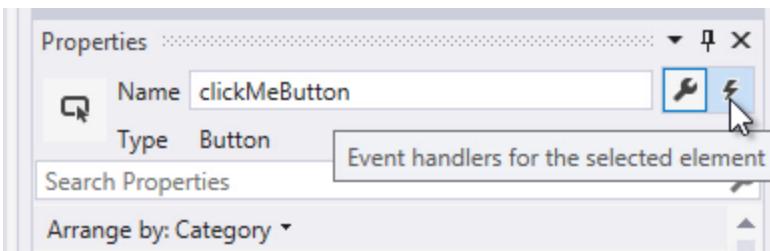


You may find it odd that, by default, XAML controls do not have a Name property already created. This is because, as we'll learn, XAML is a very concise language that is used by the Windows Phone API for layout. You only need a name for a control if you plan on accessing that control programmatically in C# code. We'll do that in a moment.

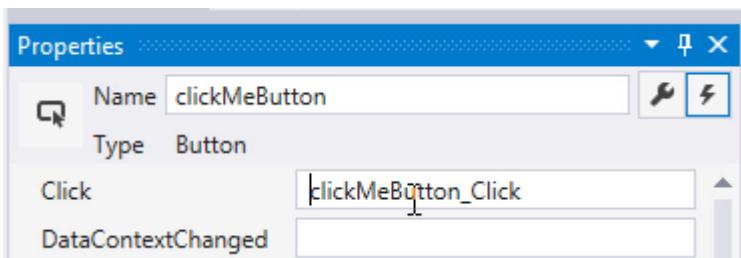
I'll also change the font size of the text that we'll fill into the TextBlock control. With the TextBlock still selected, I'll find the Text properties and set the font size to 48, which will in turn modify the FontSize attribute in the XAML code:



Next, I'll select the Button control again (whether in XAML or in the preview pane), then I'll select the lightning bolt icon next to the Name property to view the events that can be handled for this control:



Now you should be able to see all of the events for the Button. I'll double-click the white area / text box next to the Click event to automatically create a method called: `clickMebutton_Click` ...



... which will in turn create a method stub in the code behind for our `MainPage.xaml`. In other words, the `MainPage.xaml.cs` file opens in the main area of Visual Studio allowing us to write C# code in response to events that are triggered by the user (or in the case of other events, they could possibly be triggered by the life-cycle of the app itself).

```
 MainPage.xaml.cs * X MainPage.xaml* App.xaml.cs
[!HelloWorld]
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    // TODO: Prepare page for display here.

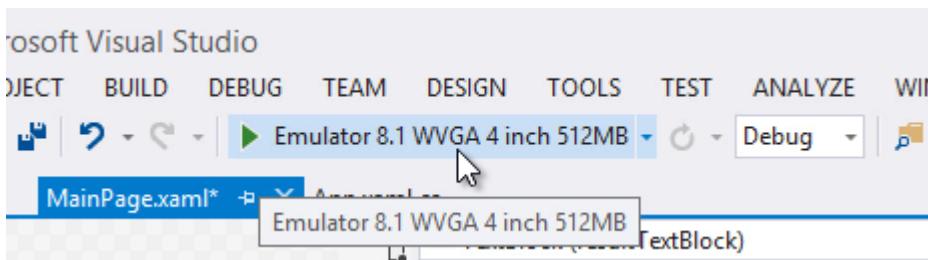
    // TODO: If your application contains multiple pages, ensure that you are
    // handling the hardware Back button by registering for the
    // Windows.Phone.UI.Input.HardwareButtons.BackPressed event.
    // If you are using the NavigationHelper provided by some templates,
    // this event is handled for you.
}

private void clickMeButton_Click(object sender, RoutedEventArgs e)
{
}
```

Inside of the method stub for the clickMeButton_Click event handler, I'll add one line of code to set the Text property of our resultTextBlock (the TextBlock we added earlier) to the string value "Hello World".

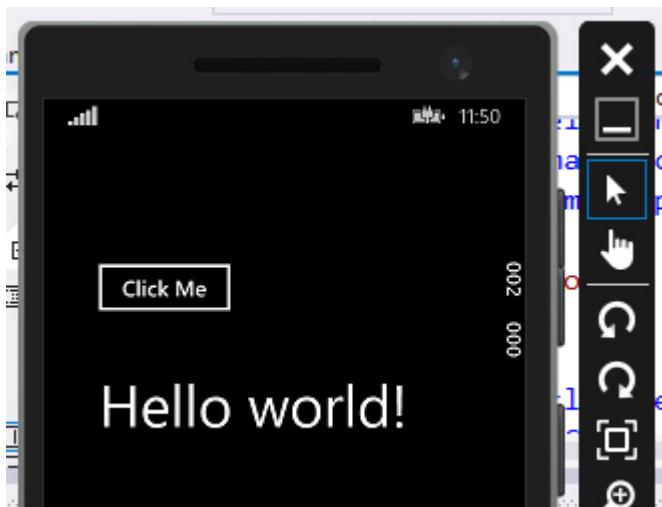
```
private void clickMeButton_Click(object sender, RoutedEventArgs e)
{
    resultTextBlock.Text = "Hello world!";
}
```

Now I'll attempt to run the app in debug mode. As we learned in the C# Fundamentals for Absolute Beginner's Series, while running our app in debug mode, we can set break points, view the values of variables, etc. When building a Widows Phone app, there's one more great feature: we can see our Phone app working without needing to deploy our app to an actual, physical Windows Phone. In this case, we'll select the green triangular Run button ... notice that we'll be launching our app in the "Emulator 8.1 WVGA 4 inch 512 MB". What does this mean? It specifies which physical phone we'll be emulating, including the screen size and memory on the phone. As you'll learn, there are several different emulators we can use that will allow us to see how our app performs on different hardware:

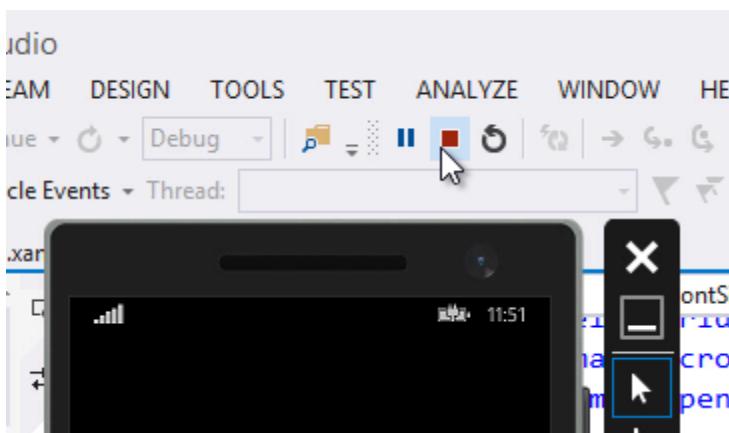


Once we run the app, we'll see it sitting inside of a software representation of a physical phone. For now, let's focus on the app itself, and we'll devote an entire lesson to learning more about the Emulator in an upcoming lesson.

If you use your mouse cursor to click the "Click Me" button (simulating the use of your finger to tap the button on the Phone's screen) it will display the message "Hello World!" in a large font:



To stop running the app in the Emulator, return to Visual Studio and press the red square Stop button:



Alternatively, you could shut down the Emulator using the large X icon in the toolbar to the right, however that is not necessary. You should get into the habit of leaving the Emulator running even when you're not debugging. There are two reasons for this. First, the Emulator is running as a virtual machine, and it takes time to "re-boot". Second, any data that your app saved to the phone between debugging runs will be lost when you shut down the Emulator.

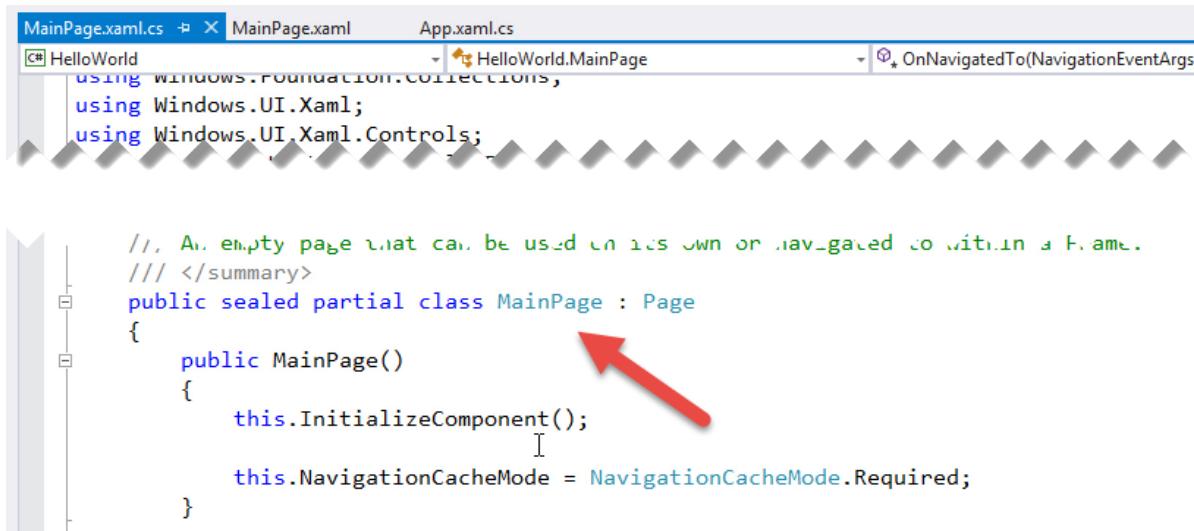
Hopefully you were able to see the relationship between the work we did in XAML and the finished app.

There are some key takeaways from this lesson as we're getting started. First of all, you're going to be able to leverage your existing knowledge of creating user interfaces and Visual Studio and apply that to creating Windows Phone Apps. It's easy and it's fun, and certainly there's a lot to learn here, but there's nothing to be intimidated about. We have the option of making changes in the most visual way by dragging and dropping items from the Toolbox onto the designer surface. We see how that created the XAML code, which looks very much like HTML. We'll learn more about this mysterious XAML syntax in the next lesson. We can also make changes in the properties window to modify the various attributes of the controls that we see on screen.

Furthermore, just like ASP.NET Web Forms and the Windows Presentation Foundation apps we created, every "screen", or rather, every Page has a code behind that's associated with it, where we can write event handler code to perform operations whenever the user interacts with our application in a given way.

The phone API provides this rich collection of controls and other options, as well as classes perform various operations or allow us to retrieve values produced by the Phone's sensors. As developers we can tap into that API to enable powerful functionality in our apps. The API is similar to working with Windows applications in so much that you have classes with properties and methods and events. Some of those affect the life-cycle of the app, some affect the visual qualities, like the visual controls or the layout of the app, and then some affect the navigation from page to page, which we haven't seen yet, but we will discuss in a later lesson. We'll talk about those in detail in just a little bit and they'll help us to build more complex and complex applications.

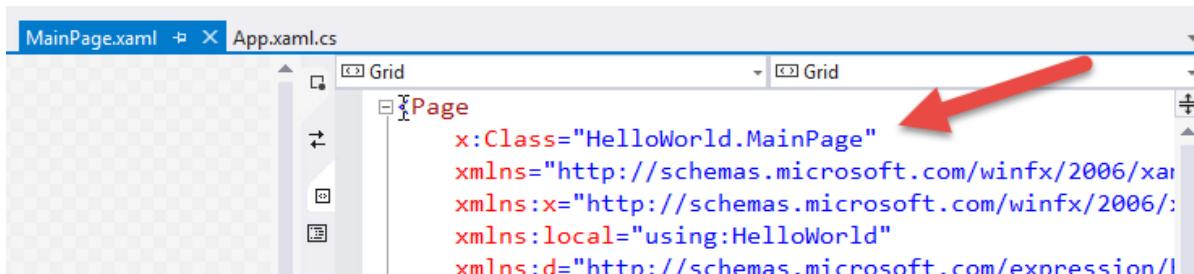
The third takeaway is that, like the Windows Presentation Foundation app example from the C# Fundamentals for Absolute Beginners Series, Phone apps are composed of a combination of both XAML and C# syntax. We can see there's a relationship between the MainPage.xaml and the MainPage.xaml.cs files because they're named similarly, and they're also visually related in the Solution Explorer. As we'll learn, these two files actually represent two parts of a whole. In the MainPage.xamlcs file, notice that this class is named MainPage:



```
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

/// An empty page that can be used in its own or navigated to within a Frame.
/// </summary>
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        this.NavigationCacheMode = NavigationCacheMode.Required;
    }
}
```

And if we go to the MainPage.xaml, we can see that this Page, too, is a class called "MainPage":



Because they two parts that are compiled together to create one “idea”, one “page” in our app. We’ll learn a little bit more about that in the next lesson.

Finally, we ran our app in debug mode inside of the Phone Emulator. The Emulator provides a fantastic way to develop and test your apps without having to deploy it to a physical device every single time you want to test some small change in your app. Furthermore, we can use techniques that we learned about in the C# Fundamental Series to set break points and to debug our app in real time as it's running. There's so much more that we can learn about the Emulator, it's various options, that we're going to devote an entire lesson to just that.

Lesson 3: Introduction to XAML

In this lesson, I want to talk about the XAML syntax we wrote in our first pass at the HelloWorld app. Hopefully you could see how the XAML we wrote impacted what we saw in the Phone preview pane. It's relatively easy to figure out the absolute basics of XAML just by looking at it, however I want to point out some of its features and functions that may not be obvious at first glance.

At a high level, our game plan in this lesson:

- (1) We'll talk about the purpose and nature of XAML, comparing it to C#
- (2) We'll talk about the special features of XAML ... little hidden features of the language that may not be obvious by just staring at it

My aim is by the end of this lesson you'll have enough knowledge that you can look at the XAML we write in the remainder of this series and be able to take a pretty good guess at what it's doing before I even try to explain what it does.

What is XAML?

In the previous lesson, I made a passing remark about XAML and how it looks similar to HTML. That's no accident.

XAML is really just XML, the eXtensible Markup Language. I'll explain that relationship in a moment, but at a higher level, XML looks like HTML insomuch that they share a common ancestry.

Whereas HTML is specific to structuring a web page document, XML is more generic. By "generic" I mean that you can use it for any purpose you devise and you can define the names of the elements and 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 meant to work together. To use XML, you define a schema, which declares the proper names of elements and their attributes. A schema is like a contract. Everyone agrees — both the producer of the XML and the consumer of the XML to write and read XML to conform to those rules, they'll abide by that contract. Now, they can communicate with each other. So, a schema is an important part of XML. Keep that in mind ... we'll come back to that in a moment.

XAML is a special usage of XML. Obviously, we see that, at least in this case, XAML has something to do with defining a user interface in our Phone's interface. So in that regard, it feels very much like HTML. But there's a big difference ... XAML is actually used to create instances of classes and set the values of the properties. So, for example, in the previous lesson we defined a Button control in XAML:

```
<Grid>

    <Button Content="Click Me"
        HorizontalAlignment="Left"
        Margin="10,100,0,0"
        VerticalAlignment="Top"
        Click="Button_Click"
        Background="Red"
        Width="200"
        Height="100"
        />

    <TextBlock x:Name="messageTextBlock"
        HorizontalAlignment="Left"
        Margin="10,200,0,0"
        TextWrapping="Wrap"
        Text=""
```

```
    VerticalAlignment="Top"  
    FontSize="48" />  
  
</Grid>
```

Comment out button:

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

Add:

```
<Grid Name="myLayoutGrid">  
  
protected override void OnNavigatedTo(NavigationEventArgs e)  
{  
    // TODO: Prepare page for display here.  
  
    Button myButton = new Button();  
    myButton.Name = "clickMeButton";  
    myButton.Content = "Click Me";  
    myButton.Width = 200;  
    myButton.Height = 100;  
    myButton.Margin = new Thickness(46, 102, 0, 0);  
    myButton.HorizontalAlignment = Windows.UI.Xaml.HorizontalAlignment.Left;  
    myButton.VerticalAlignment = Windows.UI.Xaml.VerticalAlignment.Top;  
    myButton.Background = new SolidColorBrush(Colors.Red);  
    myButton.Click += clickMeButton_Click;  
  
    myLayoutGrid.Children.Add(myButton);  
  
}
```

I've added this C# code in the `OnNavigatedTo()` method of my `MainPage` class. I'll talk about the relationship between the `MainPage.xaml` and `MainPage.xaml.cs` in just a moment, but we've already seen how we can define behavior by writing procedural C# code in the `MainPage.xaml.cs` file. Here, I'm merely writing code that will execute as soon as a new instance of the `MainPage` class is created by writing the code in the constructor of that class.

The important take away is this: XAML is simply a way to create instances of classes and set those objects' properties in a much more simplified, succinct syntax. What took us 10 lines of

C# code we were able to accomplish in just one line of XAML (even if I did separate it on to different lines in my editor, it's still MUCH SHORTER than it would have been had I used C# to create my objects.

Furthermore, using XAML I have this immediate feedback in the Phone preview pane. I can see the impact of my changes instantly. In the case of the procedural C# code I wrote, I would have to run the app each time I wanted to see how my tweaks to the code actually worked.

Introducing Type Converters

If you have a keen eye, you might notice the difference in the XAML and C# versions when it comes to the HorizontalAlignment attribute / property ...

If you tried:

```
myButton.HorizontalAlignment = "Left";
```

... you would get a compilation error. The XAML parser will perform a conversion to turn the string value "Left" into the enumeration value Windows.UI.Xaml.HorizontalAlignment.Left through the use of a Type Converter.

A Type Converter is a class that can translate from a string value into a strong type — there are several of these built into the Phone API that we'll use throughout this series. In this example, the HorizontalAlignment property, when it was developed by Microsoft's developers, was marked with a special attribute in the source code which signals to the XAML parser to run the string value through a type converter method to try and match the literal string "Left" with the enumeration value Windows.UI.Xaml.HorizontalAlignment.Left.

Just for fun, take a look at what happens when you attempt to misspell "Left":

(In the XAML misspell 'Left')

... you'll get a compilation error because the Type Converter can't find an exact match that it can convert into the enumeration value Windows.UI.Xaml.HorizontalAlignment.Left.

So, the first characteristic of XAML is that it is a succinct means of creating instances of classes. In the context of building a Phone application, it is used to create instances of user interface elements, however XAML is not just a user interface technology — it could be used for other purposes in other technologies.

Understanding XAML Namespace Declarations

Next, let's talk about all that XAML code at the very top of the MainPage.xaml file ... we ignored it until now.

At the very top of the file, we see the following:

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

While you're looking this over, remember what I said a moment ago -- about schemas being a part of XML. If that's the case, then where does this XAML promise to adhere to a schema?

See lines 3 through 7 ... there are five schemas this MainPage.xaml is promising to adhere to. Each of them are defined with the xmlns attribute. The first xmlns defined in line 3 is the default namespace -- in other words, there's no colon and no word after the colon like you see in lines 4 through 7.

The rest of the namespaces in lines 4 through 7 will use name / colon combination. So, just to be clear ... the :x or :phone is the NAMESPACE, that is associated with a SCHEMA (what we've called a contract). Each element and attribute in the rest of this MainPage.xaml MUST ADHERE TO AT LEAST ONE OF THESE SCHEMA's, otherwise the document is said to be invalid. In other words, if there's an element or attribute expressed in this XAML file that is not defined in one of these namespaces, then there's no guarantees that the compiler -- the program that will parse through our source code and create an executable that will run on the Phone -- the compiler will not be able to understand how to carry out that particular instruction.

So, in this example:

```
<TextBlock x:Name="resultTextBlock"
    HorizontalAlignment="Left"
    Margin="46,202,0,0"
    TextWrapping="Wrap"
    VerticalAlignment="Top"
```

FontSize="48"/>

We would expect the TextBlock element and attribute Background to be part of the default schema corresponding with the default namespace defined at the location in line 3.

However, x:Name is part of the schema corresponding with the x: namespace defined at the location in line 4.

I have a bright idea ... let's try to navigate to default namespace to learn more about what makes up a namespace in the first place:

<http://schemas.microsoft.com/winfx/2006/xaml/presentation>

Note: This will fail when you try it!

What?! The schema doesn't actually exist at that URL! That's because the schema is not published in the sense that you can go to that URL and view it. Instead, a schema is simply a unique name, similar to how we used Namespaces in C# to identify two classes that may have the same name — the schema (and therefore, the namespace in our XAML) keeps class names sorted out, kind of like a last name or surname.

This URL, or more properly, we should refer to it as a URI (Uniform Resource IDENTIFIER ... rather than LOCATOR) is used as a namespace identifier. The XML namespaces are instructions to the various applications that will parse through the XAML ... the Windows Runtime XAML parser will be seeking to turn it into executable code, while the Visual Studio and Blend designers will be seeking to turn it into a design-time experience.

So, the second XML Namespace defines a mapping, x: as belonging to this schema:

<http://schemas.microsoft.com/winfx/2006/xaml>

Therefore, any elements or attribute names that are preceded by the x: prefix means that they adhere to this second schema.

But, what's the difference? It's subtle, but the second schema defines the intrinsic rules for XAML in general. The first schema defines the contract / rules for Phone specific usage of XAML. In other words, the fact that we can work with the Grid, Button, TextBlock and the other XAML elements without using a prefix means that they are defined in the default namespace.

Line 5 is the local namespace ... this is the top most namespace for our project. So, any custom classes we create and want to reference, we can just append local: in front.

Lines 6 & 7 define namespaces and schemas that are used to allow Visual Studio's Phone Preview Pane on the left to

display properly. These instructions are ignored at runtime, which is what line 8 is doing -- when compiling the XAML code, ignore anything prefixed with :d.

Ok, I know there are some questions left unanswered ... we could spend a lot of time talking about the specifics, but the main takeaway is that this code at the very top of each XAML file you add to your phone project does have a purpose, it defines the rules that your XAML code must follow. You'll almost never need to modify this code, but if you remove it, you could potentially break your application. So, I would encourage you to not fiddle around with it unless you have a good reason to.

The final attribute sets a theme resource for the page. I'll talk about that in an upcoming lesson on Themes and Styles.

Understanding the relationship between the .xaml and .xaml.cs files

In Visual Studio's Solution Designer, you can see that the XAML files have an arrow which means that we can expand them to reveal a C# file by the same name, the only difference is that it has a .cs file name extension appended to the end. If you look at the .cs version of the file, you'll see that it defines a MainPage class, and furthermore, it defines it as a PARTIAL class.

```
public sealed partial class MainPage : Page
```

The other half of the equation is defined in lines 1 & 2 of the MainPage.xaml:

```
<Page  
    x:Class="HelloWorld.MainPage"  
    ...
```

While it doesn't use the term Partial like its procedural counterpart, it does indicate the relationship between the two.

Why is this important? This relationship means that the compiler will combine the output of the MainPage.xaml and the MainPage.xaml.cs files into a SINGLE CLASS. This means that they are two parts of a whole. That's an important concept ... that the XAML gets compiled into Intermediate Language just like the C# gets compiled into Intermediate Language and they are both partial implementations of a single class. This allows you to create an instance of a class in one file then use it in the other file, so to speak. This is what allows me to create an instance of the Button class called clickMeButton in XAML and then call its methods or set its properties in C#. We'll see more examples of this later in this lesson.

Understanding Default Properties

Since XAML is essentially an XML document, we can embed elements inside of other elements. We've already seen an example of this:

```
<Phone>
  <Grid ...>
    <Button ... />
    <TextBlock ... />
  </Grid>
</Page>
```

Here Page "contains" a Grid and the Grid "contains" a Button and a TextBlock. Or, perhaps more correctly in XAML parlance, that Page's Content property is set to Grid, and Grid's Children collection includes the Button and TextBlock. Depending on the type of control you're working with, the default property can be populated using this embedded style syntax. So, you could do this:

```
<TextBlock Content="Click Me" ... />
```

... or this ...

```
<Button HorizontalAlignment="Left"
        Margin="246,102,0,0"
        VerticalAlignment="Top">
  Hi
</Button>
```

... since the Content property is the default property of the Button class.

6. Understanding Complex Properties and the Property Element Syntax

In some cases, merely setting attribute values masks the complexity of what's going on behind the scenes. A good example of this is setting Background="Red". We've already seen this procedurally in C# -- to accomplish the same thing:

```
myButton.Background = new SolidColorBrush(Colors.Red);
```

... we have to create a new instance of a SolidColorBrush and pass in an enumerated Colors value. This is another great example of a property type converter that we learned about earlier in this lesson. But some attributes are simply too complex to be represented as attributes.

When a properties is not easily represented as a XAML attribute, it's referred to as a "complex property". To demonstrate this, first I'm going to remove the Background="Red" attribute from

the Button, remove "Hello World!" as the default property, and add it back with a Content="Hello World!" attribute:

Next, in the Properties pane, I'm going to set the Background property to a linear gradient brush.

- (1) I select the Brush property to reveal the Brush editor.
- (2) I make sure that Background is selected.
- (3) I select the Linear Brush tab (middle tab).
- (4) I move the selector all the way to the upper-right hand side of the color editor.

I should now see the following in the Phone Preview pane:

... but more importantly, let's look at the XAML that was generated by the Brush editor:

The XAML required to create that background cannot be easily set in a simple literal string like before when we simply used the word "Red". Instead, notice how the Background property is broken out into its own element:

```
<Button ... >
<Button.Background>
...
</Button.Background>
</Button>
```

This is called "property element" syntax and is in the form <Control.Property>.

A good example is the LinearGradientBrush. The term "brush" means that we're working with an object that represents a color or colors. Think of "brush" like a paint brush ... this particular paint brush will create a gradient that is linear — the color will change from top to bottom or left to right. Now, admittedly, you would NEVER want to do what I'm doing in this code example because it goes against the aesthetic of all Windows Phone 8 applications. But, let's pretend for now that we're expressing our individuality by using a gradient color as the background color for a Button.

As you can see (below), if we want to define a LinearGradientBrush, we have to supply a lot of information in order to render the brush correctly ... the colors, at what point that color should break into the next color, etc. The LinearGradientBrush has a collection of GradientStop objects which define the colors and their positions in the gradient (i.e., their "Offset").

However, the XAML representing the LinearGradientBrush in the code snippet above is actually SHORTENED automatically by Visual Studio. Here's what it should be:

```

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

```

Notice how the `<LinearGradientBrush.GradientStops>` and `<GradientStopCollection>` elements were omitted? This is done for conciseness and compactness and is made possible by an intelligent XAML parser. First of all, the `GradientStops` property is the default property for the `LinearGradientBrush`. Next, `GradientStops` is of type `GradientStopCollection` and implements `IList<T>`, the `T` in this case would be of type `GradientStop`. Given that, it is possible for the XAML parser to deduce that the only thing that could be nested inside the `<LinearGradientBrush ... />` is one or more instances of `GradientBrush`, each being implicitly `.Add()`'ed to the `GradientStopCollection`.

So the moral of the story is that XAML allows us to create instances of classes declaratively, and we have a granular fidelity of control to design user interface elements. Even so, the XAML parser is intelligent and doesn't require us to include redundant code — as long as it has enough information to create the object graph correctly.

Recap

To recap, we've learned about the syntax of XAML. Most of XAML is pretty straightforward, but there are a few things that are not quite as obvious.

(1) XAML is simply an implementation of XML, and relies heavily on schemas and namespaces to adhere to "contracts" so that different applications can create, interpret, display or compile the XAML code.

(2) The purpose of XAML is to allow for a compact, concise syntax to create instances of classes and set their properties. We compared the procedural version of a button created in C# versus one created declaratively in XAML and saw how much less code was required.

(3) XAML requires less code due to its built-in features like property type converters which allows a simple string value be converted into an instance of a class.

- (4) For more complex property settings, XAML offers property element syntax which harnesses the intelligent XAML parser to rely on default properties and deduction to reduce the amount of XAML code required to express a design.
- (5) We learned about the embedded syntax style and the embedded nature of elements which suggests a relationships between visual elements. For example, the Phone contains a Grid for layout, which in turn contains input or other types of controls. These are represented with opening and closing elements representing containment or ownership.
- (6) We learned about default properties; each control has a default property which can be set using that same style of embedded syntax.

Next, let's learn more about the Grid for layout, we'll learn about XAML's attached property syntax and how events are wired up in the next lesson.

Lesson 4: Understanding XAML Layout and Events

In this lesson I want to talk about layout, or in other words, how controls are positioned and arranged on your app's user interface.

So, my game plan:

- (1) We'll start by talking about the two primary elements used in layout and positioning: the Grid element and StackPanel element.
- (2) With regards to the Grid, I'll talk about defining rows and columns and various sizing options and techniques you can use to fully unlock the power of the Grid
- (3) With regards to the StackPanel, we'll discuss how to change the orientation of items inside the StackPanel, and how to affect the alignment of items as well.
- (4) Finally, we'll talk about how event handlers are "wired up" from both XAML and in C#.

Understanding the Basics of Grids

The Grid element is used for laying out other controls ... it allows you to define rows and columns, and then each control can request which row and column they want to be placed inside of.

When you use the Blank App Template, you're provided very little layout guidance to get started.

Note: I typically don't like to talk about previous versions of things that are no longer relevant to beginners, however since the following ideas I'm about to talk about were so prevalent in articles and books I feel it is necessary to talk about this.

In the past, the Page templates provided a high level <Grid> element named "LayoutRoot" that created three distinct areas:

- (1) A very short top-most row for the app name
- (2) A second short row for the page's name
- (3) One very large row (the remainder of the space available) that contained another Grid called the "ContentPanel". The intent of the ContentPanel was that you would add the remainder of your XAML layout code there.

Keep that in mind because you may see articles and books based on previous versions of Phone tooling support in Visual Studio to the "ContentPanel".

However, in the tooling support for Phone 8.1, the Page template contains a single empty grid:

```
└─<Page  
    x:Class="XAMLLayoutAndEvents.MainPage"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:local="using:XAMLLayoutAndEvents"  
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
    mc:Ignorable="d"  
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">  
  
    └─<Grid>  
        ┌─</Grid>  
    └─</Page>
```

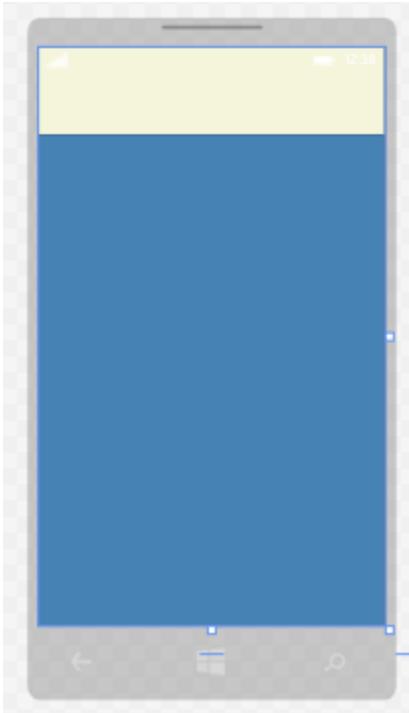
This Grid appears that there's no rows or columns defined. However, by default, there's always one RowDefinition and one ColumnDefinition even if it's not explicitly defined, and these take up the full vertical and horizontal space available to represent one large "cell" in the Grid. Any items placed between the opening and closing `<Grid></Grid>` elements are understood to be inside that single implicitly defined "cell".

Grid RowDefinitions and ColumnDefinitions and defining sizes

I'll create a quick example of a Grid that defines two rows to illustrate two primary ways of row height sizes (Project: RowDefinitions):

```
<Grid>  
    <Grid.RowDefinitions>  
        <RowDefinition Height="Auto" />  
        <RowDefinition Height="*" />  
    </Grid.RowDefinitions>  
  
    <Rectangle Height="100" Fill="Beige" Grid.Row="0" />  
    <Rectangle Grid.Row="1" Fill="SteelBlue" />  
  
</Grid>
```

The result:



First, notice how the two Rectangles place themselves inside of the two rows ... by using an attribute `Grid.Row="0"` and `Grid.Row="1"` respectively. You reference both rows and columns using a zero-based numbering scheme.

Second, notice the two different row heights ... `Height="Auto"` and `Height="*"`. There are three syntaxes that you can use to help persuade the sizing for each row and column. I used the term "persuade" intentionally. With XAML layout, heights and widths are relative and can be influenced by a number of factors. All of these factors are considered by the layout engine to determine the actual placement of items on the page.

For example, "Auto" means that the height for the row should be tall enough to accommodate all of the controls that are placed inside of it. If the tallest control (in this case, a Rectangle) is 100 pixels tall, then that's the actual height of the row. If it's only 50 pixels, then THAT is the height of the row. Therefore, "Auto" means the height is relative to the controls inside of the row.

The asterisk is known as "star sizing" and means that the height of the row should take up all the rest of the height available.

Here's a quick example of another way to use "star sizing" ... I created a project that has three rows defined in the ContentPanel. Notice the heights of each one (Project: StarSizing):

<Grid>

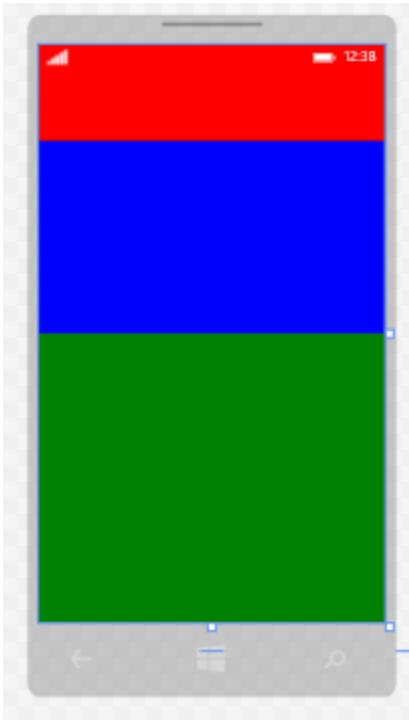
```

<Grid.RowDefinitions>
    <RowDefinition Height="1*" />
    <RowDefinition Height="2*" />
    <RowDefinition Height="3*" />
</Grid.RowDefinitions>

<Rectangle Fill="Red" Grid.Row="0" />
<Rectangle Fill="Blue" Grid.Row="1" />
<Rectangle Fill="Green" Grid.Row="2" />
</Grid>

```

This produces the following result:



Putting a number before the asterisk, I'm saying "Of all the space available, give me 1 or 2 or 3 shares of the remainder". Since the sum of all those rows adds up to six, each 1* is equivalent to 1/6th of the height available. Therefore, 3* would get half of the height available as depicted in the output of this example:

Besides Auto and star sizing, you can also specify widths and heights (as well as margins) in terms of pixels. In fact, when only numbers are present, it represents that number of pixels. Generally, it's not a good idea to use exact pixels in layouts for widths and heights because of the likelihood that screens -- even Windows Phone screens, can have different dimensions. Instead, it's preferable to use relative values like Auto and star sizing for layout.

One thing to note from this example is that control widths and heights are assumed to be 100% unless otherwise specified. That's why the rectangles take up the entire "cell" width. This is why in Lesson 2 the button first occupied the entire grid, and why I had to specify I wanted the button to be 200 x 100 pixels instead.

I want to also point out that a Grid can have a collection of ColumnDefinitions as you can see from this example app I created called GridsRowsAndColumns. Here we have a 3 by 3 grid:

```
<Page.Resources>
    <Style TargetType="TextBlock">
        <Setter Property="FontSize" Value="42" />
    </Style>
</Page.Resources>

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

    <TextBlock>1</TextBlock>
    <TextBlock Grid.Column="1">2</TextBlock>
    <TextBlock Grid.Column="2">3</TextBlock>

    <TextBlock Grid.Row="1">4</TextBlock>
    <TextBlock Grid.Row="1" Grid.Column="1">5</TextBlock>
    <TextBlock Grid.Row="1" Grid.Column="2">6</TextBlock>

    <TextBlock Grid.Row="2">7</TextBlock>
    <TextBlock Grid.Row="2" Grid.Column="1">8</TextBlock>
    <TextBlock Grid.Row="2" Grid.Column="2">9</TextBlock>

</Grid>
```

... and the result is a simple grid with a number in each "cell":



The other thing I want you to notice about this example is that when you don't specify a Grid.Row or Grid.Column, it is assumed to mean the first row (row 0) or first column (column 0). Relying on the defaults keeps your code more concise.

Grid cell Alignments and Margins

I have another example app called AlignmentAndMargins. This XAML:

```
<Grid>
    <Rectangle Fill="Blue"
        Height="100"
        Width="100"
        HorizontalAlignment="Left"
        VerticalAlignment="Top" />

    <Rectangle Fill="Red"
        Height="100"
        Width="100"
        HorizontalAlignment="Right"
        VerticalAlignment="Bottom" />

    <Rectangle Fill="Green"
```

```

    Height="100"
    Width="100"
    HorizontalAlignment="Center"
    VerticalAlignment="Center" />

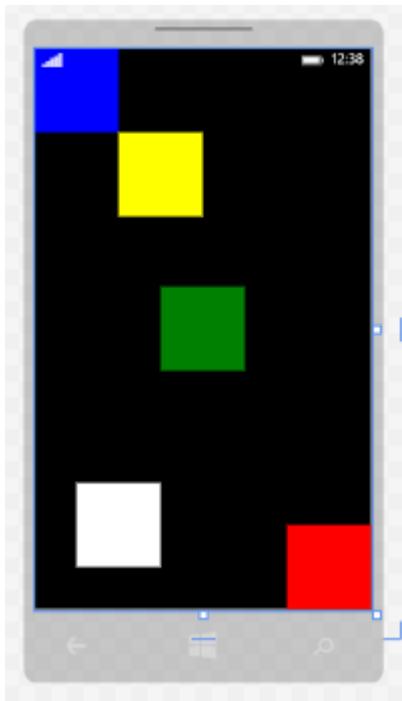
<Rectangle Fill="Yellow"
    Height="100"
    Width="100"
    HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="100,100"/>

<Rectangle Fill="White"
    Height="100"
    Width="100"
    HorizontalAlignment="Left"
    VerticalAlignment="Bottom"
    Margin="50,0,0,50"/>

</Grid>

```

Produces this result:



Most of this example should be obvious if you stare at it for a few moments, but there are several finer distinctions I want to make about Alignments and Margins. First, it points out how

VerticalAlignment and HorizontalAlignment work, even in a given Grid cell (and the same will hold true in a StackPanel as well). The __Alignment attributes PULL controls TOWARDS their boundaries. By contrast, the Margin attributes PUSH controls AWAY from their boundaries. The second observation is the odd way in which Margins are defined ... as a series of numeric values separated by commas. This convention was borrowed from Cascading Style Sheets. The numbers represent the margin pixel values in a clock-wise fashion starting from the left side. So,

Margin="10,20,30,40"

Means, 10 pixels from the left boundary, 20 pixels from the top boundary, 30 pixels from the right boundary, 40 pixels from the bottom boundary. In this case, the boundary is a single Grid cell (since the Content panel only has not defined any additional RowDefinitions and ColumnDefinitions).

A bit earlier I said that it's generally a better idea to use relative sizes like Auto and * to define heights and widths ... why, then, are margins are defined using pixels? Margins are expressed in exact pixels because they are usually just small values to provide spacing or padding between two relative values and can therefore be a fixed value without negatively impacting the overall layout of the page.

StackPanel basics

It will arrange your controls in a flow from top to bottom or from left to right depending on the Orientation property you set.

Here's an example of using A LOT of StackPanels to achieve a intricate layout:

```
<Grid>
<StackPanel Orientation="Horizontal" VerticalAlignment="Top">
    <Rectangle Width="200" Height="200" Fill="Bisque" />
    <StackPanel Orientation="Vertical">
        <StackPanel Orientation="Horizontal">
            <Rectangle Width="100" Height="100" Fill="Azure" />
            <StackPanel Orientation="Vertical">
                <Rectangle Width="100" Height="50" Fill="RosyBrown" />
                <Rectangle Width="100" Height="50" Fill="DarkCyan" />
            </StackPanel>
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <Rectangle Width="100" Height="100" Fill="Tomato" />
            <StackPanel Orientation="Vertical">
                <Rectangle Width="100" Height="50" Fill="BurlyWood" />
```

```

<Rectangle Width="100" Height="50" Fill="SaddleBrown" />
</StackPanel>
</StackPanel>
</StackPanel>
</StackPanel>
</Grid>

```

That XAML produces:



I would recommend taking a few moments to attempt to match up the StackPanels and the Rectangles to see how this is produced. The top-most StackPanel creates a “row” of content. The StackPanel’s height is dictated by the height of the largest item, which is the Bisque colored rectangle. I also set the VerticalAlignment=”top”, otherwise it will default to the middle of the Phone’s screen.

Since we’re stacking horizontally, I’ve added a StackPanel that will be positioned to the right of the Bisque colored rectangle and it will arrange items in a vertical fashion. Inside of that StackPanel I create two other StackPanels that are oriented horizontally. Thus, the intricacy continues until I have a series of rectangles that resemble something Frank Lloyd Wright would have been proud of.

As we continue throughout this series, I’ll use multiple StackPanels to organize the various XAML controls (elements) on screen. In fact, I’ve grown to the point where I prefer the flexibility of the StackPanel to the rigid nature of the Grid. I suppose that is a personal preference.

Understanding Events

If you watched the C# Fundamentals series on Channel9 or Microsoft Virtual Academy, in one of the last videos I talk about events. In the Windows Phone API, Pages and Controls raise events for key moments in their life cycle OR for interactions with the end user. In this series, we’ve already “wired up” the Click event of our “Click Me” button with a method that I called an “event handler”. When I use the term “event handler” I’m merely referring to a method that is associated with an event. I use the term “wired up” to mean that the event is tied to a

particular event handler method. Some events can be triggered by a user, like the Click event of a Button control. Other events happen during the lifetime of an event, like a Loaded event that occurs after the given Page or Control object has been instantiated by the Phone APIs Runtime engine.

There are two ways to "wire up" an event for a Page or Control to an event handler method. The first is to set it using the attribute syntax of XAML. We've already done this in our "Click Me" example:

If you'll recall, we typed:

```
Click="
```

And before we typed in the closing double-quotation mark, Intellisense asked if we wanted to select or create an event handler. We told it we wanted to create a new event handler, and Visual Studio named the event handler using the naming convention:

```
NameOfElement_EventName
```

... so in our case ...

```
clickMeButton_Click
```

Visual Studio also created a method stub in the code-behind for the XAML page, in our case, the MainPage.xaml.cs. Keep in mind that these are Visual Studio automations. We could have hit the escape key on the keyboard when Intellisense first popped up and typed that code in by hand.

A second way to associate an event with an event handler is to use the Properties window in Visual Studio.

(1) First, you must make sure you've selected the Control you want to edit by placing your mouse cursor in that element. The name of that element will appear in the Name field at the top letting you know the context of the settings below. In other words, any settings you make will be to the button as opposed to another control on the page.

(2) Click the lightning bolt icon. This will switch from a listing of Properties / attributes that can be changed in the Properties window to the Events that can be handled for this Control.

(3) Double-click on a particular textbox to create an association between that Control's event and an event handler. Double-clicking will automatically name the event and create a method stub for you in the code behind.

The third technique (that we'll use several times in this series) is to wire up the event handler in C# code. See line 35 below for an example:

The `+=` operator can be used in other contexts to mean "add and set" ... so:

```
x += 1;
```

... is the same as ...

```
x = x + 1;
```

The same is true here ... we want to "add and set" the Click event to one more event handler method. Yes, the Click event of myButton can be set to trigger the execution of MULTIPLE event handlers! Here we're saying "When the Click event is triggered, add the `clickMeButton_Click` event handler to the list of event handlers that should be executed." One Click event could trigger one or more methods executing. In our app, I would anticipate that only this one event handler, `clickMeButton_Click`, would execute.

You might wonder: why doesn't that line of code look like this:

```
myButton.Click += clickMeButton_Click();
```

Notice the open and closed parenthesis on the end of the `_Click`. Recall from the C# Fundamentals series ... the `()` is the method invocation operator. In other words, when we use `()` we are telling the Runtime to EXECUTE the method on that line of code IMMEDIATELY. That's NOT what we want in this instance. We only want to associate or point to the `clickMeButton_Click` method.

One other interesting note about event handler methods. With my addition of line 35 in the code above, I now have two events both pointing to the same event handler method `clickMeButton_Click`. That's perfectly legitimate. How, then, could we determine which button actually triggered the execution of the method?

If you look at the definition of the `clickMeButton_Click` method:

```
private void clickMeButton_Click(object sender,  
    RoutedEventArgs e)  
{  
}
```

The Windows Phone Runtime will pass along the `sender` as an input parameter. Since we could associate this event with any Control, the `sender` is of type `Object` (the type that virtually all data types in the .NET Framework ultimately derive from), and so one of the first things we'll need to do is do a few checks to determine the ACTUAL data type ("Are you a Button control?

Are you a Rectangle control?”) and then cast the Object to that specific data type. Once we cast the Object to a Button (for example) then we can access the Button's properties, etc.

The method signature in the code snippet above is typical methods in the Windows Phone API. In addition to the sender input parameter, there's a RoutedEventArgs type used for those events that pass along extra information. You'll see how these are used in advanced scenarios, however I don't believe we'll see them used in this series.

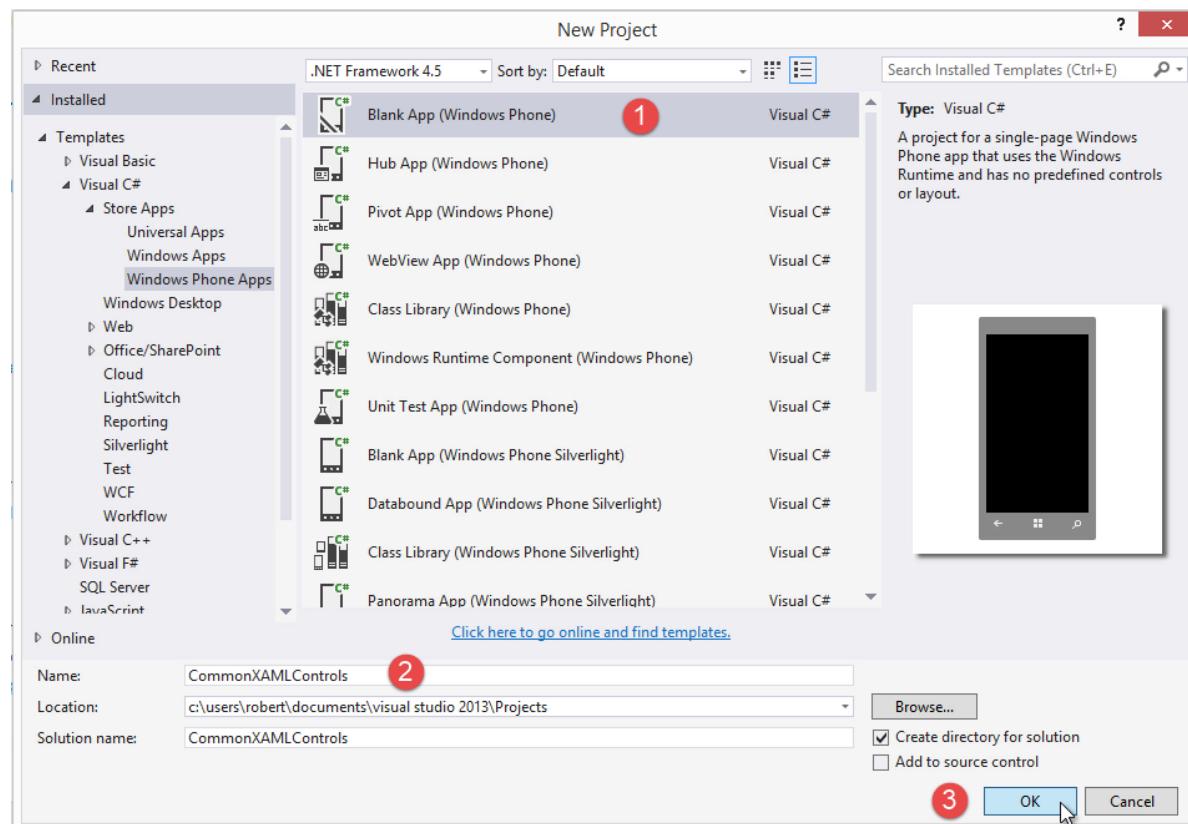
Recap

To recap, the big takeaway in this lesson was the ways in which we can influence the layout of Pages and Controls ... we looked at Grid and Stack layout. With regards to Grid layout we learned about the different ways to define the heights and widths of Rows and Columns (respectively) using Auto sizing, star sizing and pixel sizing. We talked about how VerticalAlignment and HorizontalAlignment pull controls towards boundaries while Margins push Controls away from boundaries. Then we talked about the different ways to wire up events to event handler methods, and the anatomy of an event handler method's input parameters.

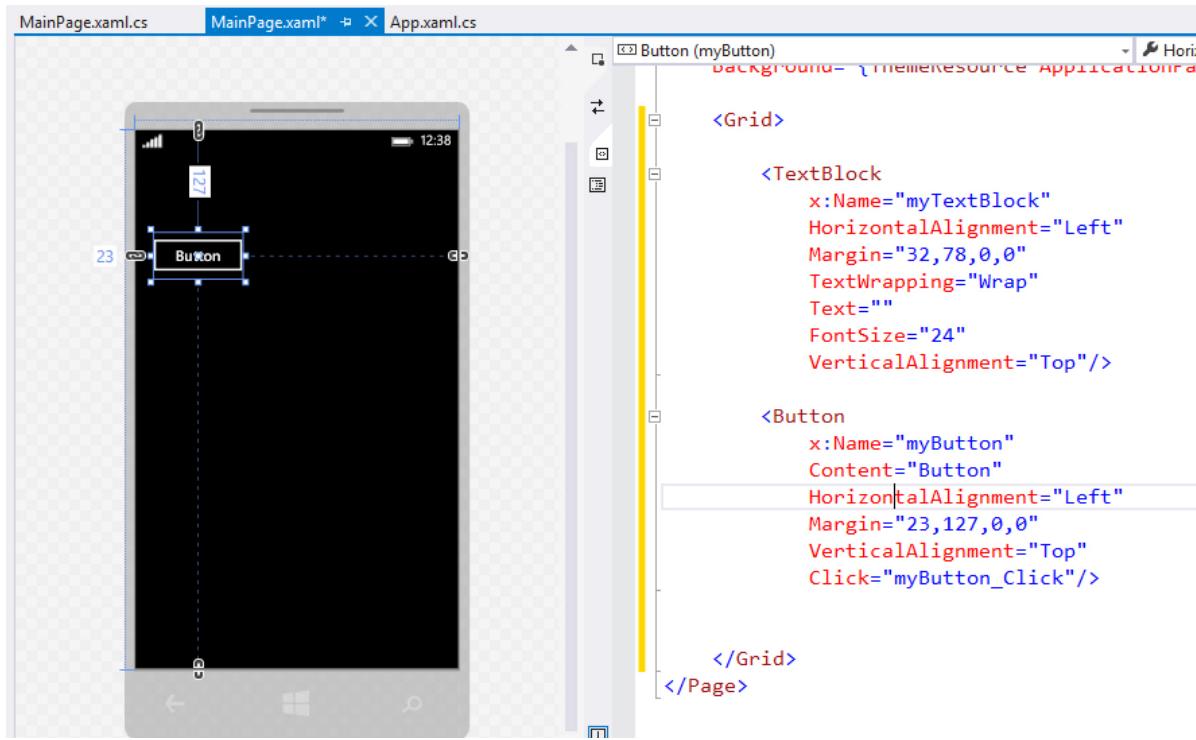
Lesson 5: Overview of the Common XAML Controls

In this lesson we'll examine many common XAML controls used for basic input and interaction in Phone apps. We're doing this as we prepare to build our first app of significance and just a lesson or two, a tip calculator. To do this, we're going to need to use different types of controls than what we've worked with up to this point. Up to now, we've only used the grid, the stack panel, the button and the text block. In this lesson we'll overview many common XAML controls. We'll continue to learn new XAML controls throughout the remainder of this series as well.

I'll start by creating a new (1) Blank App project template named (2) CommonXAMLControls and I'll (3) click the OK button to create the project.



I'll use a TextBlock and a Button control to retrieve and display values inputted in the controls we'll learn about. This will demonstrate how to retrieve values from these controls programmatically in C#.



I'll add a definition for a TextBox as follows:

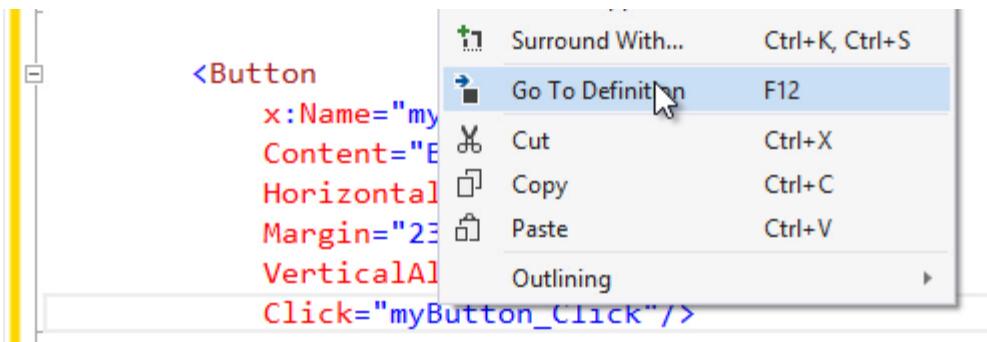
```

<TextBox
    x:Name="myTextBox"
    HorizontalAlignment="Left"
    Margin="32,197,0,0"
    TextWrapping="NoWrap"
    Text=""
    InputScope="TelephoneNumber"
    VerticalAlignment="Top"
    Width="120" />

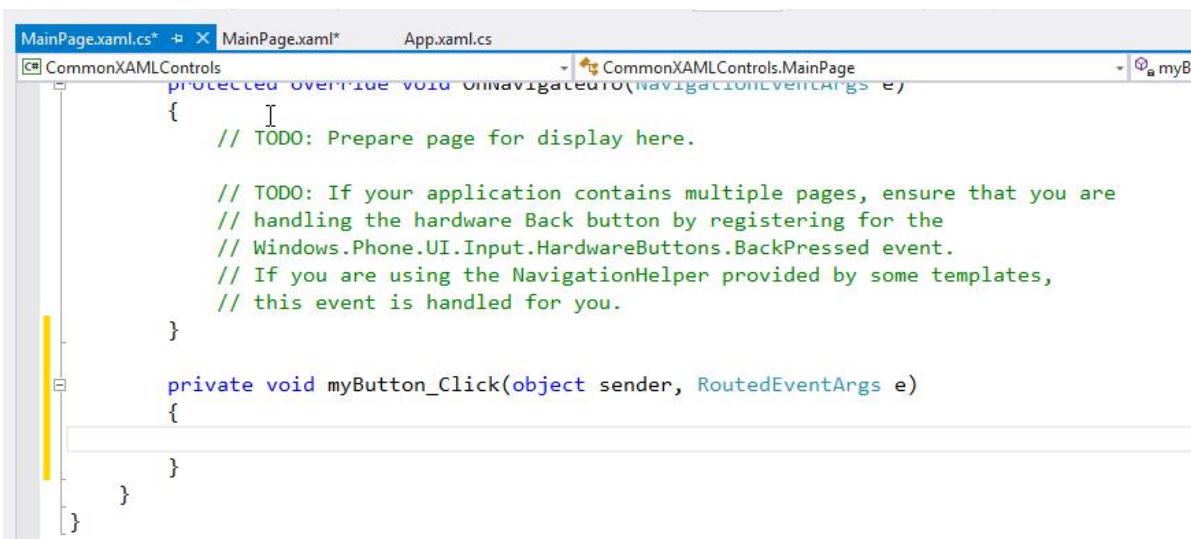
```

The TextBox will allow a user to type in text using an on-screen keyboard. The key to this example is the `InputScope="TelephoneNumber"`. As we'll see in a moment, changing the `InputScope` changes the types of information (or rather, the keyboard) that is displayed to the user.

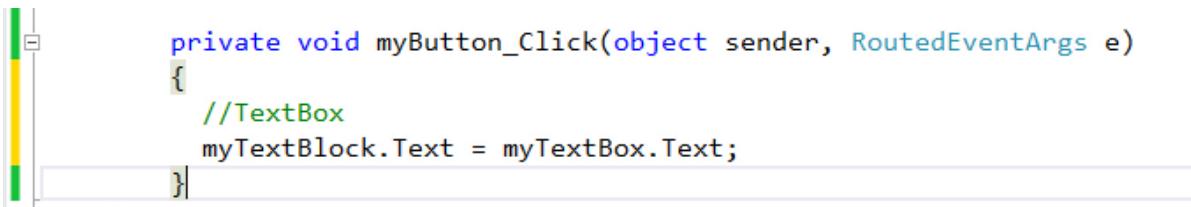
Note that `myButton`'s Click event is handled. We'll need to create an event handler method stub. To do this, I'll right-click the value "myButton_Click" and select "Go To Definition" from the context menu:



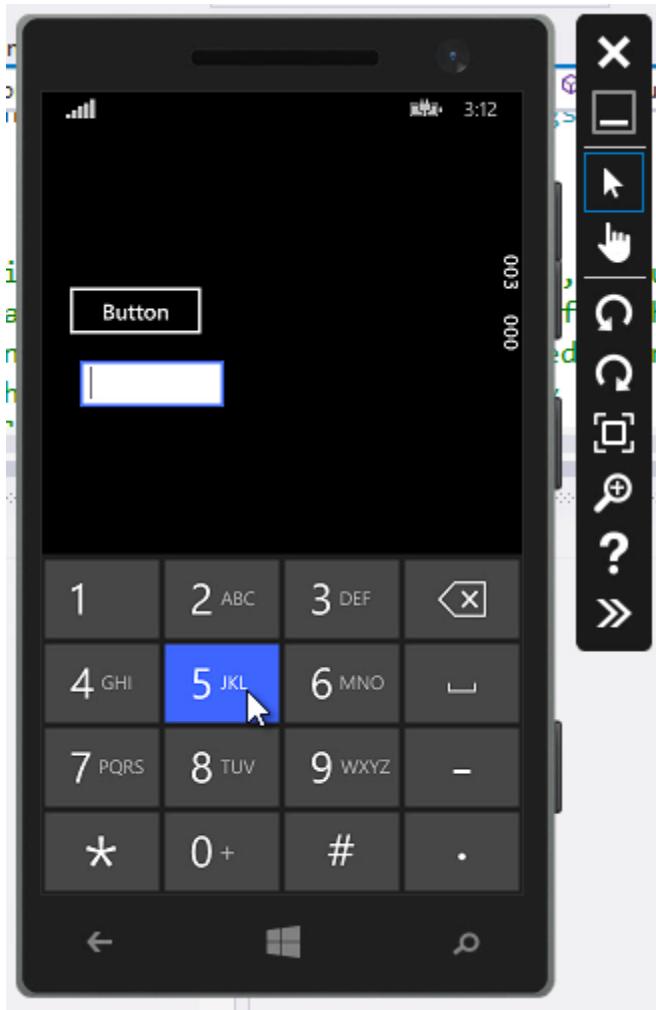
... which will create a method stub in the `MainPage.xaml.cs` code behind:



Here I'll retrieve the number typed into the `myTextBox` and display it in the `myTextBlock`:

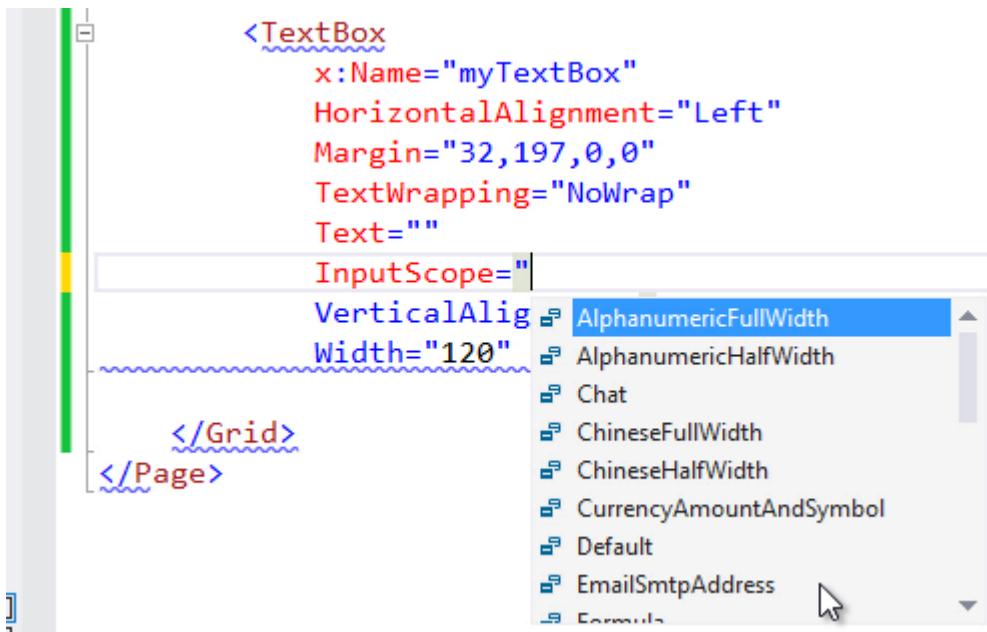


When I start debugging the app and tab the `TextBox`, I see that a keyboard appears with just numbers:

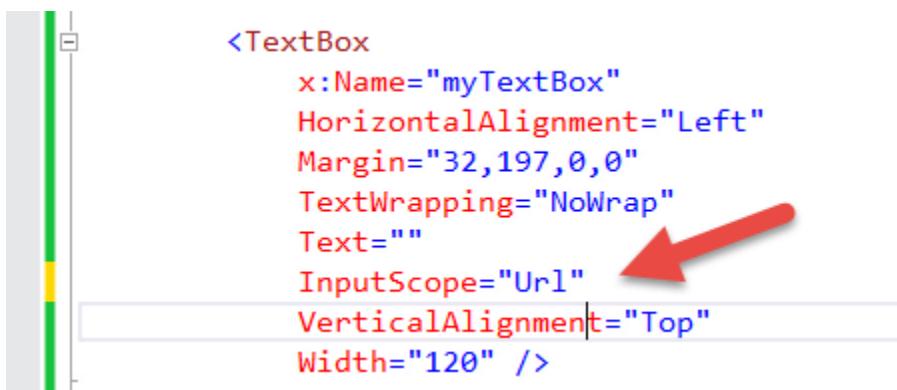


I will click the Button to display the number I typed into the TextBox retrieved and displayed in the TextBlock.

There are many different InputScope keyboard available. In the XAML editor, I can rely on Intellisense to help me see the possible enumerated values:



In this case, I'll change the `InputScope="Url"` ...



... and this time when I run the app, I can see that I have a keyboard that is geared towards helping me type in URLs (note the .com key near the space bar at the bottom).

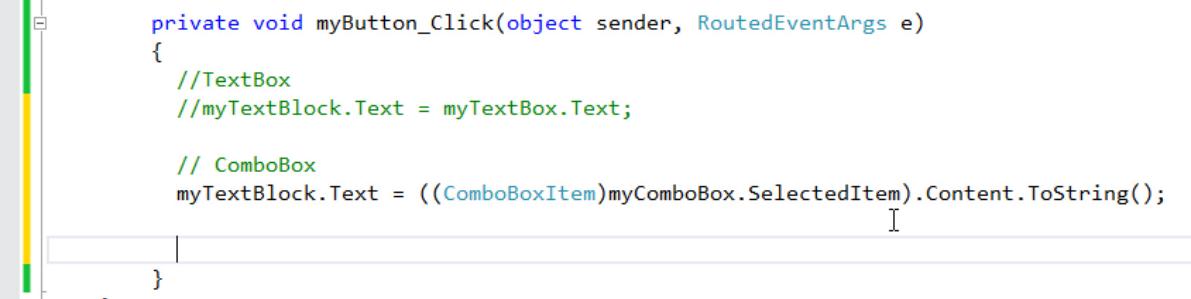


Next, we'll look at the ComboBox which allows me to display multiple possible options as a series of ComboBoxItems:



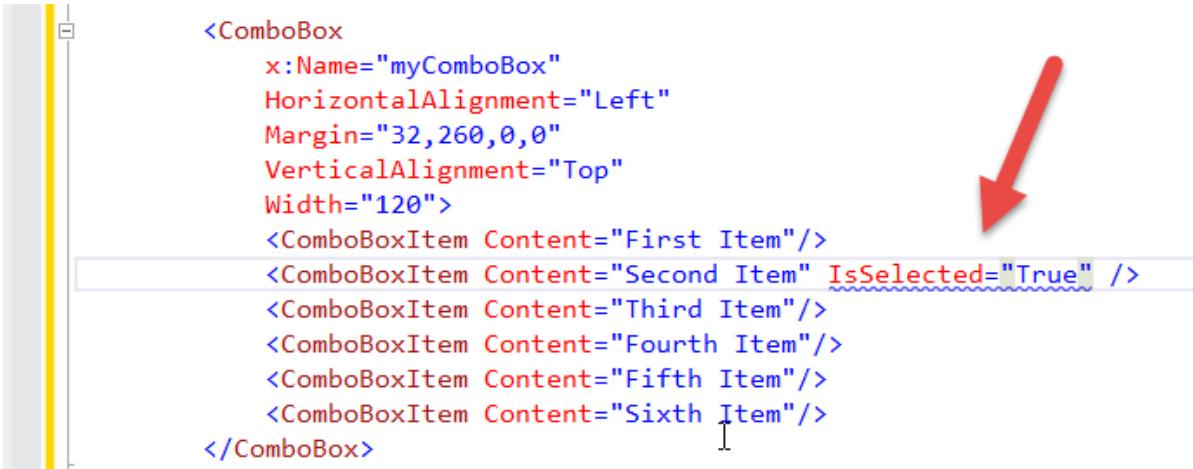
```
<ComboBox  
    x:Name="myComboBox"  
    HorizontalAlignment="Left"  
    Margin="32,260,0,0"  
    VerticalAlignment="Top"  
    Width="120">  
    <ComboBoxItem Content="First Item"/>  
    <ComboBoxItem Content="Second Item"/>  
    <ComboBoxItem Content="Third Item"/>  
    <ComboBoxItem Content="Fourth Item"/>  
    <ComboBoxItem Content="Fifth Item"/>  
    <ComboBoxItem Content="Sixth Item"/>  
</ComboBox>
```

To retrieve the selected ComboBoxItem, I'll add the following code to my myButton_Click event handler:

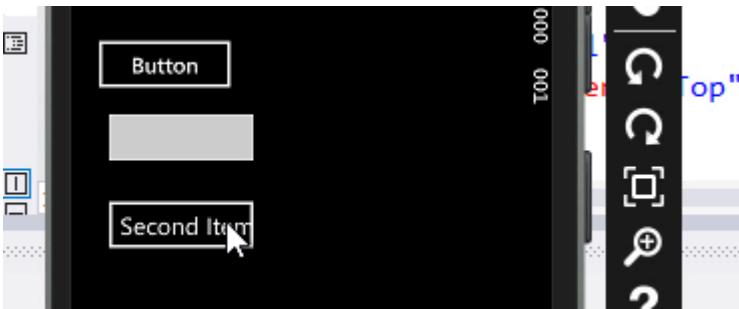


```
private void myButton_Click(object sender, RoutedEventArgs e)  
{  
    //TextBox  
    //myTextBlock.Text = myTextBox.Text;  
  
    // ComboBox  
    myTextBlock.Text = ((ComboBoxItem)myComboBox.SelectedItem).Content.ToString();  
}
```

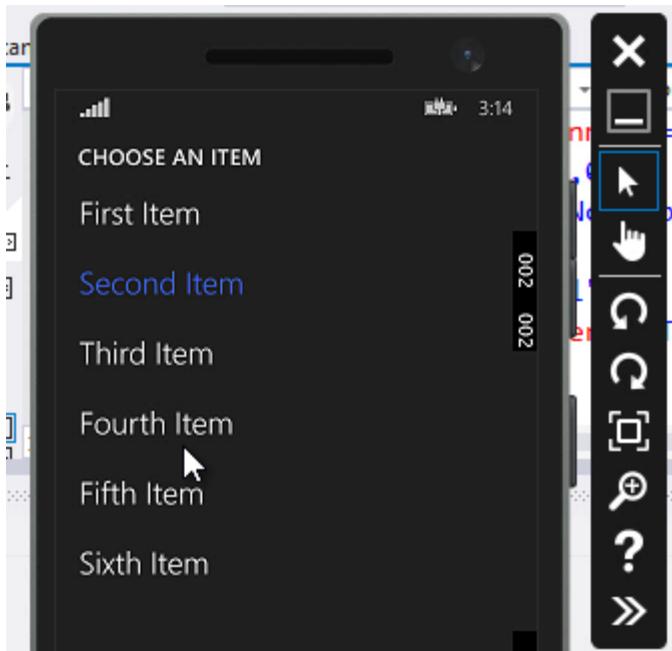
Also, to provide an initially selected / default value, I'll set the IsSelected="True" attribute value:



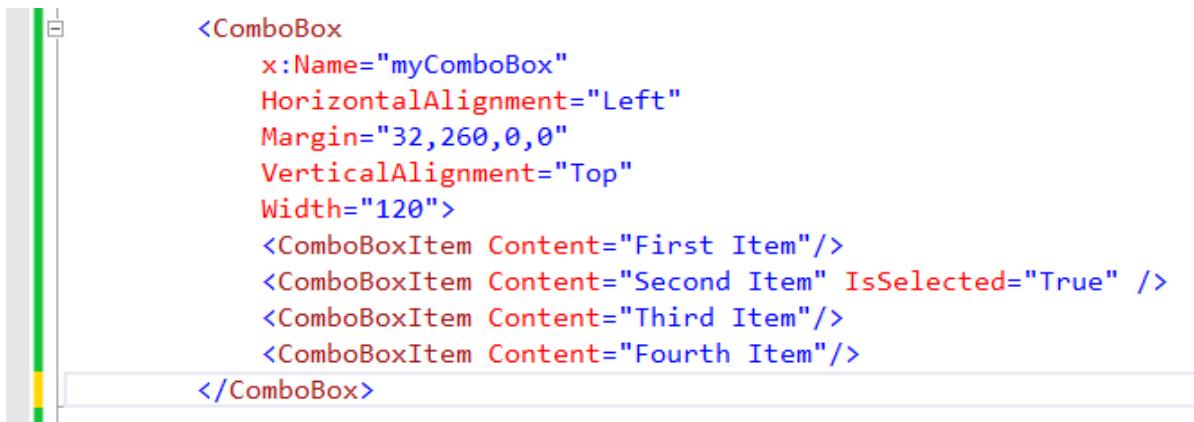
And when I run the app, notice that the ComboBox is already set to “Second Item”:



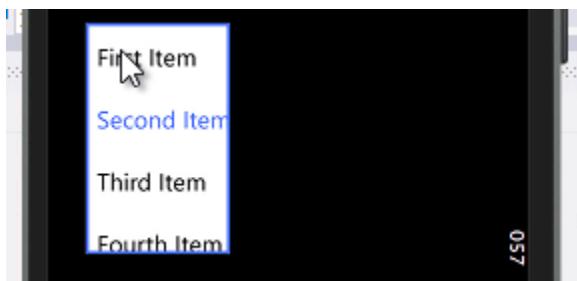
When I click on that item, a flyout control appears displaying all the possible items in the ComboBox. I can select one of them and click the checkmark icon in the command bar to accept that choice:



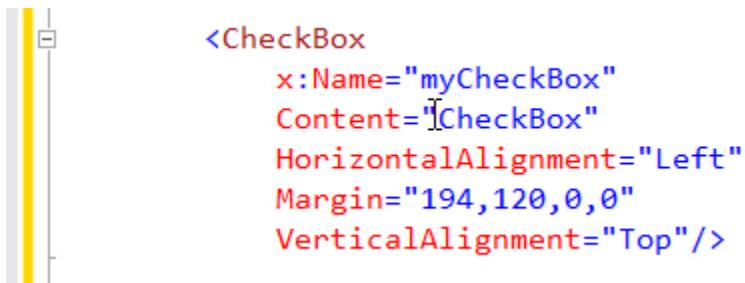
The flyout panel only appears when you have five or more items. I'll reduce this to just four ComboBoxItems:



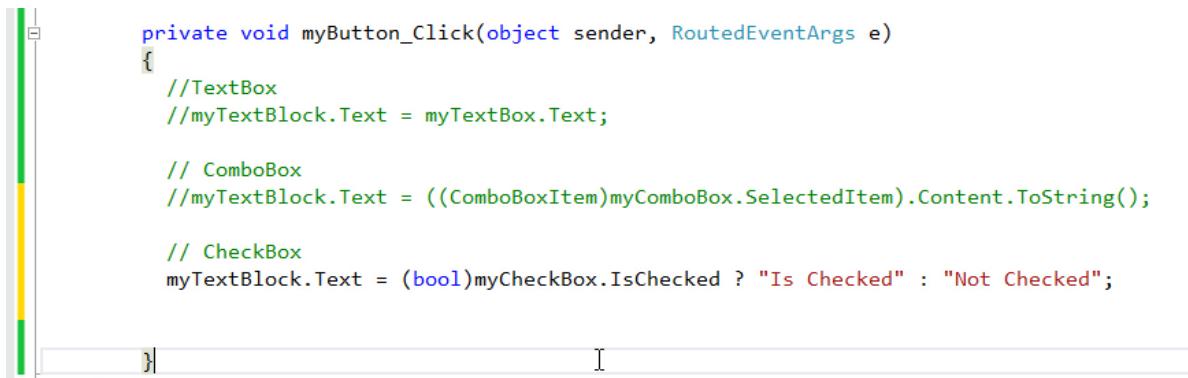
And this time when I run the app, clicking on the ComboBox reveals a list of four items:



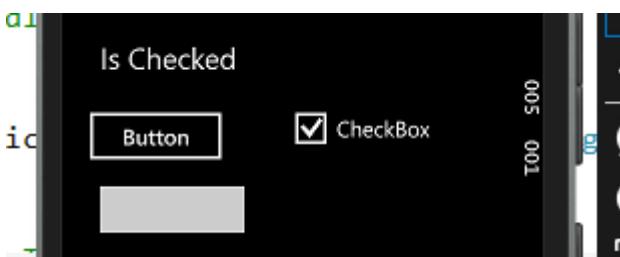
Next, we'll look at the CheckBox. It is useful when you want to retrieve a yes or no answer from the user:



To retrieve the value from the CheckBox, I'll use the IsChecked property. Here, I use the decision operator to return one of two strings depending on whether IsChecked is true or false:



When I run the app and check the CheckBox, then click the Button, I can see the value returned in each “state” of the CheckBox:



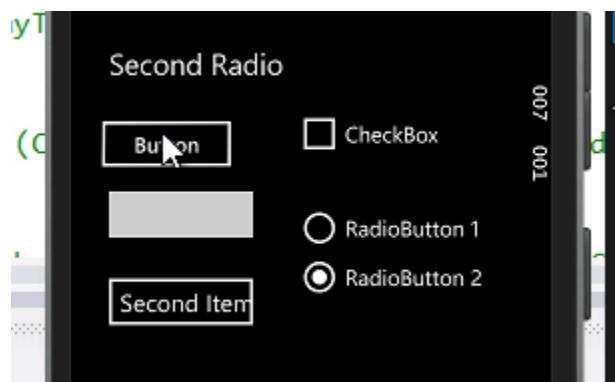
Similar to the CheckBox is the RadioButton. The only difference: RadioButtons work in groups so that no two RadioButtons in a group can be selected at the same time. This is useful where you want to present a limited set of options to the user (otherwise you should use the ComboBox). Here I create two RadioButtons:

```
<RadioButton  
    x:Name="myFirstRadioButton"  
    Content="RadioButton 1"  
    HorizontalAlignment="Left"  
    Margin="194,198,0,0"  
    VerticalAlignment="Top"/>  
  
<RadioButton  
    x:Name="mySecondRadioButton"  
    Content="RadioButton 2"  
    HorizontalAlignment="Left"  
    Margin="194,239,0,0"  
    VerticalAlignment="Top"/>
```

Here I determine which RadioButton was checked:

```
if (myFirstRadioButton.IsChecked == true)  
    myTextBlock.Text = "First Radio";  
else if (mySecondRadioButton.IsChecked == true)  
    myTextBlock.Text = "Second Radio";  
else  
    myTextBlock.Text = "Unknown";
```

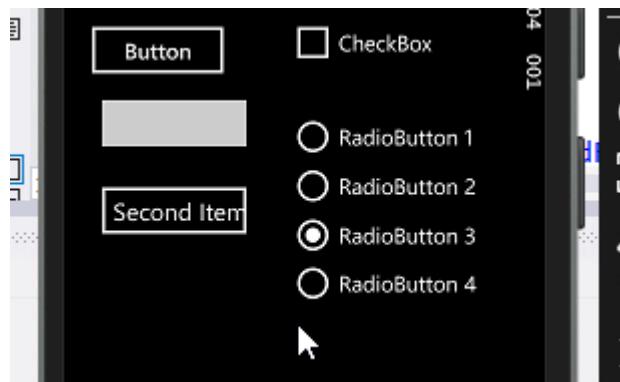
When I run the app, I can see my selection and the associated text when I choose a RadioButton then click the Button:



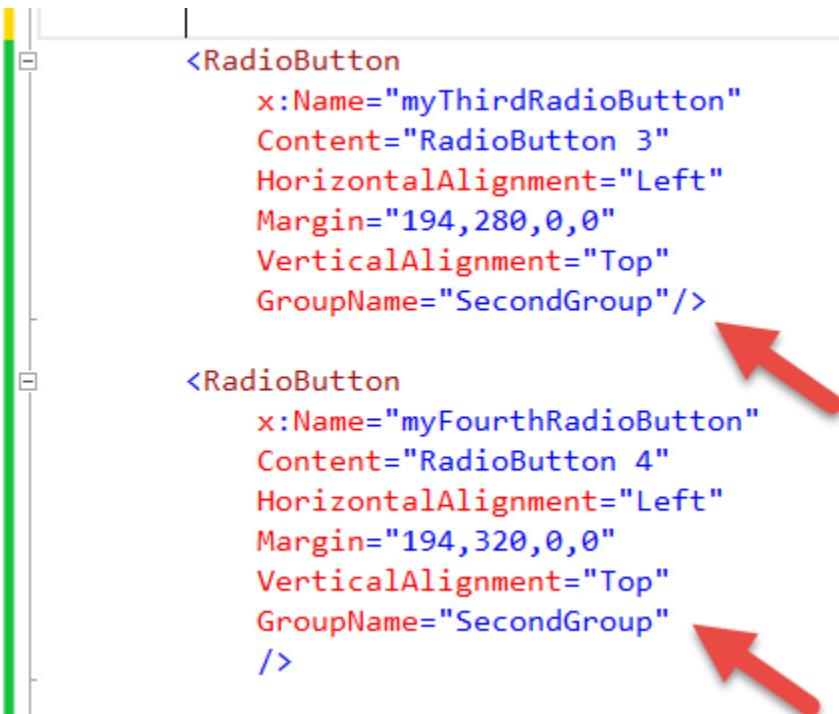
I'll add a few more RadioButtons:

```
<RadioButton  
    x:Name="myThirdRadioButton"  
    Content="RadioButton 3"  
    HorizontalAlignment="Left"  
    Margin="194,280,0,0"  
    VerticalAlignment="Top"/>  
  
<RadioButton  
    x:Name="myFourthRadioButton"  
    Content="RadioButton 4"  
    HorizontalAlignment="Left"  
    Margin="194,320,0,0"  
    VerticalAlignment="Top"  
/>
```

And currently there are four all in one group:

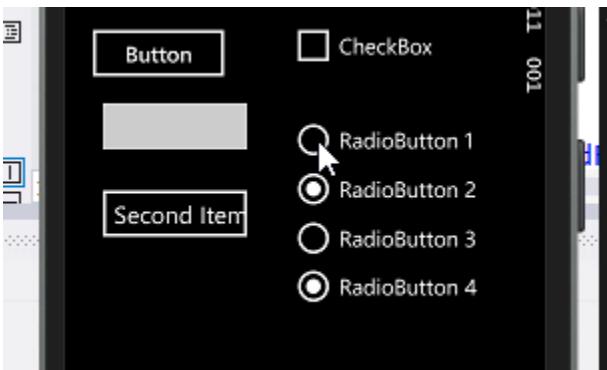


If I want to split these four RadioButtons into two separate groups, I'll give the last two RadioButton controls a GroupName. The GroupName must be spelled the same in order for them to be grouped together:



Note: If you don't add a group name to RadioButtons, they will be added to a default group. That's why our previous example worked.

When I run the app now, I have two groups of RadioButtons, each group allowing a selection of a single RadioButton:



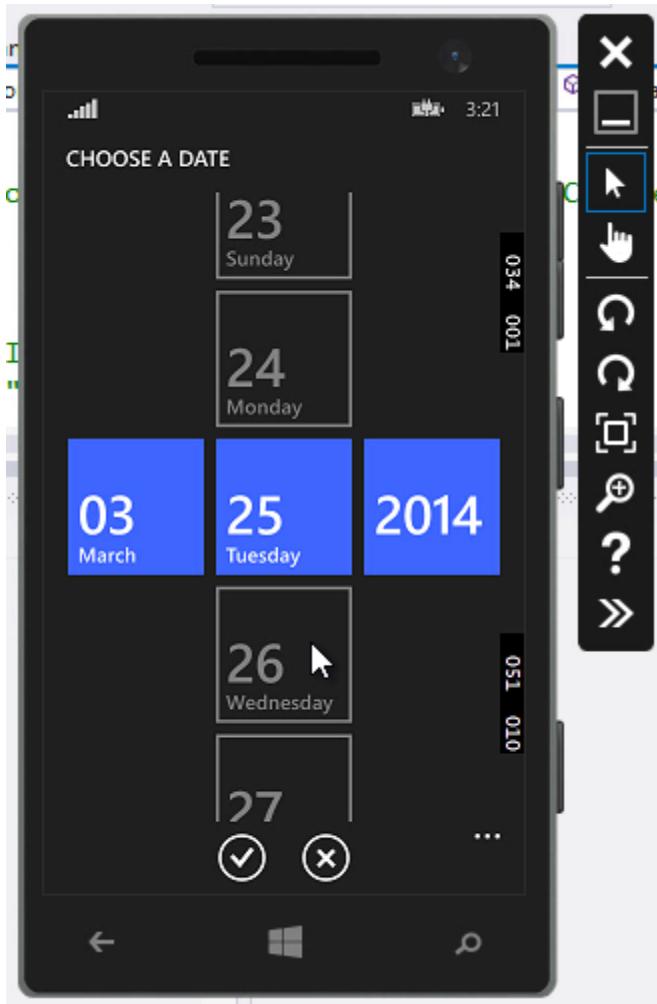
Next, we'll discuss the DatePicker which allows the user to enter a reliable date using a special flyout:

```
<DatePicker  
    x:Name="myDatePicker"  
    HorizontalAlignment="Left"  
    Margin="32,349,0,0"  
    VerticalAlignment="Top"  
/>
```

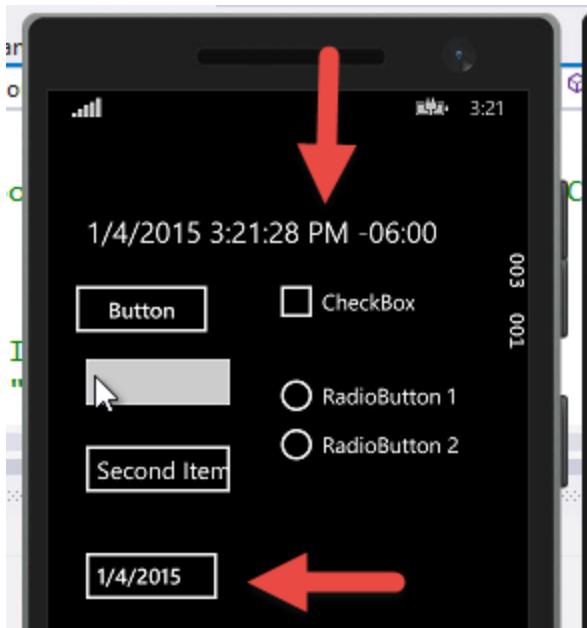
To retrieve the selected date from the DatePicker, I add the following C# code:

```
myTextBlock.Text = myDatePicker.Date.ToString();
```

When I run the app and tap the DatePicker, a flyout displays a three-column selection control. I can change the month, day and year independently by scrolling the date part up or down. Once I've found the date I'm looking for, I select the check mark icon to accept the selection or the x icon to cancel:



When I select a date and click the Button, I retrieve the date and display it in a raw format. I could use the many built-in DateTime formatting options on this string before I display it or I could use String.Format's special formatting codes:



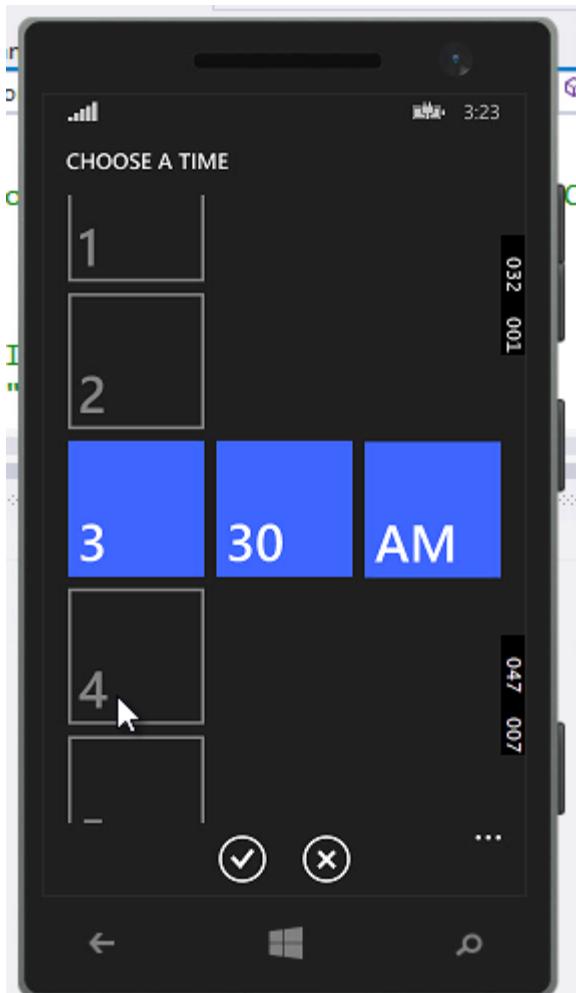
Next, we'll talk about the TimePicker. This is almost identical to the DatePicker except it only deals with the Time part:

```
<TimePicker  
    x:Name="myTimePicker"  
    HorizontalAlignment="Left"  
    Margin="160,349,0,0"  
    VerticalAlignment="Top"/>
```

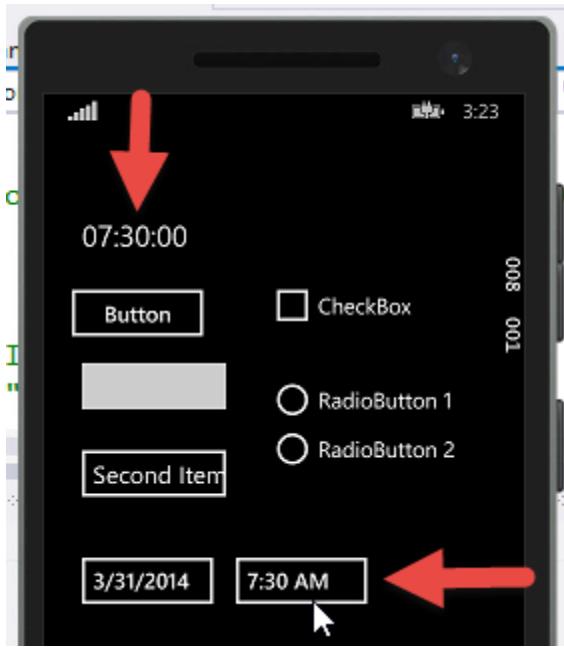
Here I'll retrieve the current selected Time from the TimePicker and display it in the TextBlock:

```
myTextBlock.Text = myTimePicker.Time.ToString();  
}
```

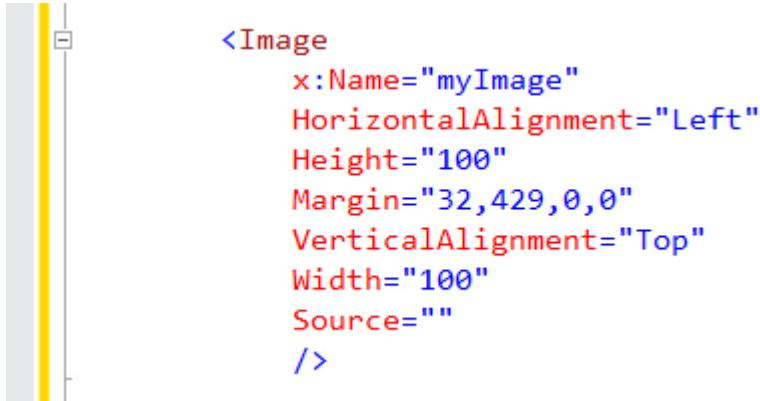
Again, when I run the app and tap the TimePicker, I see a three-column display of time parts, the hour, minute and AM/PM:



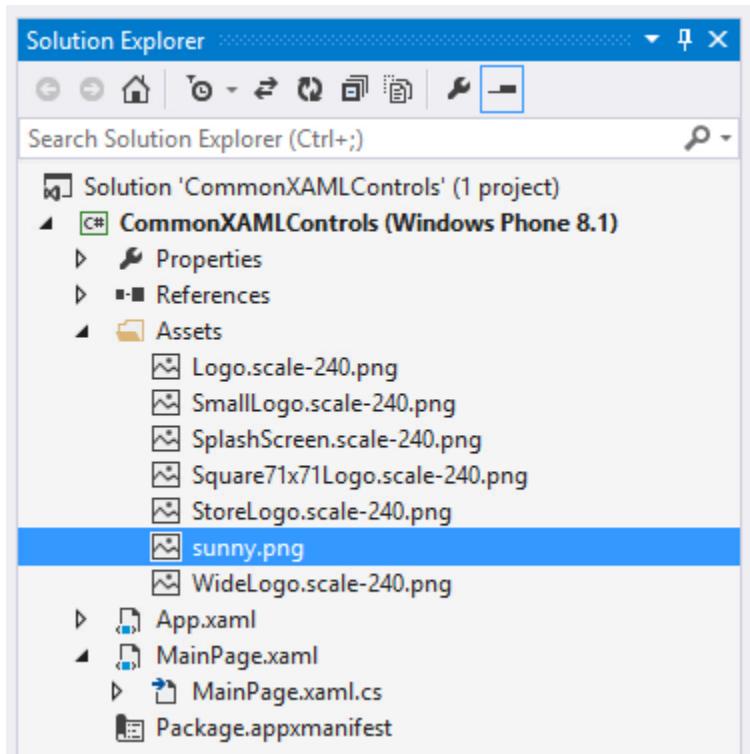
I can choose a time, then tap the check mark to accept the new time or the x icon to cancel the selection. When I choose a time and click the Button, I retrieve the Time in a raw format. Again, I could use the built-in DateTime functions or String.Format to nicely format the time as desired:



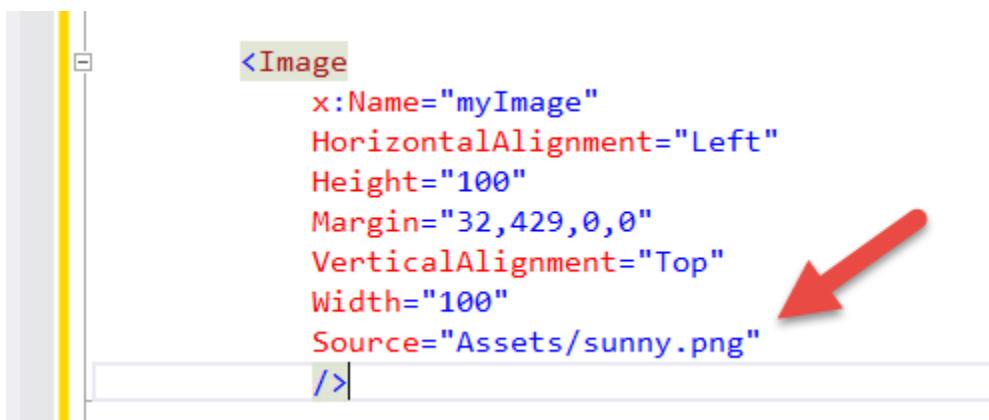
Next, we'll look at the Image control:



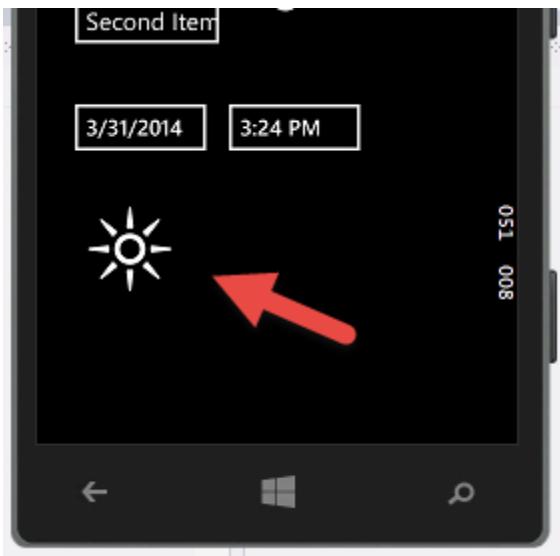
To demonstrate this control, I'll need to add the `sunny.png` file from the Assets folder in the `Lesson5.zip` file that accompanies this lesson. You can drag that `sunny.png` file from Windows Explorer into the Assets folder of the Solution Explorer. When you're finished you should see the image file listed in your Assets folder like so:



Now I'll edit the Image code I added a moment ago to set the Source attribute to: "Assets/sunny.png":

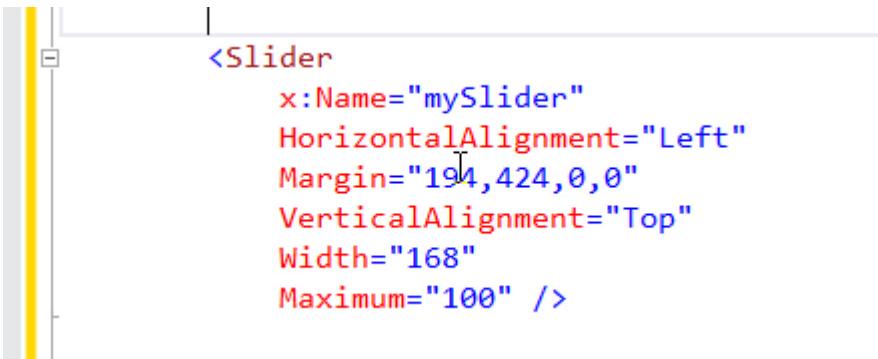


When I run the app, I can see the image:



Later in this series, we'll learn how to programmatically set the Source property of the Image control by constructing a URI object. More on that later.

Next, we'll work with the Slider control which allows a user to choose a value between a Minimum and Maximum value. In this case, the Minimum will be 0 (the default) and the Maximum will be 100.



When we run the app, you can drag the slider to the right (higher value) or to the left (lower value):



The Slider simplifies numerical input as long as there's only a limited set of values and precision is not important.

Next, I'll demonstrate the use of a ProgressBar. You typically use a ProgressBar to display feedback to the end user as to the progression of a given operation. As the bar creeps from left to right, it lets the user know the approximate progress and pace of the operation.

In our case, we'll use the ProgressBar to mirror the current value of the Slider control. In other words, as you drag the Slider left and right, we'll retrieve the value of the Slider and set it to the value of the ProgressBar. We certainly could accomplish this in C#, however in this case we'll use a special feature of XAML called binding. Here, we'll set the Value attribute of the ProgressBar to the Value of the mySlider control using a special binding syntax:



```
<ProgressBar  
    x:Name="myProgressBar"  
    HorizontalAlignment="Left"  
    Height="10"  
    Margin="195,483,0,0"  
    VerticalAlignment="Top"  
    Width="100"  
    Maximum="100"  
    Value="{Binding ElementName=mySlider, Path=Value, Mode=OneWay}" />
```

We'll learn more about this binding syntax later. For now, when we run the app and change the value of the Slider, the ProgressBar's value changes as well.



Most of this series will rely on binding, but not to other controls. Rather, we'll be binding to data that represents information our app will manage in memory and to the Phone's storage.

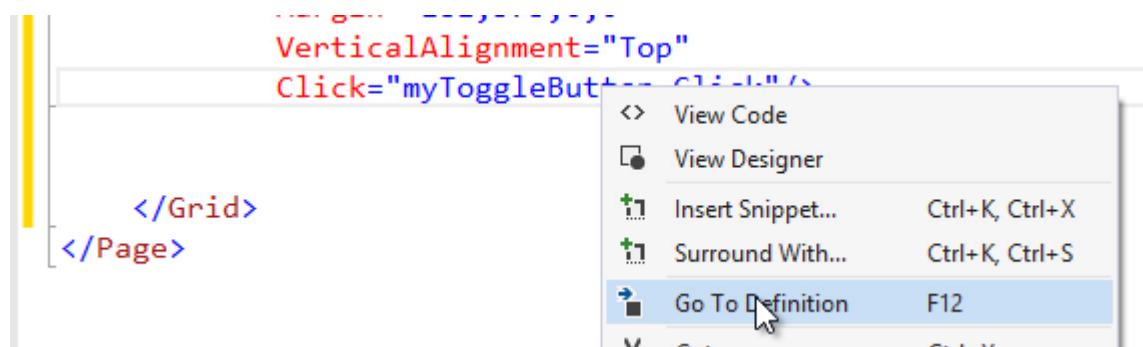
Similar to the ProgressBar is the ProgressRing. While the ProgressBar provides feedback as to the estimated completion of a given task, the ProgressRing only provides feedback to the user that the operation is in progress with no indication how long until completion. We'll add the ProgressRing's XAML to our MainPage.xaml:

```
<ProgressRing  
    x:Name="myProgressRing"  
    HorizontalAlignment="Left"  
    Margin="32,562,0,0"  
    VerticalAlignment="Top"  
    Height="76"  
    Width="76"/>
```

And I'll pair this control up with another common control, the ToggleButton. The ToggleButton has three states: On, Off and Unknown. First, we'll add the ToggleButton:

```
<ToggleButton  
    x:Name="myToggleButton"  
    Content="ToggleButton"  
    HorizontalAlignment="Left"  
    Margin="181,573,0,0"  
    VerticalAlignment="Top"  
    Click="myToggleButton_Click"/>
```

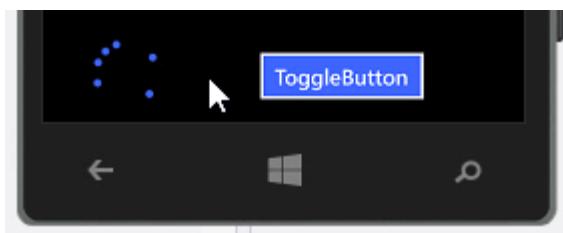
I'll want to respond each time the user taps the ToggleButton, so I'll handle the Click event. Once again, I'll right-click the event handler method name "myToggleButton_Click" and select "Go To Definition" from the context menu:



... to create a method stub. Inside the method, I'll add the following code:

```
private void myToggleButton_Click(object sender, RoutedEventArgs e)
{
    if (myToggleButton.IsChecked == true)
        myProgressRing.IsActive = true;
    else
        myProgressRing.IsActive = false;
}
```

Based on the `IsChecked` property of the `ToggleButton`, I'll set the `ProgressRing`'s `IsActive` property to true (turns it on) or false (turns it off). When I run the app and click the `ToggleButton` to the On state, the ring pulsates:



These are some of the very most common controls. Some of them like our `DatePicker`, `TimePicker`, the `ComboBox` employ other controls like the `Flyout` that will display other options on another “page” and when you make the selection, it will return to the original page with the given control set to the selection the user made.

There are literally hundreds of additional properties, dozens for each of the controls that are displayed here for specific purpose, and as we go through the series you'll learn a few. But ultimately, if you can imagine some desired interaction or display based on previous experience working with the Phone or with apps in general there's probably going to be a property that will allow that will facilitate that feature. I recommend perusing the list of properties for a given control to get an idea of what it can do. When you come across a property you do not understand, take a moment and search MSDN for more information to see if it can help you accomplish what you need.

Lesson 6: Themes and Styles XAML

The Windows Phone operating system has a distinctive look and feel. I'm not just talking about the tile-based user interface, but also the colors and typography. If you search online for "windows phone style guide" you'll find official documents from Microsoft that outline these suggestions. However, even inside of those guidelines, there is room for creative expression and branding. So, you'll want to know the rules and know when to break them.

In this lesson, I'm going to talk about the technical aspects of working with XAML to style your app. Primarily, I'll talk about creating re-usable resources and styles that can be shared between many elements whether on the same page or the same application. Finally, we'll talk about the pre-built themes that are available so that forces consistency across all apps on a given user's phone.

Suppose you have a particular color or setting that you know you'll want to re-use throughout a page or the entire app, or a color or setting that you may want to change occasionally and you want to make the change in one place and have it reflected everywhere that style is used, similar to a Cascading Style Sheet style. In that case, you can create a resource and then bind to it as needed.

I created a simple Phone project called XAMLResources with the simplest {StaticResource} example I could think of:

(1) I add <Page.Resources> as a child element of <Phone> to hold Local (or rather, page-level) Resources. Any resources or styles created here will only be available on this page, not other pages.

(2) I create a SolidColorBrush resource that I can reference with the key "MyBrush".

```
<Page.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Brown" />
</Page.Resources>
```

(3) To utilize this SolidColorBrush with the key "MyBrush", I'll bind to it like so:

```
<TextBlock Text="APP NAME"
    Foreground="{StaticResource MyBrush}" />
```

The result:



The key is the binding syntax which we'll see often for different purposes in XAML. Binding to a resource takes the form of:

```
{StaticResource ResourceName}
```

The curly braces indicate binding. The first word “StaticResource” define the type of binding ... we’re binding to a resource that is defined in XAML, as opposed to a more dynamic form of binding for data which we’ll talk about later.

Beyond simple resources, styles allow you to collect one or more settings that can be re-used across a Page or across an entire app for a specific control. Then, when you want to re-use that style, you set a given control’s Style attribute and bind it to that style you defined.

Virtually every control and even the Page itself has features of its appearance that can be customized, whether it be fonts, colors, border thickness, width and height or what have you. All of these attributes can be set on an individual basis on each control in your XAML.

Here I create the following style:

```
<Page.Resources>
...
<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>
```

Here I create a Style that I can reference with the key "MyButtonStyle". Notice that the TargetType is set to Button ... this style only applies to Button controls. Inside of this style, I can set individual attributes of the Button. I have three attributes set using `<Setter>` elements that include the Property (attribute on the Button) I want to set and the value I want it set to. So, in this case, I create a style which sets the Background attribute to the SolidColorBrush "Blue" as well as certain attributes of the Font to give it a distinctive appearance.

(4) Next I bind my Button's Background attribute to the value bound to MyButtonStyle.

```
<Button Content="My Brush Example"
    Height="100"
```

```
Style="{StaticResource MyButtonStyle}" />
```

Which produces this result:



In this overly simplistic example, it may not be readily apparent the value of this approach. As your application grows large and you want to keep a consistent appearance between the controls on the page, you may find this quite handy. It keeps the XAML concise and compact. And, if you need to change the style or color for any reason, you can change it once and have the change propagate to everywhere it has been applied.

I created these Local Resources on the page, meaning they are scoped to just the MainPage.xaml. What if I wanted to share these Resources across the entire application? In that case, I would remove them from the Page's <Page.Resources> and add them in the App.xaml's <Application.Resources> section as a System Resource:

```
<Application
  ...
  <Application.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>
  </Application.Resources>
</Application>
```

And there should be no change to the appearance of the Button on my original MainPage.xaml.

Discovering Theme Resources

In addition to styles, there are Themes. Themes are built-in styles available to all Windows Phone apps and their color and other appearance settings are set by the user of the phone. So, on my Phone, I go to Settings -> theme and I choose the dark background and the yellow accent color. I would like to see apps utilize this color scheme to stay consistent with my preferred colors. As a developer, I can make sure I adhere to those settings and others by binding to a ThemeResource.

If you look at the Page template, it utilizes a theme resource:

```
<Page  
    x:Class="XAMLResources.MainPage"  
    ...  
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

And I can even use theme resources inside of my styles. Here I'll modify my application scoped style to use the accent color the user has chosen:

```
<Application.Resources>  
    ...  
    <Style TargetType="Button" x:Key="MyButtonStyle">  
        <Setter Property="Background"  
            Value="{ThemeResource PhoneAccentBrush}" />  
        ...  
    </Style>  
</Application.Resources>
```

Themed Resources should be leveraged to keep your apps looking like they belong on the Windows Phone. You should resist the urge to use custom colors, fonts and the like unless you have a good reason to (such as to match your company's established branding elements, etc.)

Furthermore, there are built-in styles available to your app. For example, if I were to modify my MainPage.xaml like so:

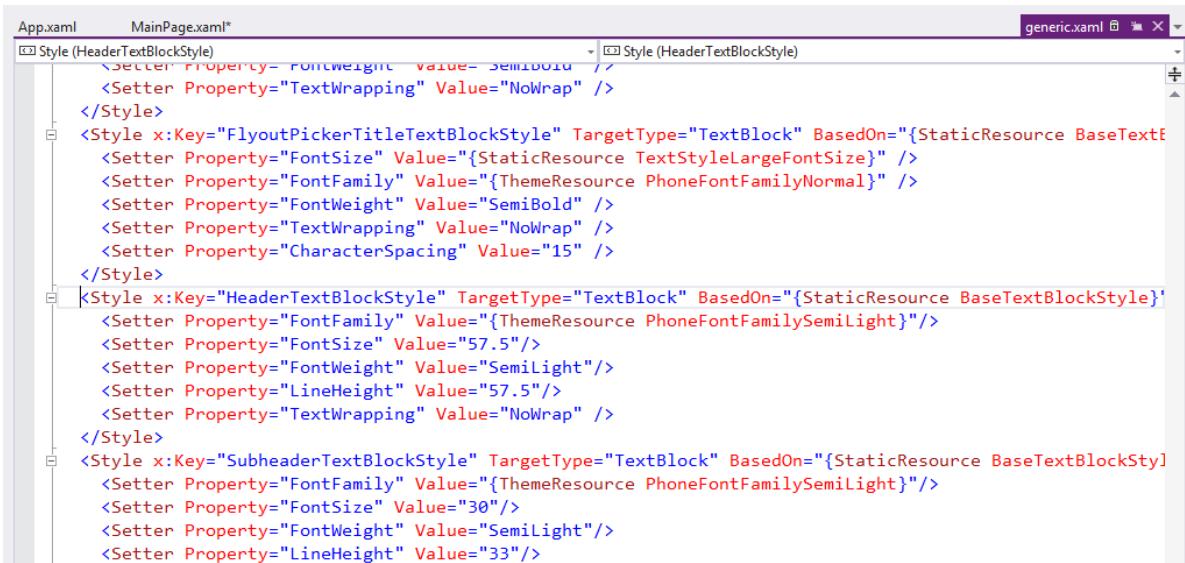
```
<TextBlock Text="page name"  
    Style="{StaticResource HeaderTextBlockStyle}" />
```

It will produce the following appearance:

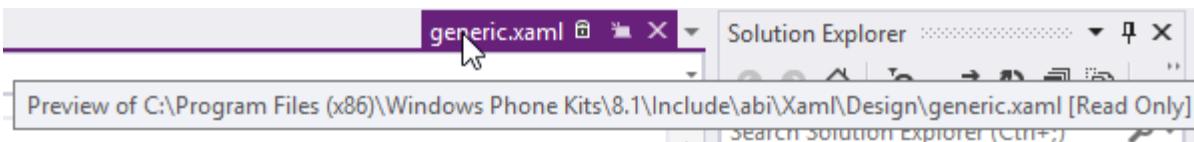


But where did this StaticResource come from? I did not define it.

If you right-click on the style name and select “Go To Definition” you’ll see that it opens a read-only file from your hard drive:



If you hover over the preview tab, you’ll see the location of this file and the fact that it is read-only:



These are pre-built styles for you to use to ensure consistency across apps.

It's also worth noting that many of the styles are based on styles which are based on yet other styles. Notice the definition of the style we're investigating:

```
<Style x:Key="HeaderTextBlockStyle"
    TargetType="TextBlock"
    BasedOn="{StaticResource BaseTextBlockStyle}">
    ...

```

</Style>

The key here is the BasedOn attribute. So, this style builds on the BaseTextBlockStyle and adds several modifications to it. In this way, we can override these styles with our own. This visual inheritance allows developers to avoid repeating the attribute settings that will be common across variations of the styles, much like how Cascading Style Sheets work in web development.

I said earlier that there were different binding expressions in the Windows Phone for different purposes. The {StaticResource } is generally a resource or style that has been created by you or was created by Microsoft (as we have just seen). The {ThemeResource } is for binding to a theme. The last binding expression we'll see is simply {Binding } which is used for binding data (i.e., usually generic lists of custom types with their properties set to the data we want to work with in our app) to on page elements. We'll see this at work much later in this series.

Recap

Just to recap, the big takeaway from this lesson was the ways in which we can style our app to make it look like it belongs on the Windows Phone while also expressing our own individuality. We learned how to bind to StaticResources like styles and resources, and ThemeResource which borrow from the user's selections.

Lesson 7: Understanding the Navigation Model

Many apps you will want to build require more than one Page, therefore you'll need to learn how to navigate from one page to another and learn how to pass important data between pages. Furthermore, you will want to learn how to manipulate the navigation history so that you can control what happens when the user uses the Phone's physical back button.

Let's start with the basics. We'll create a new Blank App project template called NavigationExample. Immediately, I'll add a new Blank Page called Page2.xaml.

Back on the MainPage.xaml, I'll replace the <Grid> with the following code:

```
<StackPanel Margin="30,30,0,0" >
    <TextBlock Text="MainPage.xaml" />
    <Button Content="Go To Page 2"
        Click="Button_Click" />
</StackPanel>
```

I'll right-click the Button_Click event name and select "Go To Definition" to create the event handler method stub. I'll add the following code:

```
// Simple navigation, no parameters
Frame.Navigate(typeof(Page2));
```

On Page2.xaml, I'll replace the <Grid> with the following code:

```
<StackPanel Margin="30,30,0,0">
    <TextBlock Text="Page2.xaml" />
    <TextBlock x:Name="myTextBlock"
        Text=""
        FontSize="32" />
    <Button Content="Go To Page 3"
        Click="Button_Click" />
</StackPanel>
```

When I run the app, I can click the button to navigate from the MainPage.xaml to Page2.xaml.

The key to this example is the Frame.Navigate() method. The Frame object allows you to load and unload Page objects into and out of the viewable area. Furthermore, it will keep track of the order of pages as they are visited so that you can use methods like GoBack() and

`GoForward()` to traverse the history of pages visited. In that respect, the Frame is very much like the history feature of your web browser.

In addition to simple navigation, you are able to pass data from one page to the next. To do this, we'll add the following code in the `Button_Click` event of the `MainPage.xaml`:

```
// Simple navigation, no parameters
//Frame.Navigate(typeof(Page2));

// Passing a simple type, like a string
Frame.Navigate(typeof(Page2), "Hola from MainPage");
```

We'll retrieve the values that were "sent" via the `OnNavigatedToEvent` that should already be generated for you in the `Page` template. Here's the code in my version:

```
/// <summary>
/// Invoked when this page is about to be displayed in a Frame.
/// </summary>
/// <param name="e">Event data that describes how this page was reached.
/// This parameter is typically used to configure the page.</param>
protected override void OnNavigatedTo(NavigationEventArgs e)
{
}
```

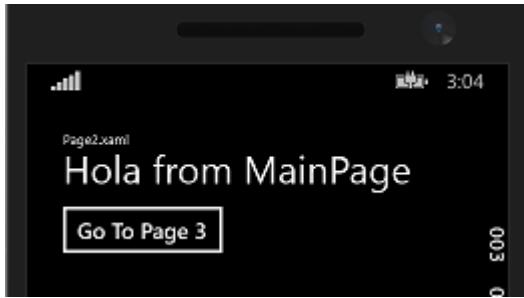
As you can read in the comments, this `Page` event is invoked when the page is about to be displayed in the `Frame`. You can use the input parameter of type `NavigationEventArgs` to retrieve the value passed from the calling page (if anything was, in fact, passed). In our case, we'll add the following code:

```
myTextBlock.Text = e.Parameter.ToString();
```

In addition to the `OnNavigatedTo()` method, there's also a `OnNavigatedFrom()` which will be triggered right before the current page is removed from the `Frame`. This might be a good opportunity to clean up any resource or save the current state of the page depending on your needs.

(Note: I also needed to implement the `Button_Click` event handler method stub on `Page2.xaml`. I right-click the name `Button_Click` on the `Page2.xaml` and select "Go To Definition". This will allow the app to be compiled.)

When I run the app in the emulator, I can see the following when navigating to `Page2`:



Typically, sending a simple type is not enough. You may need to send several pieces of information from one page to the next. To accomplish this, I usually create a class I call `NavigationContext` that contains the data I need to pass. I'll include properties for each data element I may need.

So, I'll add a new class in my project called `NavigationContext.cs` and fill it with the following class definition:

```
public class NavigationContext
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
}
```

Now, in `MainPage.xaml`, I'll add the following code in the `Button_Click` event handler:

```
// Simple navigation, no parameters
//Frame.Navigate(typeof(Page2));

// Passing a simple type, like a string
//Frame.Navigate(typeof(Page2), "Hola from MainPage");

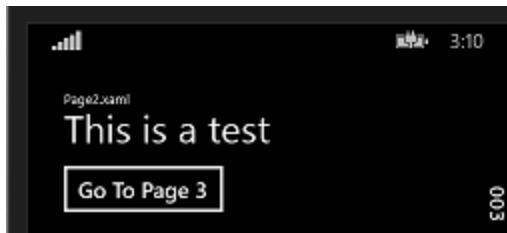
// Passing an instance of a 'Navigation Context' object
NavigationContext nav = new NavigationContext()
{
    ID = 7,
    Name = "Test",
    Description = "This is a test"
};

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

And then in the OnNavigatedTo event handler of the Page2.xaml.cs page:

```
//myTextBlock.Text = e.Parameter.ToString();  
  
var nav = (NavigationContext)e.Parameter;  
myTextBlock.Text = nav.Description;
```

Now, when I navigate from the MainPage to Page2, I see the following:



Finally, I want to show you how you can manipulate the Frame's history. I'll add a third page called Page3.xaml and replace its <Grid> with the following:

```
<StackPanel Margin="30,30,0,0">  
    <TextBlock Text="Page 3" />  
    <TextBlock x:Name="pagesTextBlock"  
        Text="" />  
    <Button Content="Go Back"  
        Click="Button_Click" />  
</StackPanel>
```

In the Page3.xaml.cs I'll implement the following

```
protected override void OnNavigatedTo(NavigationEventArgs e)  
{  
    string myPages = "";  
    foreach (PageStackEntry page in Frame.BackStack)  
    {  
        myPages += page.SourcePageType.ToString() + "\n";  
    }  
  
    pagesTextBlock.Text = myPages;  
  
    Frame.BackStack.RemoveAt(Frame.BackStackDepth - 1);
```

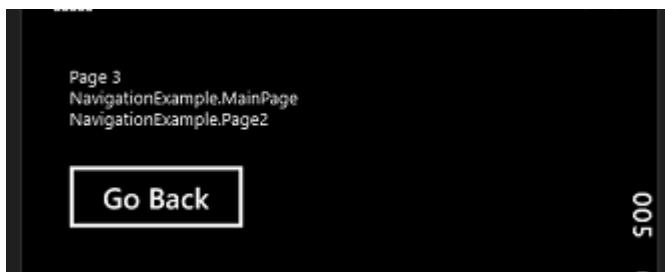
```
}
```

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Frame.GoBack();
}
```

Returning to Page2.xaml.cs, I'll implement:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Frame.Navigate(typeof(Page3));
}
```

When I navigate from the MainPage to Page2 to Page3 by clicking buttons, I will see the following displayed on Page 3:

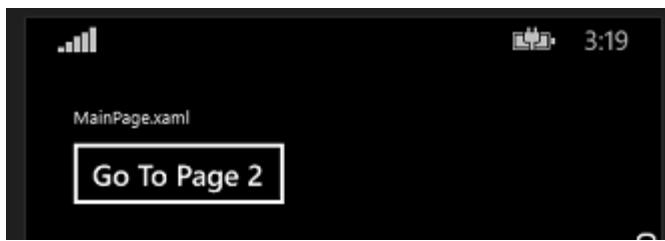


The key to this example is traversing the `Frame.BackStack` which is an `IList<PageStackEntry>`. A `PageStackEntry` represents a page that was navigated to.

After I display the pages that we've travelled through to get to Page 3, I do the following:

```
Frame.BackStack.RemoveAt(Frame.BackStackDepth - 1);
```

This allows me to find the last page (`Frame.BackStackDepth - 1`) and remove it from the collection of pages. The result? When you click the "Go Back" button on Page 3, we navigate through the `Frame.BackStack` using the `Frame.GoBack()` method which will navigate to the last item in the `Frame.BackStack` ... that should be the MainPage since we removed the Page2 `BackStackEntry`.



In addition to the `Frame.GoBack()`, there's also a `Frame.GoForward()` which allows us to return to the previous page in the BackStack.

Recap

In this lesson we learned the absolute basics of navigation. We learned about the purpose of the `Frame` object, and how to handle the `OnNavigatedTo()` method to retrieve values that are passed from one page to the next. We learned how to create a `NavigationContext` class to allow us to pass more than one simple type between pages, and finally learned about the `BackStack` which is a collection of `PageStackEntry` objects, each object representing a page we've navigated through, and how to traverse the `BackStack` using the `GoBack()` and `GoForward()` methods.

Lesson 8: Working with the package.appxmanifest

The Package.appxmanifest contains meta-data about your application. In other words, it describes your app to the operating system including the friendly name of the app, icons and images that will be used on the Start page, what the phone's hardware must support in order for your app to work properly, and much more.

I just want to take a quick tour of the Package.appxmanifest ... admittedly, this will lack the context of when during the development lifecycle you might need to employ some of these settings. If nothing else, this will give you some insight into what's possible when building apps for the Windows Phone, even if I don't demonstrate the specifics of how to implement a given feature.

First, the Package.appxmanifest is used during compilation of your source code into a install package. The install package will have the file extension appx. A manifest introduces your app to the phone telling it which assemblies are contained in the appx, which asset files should be used for the start and apps screen of the phone, as well as the app's name, and the features of the phone that the app uses, and so on. This is why it is sometimes referred to as the metadata about your app.

Second, when you open it in the main area, you see this tabbed designer:

Package.appxmanifest X

The information the system needs to deploy, display, or update your app is contained in the Package.appxmanifest file, and the information used for the Store listing is contained in the StoreManifest.xml file. You can use the Manifest Designer to modify the properties in these files.

Application Visual Assets Requirements Capabilities Declarations Content URLs Packaging

Use this page to set the properties that identify and describe your app.

Display name: PhoneControls

Entry point: PhoneControls.App

Default language: en-US [More information](#)

Description: PhoneControls

Supported rotations: An optional setting that indicates the app's orientation preferences.

 Landscape

 Portrait

 Landscape-flipped

SD cards: Prevent installation to SD cards

Notifications:

Toast capable: (not set)

Lock screen notifications: (not set)

Tile Update:

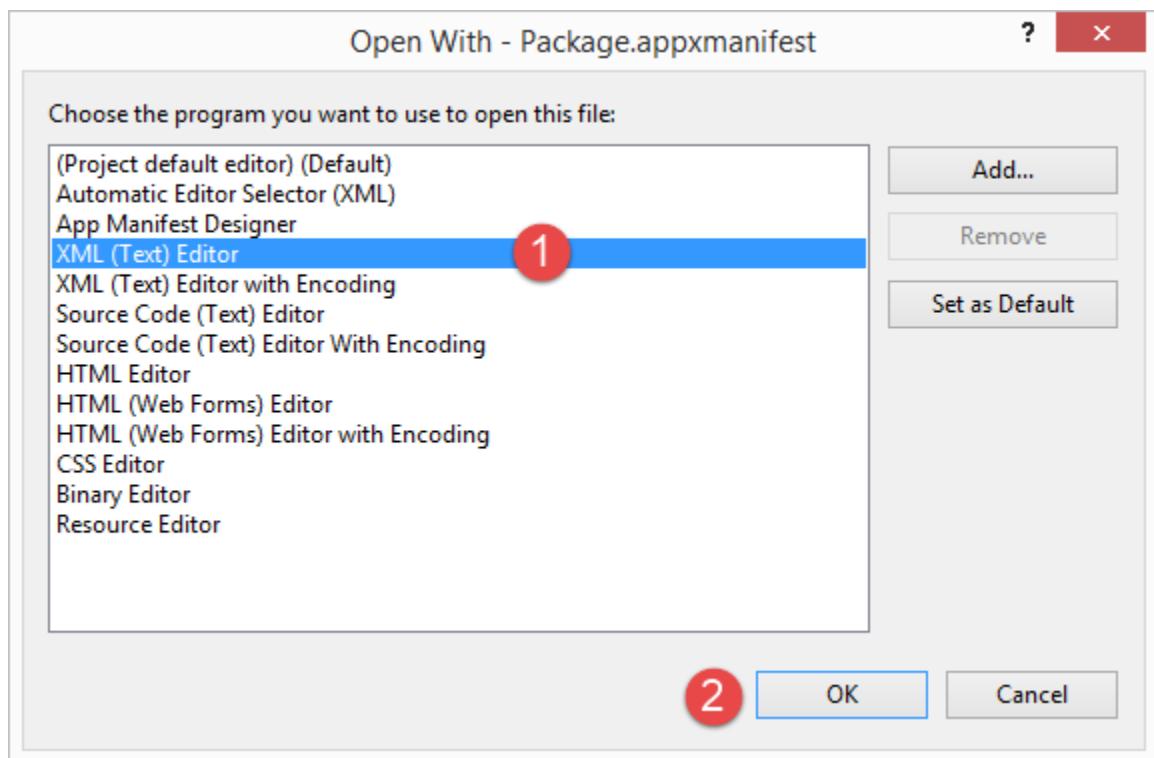
Updates the app tile by periodically polling a URI. The URI template can contain "{language}" and "{region}" tokens that will be replaced at runtime to generate the URI to poll.

[More information](#)

Recurrence: (not set)

URI Template:

However, the file is actually just an XML file and this property designer simply modifies its values. Just for fun, I'll right-click the Package.appxmanifest in the Solution Explorer and select Open With ... in the context menu. The Open With dialog appears.



(1) Select XML (Text) Editor

(2) Click OK

You'll see the file in its "naked" form. I would recommend you do not change anything about this file, at least not unless you have a very specific reason to do so. Close the tab to ensure you do not corrupt it.

We'll re-open the Package.appxmanifest property designer and briefly review the purpose of each of the settings. Frankly, most of the settings are self-explanatory and yet the nuanced reasons why you may choose to change these settings could require more time than I have in this lesson. My goal here is to just do a quick review and call to your attention the fact that they exist so that if you're following some other tutorial in the future, you'll have some idea of where to look for a given settings.

There are seven tabs:

- Application
- Visual Assets
- Requirements
- Capabilities
- Declarations

Content URIs

Packaging

We'll start with the Application tab. I would classify this as a catch-all for properties that don't fit nicely in other tabs. Many of the settings are utilized by the operating system, such as the Display Name, the Default Language, and the Supported Rotations and whether or not you would want to prevent installation of the app to SD cards to prevent users from sharing your app.

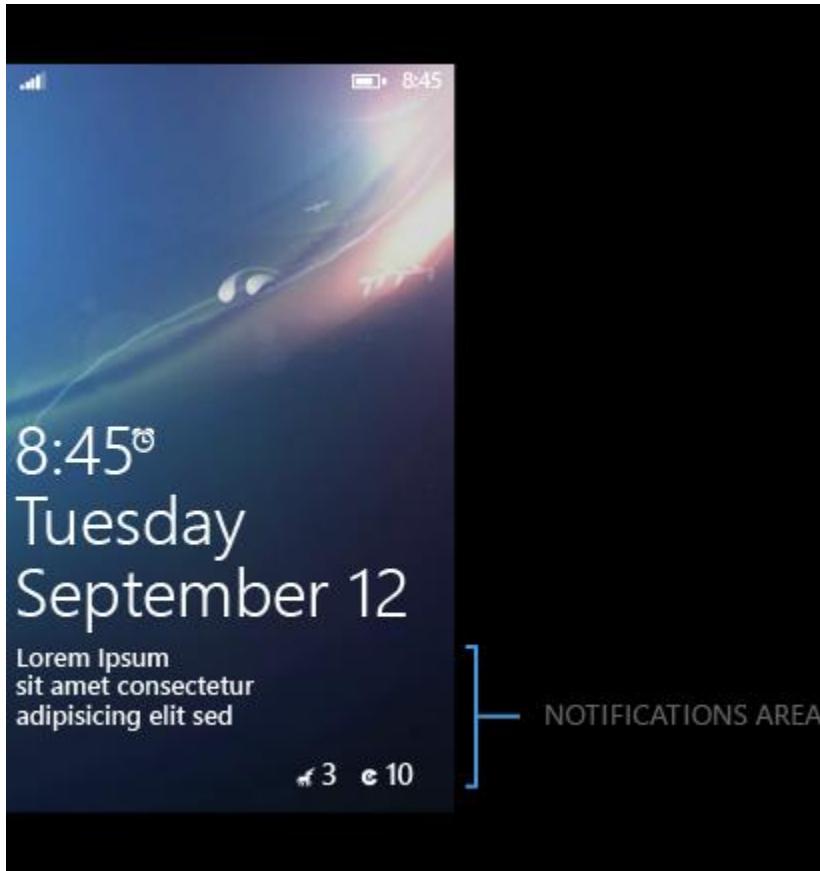
The Notifications settings allow you to tell the operating system whether you intend to use Toasts and Lock Screen Notifications in your app.

A toast displays at the top of the screen to notify users of an event, such as a news or weather alert.



A toast notification displays for about 10 seconds unless the user dismisses it with a flick to the right. If the user taps the toast, by default, your app's start screen launches. Or, you can choose to specify which screen of your app will launch.

Lock screen notifications allow you to update the lock screen with text messages or numbers usually indicating that there's a number of updated items the user should be aware of.



Your users can choose to customize this area with any eligible third-party apps they choose. As a developer you can design your app to be an app that the phone user can choose to customize the lock screen notifications area.

The lock screen's app icon, count, and text are pulled directly from the app's default Tile. Info appears on the lock screen only if the default Tile contains this info. For example, a count will appear on the lock screen only if the Tile displays it.

The next tab is for setting the app's Visual Assets:

Application Visual Assets Requirements Capabilities Declarations Content URIs

Windows Phone apps should support displays of different resolutions. Windows Phone provides a simple way to do this via resource loading in the manifest.

[More information](#)

All Image Assets

Title:

Short name:

Show name: Square 150x150 Logo
 Wide 310x150 Logo

Default size: (not set)

Background color: #464646

Square 150x150 logo:

Assets\Logo.png [X](#)

Scaled Assets

		
360 x 360 px	210 x 210 px	150 x 150 px
...

We'll use this later as we build a real app to change the tiles, the splash screen, and more. Notice that there are different sizes required. These are required for different states of the tile on the Start or Apps page on the Phone. On the Start page, the user can decide whether the app should be small, medium or large so you need to provide assets for each of these scenarios.

The next tab is for hardware requirements. For example, if you were building an app that needed the NFC capability, it wouldn't make sense for a user to install it on a phone without that feature.

Application Visual Assets Requirements Capabilities Declarations Content URIs

Use this page to specify the hardware requirements of your app. Your app installs only on devices that meet specified requirements.

Requirements:

Gyroscope
 Magnetometer
 NFC
 Front Camera
 Rear Camera

Description:

Requires a phone that contains a gyroscope to function.
[More information](#)

The next tab is for Capabilities. A capability is a request for permission. Suppose your app needs (or wants) to access a given feature of the phone or the data on the given phone. You can't just grab it. The app has to ask for the user's permission to use that capability because it is usually handling some potentially private information that the user may not want to share with just any app.

This screenshot shows the 'Capabilities' tab of the Windows Phone 8.1 developer portal. The tab bar includes Application, Visual Assets, Requirements, Capabilities (which is selected), Declarations, Content URLs, and Packaging. Below the tabs, a message says 'Use this page to specify system features or devices that your app can use.' The main area is divided into two sections: 'Capabilities:' and 'Description:'. The 'Capabilities:' section contains a list of checkboxes for various permissions, with 'Appointments' being checked. The 'Description:' section provides a detailed explanation of what the 'Appointments' capability does, mentioning access to the appointment store and the ability to create new calendars. A link to 'More information' is also present.

The next tab is Declarations.

This screenshot shows the 'Declarations' tab of the Windows Phone 8.1 developer portal. The tab bar includes Application, Visual Assets, Requirements, Capabilities, and Declarations (which is selected). Below the tabs, a message says 'Use this page to add declarations and specify their properties.' The main area is titled 'Available Declarations:' and contains a dropdown menu labeled 'Select one...' and an 'Add' button. A list of declaration types is shown, with 'File Open Picker' currently selected, indicated by a blue highlight and a cursor icon pointing at it.

Think of declarations as other ways that the phone's operating system can execute or enable certain features of your application. For example, we may want to allow our application to execute every half-hour or so (you don't get to decide the exact time) so that it can change

the phone's lock screen image. In that scenario we would add the Background Task declaration and configure the Timer, Executable and Entry Point attributes (perhaps others) to launch that part of your app that changes the image on the lock screen.

Or, in the case of the File Open Picker, you can allow other apps to use your app to choose a given file managed by your app.

The next tab is the Script Notify.

Use this page to specify the HTTPS URLs that can use window.external.notify to send a ScriptNotify event to the app. HTTPS URLs can include wildcards, for example, https://*.microsoft.com or https://*.*.microsoft.com.

URI: Rule:

The idea behind this is pretty cool. In a few lessons we'll learn about the Web View control. It can open up and display web pages. We'll be using it to display web pages that are hosted locally in the app's project. However, you can also open up external websites as well. Now, if that external website happens to be running JavaScript that executes a command window.external.notify, it could pass data into your app. The trick is, you have to indicate which URI's you want to allow to pass information into your app on this page, and it has to be one with an SSL certificate. On your side, you handle the WebView_ScriptNotify event, retrieve the data that is being sent.

Finally, the Packaging tab:

Use this page to set the properties that identify and describe your package when it is deployed.

Package name:

Package display name:

Version: Major: Minor: Build: Revision:

Publisher display name:

Package family name:

Generate app bundle: [What does an app bundle mean?](#)

As the instructions on the tab explain, they deal with information about the packaging of the app for deployment. The Package Name is a globally unique identifier, or GUID, that differentiates it from all other apps ever created for the Phone, while the display name, Version

and Publisher display name are more human readable and descriptive of the app inside the package. The last setting, Generate app bundle, allows you to split up your app into multiple parts so that it takes up less space and requires less time to download for the end user. You might choose to do this if you target multiple languages and have image assets that target different screen sizes and split up your app appropriately into a series of packages that segment those resources. In these cases, the end user would only download the packages that apply to their language and their screen size.

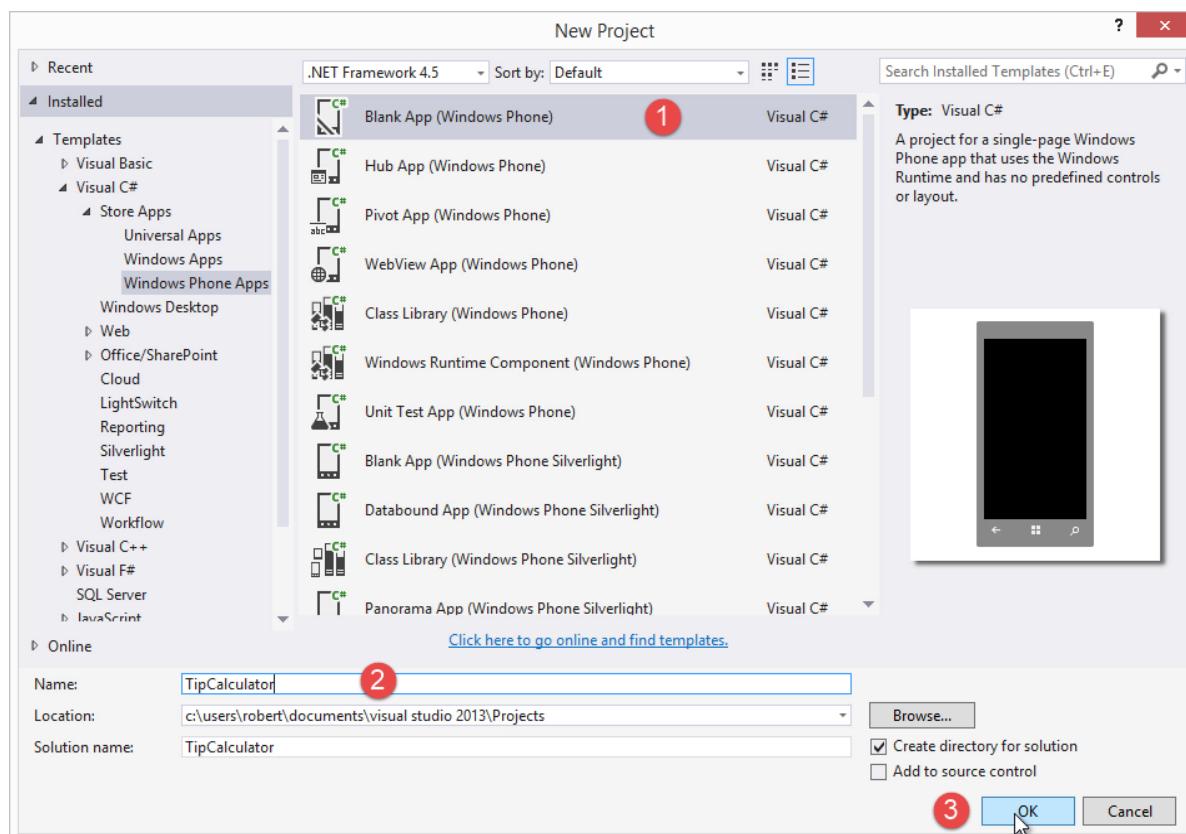
Recap

At the outset, I said that the value of this lesson is in calling to your attention that fact that these settings exist and how you would go about changing them if you needed a given feature. Furthermore, it helps you see what is possible — the features you may want to employ in your app, even if we don't demonstrate them in this series of lessons.

Lesson 9: Exercise: Tip Calculator

In this lesson we'll build our first complete app, a Tip Calculator. It will help solve one of the fundamental problems that I have whenever I'm out to a restaurant, and I'm trying to figure out how much to tip the waitress based on the service. Usually, I'm a pretty generous tipper. However, for the sake of this app that we're going to build, we're going to give three options to help calculate either 10% for mediocre service, 18% for good service, or 25% for exceptional service. You could expand this out for any type of simple calculation.

To begin, we'll create a new (1) Blank App project (2) named TipCalculator. I'll (3) click the OK button to create the project:



I'll begin by creating a number of RowDefinitions in the default Grid on the MainPage.xaml:

```
<Page  
x:Class="TipCalculator.MainPage"  
...  
>
```

```
<Grid>
```

```
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition Height="100" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
```

I'll use the top two rows for the app name and instructions using two TextBlocks like so:

```
<TextBlock Margin="20, 0, 20, 0"
  Grid.Row="0"
  Style="{StaticResource TitleTextBlockStyle}">
  Tip Calculator
</TextBlock>

<TextBlock Margin="20, 0, 20, 0"
  Grid.Row="1"
  Style="{StaticResource TitleTextBlockStyle}"
  FontSize="48">
  Enter the Bill Amount
</TextBlock>
```

Next, I'll add a StackPanel and it will comprise the third row which uses star sizing to take up the remainder of the height remaining. I'll add the controls needed for this app to this StackPanel. In fact, I'll begin by adding a TextBlock that will serve as a label for the TextBox control beneath it:

```
<StackPanel Name="myStackPanel" Grid.Row="2" Margin="20, 0, 20, 0">

  <TextBlock HorizontalAlignment="Left"
    TextWrapping="Wrap"
    Text="Bill Amount"
    FontSize="24"/>

  <TextBox Name="billAmountTextBox"
    Text="$0.00"
    TextAlignment="Right"
    HorizontalAlignment="Left"
```

```

    TextWrapping="Wrap"
    VerticalAlignment="Top"
    InputScope="Number"
    Width="100"
    FontSize="24"
    LostFocus="amountTextBox_LostFocus"
    TextChanged="billAmountTextBox_TextChanged"
    GotFocus="amountTextBox_GotFocus" />

</StackPanel>

```

The billAmountTextBox allows the user to type in the amount printed on the receipt. Obviously, this will be a large part of the calculation for the tip.

Next, I'll add a TextBlock (again, used as a label) and a series of RadioButtons to allow the user to choose the percentage to tip based on the quality of the service:

```

<TextBlock HorizontalAlignment="Left"
    TextWrapping="Wrap"
    Text="Percent To Tip:"
    VerticalAlignment="Top"
    FontSize="24"
    Margin="0,20,0,0"/>

<RadioButton Content="10% - Horrible Service"
    Tag="0.1"
    GroupName="percentRadio"
    Click="RadioButton_Click" />

<RadioButton Content="18% - Acceptable Service"
    Tag="0.18"
    GroupName="percentRadio"
    IsChecked="True"
    Click="RadioButton_Click" />

<RadioButton Content="25% - Great Service"
    Tag="0.25"
    GroupName="percentRadio"
    Click="RadioButton_Click" />

```

Notice that I'm using the Tag property in each RadioButton. The Tag property is a holdover from many years ago. It allows you to add anything you want. What I've used it for, in this particular case, is the actual percentage amount that will be used for the calculation. Instead of having to do some switch statement determine the percentage, I just input the actual percentage inside of the Tag, so when a radio button is selected, I can programmatically retrieve the Tag, convert it to a decimal and then use that in my calculation for the percentage ticked.

.1 for 10%
.18 for 18%
.25 for 25%

Next, I'll display the calculated tip amount. The first TextBlock is just the label describing the purpose for the next TextBlock which actually will display the tip amount:

```
<TextBlock HorizontalAlignment="Left"
    TextWrapping="Wrap"
    Text="Amount to Tip:"
    FontSize="24"
    Margin="0,20,0,0"
/>
<TextBlock Name="amountToTipTextBlock"
    HorizontalAlignment="Left"
    TextWrapping="Wrap"
    VerticalAlignment="Top"
    Text="$0.00"
    FontSize="36"
/>

```

Finally, I'll display the total amount of the bill which includes the pre-tip amount along with the tip. This will be the amount charged to my credit card:

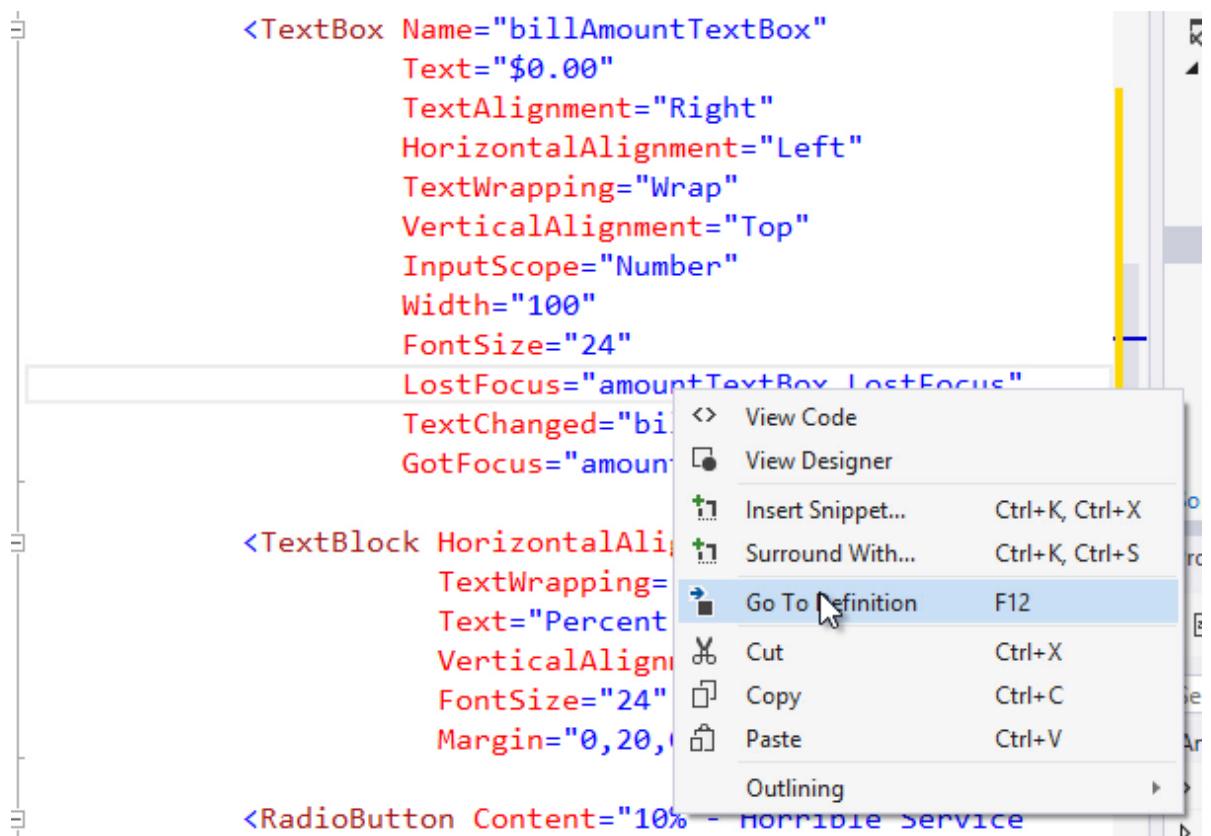
```
<TextBlock HorizontalAlignment="Left"
    TextWrapping="Wrap"
    VerticalAlignment="Top"
    Text="Total Bill:"
```

```
    FontSize="24"
    Margin="0,20,0,0"
/>
<TextBlock x:Name="totalTextBlock"
    HorizontalAlignment="Left"
    TextWrapping="Wrap"
    VerticalAlignment="Top"
    Text="$0.00"
    FontSize="36"
/>

```

The billAmountTextBox (as well as the RadioButton controls) have a number of events that we'll want to handle for various purposes. For example, when the user taps the billAmountTextBox to enter an amount, we will want to clear the text and allow them to type in what they want. When the user is typing or editing the number, we want to perform the calculation instantly. When they move their mouse cursor out of the billAmountTextBox, we want to nicely format the number they typed as dollars and cents with a dollar sign symbol.

To set this up, I'll put my mouse cursor in each of the event handler names I created and will right-click and select "Go To Definition". Alternatively, I'll put my mouse cursor in each event handle name and select F12 on the keyboard to perform the same operation:



The result is a series of stubbed out event handler methods in the MainPage.xaml.cs:

A screenshot of the MainPage.xaml.cs code editor. It shows four stubbed event handler methods:

```
private void amountTextBox_LostFocus(object sender, RoutedEventArgs e)
```

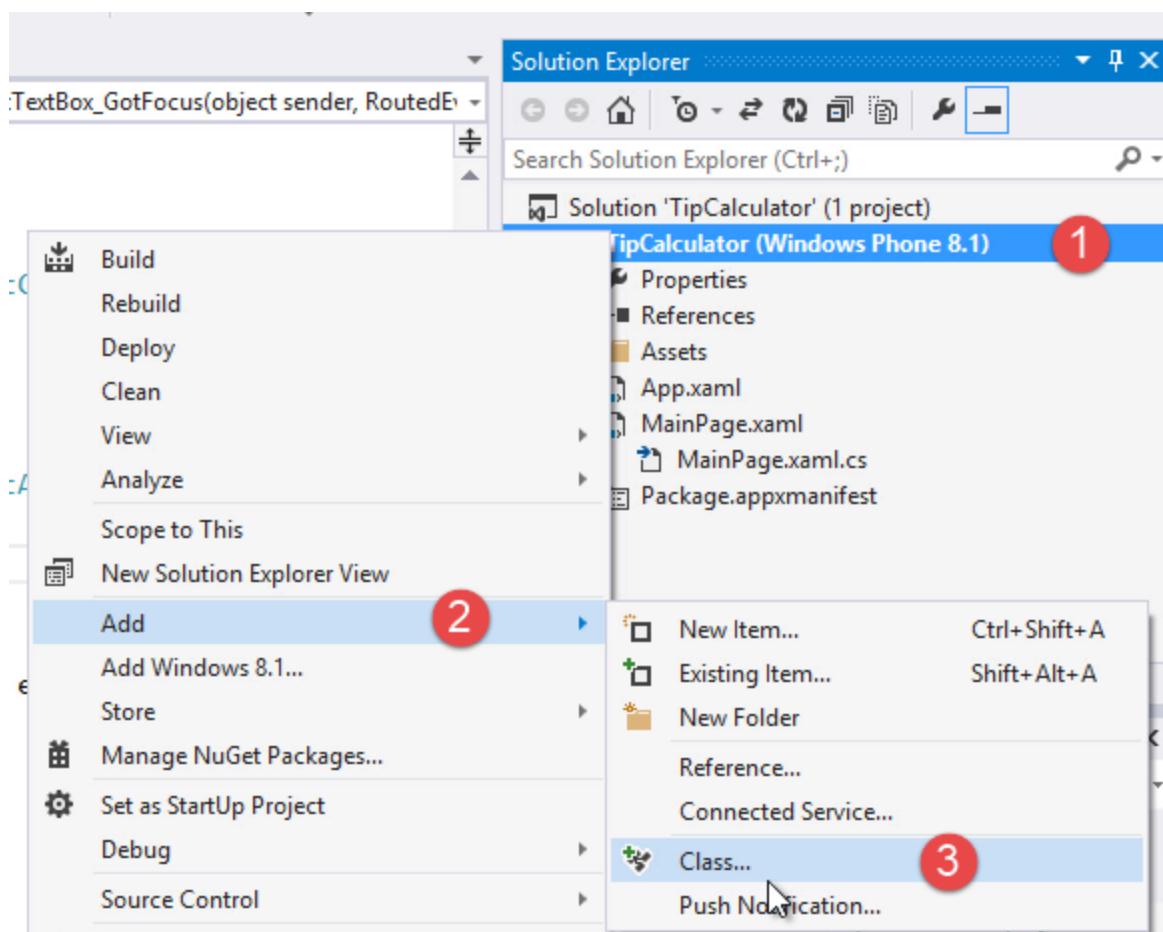
```
private void billAmountTextBox_TextChanged(object sender, TextChangedEventArgs e)
```

```
private void amountTextBox_GotFocus(object sender, RoutedEventArgs e)
```

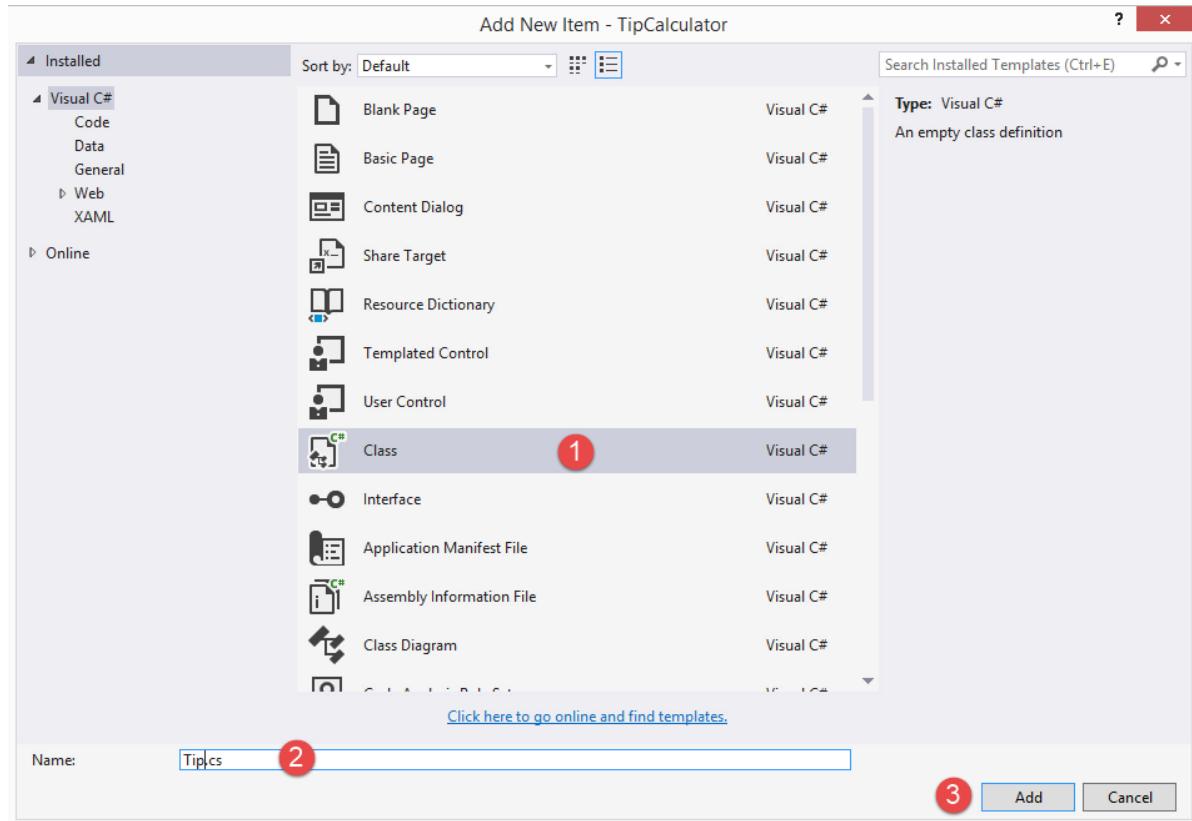
```
private void RadioButton_Click(object sender, RoutedEventArgs e)
```

Now I need to perform the calculation of the tip and the other amounts, and format them for display. I could go about this in many different ways. I could just create all the code right here in my MainPage.xaml.cs, but I have larger aspirations for this app. Sure, some people might be using it on their phone, but some people might have their Microsoft Surface or other Windows 8 tablet with them while they're out to eat. They may want to use the app in that context instead. In the next lesson, we'll create a Universal app version of the Tip Calculator to build both the phone version of the app, and a Windows store version of the app for use on Windows tablets or on desktop.

With that in mind, I'm going to add a class that will know how to calculate the tip and format the results as dollar values. This will make it easier for me in the future keeping all of the important stuff, like the calculations and the data members, isolated into their own class. I'll (1) right-click the project name in the Solution Explorer, (2) select Add from the context menu, and (3) select Class ... from the sub menu:



In the Add New Item dialog, (1) I'll make sure Class is selected, (2) I'll rename the file: Tip.cs, and (3) click the Add button:



I'll add the following code to the Tip class. First, note that I change this to a public class. Second, I create three auto-implemented properties, BillAmount, TipAmount and TotalAmount:

```
namespace TipCalculator
{
    public class Tip
    {
        public string BillAmount { get; set; }
        public string TipAmount { get; set; }
        public string TotalAmount { get; set; }
    }
}
```

You might be wondering why I am using strings instead of double or decimal. The reason is because I want to format these and make them publicly available, so I can more easily associate the values, and not have to do the conversion of the values inside of the MainPage.xaml.cs file. I want to just reference the BillAmount property, and automatically assign its formatted text value to the TextBox. By keeping that formatting inside of my tip class, it will make it more

easily reusable for the Universal app for Windows 8 or for my phone. At least, that's my thought process.

Next, we'll create a constructor. Inside of this constructor, I'll initialize each of the properties to String.Empty.

```
public Tip()
{
    this.BillAmount = String.Empty;
    this.TipAmount = String.Empty;
    this.TotalAmount = String.Empty;
}
```

Next, I'll create a method that will actually calculate the tip:

```
public void CalculateTip(string originalAmount, double tipPercentage)
{
    double billAmount = 0.0;
    double tipAmount = 0.0;
    double totalAmount = 0.0;

    if (double.TryParse(originalAmount.Replace('$', ' '), out billAmount))
    {
        tipAmount = billAmount * tipPercentage;
        totalAmount = billAmount + tipAmount;
    }

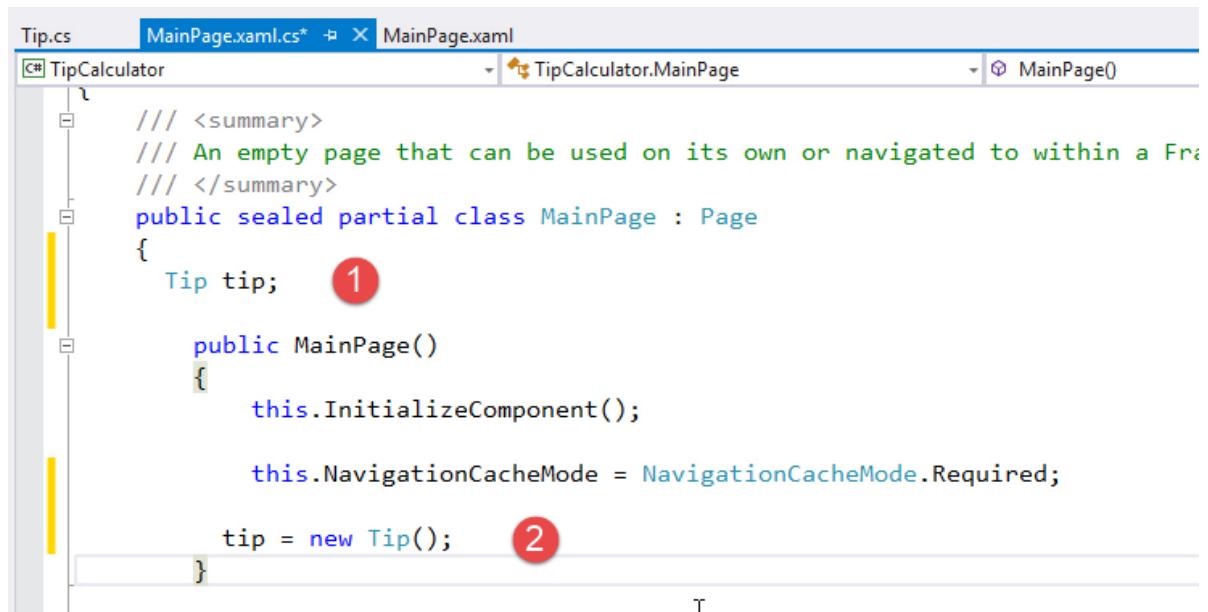
    this.BillAmount = String.Format("{0:C}", billAmount);
    this.TipAmount = String.Format("{0:C}", tipAmount);
    this.TotalAmount = String.Format("{0:C}", totalAmount);
}
```

Frankly, this is a very simple method. I accept the original bill amount as well as the tip percentage as input parameters. Next, since the original bill amount is typed in as a string, I'll do some minor checking to ensure that the string can be converted to a double. I'll remove the dollar sign to help ensure the success of the double.TryParse() method. TryParse() will return true if the value can successfully be turned into a double, false if it cannot (i.e., the user typed

in non-numeric values). Furthermore, it will return the parsed value as an out parameter called billAmount. We talked about out parameters, as well as other TryParse() style methods in the C# Fundamentals for Absolute Beginners series so please review if you've forgotten how this works.

Assuming the original bill amount can be parsed into a double, we perform the calculation for tip and for the total bill amount. Lastly, we format each of those values and set the public auto-implemented properties so that we can access from from our MainPage.xaml.cs file.

Now that we've implemented our business rules in the Tip.cs class, we'll utilize its properties and methods back in our MainPage.xaml.cs file. (1) I'll create a private field to hold on to a reference to our Tip class, and (2) in the MainPage() constructor, I'll create a new instance of Tip and set it to the private field tip:



Since there are several event handler methods from which we'll want to call the Tip's CalculateTip method, I'll implement that call in a private helper method called performCalculation():

```
private void performCalculation()
{
    var selectedRadio = myStackPanel.Children.OfType<RadioButton>().FirstOrDefault(r =>
r.IsChecked == true);
```

```

tip.CalculateTip(billAmountTextBox.Text, double.Parse(selectedRadio.Tag.ToString()));

amountToTipTextBlock.Text = tip.TipAmount;
totalTextBlock.Text = tip.TotalAmount;

}

```

The `performCalculation` first determines which `RadioButton` was checked by using a clever LINQ statement. Here, we look at all children objects of the `myStackPanel`. Then, we narrow it down to just those who are of type `RadioButton`. Finally, we look for the first `RadioButton` whose `IsChecked` property is set to true. I like this instead of using a long switch statement that would require me to add code each time a new `RadioButton` is added to the `StackPanel`.

Once I know which `RadioButton` was selected, I'm ready to call the `Tip`'s `CalculateTip()` method. I send in the `billAmountTextBox.Text` and then use the `Tag` property of the selected `RadioButton` as the tip percentage. Since the `Tag` property is of type string, we'll have to call `double.Parse()` to pass it to `CalculateTip()` correctly as a double.

Now, we can use `performCalculation()` in the two places where I anticipate it will be useful, namely, as the user is typing in a new bill amount number (`billAmountTextBox_TextChanged`) and when a different `RadioButton` is selected (`RadioButton_Click`):

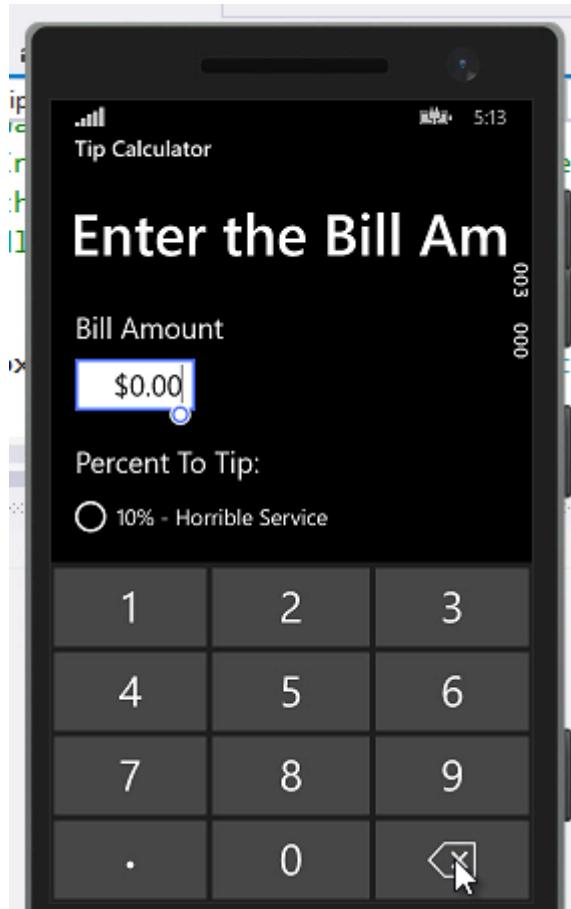
```

private void billAmountTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    performCalculation();
}

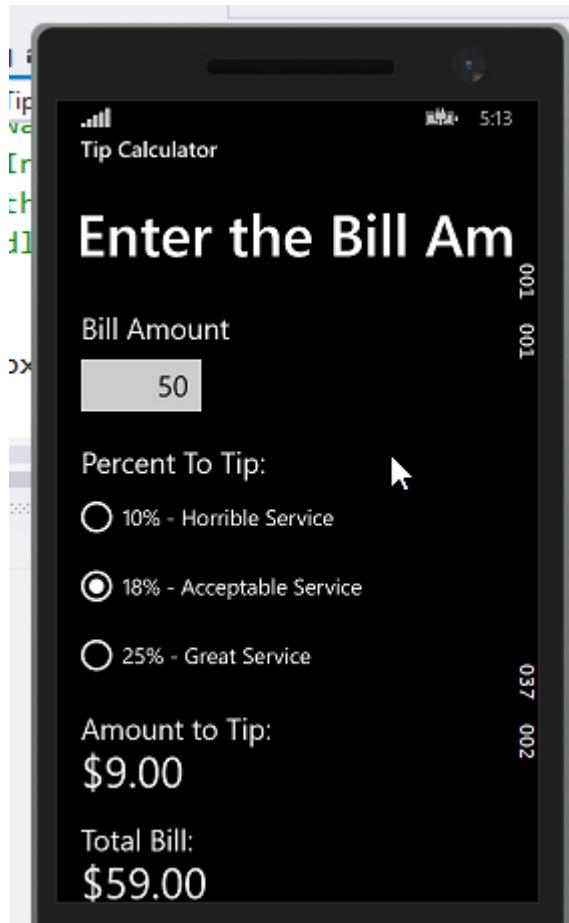
private void RadioButton_Click(object sender, RoutedEventArgs e)
{
    performCalculation();
}

```

At this point, when testing the app, it works correct, but there are a few inconveniences. Ideally when I tap the Bill Amount TextBox to edit the value, it would remove the number and not require I use the delete button to remove the existing value before typing in a new one:



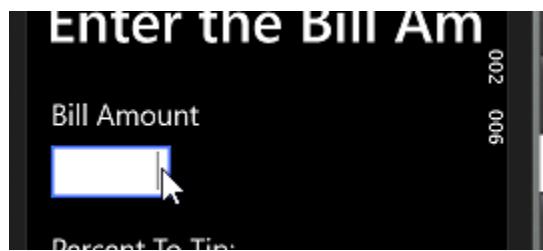
Also, I would like for the new value to be formatted as dollars and cents correctly when I am finished typing it in the Bill Amount TextBox:



To accommodate these desired features, I'll first clear out the bill amount TextBox when it gets focus like so:

```
private void amountTextBox_GotFocus(object sender, RoutedEventArgs e)
{
    billAmountTextBox.Text = "";
}
```

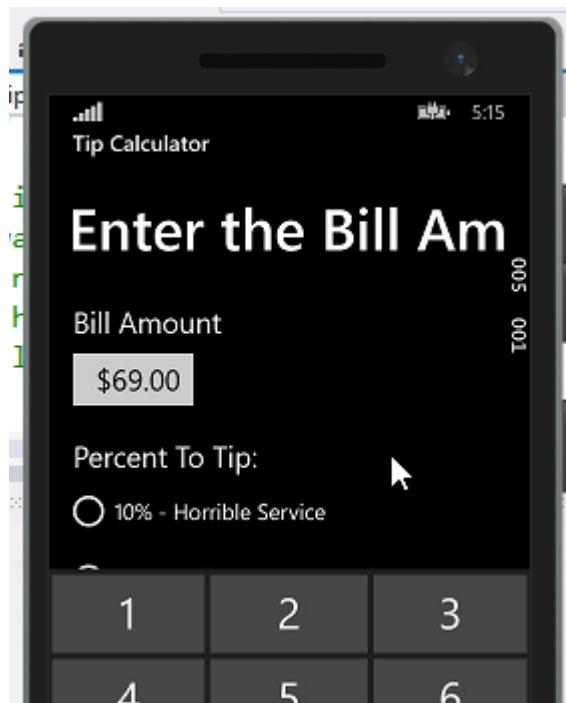
Now, when I tap in the TextBox, the previous value is cleared out. Perfect!



To accommodate the desire for the value to be properly formatted when the user exits the bill amount TextBox, I'll retrieve the BillAmount property counting on the fact that this property is properly formatted from calls to the Tip class' CalculateTip() method:

```
private void amountTextBox_LostFocus(object sender, RoutedEventArgs e)
{
    billAmountTextBox.Text = tip.BillAmount;
}
```

Now, when I leave the bill amount TextBox, the value I typed will be nicely formatted as dollars and cents:



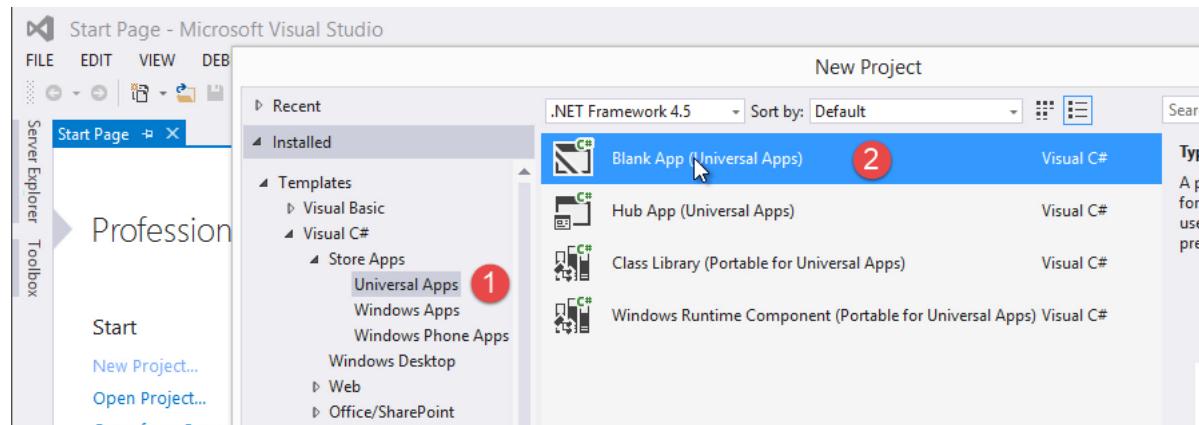
Admittedly, if this were a complete app, I would want to add custom tiles, a splash screen, etc. We'll do that when we build our next app several lessons from now.

In the next lesson, we'll rebuild this app as a Universal app that can be submitted to both the Phone and Windows 8 stores.

Lesson 10: Exercise: Tip Calculator as a Universal App

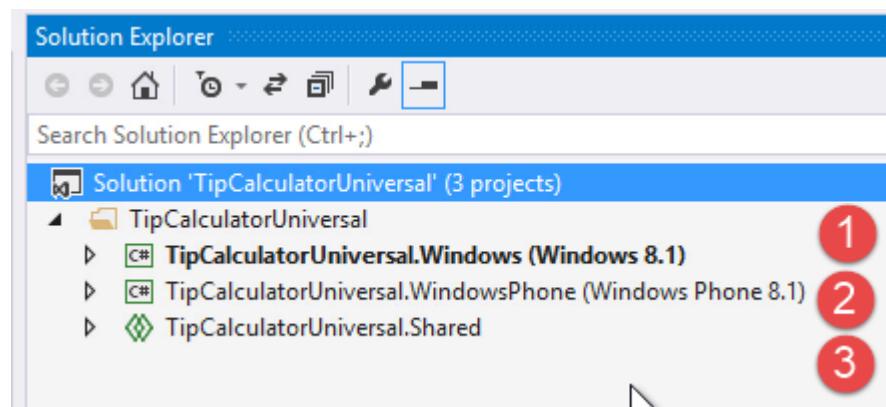
In this lesson we're going to take the work that we did in the previous lesson and translate it into a Universal App, which will allow us to distribute our Tip Calculator to both the phone and to the Windows 8.1 App Store for inclusion there.

To begin, (1) In the New Project template, select Templates > Visual C# > Store Apps > Universal Apps. (2) Select the Blank App (Universal Apps) project template.



Name the project “TipCalculatorUniversal” and click the OK button to close the New Project dialog and create the new app.

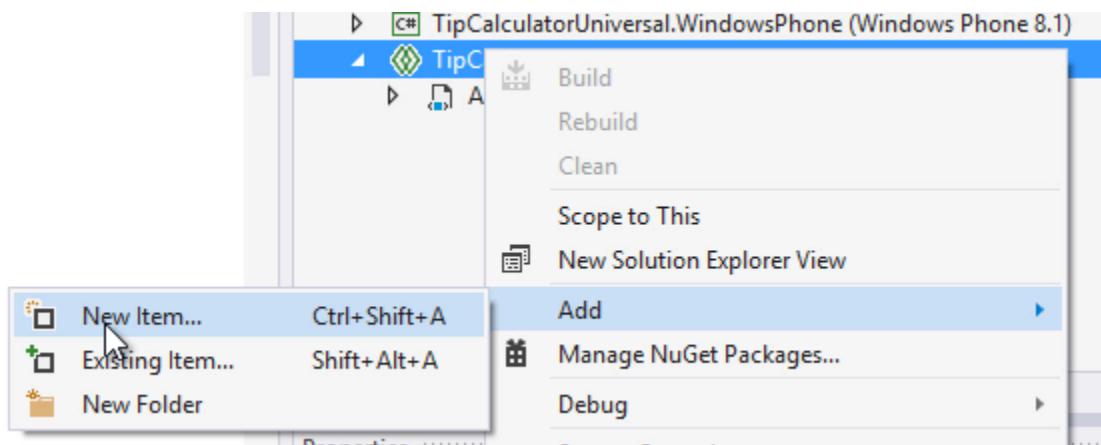
Once the new app is created, the Solution Explorer reveals three projects all prefixed with the solution name, then a dot, then:



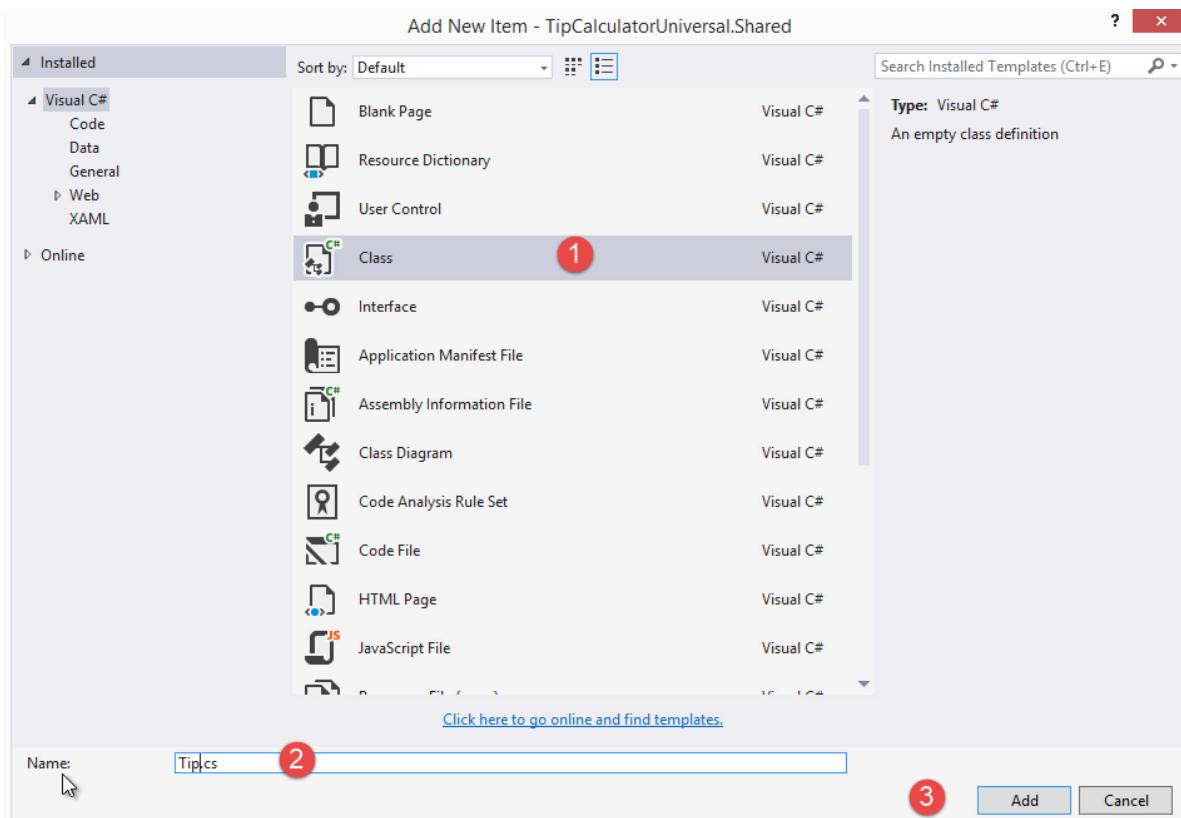
- (1) Windows
- (2) WindowsPhone
- (3) Shared

Basically, the Windows and the Windows phone projects should only be different in so much that they have similar but different XAML to lay out based on screen size, for their specific differences. Then, anything that we can share should be added to this Shared project. So for our Tip Calculator app, this architecture is very straight forward. We'll put our Tip.cs class in the shared project and then reference it from both our WindowsPhone project and the Windows project.

To begin, let's start with our shared project. I'm going to right-click the shared project and select Add > New Item ...:

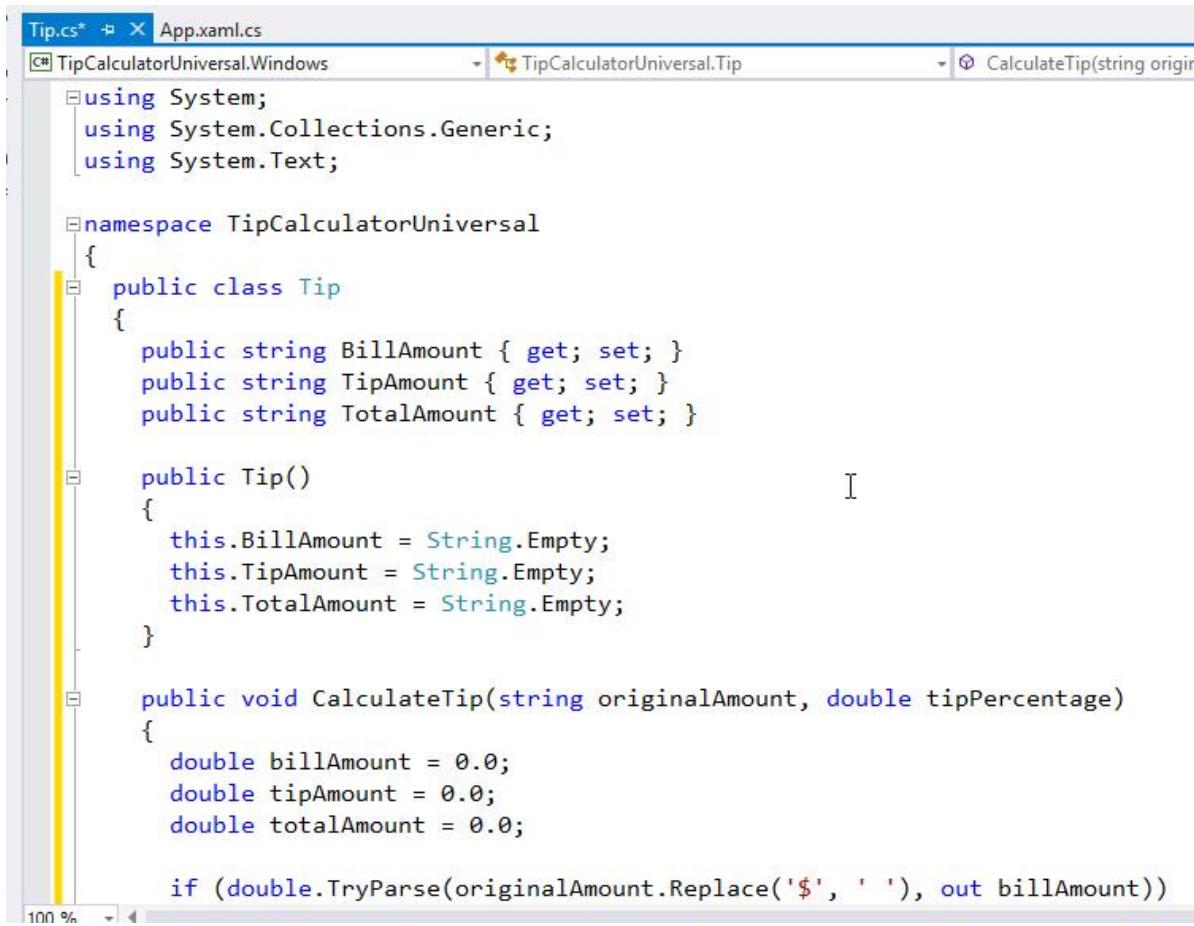


In the Add New Item dialog:



- (1) Choose Class
- (2) Name it: Tip.cs
- (3) Click Add

I'll copy and paste the work that I did in the previous lesson so we have exact duplicate. No changes required here.



The screenshot shows the Visual Studio IDE with the Tip.cs file open. The code defines a class named Tip with properties for BillAmount, TipAmount, and TotalAmount, and a constructor that initializes them to empty strings. It also contains a CalculateTip method that parses the original amount and calculates the tip and total amounts.

```
Tip.cs* + X App.xaml.cs
TipCalculatorUniversal.Windows      TipCalculatorUniversal.Tip      CalculateTip(string origin)

using System;
using System.Collections.Generic;
using System.Text;

namespace TipCalculatorUniversal
{
    public class Tip
    {
        public string BillAmount { get; set; }
        public string TipAmount { get; set; }
        public string TotalAmount { get; set; }

        public Tip()
        {
            this.BillAmount = String.Empty;
            this.TipAmount = String.Empty;
            this.TotalAmount = String.Empty;
        }

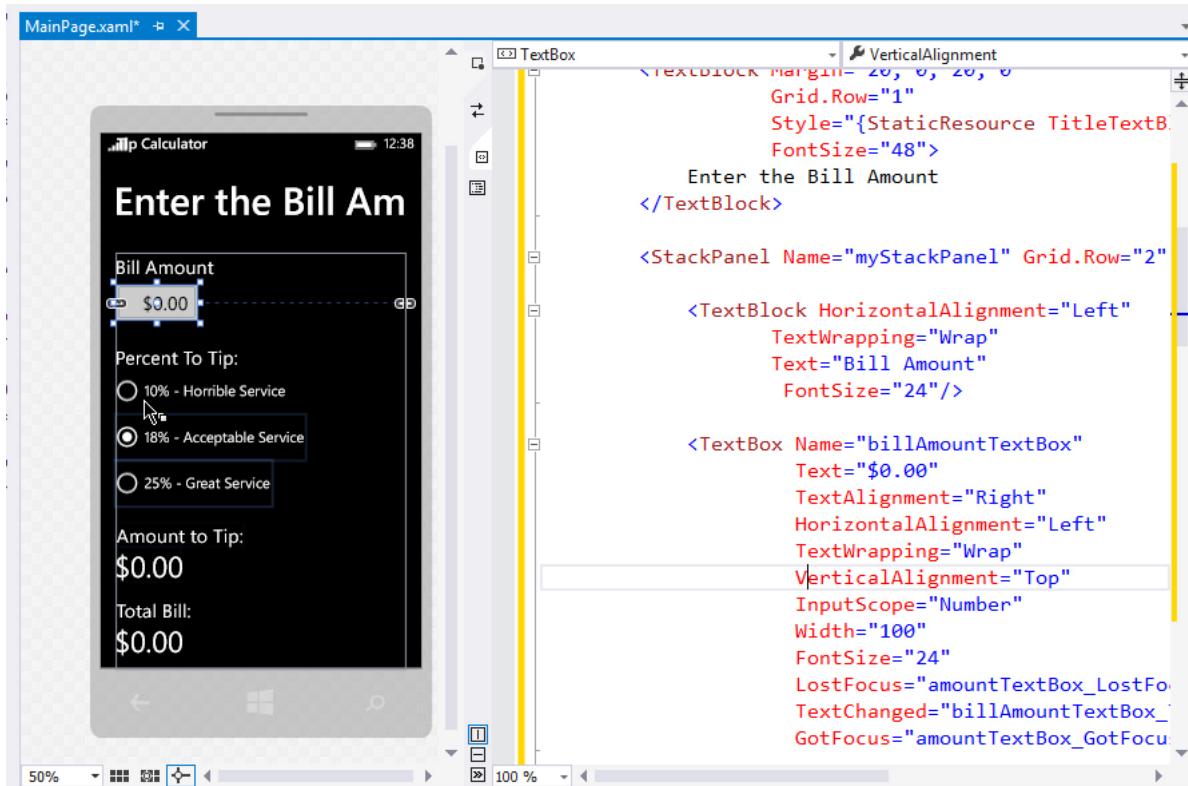
        public void CalculateTip(string originalAmount, double tipPercentage)
        {
            double billAmount = 0.0;
            double tipAmount = 0.0;
            double totalAmount = 0.0;

            if (double.TryParse(originalAmount.Replace('$', ' '), out billAmount))

```

Now that Tip class will be available to both my Windows and Windows phone projects.

Next, we'll work on the Windows Phone project since we've already created that once. This should be just a matter of going to my MainPage.xaml and copying and pasting all the XAML that I created previously. After copying and pasting, I have the following result:



Next, I'll re-create the event handler method stubs by selecting each one in XAML and press the F12 keyboard command. When finished, the MainPage.xaml.cs should look similar to this:

```

private void billAmountTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
}

private void amountTextBox_GotFocus(object sender, RoutedEventArgs e)
{
}

private void RadioButton_Click(object sender, RoutedEventArgs e)
{
}

```

In my MainPage.xaml.cs, I'll add the PerformCalculation() helper method (and that should be the same as what I created in the previous lesson).

```
private void performCalculation()
{
    var selectedRadio = myStackPanel.Children.OfType<RadioButton>().FirstOrDefault(r =>
        r.IsChecked);
    tip.CalculateTip(billAmountTextBox.Text, double.Parse(selectedRadio.Tag.ToString()));

    amountToTipTextBlock.Text = tip.TipAmount;
    totalTextBlock.Text = tip.TotalAmount;
```

Near the top, we'll need to create a private field that will hold a reference to the Tip class:

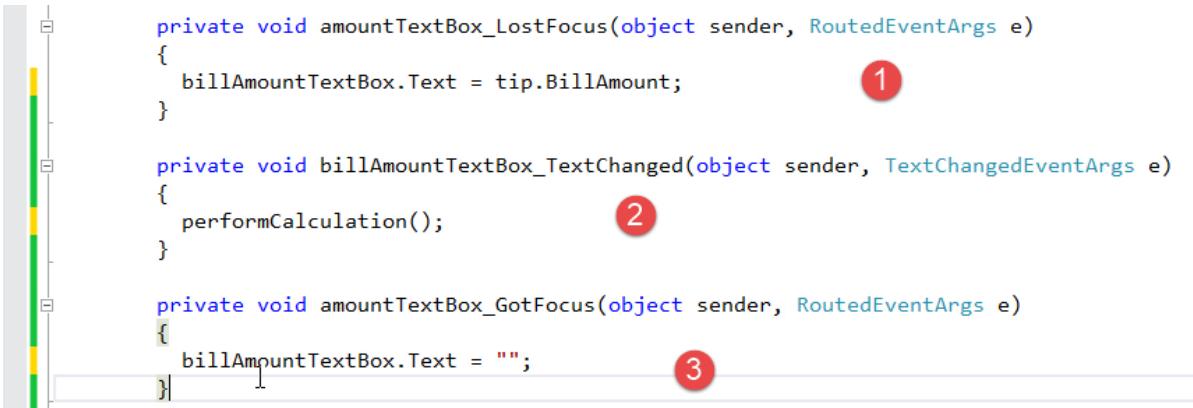
```
public sealed partial class MainPage : Page
{
    private Tip tip;
```

In the RadioButton_Click event handler, I want to call the performCalculation() helper method.

```
private void RadioButton_Click(object sender, RoutedEventArgs e)
{
    performCalculation();
}
```

Next:

- (1) In amountTextBox_LostFocus, I want to make sure to set the billAmountTextBox.Text to the tip.BillAmount to make sure it is formatted correctly.
- (2) In billAmountTextBox_TextChanged, I want to performCalculation() to make sure we're updating the new values.
- (3) In billAmountTextBox_GotFocus, I want to clear the contents when the TextBox control receives focus so that I can enter fresh values.



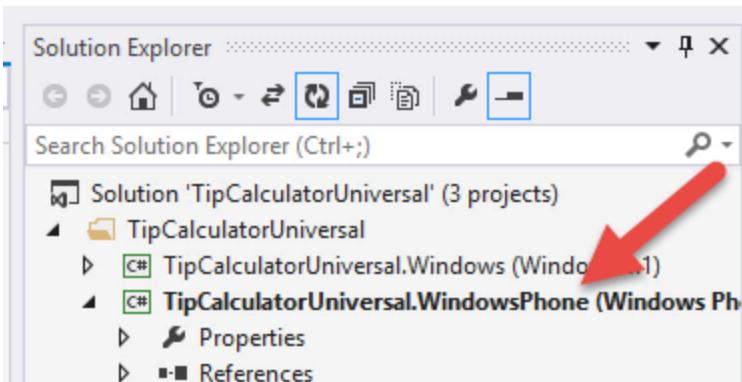
```
private void amountTextBox_LostFocus(object sender, RoutedEventArgs e)
{
    billAmountTextBox.Text = tip.BillAmount; ①
}

private void billAmountTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    performCalculation(); ②
}

private void amountTextBox_GotFocus(object sender, RoutedEventArgs e)
{
    billAmountTextBox.Text = ""; ③
}
```

It's time to test the app in the Emulator.

Important: Currently the Windows project is selected as the Start Up Project. You can see that because it is in bold in the Solution Explorer. To change the Start Project that will execute when we start debugging to the Windows Phone project, right-click the TipCalculatorUniversal.WindowsPhone project in the solution explorer and select "Set as Start Up Project". As you can see, the project name is in a bold font as it appears inside of the Solution Explorer.

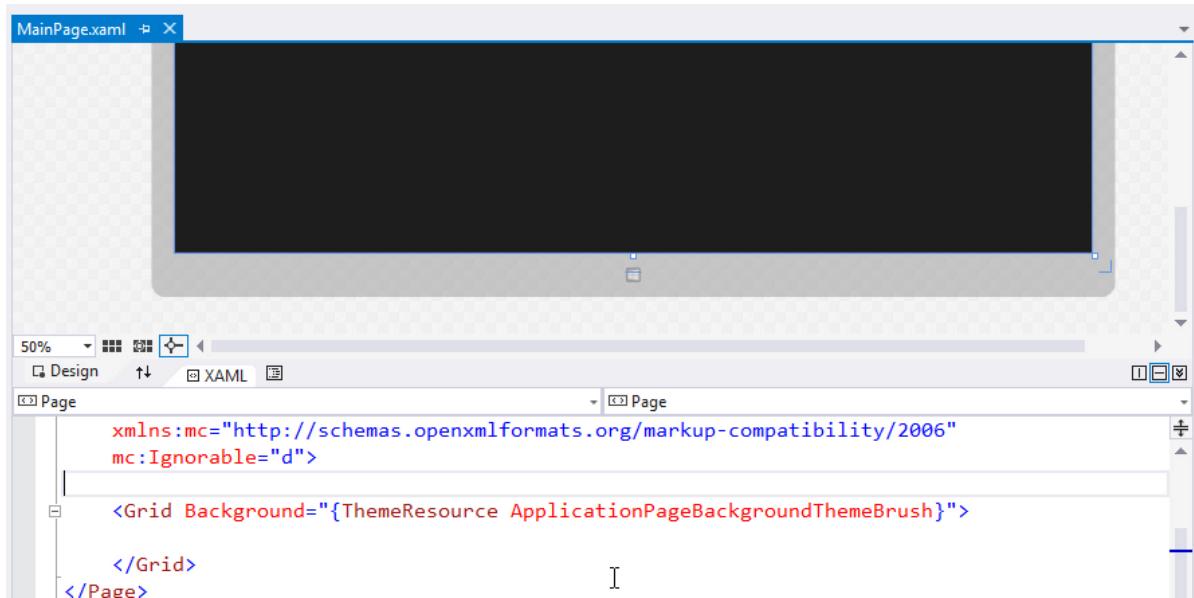


Select the start debugging button to run the app in the Emulator. It should work exactly as it did in the previous lesson.

Now, let's go ahead and create the Windows version of the app.

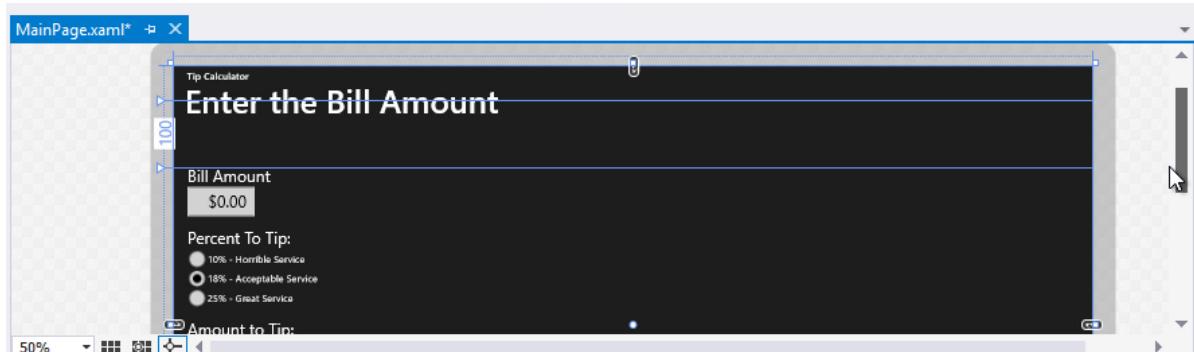
First, I'll reset the TipCalculatorUniversal.Windows project as the Start Up Project using the technique I just demonstrated.

Next, I'll open the MainPage.xaml of the Windows project. You can see we get more of the Microsoft Surface preview in the XAML Editor.



The purpose of this lesson is to demonstrate how little effort will be required to take our Windows Phone app and turn it into a Windows app. To that end, I'll begin by literally copying all of the XAML from my Phone's user interface and pasting it into the XAML editor for the Windows app.

Once I copy and paste everything in the body of my page, I can see the preview displays my work:

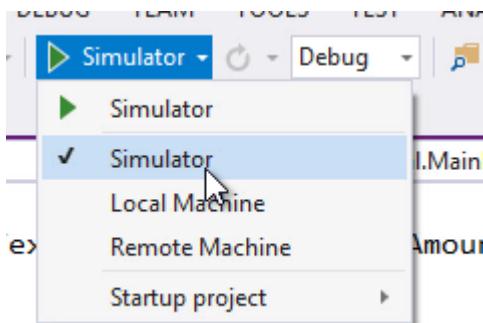


Admittedly, there's a lot of blank space on the right-hand side that we're not using, and I certainly could re-work a lot of this to make it utilize some of that blank space, however as a first pass, I'm pretty happy with this. It's the promise and the value of XAML that I'm able to reuse it both these contexts, and it works well.

Beyond the XAML layout, we'll need to modify the MainPage.xaml.cs using the technique I demonstrated earlier, using F12 to Go To Definition ... and create the event handler method stubs.

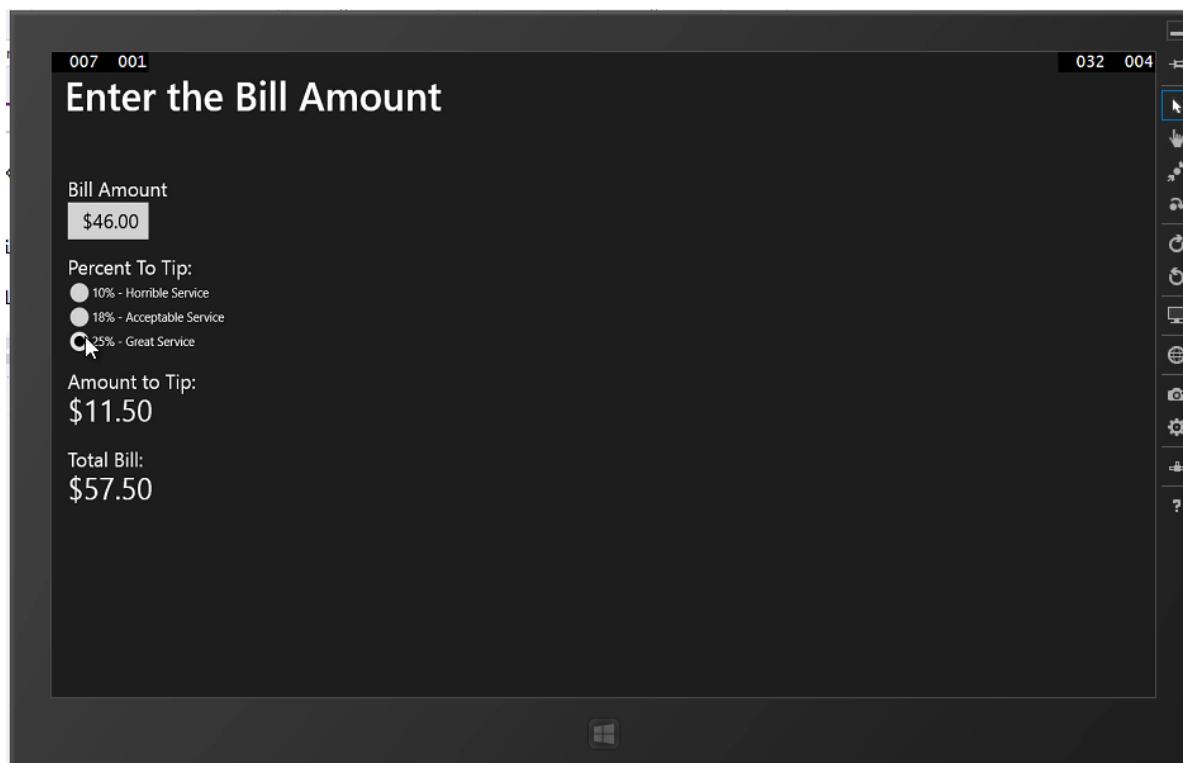
The code in the code behind is identical to what we created before, so use the techniques and the code we used earlier in this lesson for the event handlers, the reference to the Tip class, and the performCalculation() helper method.

Once you've finished that, select the arrow next to the debug button that is currently set to Local Machine and change it to Simulator:



Click the debug button.

In the simulator, you can work with the app as expected:



As you can see, you can create a Universal App using one code base (the Tip class) and almost identical XAML for layout. Admittedly, I could tweak the Windows version of the XAML if I wanted to make it look good on the larger form factor, but even when I didn't modify it, it looked just fine, at least for this initial version.

So I encourage you to do this: Whatever you're going to build, build it for both Windows and Windows Phone. Structure your shared logic with that in mind. Once you keep your shared logic / business rules / data access in the Shared project, you'll just reference methods and so forth that are inside of the classes of your Shared project from both the Windows and the Windows phone projects.

Lesson 11: Working with the Windows Phone 8.1 Emulator

We've seen the Windows Phone Emulator at work in this series. It is a crucial component to developing apps for the Windows Phone platform, so I wanted to spend a little time becoming more familiar with it and point you in the right direction for more information.

Our game plan in this lesson ...

- (1) We'll learn about what the Windows Phone Emulator really is, and how it supplies different versions for different deployment scenarios.
- (2) We'll learn about the function of the Emulator, including keyboard shortcuts that emulate device buttons.
- (3) We'll learn about the controls to resize, rotate and simulate the act of handling the virtual device as if it were a physical one, with accelerometer support, GPS support and more.

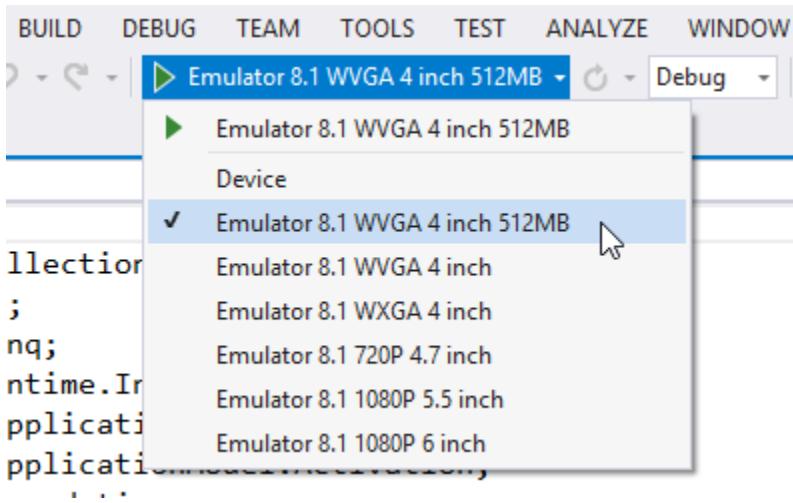
What is the Windows Phone Emulator?

In a nutshell, the Windows Phone Emulator is a desktop application that simulates a Windows Phone device, and actually provides similar performance to a physical Windows Phone device. It provides a virtualized environment in which you can debug and test Windows Phone apps without a physical device -- in fact the emulator is running Microsoft's Hyper-V.

Now, while the Emulator is great for development and quick debugging sessions, before you publish your app to the Windows Phone Store, Microsoft recommends that you actually test your app on a real Windows Phone.

Choosing different versions of the Emulator for debugging

Up to now, when we clicked the Run / Debug button on Visual Studio's toolbar, we were running the Emulator in its default configuration ... something called WVGA 512MB.



What does WVGA and 512MB really mean?

The 512MB indicates that we're running in a memory constrained environment ... The default emulator image in Visual Studio is Emulator WVGA 512MB, which emulates a memory-constrained Windows Phone 8.1 phone. So, for example, the Lumia 610 is an inexpensive entry level Windows Phone 8.1 device which sports only 256MB of RAM. By contrast, the Lumia 920 has 1 GB of RAM. On lower-RAM devices, having multiple apps running at the same time or creating a memory intensive app could cause performance problems. So, to be sure that your app will run well on lower-RAM devices you can test your app using a realistic Emulator image.

There are a number of great articles about RAM usage on MSDN ... let me point out the best starting point to learn more:

App performance considerations for Windows Phone

[http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff967560\(v=vs.105\).aspx](http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff967560(v=vs.105).aspx)

Optimizing Apps for Lower Cost Devices

http://blogs.windows.com/windows_phone/b/wpdev/archive/2012/03/07/optimizing-apps-for-lower-cost-devices.aspx

What does "WVGA" as well as those other acronyms stand for?

The Emulator allows you to test your app on a unique emulator image for each of the screen resolutions supported by Windows Phone. This default selection encourages you to target the largest possible market for your Windows Phone 8 app.

- WVGA (800 × 480)
- WXGA (1280 × 768)
- 720p (1280 × 720)

- 1080p (1920 x 1080)

If you run the default, then go to the Settings app in the About button, you can see this confirmed.



How does this translate into the actual phones on the market?

Lumia 1520

Display Size: 6 inch

Display Resolution: Full 1080p (1920 x 1080)

Lumia Icon

Display Size: 5 inch

Display Resolution: Full 1080p (1920 x 1080)

Lumia 1020 and 920

Display size: 4.5 inch

Display resolution: WXGA (1280 x 768)

Lumia 820

Display size: 4.3 inch

Display resolution: WVGA (800 x 480)

Lumia 610

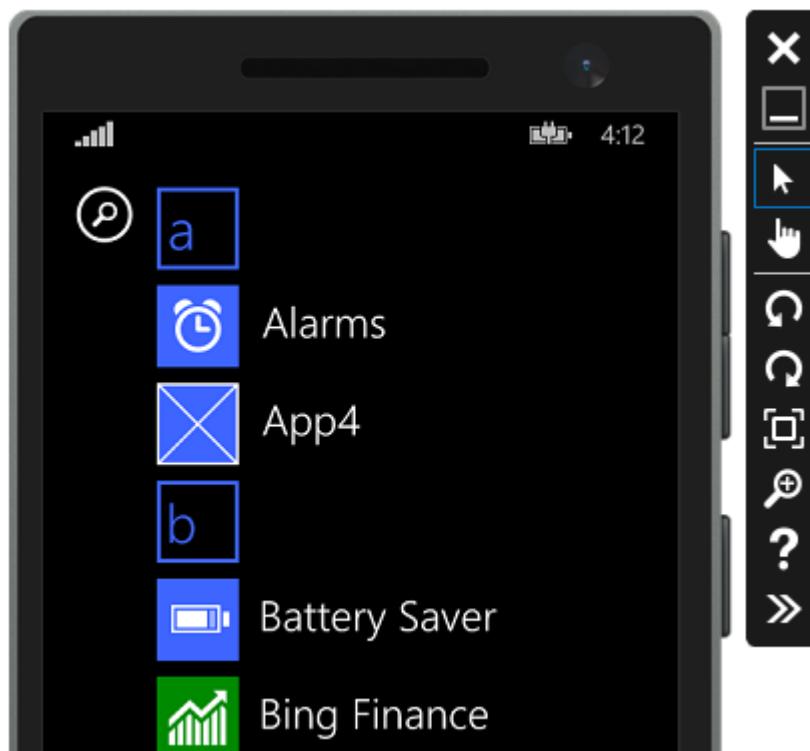
Display size: 3.7 "

Display resolution - WVGA (800 x 480)

I realize by the time you watch this there may be newer phone models. The point is that you should be aware of the fact that you'll need to support different screen resolutions and memory constraints. Like I talked about in the lesson on Layout using XAML, you will want to eliminate specific pixel sizes for layout (except for margins). Choosing from the different Emulator sizes, you can make sure you're on the right track.

Working with the Phone Emulator's special features

I'll not spend a lot of time on the phone's basic navigation. A lot of this you'll pick up by using the Emulator throughout the series if you don't already have a physical phone in your possession. Essentially, you'll have an almost identical experience, complete with the Start and Apps page, search, web browsing, clock, battery, etc.



You have the same hardware buttons on the bottom of the Emulator as you would have on a Windows Phone.



However, you'll be missing the buttons on the side of the phone. For example, my Lumia 920 has three buttons on the side ... a volume up and down, a power button, and a camera button. These can be accessed in the Emulator with keyboard function keys.

There's a list of keyboard mappings here:

[http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff754352\(v=vs.105\).aspx#BKMK_KeyboardMapping](http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff754352(v=vs.105).aspx#BKMK_KeyboardMapping)

F6 - Camera button half-way

F7 - Camera button

F9 - F10 raise and lower the volume

F11 - plays / pauses audio ... it simulates an in-line headphone button that pauses music and answers incoming phone calls. If you double-tap the button, it'll skip to the next audio track on your playlist or album

F12 - Power button / lock screen

F1 - Back button

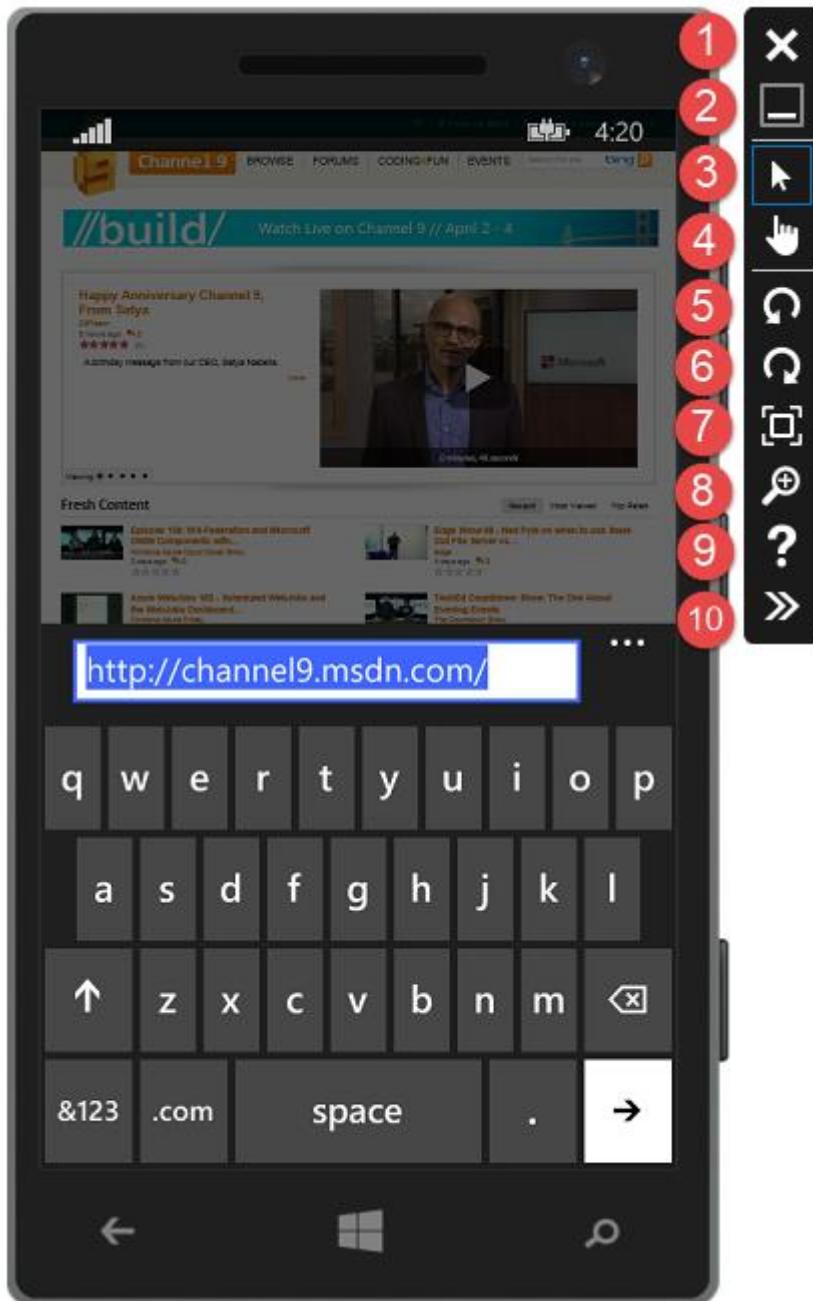
F2 - Windows key

F3 - Search button

PAGE DOWN button - When a textbox is the target input, PAGE DOWN disables the virtualized "hardware" keyboard down, and you can use your physical keyboard for input

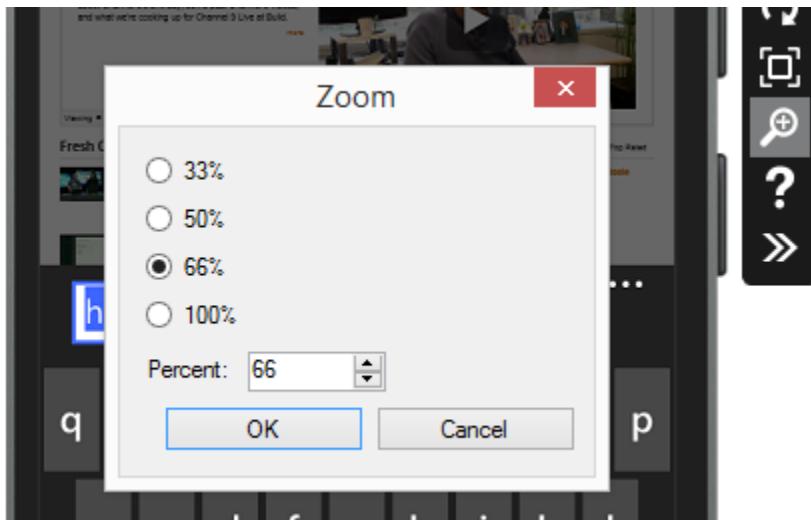
PAGE UP button - When a textbox is the target input, PAGE UP enables the virtualized "hardware" keyboard.

PAUSE/BREAK button - Toggles the keyboard, so you could just use that all the time.



In addition to the "virtualized" phone on screen, there's a floating menu off to the right. A little experimentation will reveal that the first six buttons, in order:

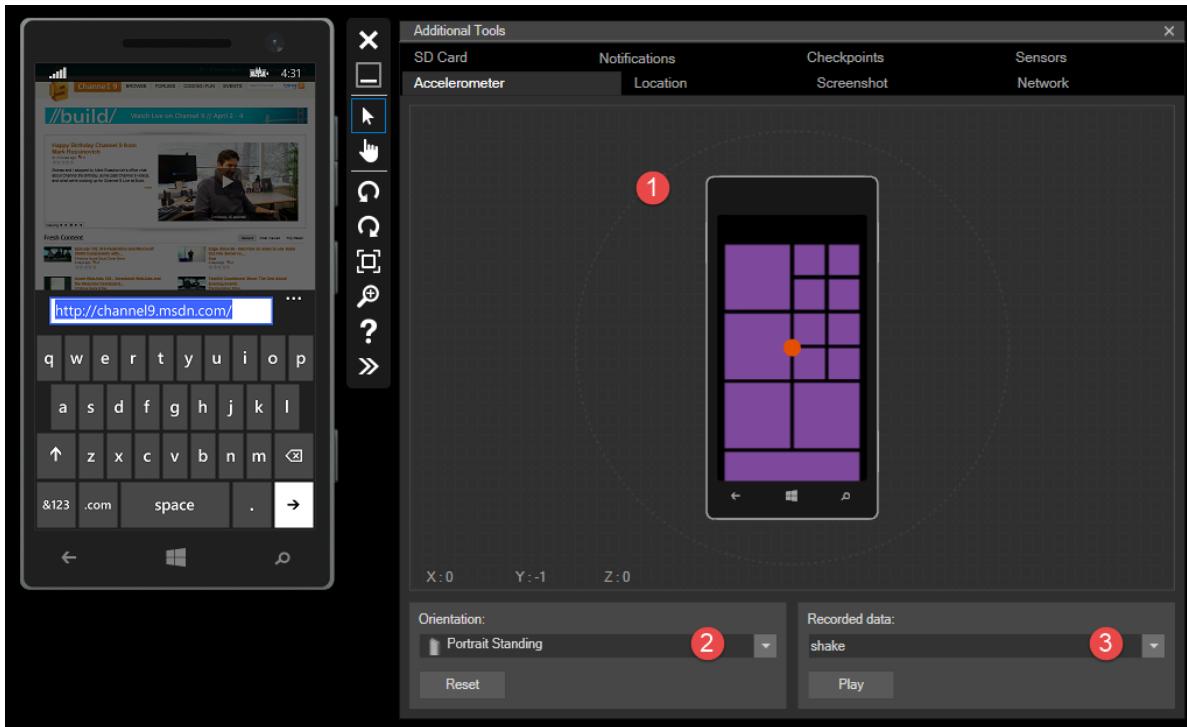
- (1) Shuts down the emulator
- (2) Minimizes the emulator
- (3) Changes to single-finger selection mode
- (4) Changes to two-finger selection mode (for pinch gesture)
- (5) Rotates the emulator 90 degrees counter clockwise
- (6) Rotates the emulator 90 degrees clockwise
- (7) Expands the emulator to the largest size that can fit comfortably on your computer screen
- (8) Brings up the zoom dialog



(9) Presumably this will open up a help portal online, however that hasn't been implemented at the time I write this.

(10) The double chevron at the bottom opens up the additional tools dialog.

There are eight tabs on the Additional Tools dialog. The first is the Accelerometer which can be used to test apps that utilize the accelerometer sensor.



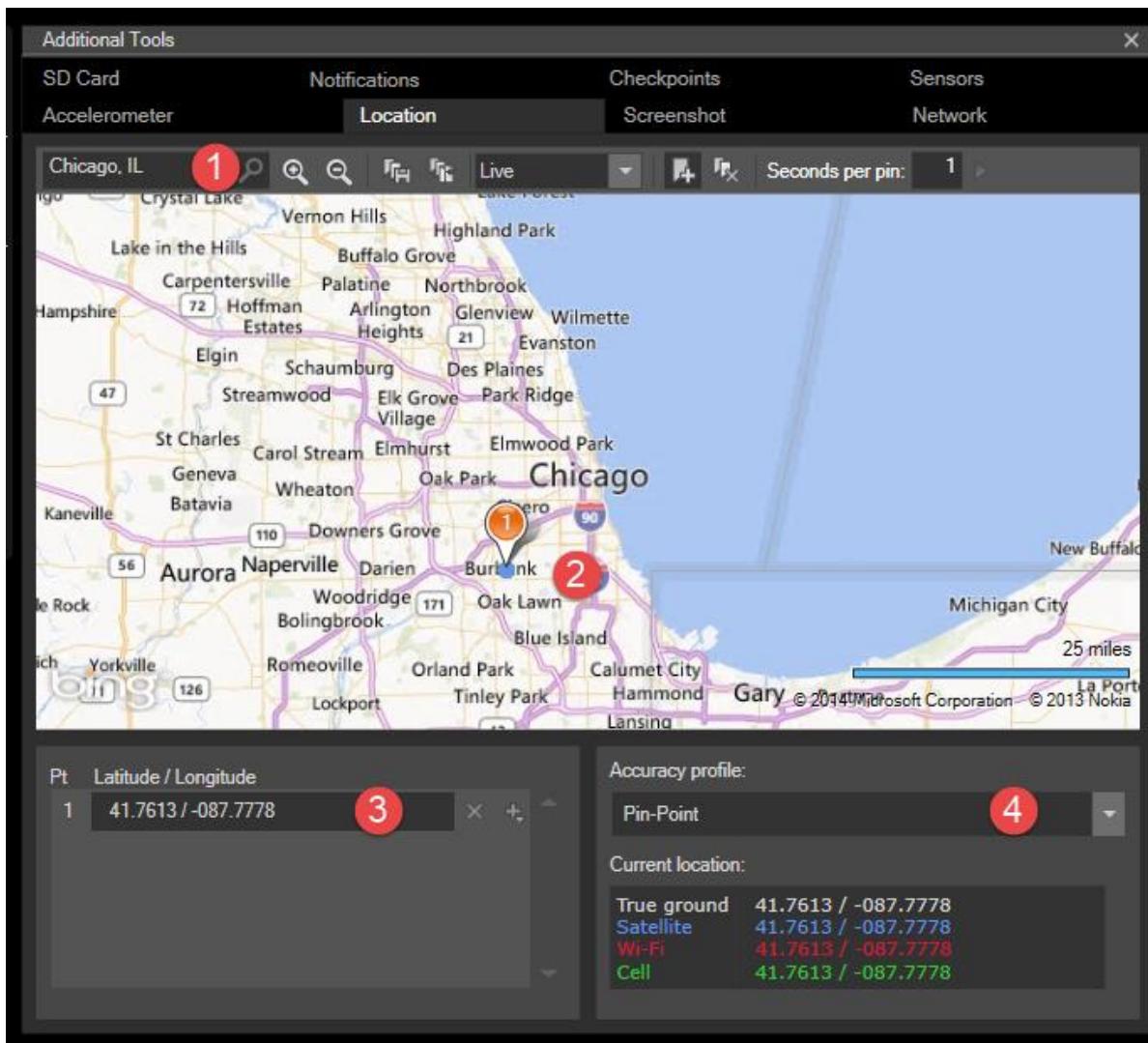
- (1) You can change the center point of the phone to change its position in 3D space by clicking and dragging the orange dot in the circle.
- (2) You can change the Orientation to one of the presets
- (3) You can play recorded data, like a "shake" motion.

For more information about the Emulator and the Accelerometer sensor in the Windows Phone SDK, a good starting spot would be this article:

How to test apps that use the accelerometer for Windows Phone

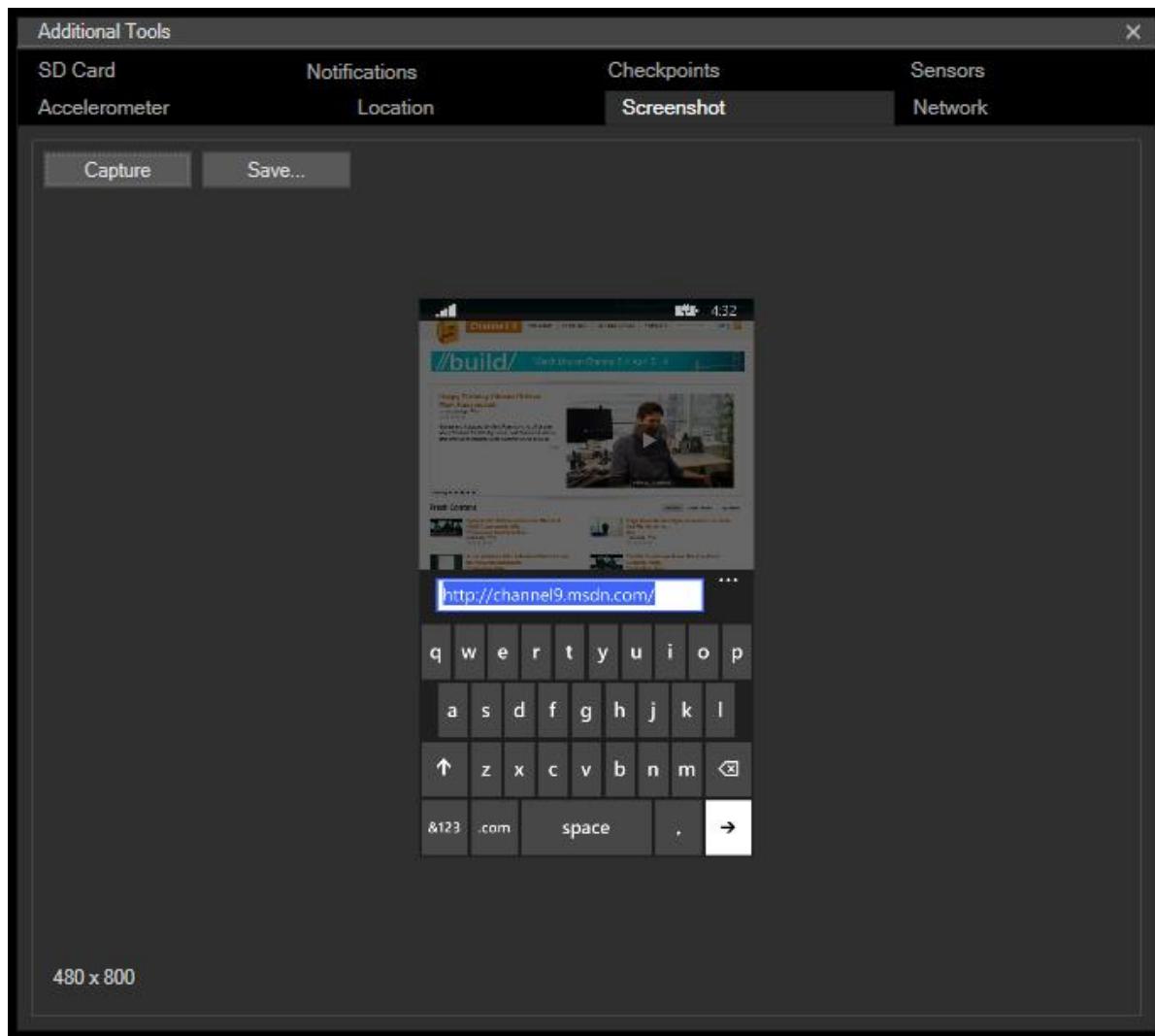
[http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh202936\(v=vs.105\).aspx](http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh202936(v=vs.105).aspx)

The next tab is the Location tab which allows you to set the current location of the phone. So, even though I'm sitting in Dallas, Texas, I can act like I'm testing my phone in Seattle or any other place in the world. To do this:

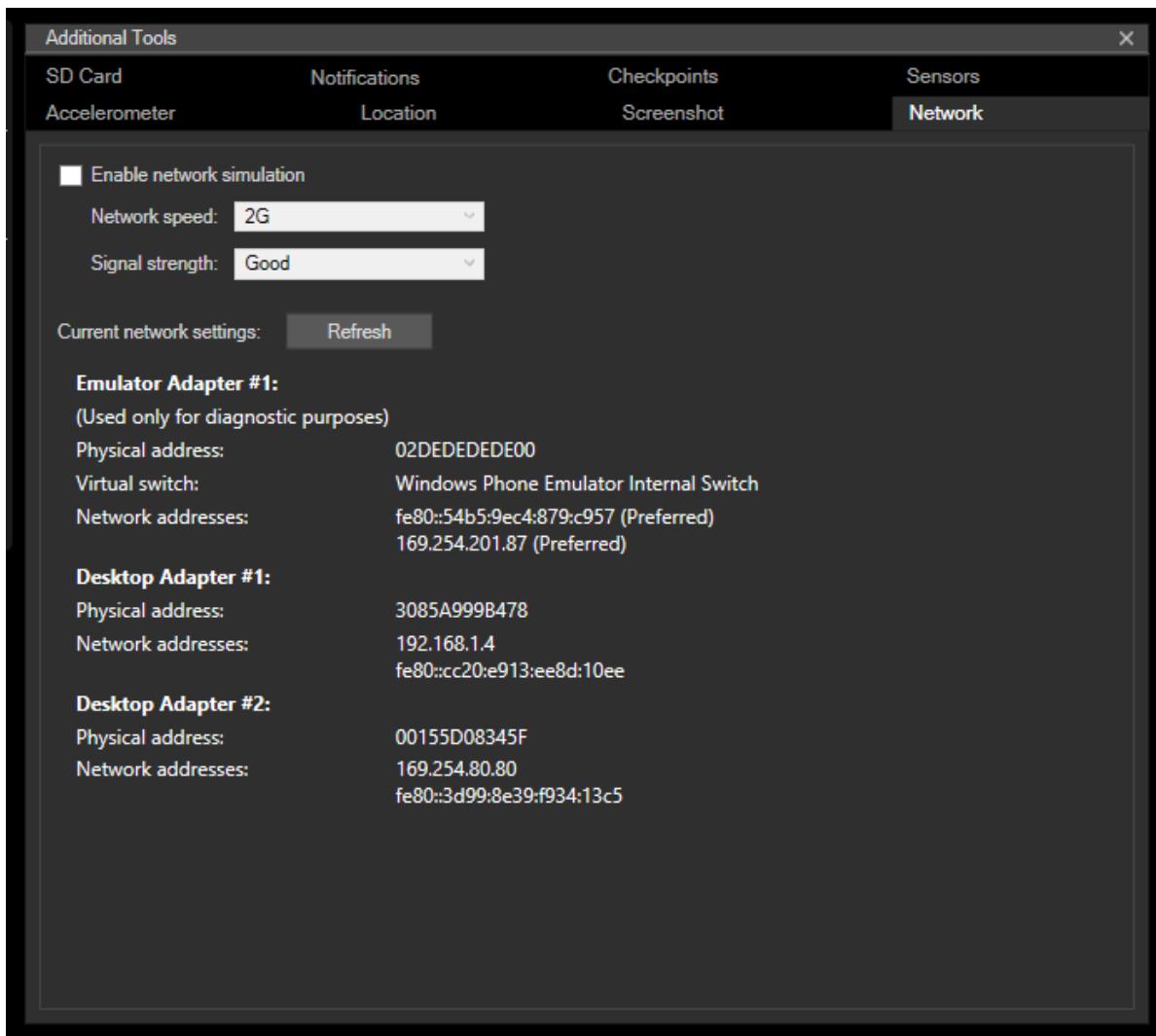


- (1) I'll perform a search for a particular location, like "Chicago, IL".
- (2) I'll right-click the map to create a pin in the suburb where I grew up.
- (3) I'll verify the exact location in the list of pins marked by their latitude and longitude.
- (4) I can introduce variance to the accuracy. In some areas, there are fewer ways to determine the exact location because there are fewer physical resources (like cell towers, etc.) to use to triangulate the location of the phone. The Accuracy profile allows you to adjust this to test your app and how it performs in various less accurate scenarios.

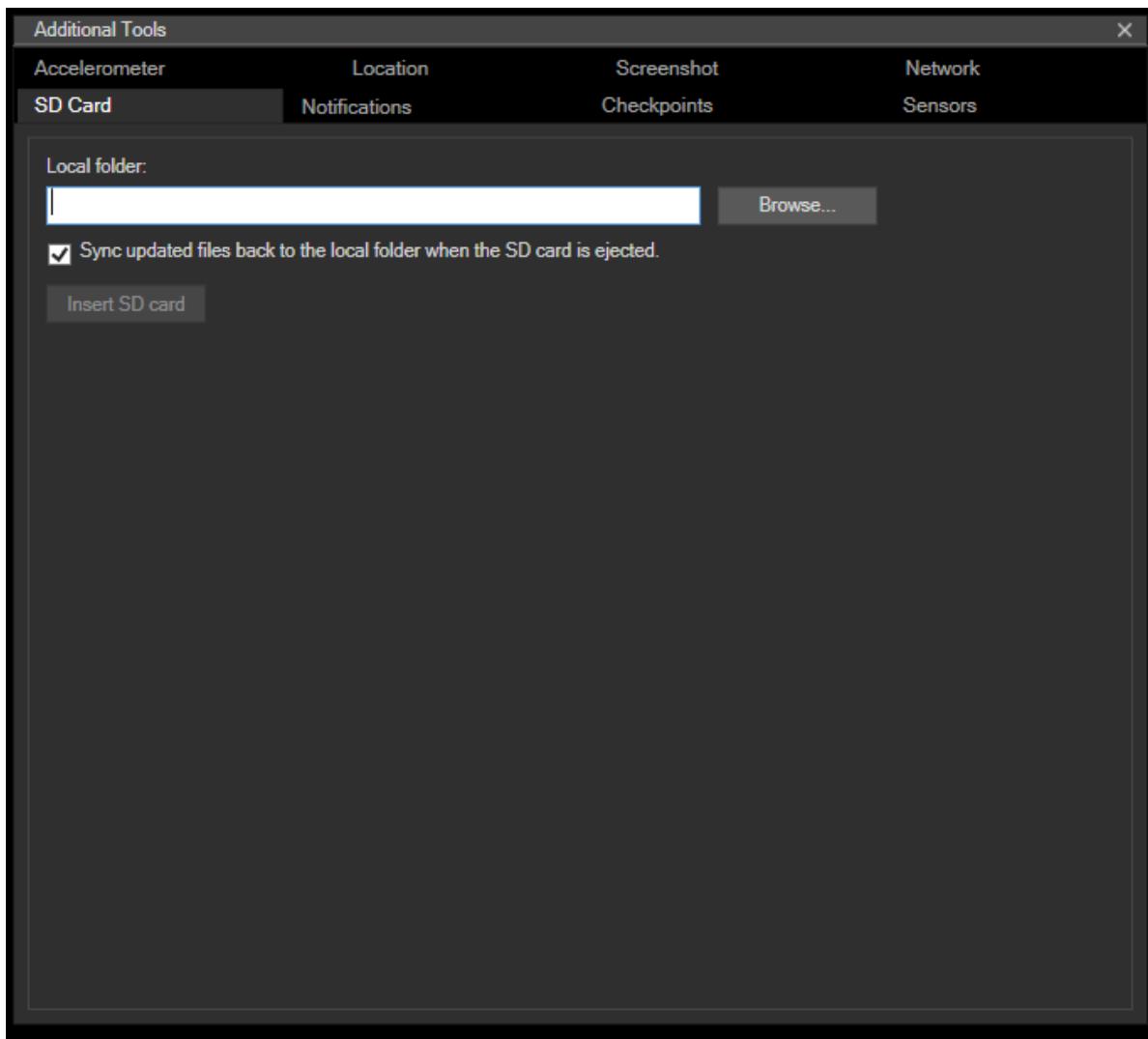
The third tab is for creating screen shots, which can be helpful when you're creating documentation or bug reports.



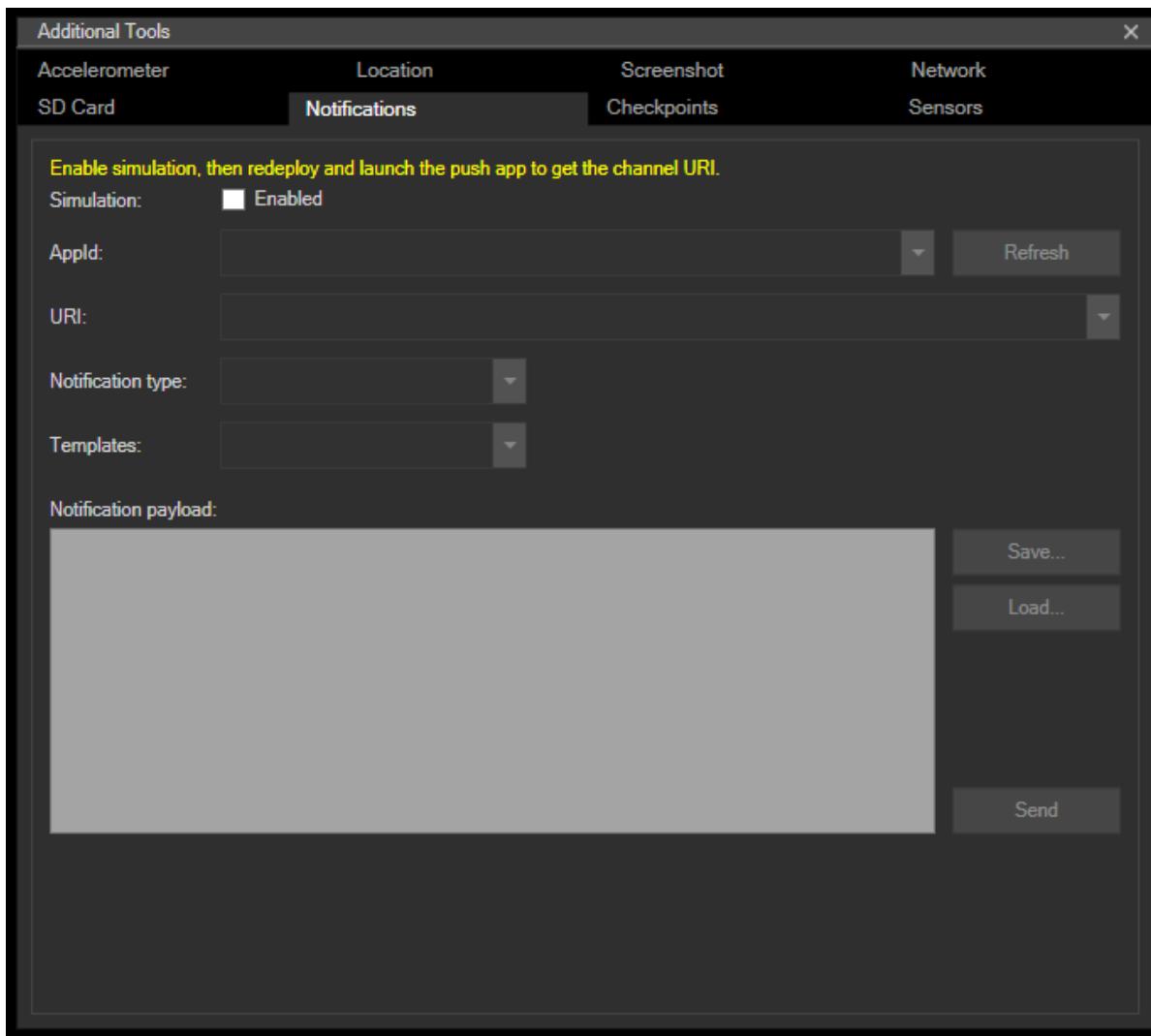
The fourth tab is the Network tab. In addition to viewing the IP address of the phone on the network (which may be helpful if you're building some network monitoring app) but also you can throttle the phone's network access to test your app's performance when in bandwidth challenged situations.



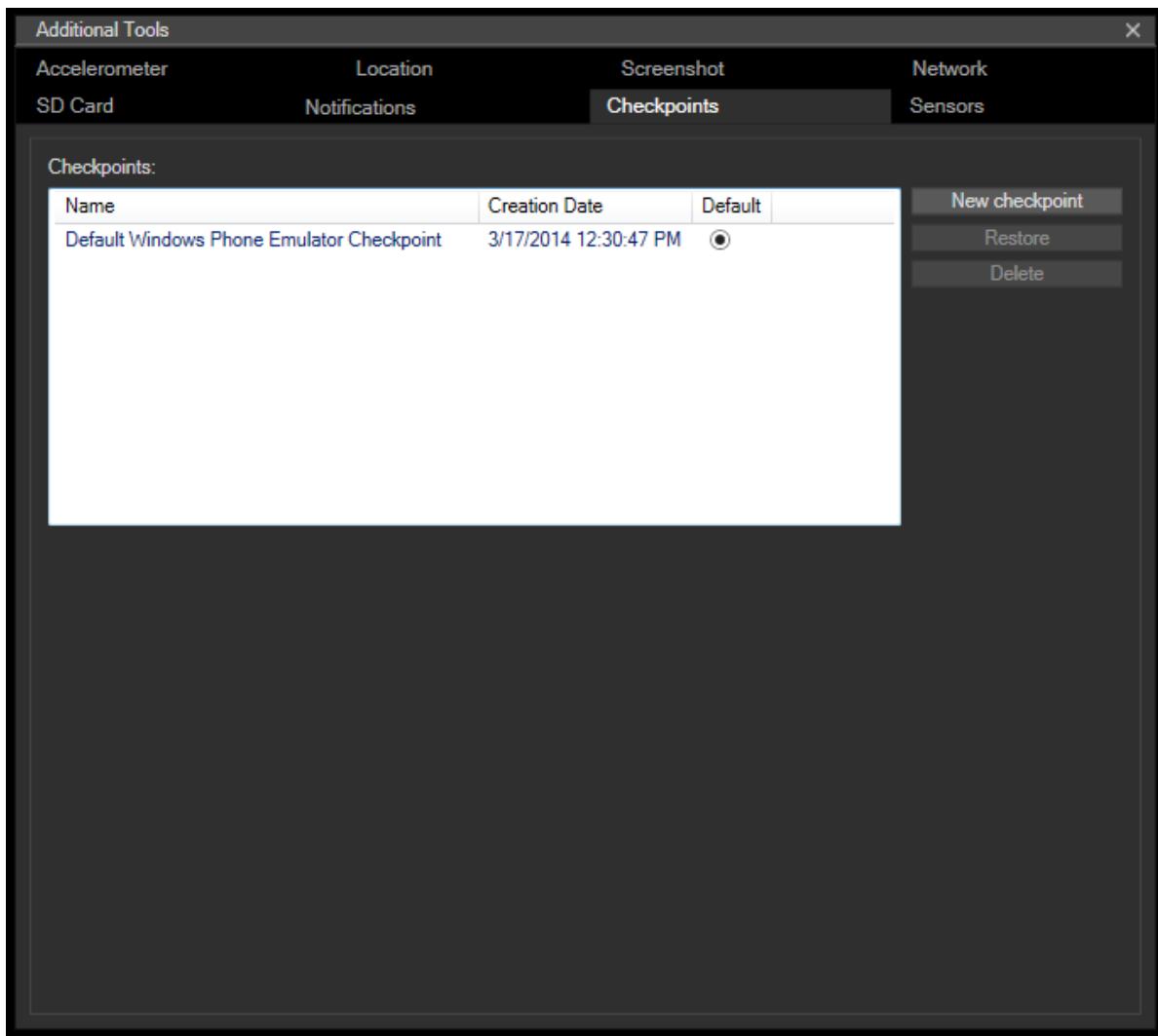
Fifth, you can designate a location on your hard drive that will simulate the use of the SD Card so that you can test whether your app is correctly saving or working with the file system of a removable SD card.



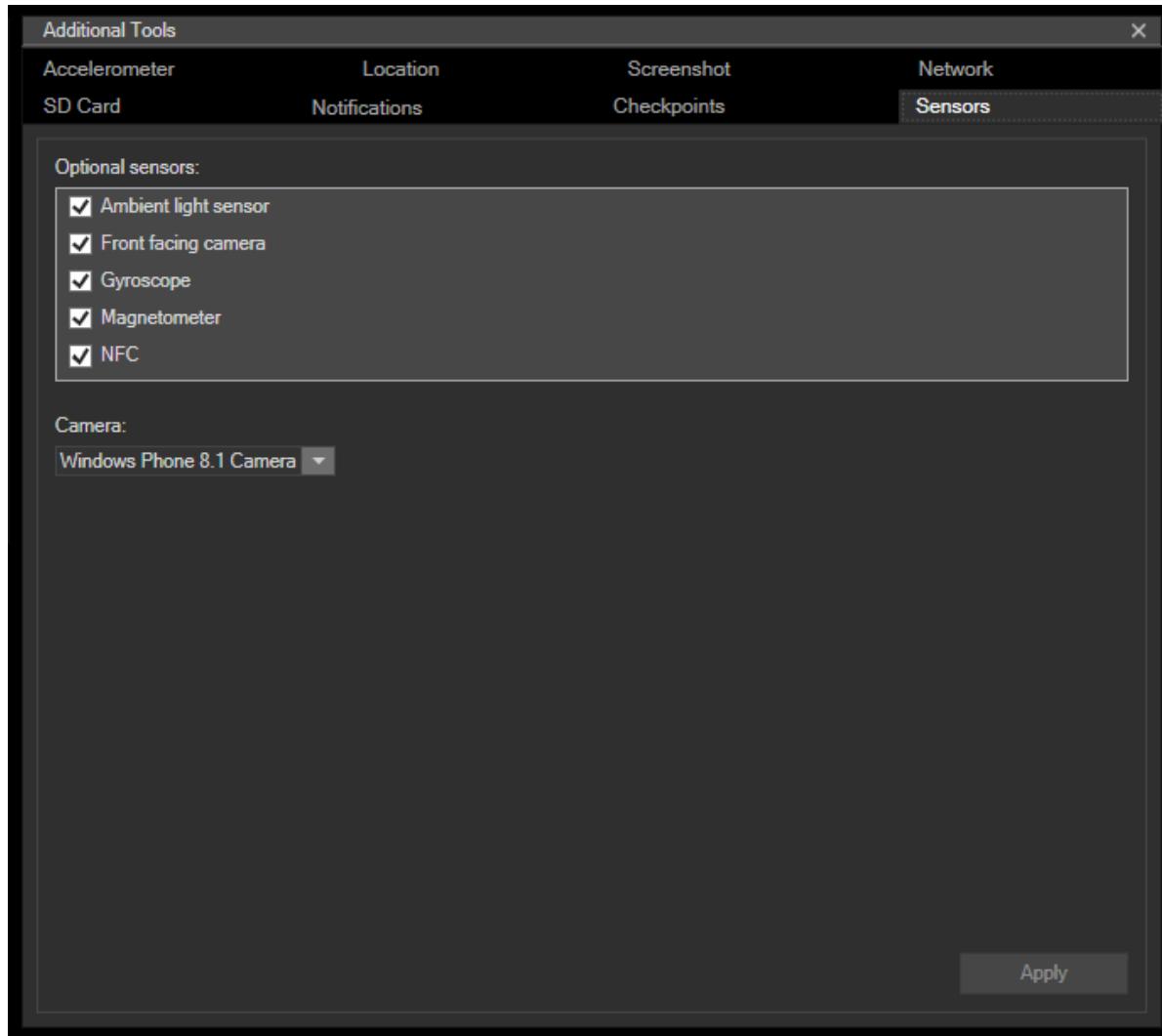
The sixth tab allows you to simulate push notifications.



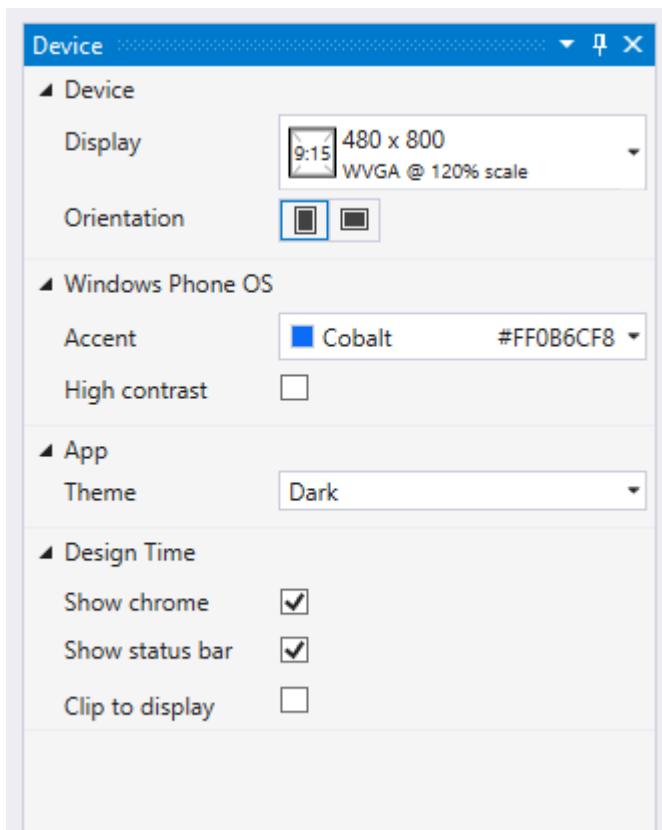
The seventh tab allows you to save the current state of the emulator as a checkpoint. This allows you to test a scenario where the phone is configured in a certain way over and over by restoring the state of the phone to a snapshot of its previous state. Think: virtual machine snapshots, because that is what is happening here.



Finally, the Sensors tab allows you to turn on and off certain sensors to test how your app works should it rely on one of those sensors and they don't exist in the user's phone.



In addition to the Emulator itself, Visual Studio has a Device window with some tooling that can affect how the XAML designer displays the phone ... orientation, theme and color, chrome, etc.



So as you see, you have so many ways to test and monitor your app during development time so you gain some confidence in how it will perform when you distribute it on the Store.

Recap

The last tip I would give about the emulator is that it's ok to just leave it up and running during development. There's no need to shut it down. When you launch and re-launch your app in debug mode from Visual Studio, part of that process is deploying the latest version of the app to the phone. It's time consuming to re-launch the Emulator each time, and Visual Studio will connect to an existing Emulator instance when it needs to.

So to recap, the Emulator is a great tool that allows you to quickly deploy your app during development to a virtualized Windows Phone. We looked at how to test different device screen sizes and memory constraints, how to manipulate the emulator to test for rotation and motion, how to make the phone think it is located geographically in a specific place, and much more. We'll use these skills extensively throughout this series.

Lesson 12: Understanding the App's Lifecycle and Managing State

In this lesson, we'll briefly talk about the important moments in the life cycle of your app. Hopefully this will give you some insight as to what is going on in your app even when it is no longer running.

In a nut shell, your app is either running, it is not running (terminated) or it is suspended. What's the difference between terminated and suspended? When a user switches to another program, chances are likely your app will be suspended for a while until the user switches back to your app. During this time, the app is in hibernation, in other words, the code in your app is not running, but the state of your application's objects and variables including the Frame and its BackStack is kept in memory. This allows the user to switch between apps quickly and fluidly.

However, it is possible the Phone's operating system will choose to terminate your app, which means it is removed from memory along with your application's objects and variables. This means that the next time the user launches your app, it will have no recollection of the state of the app when it last ran. This means any work the user was doing is lost (unless you saved it), and it means that the Frame's last page is forgotten, as is the entire BackStack.

Now, why would the Phone take a suspended app and then terminate it? There are a couple of reasons why. Usually this happens because it needs the RAM your app occupies. But it could also happen if the Phone runs out of battery or the user reboots the phone. The key is that, as an app developer, your app is notified that it is about to be suspended, however once it is suspended it is not notified about termination.

Fortunately, you can handle the `Suspending()` event to quickly save the state of your app before it is suspended.

Most of this work will happen in the App class. The App class is implemented in `App.xaml.cs`. You override those methods of the App class that you want to handle in your app. The most important is the `OnLaunched` method that executes each time your app is initially launched, and the `OnSuspending()` method which gives you a tiny window of opportunity to save the state of the app just in case it is terminated while suspended.

The App class provides a few important services that are very low level:

1. It's the entry point for the app
2. It performs lifetime management, so when the user uses the physical buttons on the phone, or the operating system tells our app to go away for now, the App class implements methods that handle those events.
3. It manages app-scoped resources, not the least of which is the Frame object for navigation purposes. More about that in a moment. You'll see later where I add a public

property to the App class so that I can reference it throughout all pages in the app. Also, recall from the lesson on styling your app that you can create global resources and styles by defining them in the `<Application.Resources>` section of the App.xaml file.

4. It will also allow you to catch with unhandled exceptions in your app.

The operating system provides your app a Window to display in. Instead of allowing you to work directly with the Window, your app creates a Frame object that represents the Window, but adds navigation features ... As we discussed in the lesson about navigation, the Frame object can load and unload objects that derive from Windows.UI.Xaml.Controls.Page (which each of your Pages will derive from). It also keeps a history of all the pages that were loaded and in which order in the BackStack, an `IList<PageStackEntry>`. Much of the setup work for the Window and Frame happens in the `OnLaunched` event handler.

Ok, so honestly, there are many different ways to save your data while your app is running or prior to it going into a suspended state. I'm going to demonstrate three easy ways to enable this.

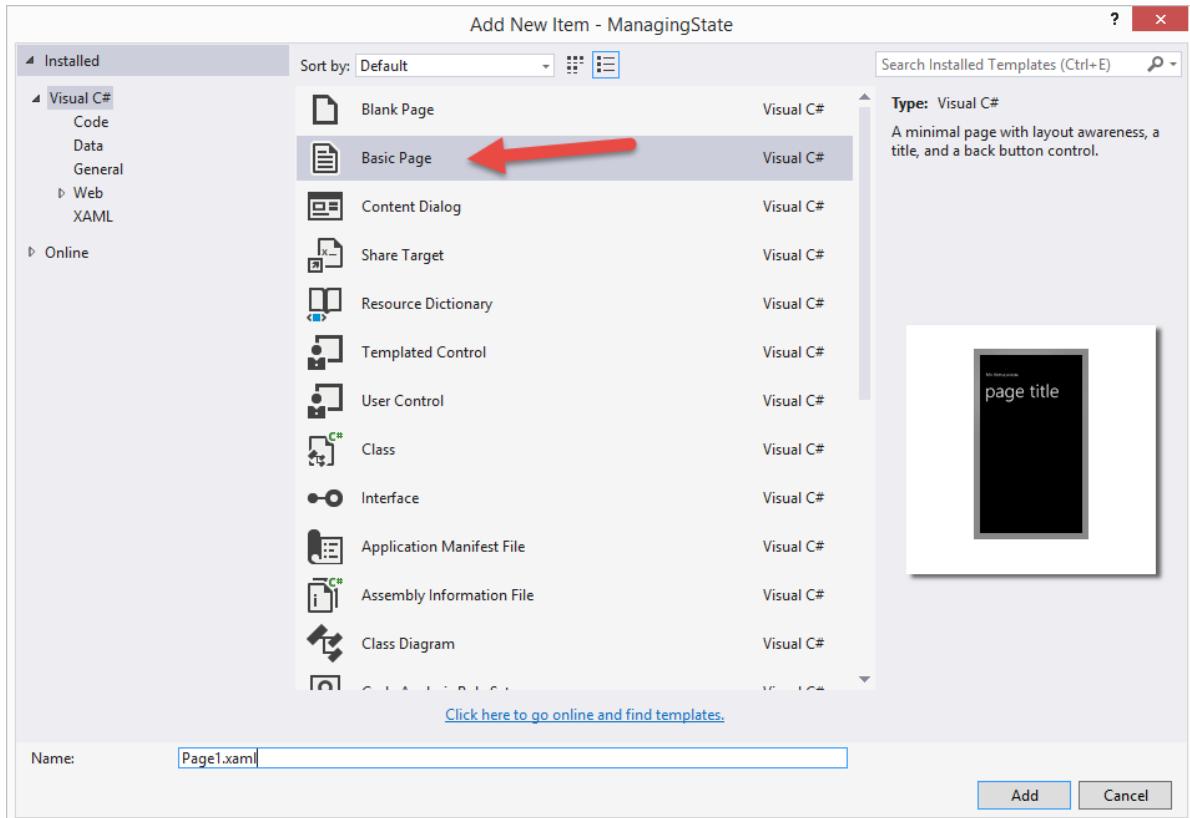
The first technique is to employ the `SuspensionManager`. The second technique is to use `Windows.Storage.ApplicationData.Current.LocalSettings`. The third is to use the `NavigationHelper`. Now, fair warning, the `SuspensionManager` and the `NavigationHelper` are not part of the Phone API. Instead, they are helper classes that are added to your project via a project or page template. The Blank App template does NOT include these by default, however I'll show you how we'll add them into your project easily.

Let's start with using no state saving techniques just to show you what the problem is. Then, we can use these three techniques to solve the problem.

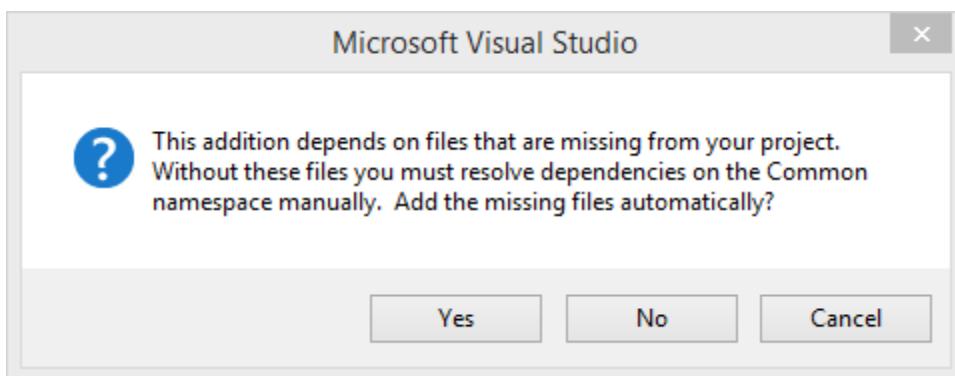
We'll create a new Blank App project called "ManagingState".

I'll delete the `MainPage.xaml`. At this point, the project will not compile because the `App.xaml.cs` needs a page to load at start up. We'll fix that later.

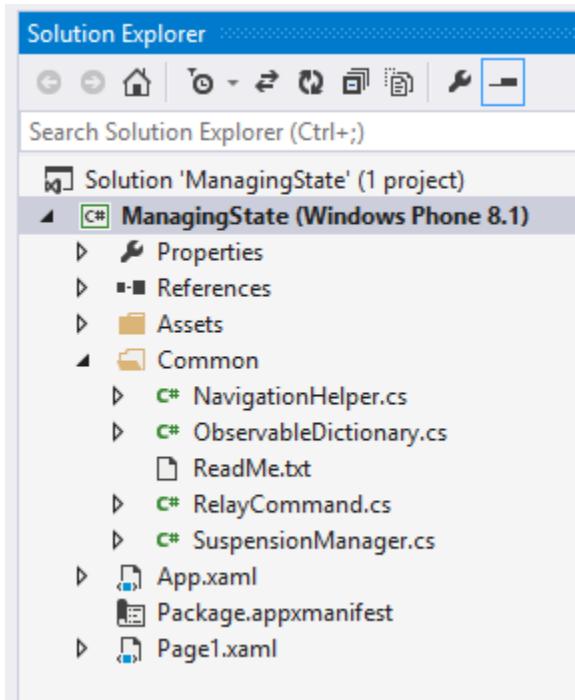
Add a Basic Page called `Page1.xaml`:



When you click the Add button, you'll see the following message box:



Click "Yes". This will add both the Page1.xaml and a folder called \Common with a number of new files in it. We will not use these files just yet, but soon. Your Solution Explorer should look like this:



Add two more Basic Pages, called Page2.xaml and Page3.xaml, respectively.

In all three pages, add the following code inside of the <Grid> named “ContentRoot”:

```
<StackPanel>
    <Button Content="Go To Previous Page"
        Click="previousButton_Click" />

    <Button Content="Go To Next Page"
        Click="nextButton_Click" />

    <TextBox Name="valueTextBox" />

</StackPanel>
```

Also, change the TextBlock that displays the page title like so, using the appropriate name:

```
<TextBlock Text="page 1" Margin="0,12,0,0" Style="{ThemeResource
HeaderTextBlockStyle}"/>
```

When you’re finished, Page1.xaml should look like this in the designer:



... and the other two pages should look similar.

Add the following code to Page1.xaml.cs, at the very bottom of the class definition (right after a code region named “NavigationHelper registration”):

```
private void previousButton_Click(object sender, RoutedEventArgs e)
{
    if (Frame.CanGoBack)
        Frame.GoBack();
}

private void nextButton_Click(object sender, RoutedEventArgs e)
{
    if (Frame.CanGoForward)
        Frame.GoForward();
    else
        Frame.Navigate(typeof(Page2));
}
```

In a similar fashion, add the following code to Page2.xaml.cs:

```
private void previousButton_Click(object sender, RoutedEventArgs e)
{
    Frame.GoBack();
}

private void nextButton_Click(object sender, RoutedEventArgs e)
{
```

```

if (Frame.CanGoForward)
    Frame.GoForward();
else
    Frame.Navigate(typeof(Page3));
}

```

And finally, in a similar fashion, add the following code to Page3.xaml.cs:

```

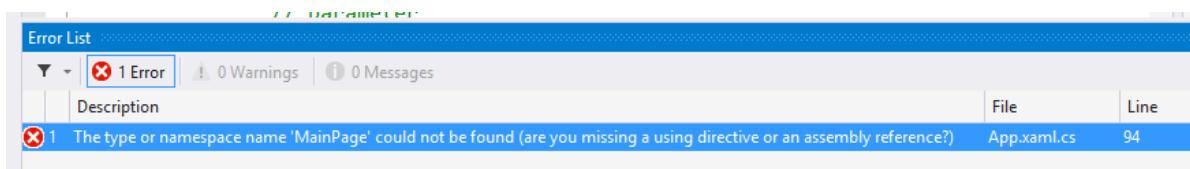
private void previousButton_Click(object sender, RoutedEventArgs e)
{
    Frame.GoBack();
}

private void nextButton_Click(object sender, RoutedEventArgs e)
{
    if (Frame.CanGoForward)
        Frame.GoForward();
    else
        Frame.Navigate(typeof(Page1));
}

```

Unfortunately, all of this prep work was necessary to create a viable scenario.

If you attempt to run the application at this point, you'll get an error:

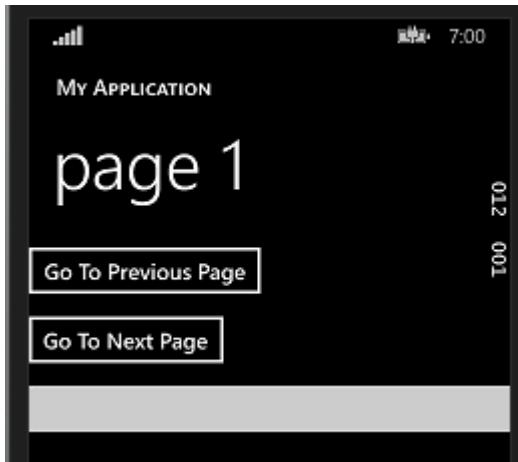


Double-click that entry in the Error List to go to the line of code that references MainPage which no longer exists in our app.

Change this line:

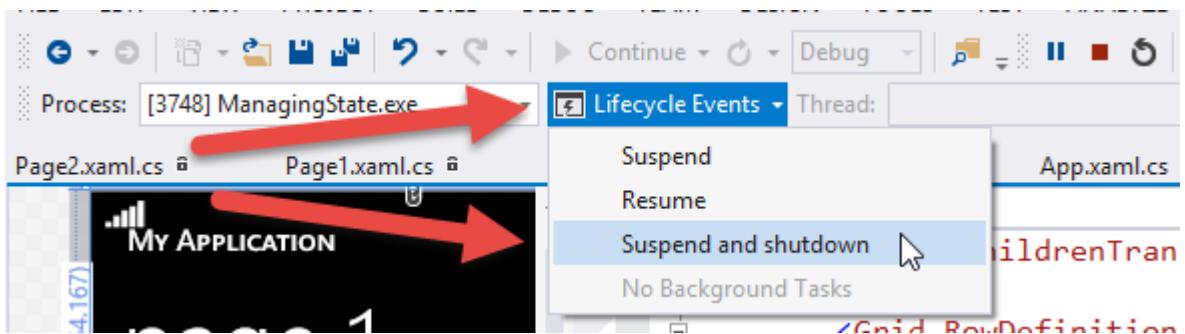
```
if (!rootFrame.Navigate(typeof(MainPage), e.Arguments))
```

... to ...



Click the “Go To Next Page” button until you reach Page 3 ... you can even use the “Go To Previous Page” a couple of times if you wish just to see the Frame.GoBack() and Frame.GoForward() methods at work.

Next, go back to Visual Studio and select the Lifecycle Events dropdown (which is only displayed while you are debugging your app) and select “Suspend and shutdown”.



The app will terminate, you'll leave Debug mode, and Visual Studio will return to the design time experience.

Now, restart the app in Debug mode. Notice that you will always start back on Page1.xaml. Attempt to “Go Back”, however the the Frame.BackStack from your previous session is lost because the app was terminated and with it, all session state.

We'll remedy this easily with the SuspensionManager class. I won't go into the details of the class, however by inserting a couple of lines of code in the App.xaml.cs file's OnNavigated and OnSuspending methods, we can at least get the Frame's BackStack back.

In the App.xaml.cs, in the OnLaunched() method, find these lines of code:

```
if (rootFrame == null)
{
    // Create a Frame to act as the navigation context and navigate to the first page
    rootFrame = new Frame();
```

Paste this line of code below them:

```
ManagingState.Common.SuspensionManager.RegisterFrame(rootFrame, "appFrame");
```

In a nutshell, the SuspensionManager.RegisterFrame tells the SuspensionManager which object (rootFrame, the only Frame in our app) that we'll be suspending.

Next, in the OnSuspending event, find this comment:

```
// TODO: Save application state and stop any background activity
```

... and add this line of code:

```
await ManageAppLifecycle.Common.SuspensionManager.SaveAsync();
```

This will force you to add the `async` keyword into the `OnSuspending` method declaration like so:

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
```

I devote a lesson to explaining the `await` and `async` keywords. For now, just add the `async` keyword and continue on.

At this point, we've registered the Frame with the SuspensionManager class and have called the `SaveAsync()` method which will save the current state of the Frame each time the app is suspended.

Finally, we'll want to Load the saved Frame when the app has terminated. Remember: if the app doesn't terminate, we won't need to restore the state, but we do not get a notification when the app is about to be terminated. Therefore, we can't take any chances and have to save state each time the app is suspended.

Back in the `OnLaunched()` method, locate these lines of code:

```
if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
    // TODO: Load state from previously suspended application
```

... and paste the following line of code beneath the TODO comment:

```
await ManagingState.Common.SuspensionManager.RestoreAsync();
```

Once again, the presence of the await keyword will force us to modify the OnLoaded method signature by adding the async keyword like so:

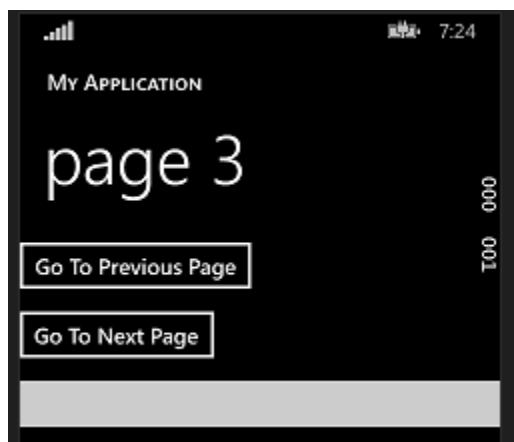
```
protected async override void OnLaunched(LaunchActivatedEventArgs e)
```

Now, shut down the emulator. Why? We've already terminated the app once. As a result, the code we just added (SuspensionManager.RestoreAsync()) will attempt to run, but there will be no state saved and we'll encounter an unhandled exception. This only happens because I asked you to run this app once prior to adding the SaveAsync() code. You'll never have to do this again.

Run the app, navigate from page 1 to page 2 to page3 using the "Go To Next Page" button.

Next, use the Lifecycle Events dropdown to "Suspend and Shutdown" the app.

Now, immediately run the app again. Notice that you'll start with Page 3 since that was the last page prior to termination:



Furthermore, click the “Go To Previous Page” button a couple of times to return to page 2 and page 1. The moral of the story is that we’ve now saved both the last page the user was working on AND the Frame.BackStack!

What about application data? How can that be retained between termination sessions? In other words, if the user is typing in entries to the text boxes, how can those be entries be saved should the app terminate before they are permanently saved or used?

There are two techniques to save application data, as I alluded to earlier.

The first is to utilize Windows.Storage.ApplicationData.Current.LocalSettings.

LocalSettings is a simple dictionary that is saved to the app’s assigned storage area. There’s also a similar storage area called Windows.Storage.ApplicationData.Current.RoamingSettings which will save data into a folder that gets synchronized across all of your devices. For now, we’ll prefer LocalSettings.

In Page1.xaml, modify the valueTextBox adding a Textchanged event handler:

```
<TextBox Name="valueTextBox"
        TextChanged="valueTextBox_TextChanged" />
```

Right-click the new event handler name and select Go to Definition or the F12 function key. In the new valueTextBox_TextChanged event handler method, add the following code:

```
private void valueTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    Windows.Storage.ApplicationDataContainer localSettings =
        Windows.Storage.ApplicationData.Current.LocalSettings;
    localSettings.Values["value"] = valueTextBox.Text;
}
```

This code will save the current Text property of the valueTextBox control each time the text changes. This may seem excessive, however this is one sure way to guarantee the application data state is saved in case the phone’s battery dies or any other termination scenario.

How to load the value back? Look for the NavigationHelper_LoadState() event handler defined earlier on the Page1.xaml.cs and add the following code:

```
private void NavigationHelper_LoadState(object sender, LoadStateEventArgs e)
```

```

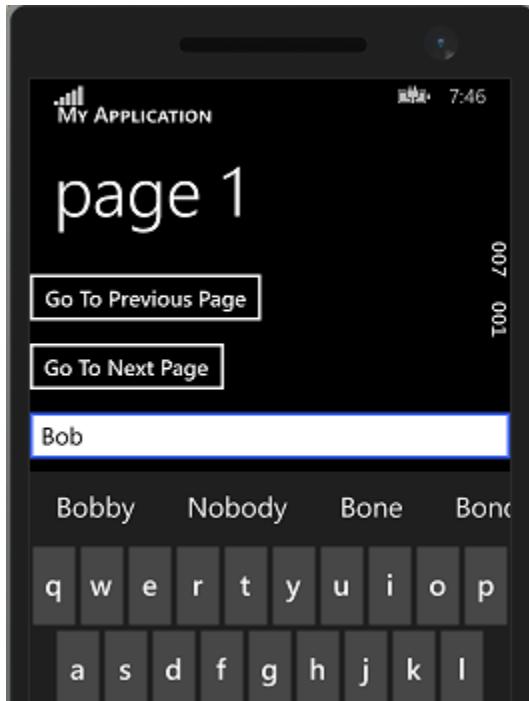
{
    Windows.Storage.ApplicationDataContainer localSettings =
        Windows.Storage.ApplicationData.Current.LocalSettings;

    if (localSettings.Values.ContainsKey("value"))
    {
        valueTextBox.Text = localSettings.Values["value"].ToString();
    }
}

```

First, this ensures that the key “value” exists, and if it does, then load its value into the valueTextBox.Text property.

To test this, run the app, type in text like so:



... then back in Visual Studio, select the Lifecycle Events “Suspend and shutdown”. Immediately re-run the app. You should be returned to page 1 and “Bob” should re-appear in the text box.

The second technique for saving application data state is something built into the NavigationHelper class. You’ll learn more about this helper class later in this series when we begin working with the Hub App template. For now, just know that it is part of the Basic Page template, pre-wired and ready to go. If you create a new Blank page, you’ll have to do a lot of work to integrate it, but it’s not impossible ... just easier to use the Basic Page template.

In a nutshell, the NavigationHelper was really meant for this scenario ... to help you maintain application state. We've already used the NavigationHelper_LoadState method in the previous example. It fires each time the OnNavigatedTo() event for the page fires. How that all works, I'll leave to a future discussion.

We'll focus on the NavigationHelper_SaveState() method that's defined on each page. Let's look at Page2.xaml.cs:

```
private void NavigationHelper_SaveState(object sender, SaveStateEventArgs e)
{
}
```

The SaveStateEventArgs has a property called PageState, which is a dictionary. So, we can add this line of code:

```
e.PageState["value"] = valueTextBox.Text;
```

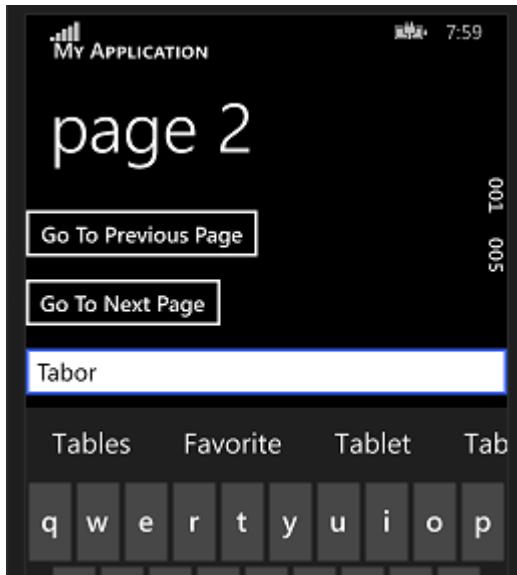
Now, whenever the app is suspended, the NavigationHelper_SaveState() method is called on each page the Frame knows about. It will add a key called "value" and set it to the string typed into the valueTextBox.

To restore the state of this data, we'll work again with the NavigationHelper_LoadState:

```
private void NavigationHelper_LoadState(object sender, LoadStateEventArgs e)
{
    if (e.PageState != null && e.PageState.ContainsKey("value"))
    {
        valueTextBox.Text = e.PageState["value"].ToString();
    }
}
```

Again, we do a check to make sure the LoadStateEventArgs has something in it called "value" and if it does, then set the valueTextBox.Text property to that value.

Run the app, navigate to page 2, type in something like so:



Then back in Visual Studio, select Lifecycle Events “Suspend and Shutdown”. Re-run the app, navigate back to Page 2 and the text box should be populated with your original entry.

Recap

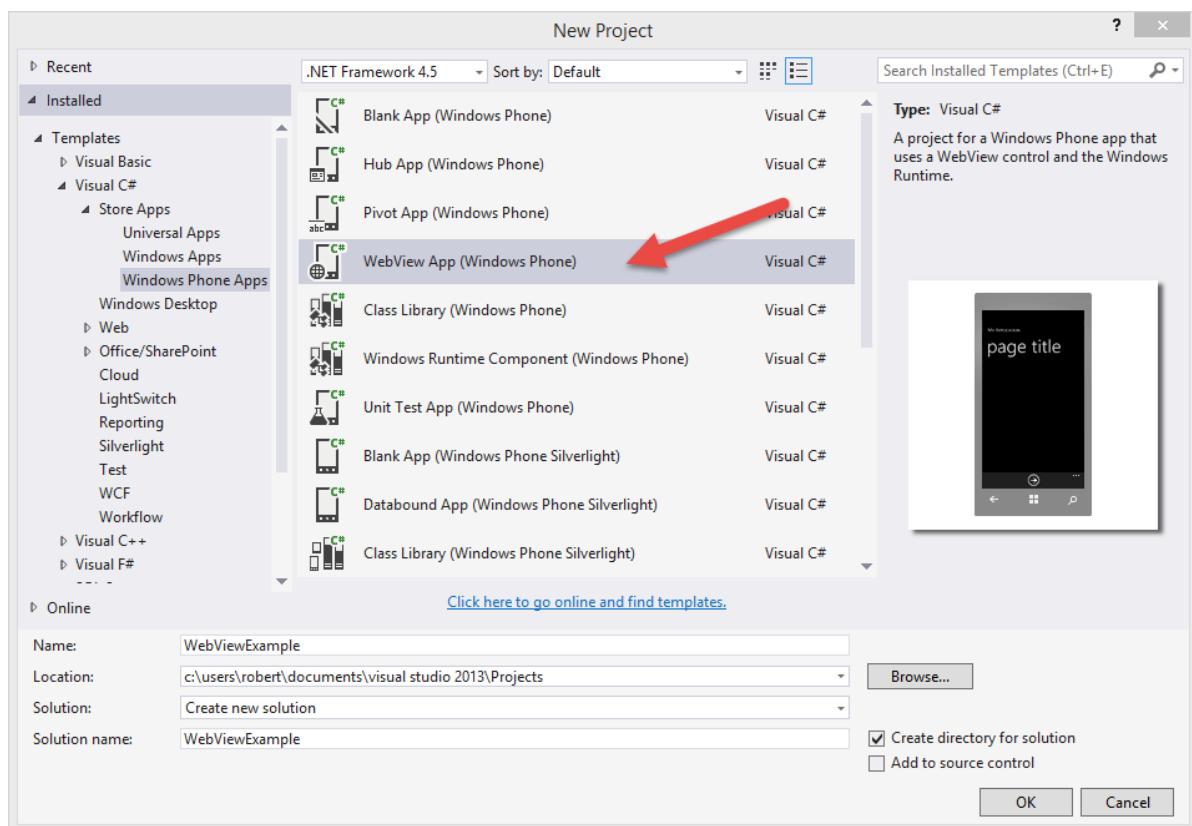
The key is to understand that your app has a life cycle, and you have the tools to prepare for the move from suspension to termination ... you can save the application state such as the Frame's BackStack and current page, as well as the application's data.

Lesson 13: Working with the Web View App Template

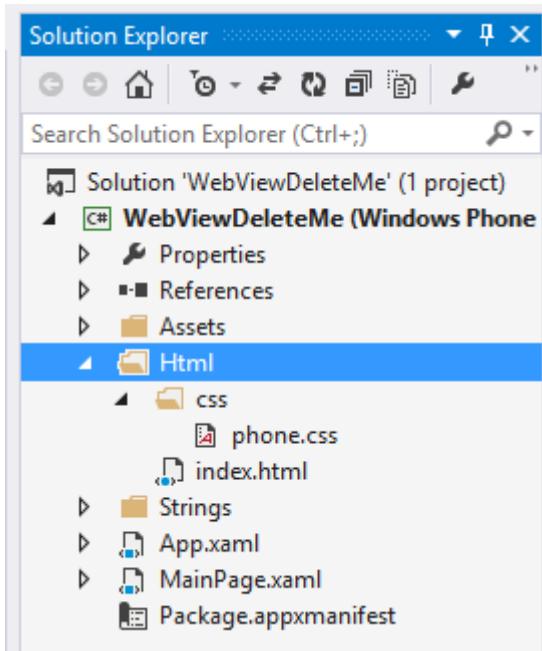
In my personal opinion, the WebView App template featuring the Web View control, is probably the easiest way to create an app IF you already have basic HTML, CSS and JavaScript knowledge. In this lesson, we'll look at the basics, in the next lesson we'll build an app based on something from the JavaScript and JQuery Fundamentals for Absolute Beginners series just to demonstrate what's possible and to demonstrate the types of issues you might face when converting an existing JavaScript-based app to the Web View App Template.

Consequently, these are the only two lessons in this series that do not primarily use XAML and the Phone API for layout, navigation, etc. Feel free to skip these lessons if you do not intend to utilize the web-stack of technologies for building apps.

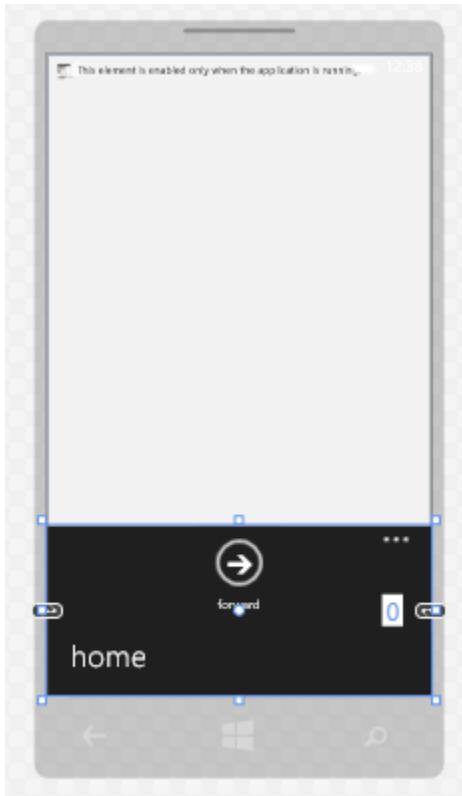
(1) I'll begin by creating a new WebView App project called "WebViewExample".



You'll immediately notice the presence of the \Html subfolder along with an index.html file and a \css subfolder with a phone.css file.



If you take a look at the `MainPage.xaml`, it hosts a `WebViewControl` and a `CommandBar` with a `Forward` and `Home` app bar button.



In the `MainPage.xaml.cs`, there's a `HomeUri` referencing the `\Html\index.html`.

```
private static readonly Uri HomeUri = new Uri("ms-appx-web:///Html/index.html",
UriKind.Absolute);
```

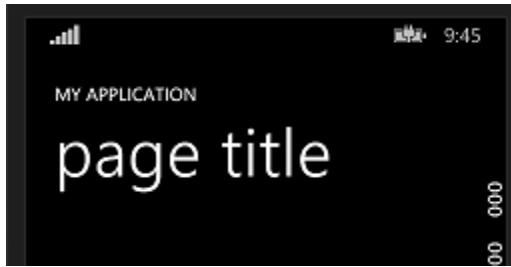
In the `OnNavigatedTo()` method, the `WebViewControl` is told to navigate to the `HomeUri` as well as some event handlers for the physical back button and the forward `AppBarButton` in the `CommandBar`. In fact, most of the code on this page is attempting to manipulate the “page stack” — page that have been visited, going back, going forward, etc. We’ll see those at work in a moment.

In fact, you may notice that the `WebViewControl` has a very similar set of `Navigate` methods as the `Frame`. We can leverage what we already know about the `Frame` to take control of the navigation experience if we want to, but most of that has been taken care of for us already by the `WebView` project template.

(2) The templated `index.html` has the following definition:

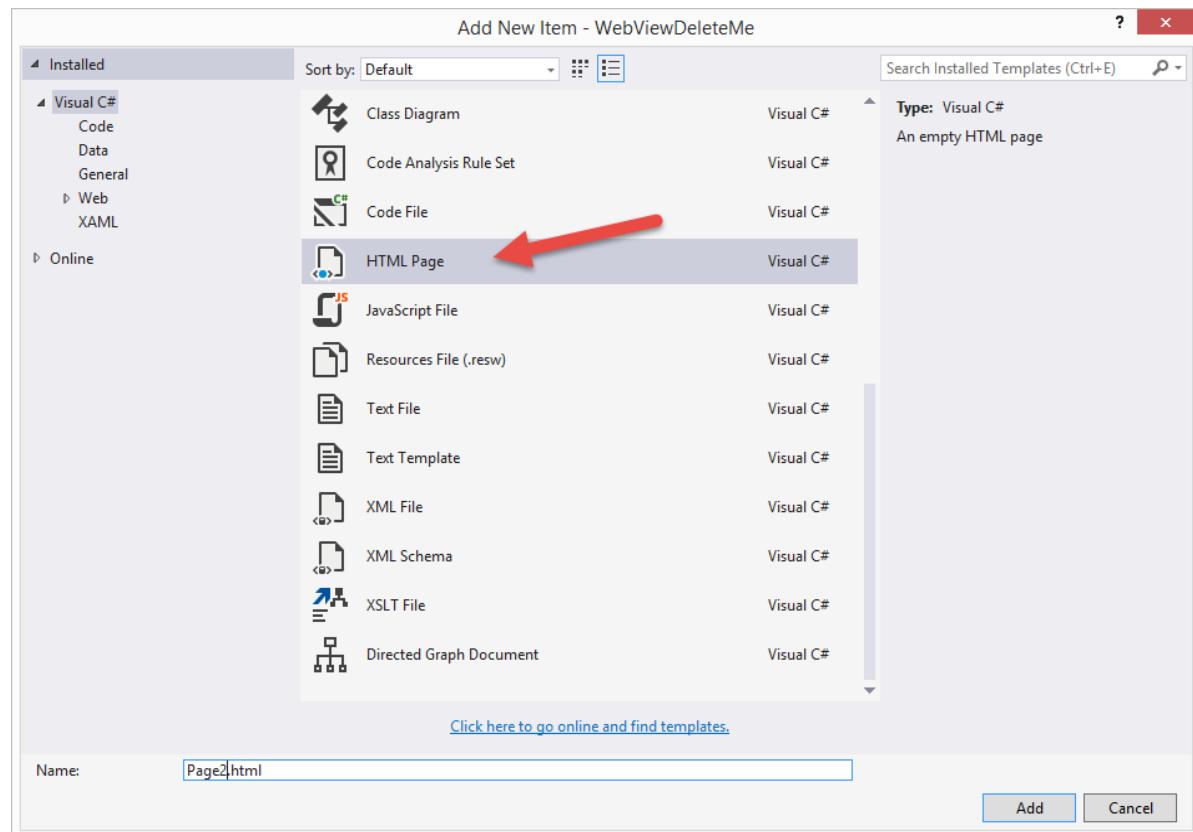
```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" href="css/phone.css" />
    <title>Windows Phone</title>
</head>
<body>
    <div>
        MY APPLICATION
    </div>
    <div id="page-title">
        page title
    </div>
</body>
</html>
```

Two items to note: `<link>` references the `\css\phone.css` file, and there are two `<div>` elements to create the now familiar app name and page title. If we were to run the app right now, we would see the following:



Take a look at the phone.css file ... match up the styles defined in the CSS with what you see on screen. Notice the first two @media definitions are to help with orientation, while the remaining styles affect the layout of the page. You are free to add to this file, but I would recommend that you add additional CSS files and link to them so that, if Microsoft ever updates this file, your changes will not have to be merged.

(3) Add a second html page called “Page2.html”.



In my version of the page, it does not have the link to the phone.css nor does it have the the <div> tags. Instead, you get this:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>

</body>
</html>
```

So, I'll copy the HTML from index.html into the new Page2.html to make sure they're consistent. I'll then modify Page2.html so that it has the proper app name and page title:

```
<body>
<div>
    WEBVIEWCONTROL EXAMPLE
</div>
<div id="page-title">
    page two
</div>

</body>
```

And I'll modify index.html in a similar way:

```
<div>
    WEBVIEWCONTROL EXAMPLE
</div>
<div id="page-title">
    page one
</div>
```

(4) Add these lines to index.html:

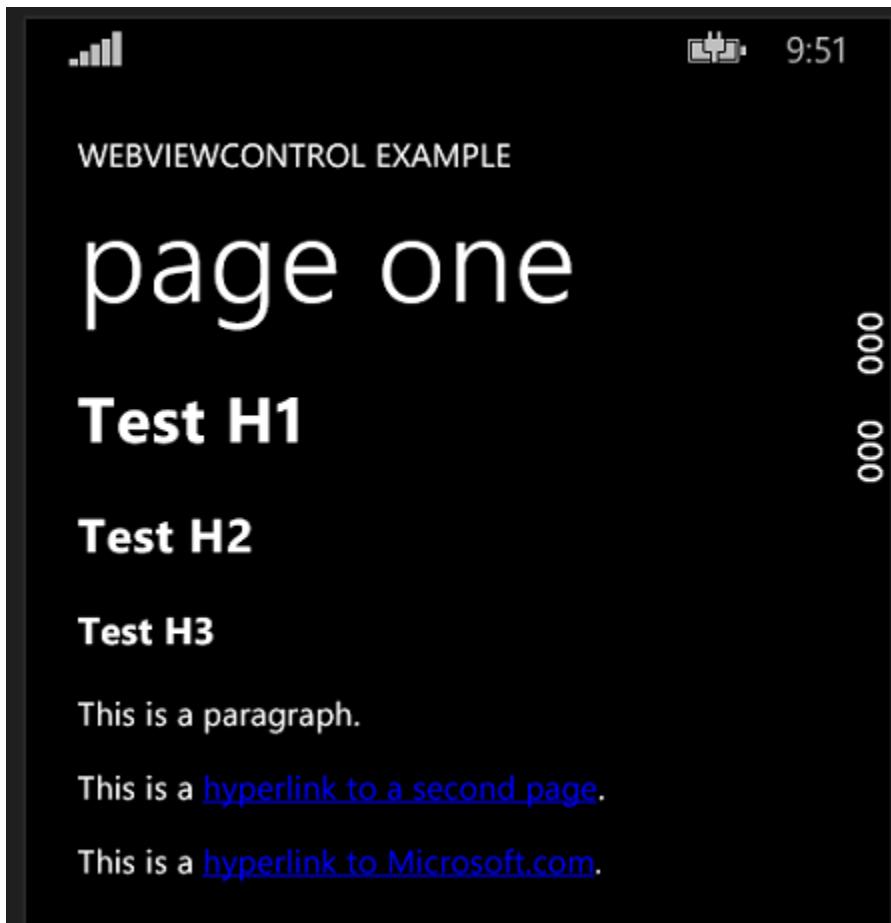
```
<h1>Test H1</h1>
```

```
<h2>Test H2</h2>
<h3>Test H3</h3>
<p>This is a paragraph.</p>

<p>This is a <a href="Page2.html">hyperlink to a second page</a>.</p>
<p>This is a <a href="http://www.microsoft.com">hyperlink to Microsoft.com</a>.</p>
```

(5) Test your changes by Debugging the app.

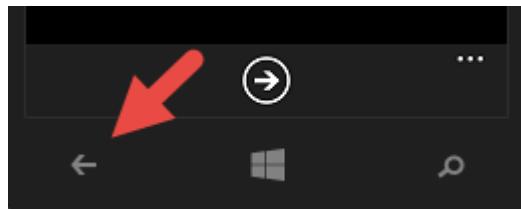
You should now see how common tags are styled by default:



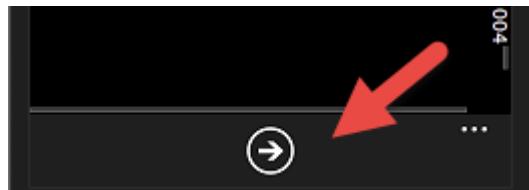
Additionally, if you click the “hyperlink to a second page”, Page2.html should appear:



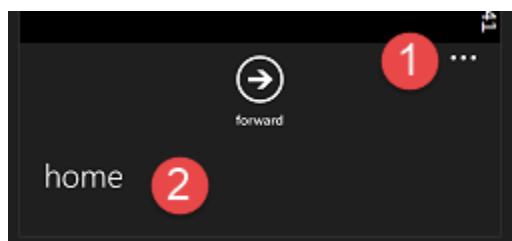
If you click the back arrow on the Phone's face, you should be returned to index.html:



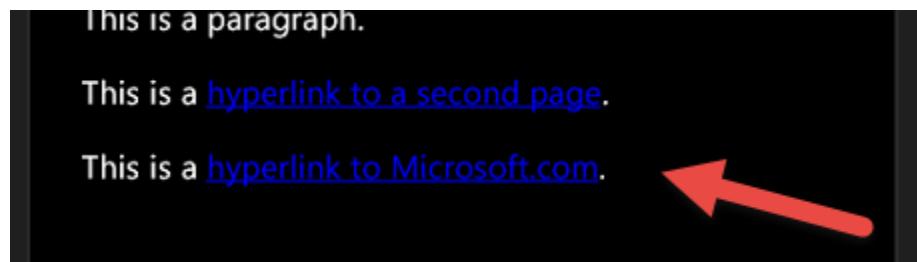
When back on index.html, you should be able to click the Forward arrow AppBarButton on the CommandBar to return to Page2.html:



While on Page2.html, you should be able to click (1) the ellipsis to expand the CommandBar, and (2) click the “home” AppBarButton (menu) to return to index.html:



And finally, when you're on index.html, you should be able to click the “hyperlink to Microsoft.com” ...



... and be able to navigate to Microsoft's home page:



Recap

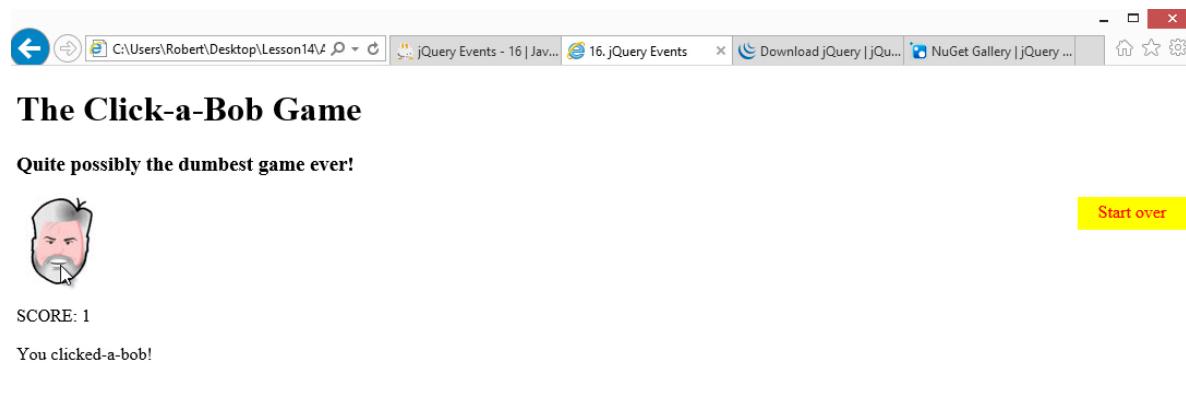
Those are the basics of using the `WebViewControl` and the `Web View Project` template. Note that you're not limited to just HTML and CSS, you can also add JavaScript and jQuery and I'll demonstrate how to do that in the next lesson with a more full-fledged example featuring an interactive mini-game.

Lesson 14: Exercise: Whack-a-Bob App

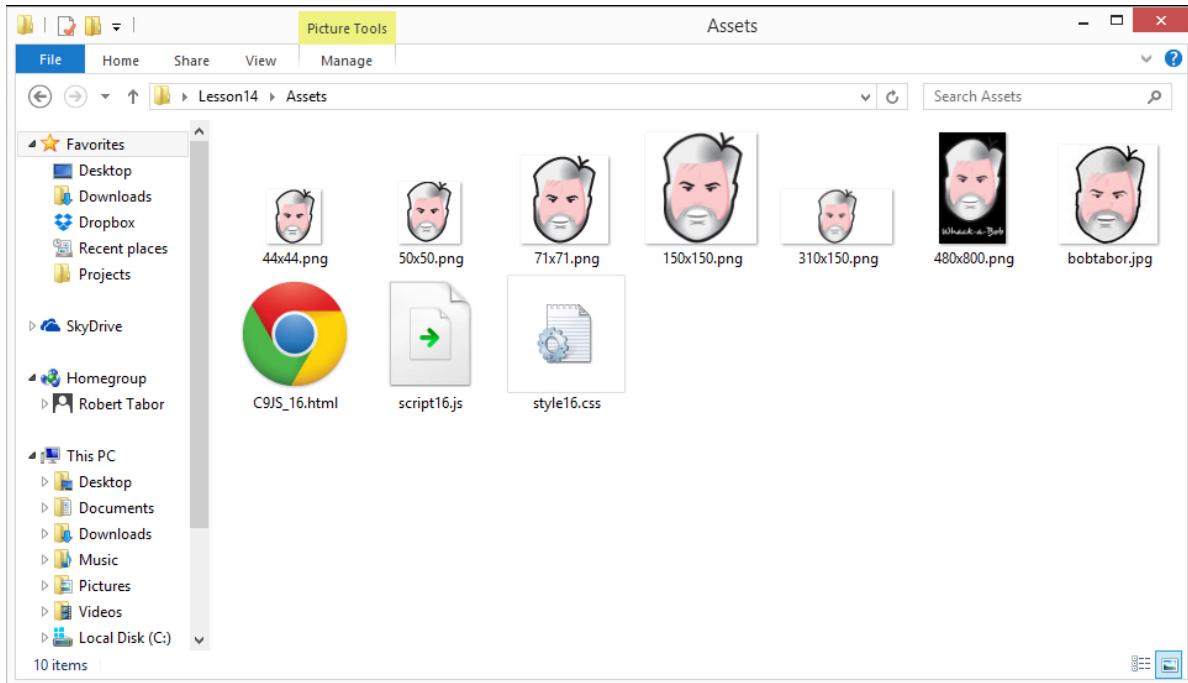
In this lesson we're going to take what we've learned previously and we're going to create an entire app, a very silly little game using the WebView App project template featuring the WebView control. We'll base our game on this app that I've built for the Javascript Fundamentals for Absolute Beginners series on Channel9. You can find it here:

<http://channel9.msdn.com/Series/Javascript-Fundamentals-Development-for-Absolute-Beginners/jQuery-Events-16>

In that lesson I explained jQuery Events and to demonstrate the concepts of that lesson I created the Click-a-Bob game. It features a little cartoon "bob head" and you keep clicking on the cartoon bob head and it gives you a score. You can reset the score by hovering over the Start Over button on the right.



Important: Make sure you download the assets that are available for this lesson in the Lesson14.zip file. Inside of that .zip file there should be an Assets folder containing the icons, the images, the source code, etc. required to follow along and complete this lesson.

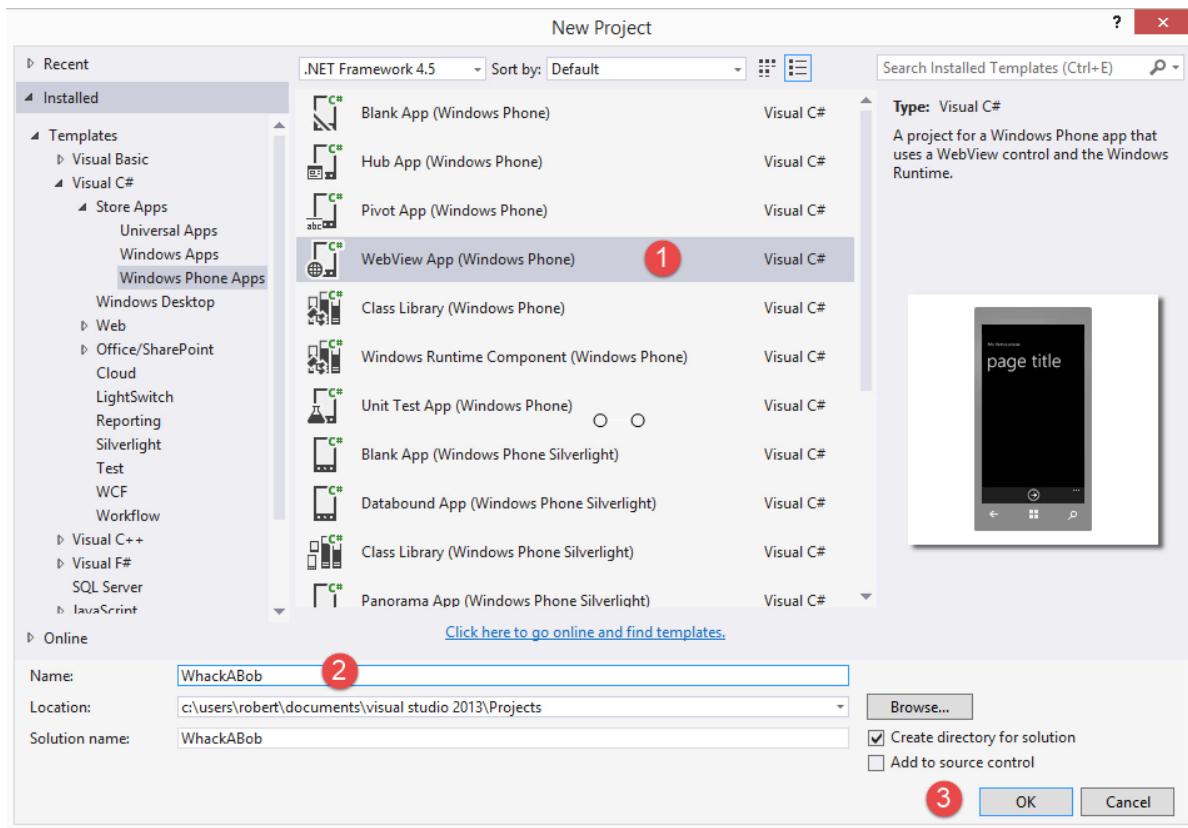


Note: This project was inspired by the Clow Clicker app. The guy who built it just wanted to parody some of those mindless social games and, to his surprise, it became an instant success. People just kept clicking the cow to get more click points.

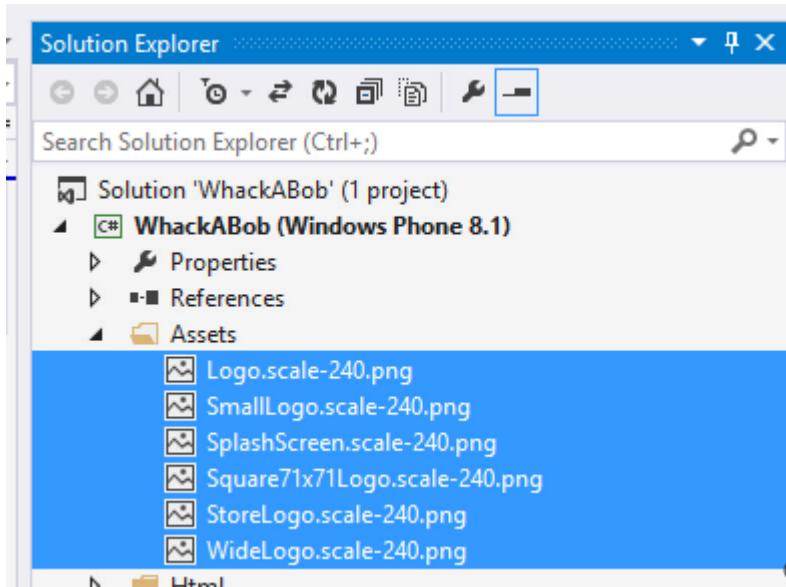
http://en.wikipedia.org/wiki/Cow_Clicker

Our plan is to take the Click-a-Bob and turn it into a Phone app called Whack-a-Bob.

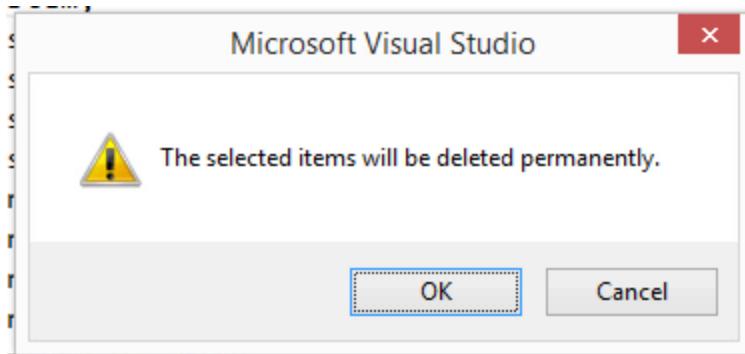
To begin, we'll create a new (1) WebView App project (2) named WhackABob and (3) clicking OK:



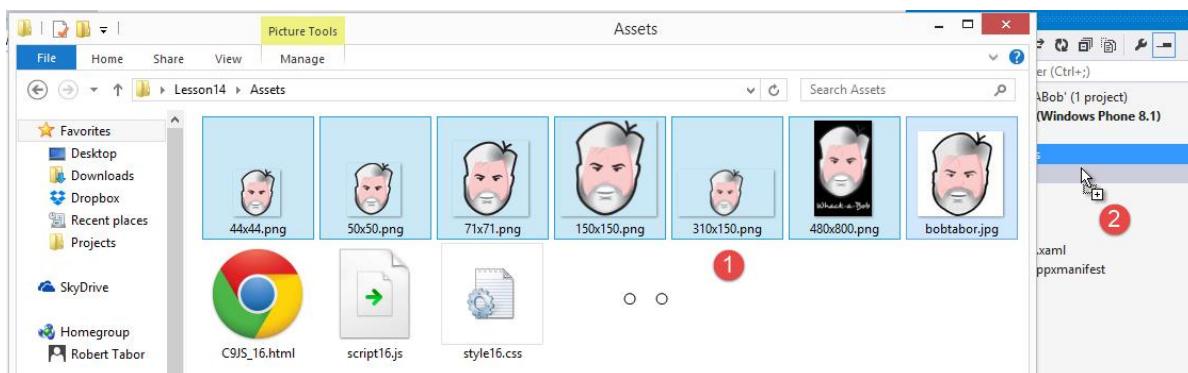
The first task is to copy over all of the assets and code from the Assets folder. To begin I'll select all of the current files in the project's Assets folder:



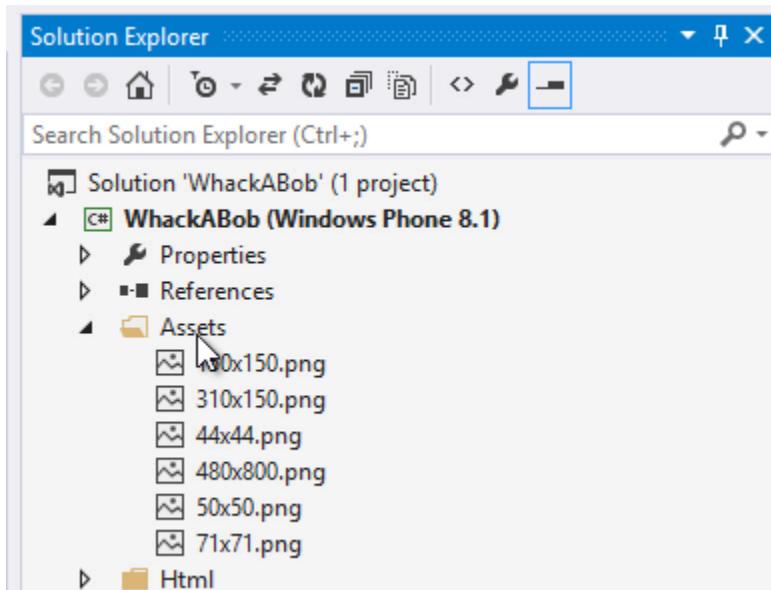
... and select the Delete key on my keyboard. It will ask me to confirm the deletion and I'll select the OK button.



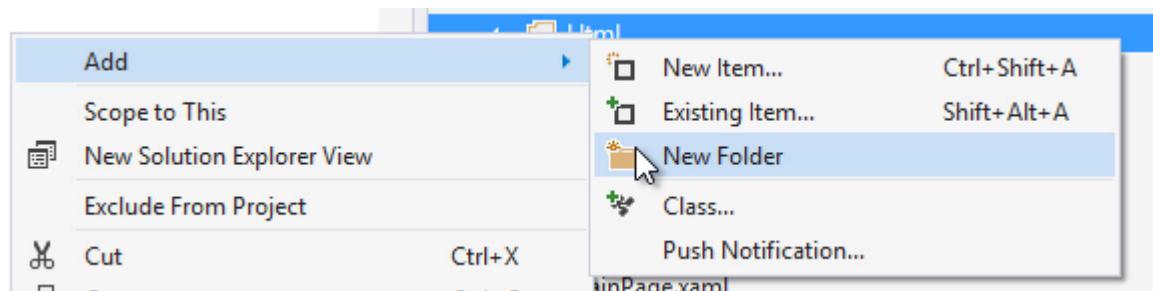
Next, I'll select all of the numerical bob head images for use as app tiles in Windows Explorer and drag and drop these into the Assets folder in Visual Studio's Solution Explorer:



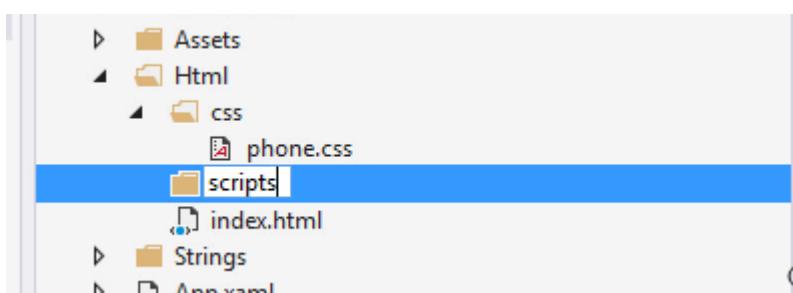
Your Assets folder should now look like this:



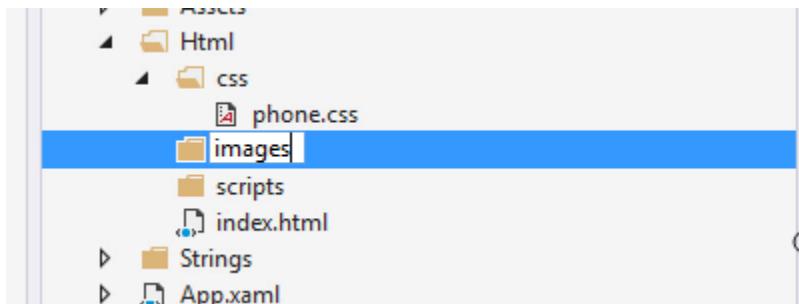
Next, I'll create a sub-folder called scripts and images. I'll right-click on the Html folder and select New Folder:



And name the first folder: scripts



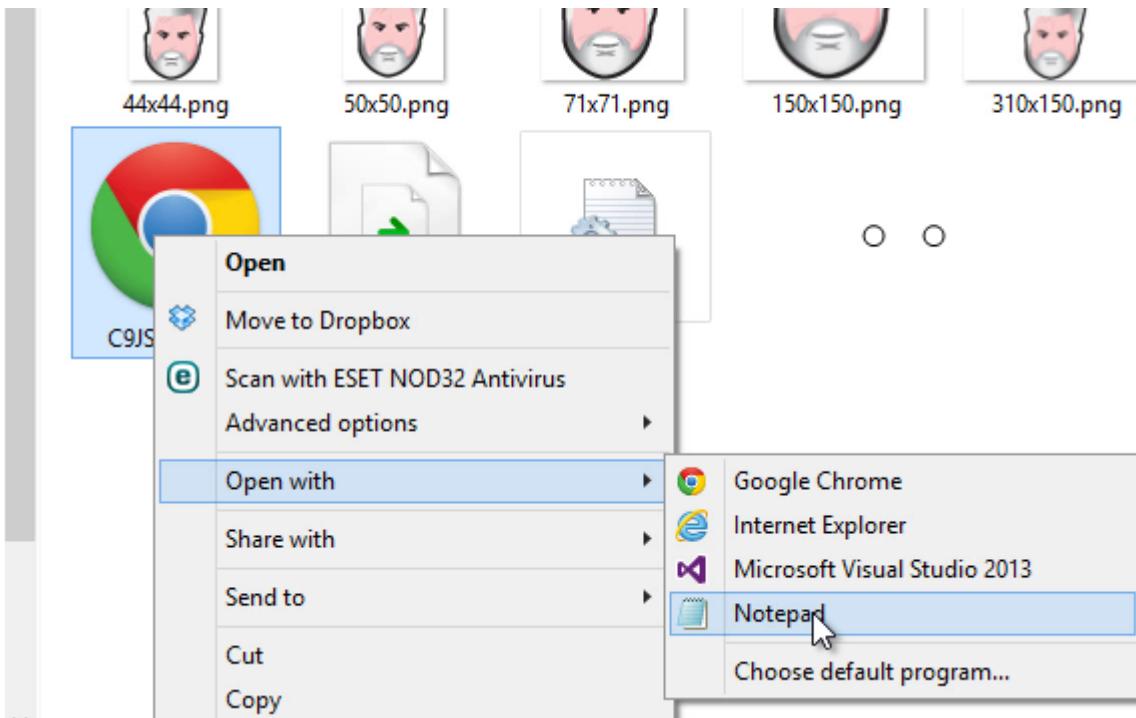
I'll repeat, this time naming the sub-folder: images



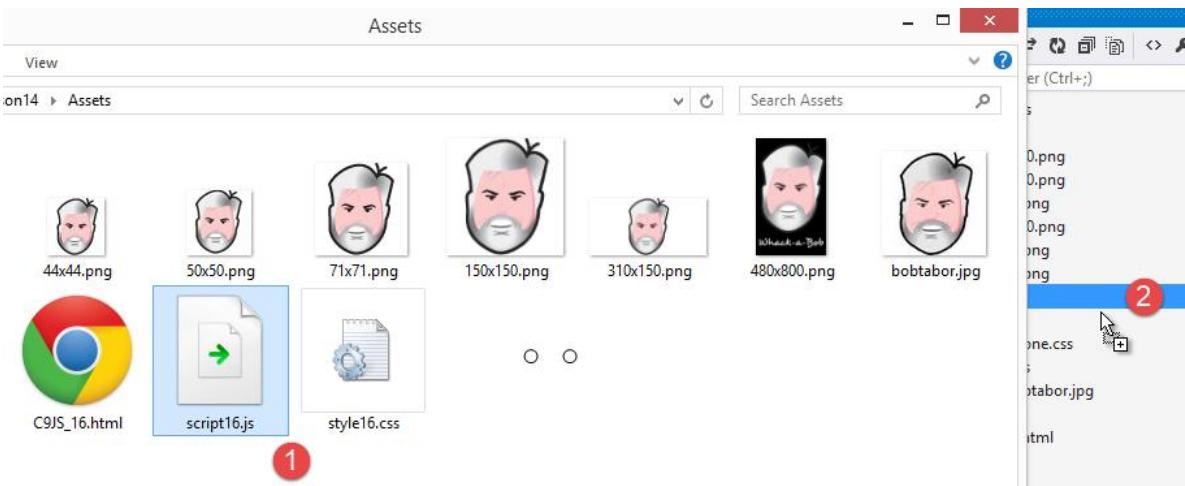
Next, I'll (1) drag the bobtabor.jpg image from Windows Explorer into (2) the new images sub-folder in Visual Studio:



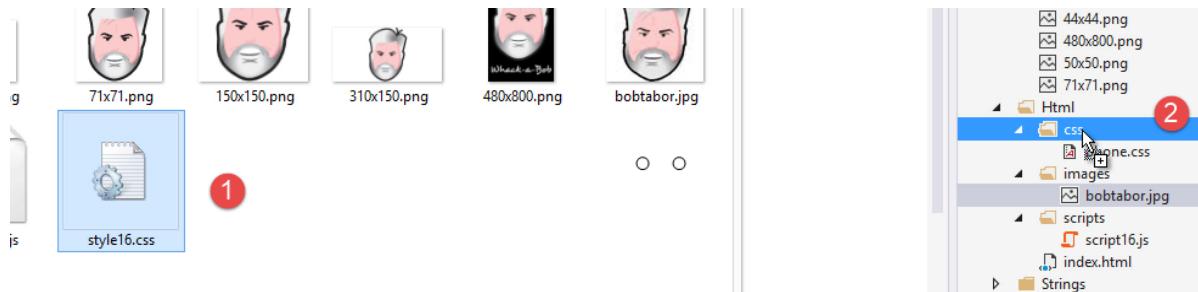
In preparation for copying the HTML from the original web page into the new index.html page in Visual Studio, I'll right-click the C9JS_16.html file in Windows Explorer and select Open with > Notepad.



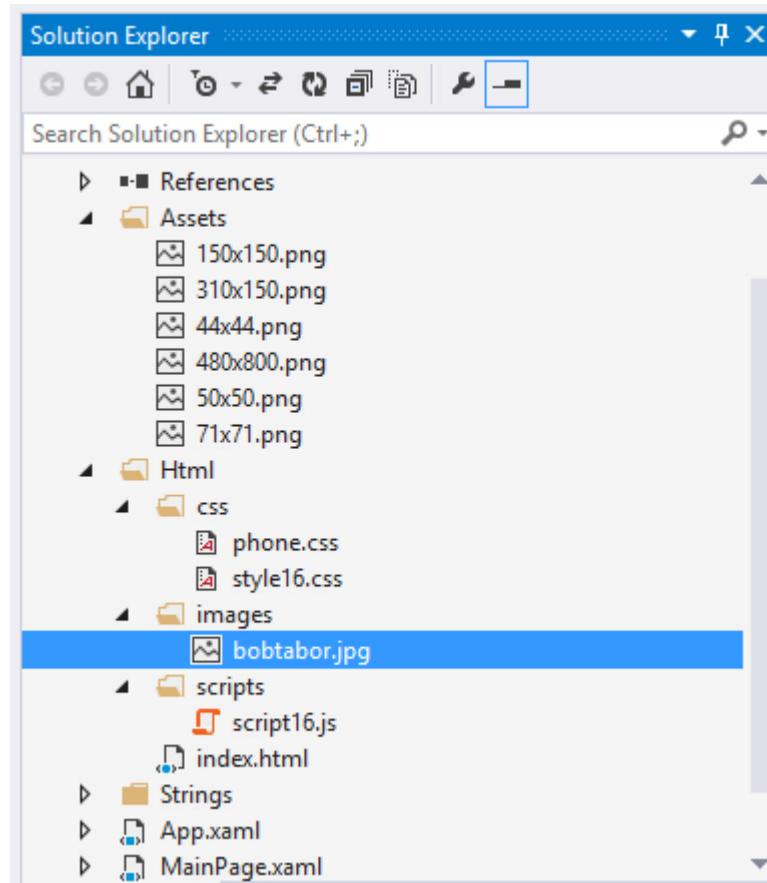
Next, I'll (1) drag and drop the script16.js file from Windows Explorer into (2) the new scripts sub-folder in Visual Studio's Solution Explorer:



Finally, I'll (1) drag and drop the style16.css file from Windows Explorer into (2) the css sub-folder in Visual Studio's Solution Explorer:



When finished, your Solution Explorer should have the same file structure as mine:



Next, we'll begin to edit the index.html page by copying HTML from the original C9JS_16.html page (which we opened in Notepad). I'll highlight and copy the references to jQuery to our Script16.js and to our style16.css files ...

C9JS_16.html - Notepad

```
<!DOCTYPE html>
<html>
<head>
    <title>16. jQuery Events</title>
    <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.7.1.js"></script>
    <script type="text/javascript" src="script16.js"></script>
    <link rel="stylesheet" type="text/css" href="style16.css" />
</head>
<body>
    <h1>The Click-a-Bob Game</h1>
    <h3>Quite possibly the dumbest game ever!</h3>
```

... and paste into the head section of the index.html document:

index.html* App.xaml.cs

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" href="css/phone.css" />
    <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.7.1.js"></script>
    <script type="text/javascript" src="script16.js"></script>
    <link rel="stylesheet" type="text/css" href="style16.css" />
```

(1) I want to make sure I reference the correct locations for these files so to make sure I'm referencing the scripts folder, and (2) I'll remove the reference to jquery in the CDN.

index.html

```
<link rel="stylesheet" type="text/css" href="css/phone.css" />
<script src=""></script> 2
<script type="text/javascript" src="scripts/script16.js"></script> 1
<link rel="stylesheet" type="text/css" href="css/style16.css" />

<title>Windows Phone</title>
```

When you're building an app on the phone you don't want to include any references to outside resources even though jQuery is best served up from a Content Delivery Network when building web page consumed by web browsers over the internet. So, we'll copy jQuery down into our local project and reference it from there.

If you go to:

[Http://jquery.com/download](http://jquery.com/download)

... you can download the compressed version, the production version of jQuery, the latest version that's available, 2.1.0.

jQuery 2.x

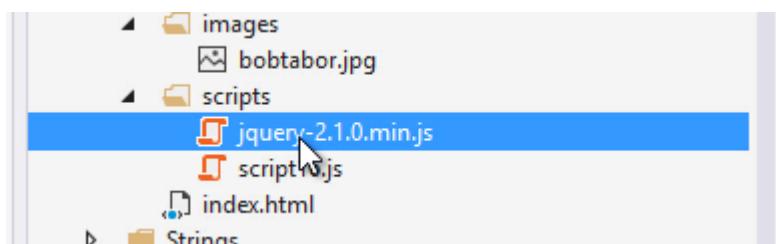
jQuery 2.x has the same API as jQuery 1.x, but *does not support Internet Explorer 6, 7, or 8*. All the notes in the [jQuery 1.9 Upgrade Guide](#) apply here as well. Since IE 6/7/8 are still relatively common, we recommend using the 1.x version unless you are certain no IE 6/7/8 users are visiting the site. Please read the [2.0 release notes](#) carefully.

Download the compressed, production jQuery 2.1.0  [Download the uncompressed, development jQuery 2.1.0](#)

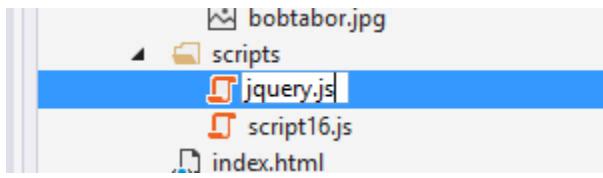
If you attempt to save this file, Internet Explorer gives you a little warning here. "Hey, this file can't be verified. Be careful about this." Click Save:



Once saved locally, I'll drag it into my scripts sub-folder ...



... and for ease of working with it I'm just going to rename this, just jquery.js:



So then in my HTML what I'll do is reference it as scripts/jQuery.js:

```
<script src="scripts/jquery.js"></script>  
<script type="text/javascript" src="scripts/script16.js"></script>  
<link rel="stylesheet" type="text/css" href="css/style16.css" />
```

Back to my original HTML, I'm going to select all the HTML in the body and copy it from Notepad ...

C9JS_16.html - Notepad

File Edit Format View Help

```
<!DOCTYPE html>
<html>
<head>
    <title>16. jQuery Events</title>
    <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-
1.7.1.js"></script>
    <script type="text/javascript" src="script16.js"></script>
    <link rel="stylesheet" type="text/css" href="style16.css" />
</head>
<body>
    <h1>The Click-a-Bob Game</h1>O_O
    <h3>Quite possibly the dumbest game ever!</h3>

    <div id="game">
        
        <p>SCORE: <span id="score">0</span></p>
        <p id="success">You clicked-a-bob!</p>
    </div>

    <div id="startover">Start over</div>

</body>
```

... and paste it beneath the title div in the index.html page in Visual Studio:



```
index.html*  X App.xaml.cs
<title>Windows Phone</title>
</head>
<body>
    <div>
        MY APPLICATION
    </div>
    <div id="page-title">
        page title
    </div>

    <h1>The Click-a-Bob Game</h1>
    <h3>Quite possibly the dumbest game ever!</h3>

    <div id="game">
        
        <p>SCORE: <span id="score">0</span></p>
        <p id="success">You clicked-a-bob!</p>
    </div>

    <div id="startover">Start over</div>
</body>
</html>
```

Furthermore, I'm going to edit the head and title div (beneath the body tag), I'll change all text from "clicked" to "whacked", and I'll modify the image location to the images/bobtabor.jpg.



The screenshot shows the Microsoft Visual Studio code editor with the file "index.html" open. The code is written in HTML and includes references to external CSS and JavaScript files. It features a title, a main message, a title for the game, and a game section containing a score and a success message. A "Start over" button is also present.

```
<link rel="stylesheet" type="text/css" href="css/phone.css" />

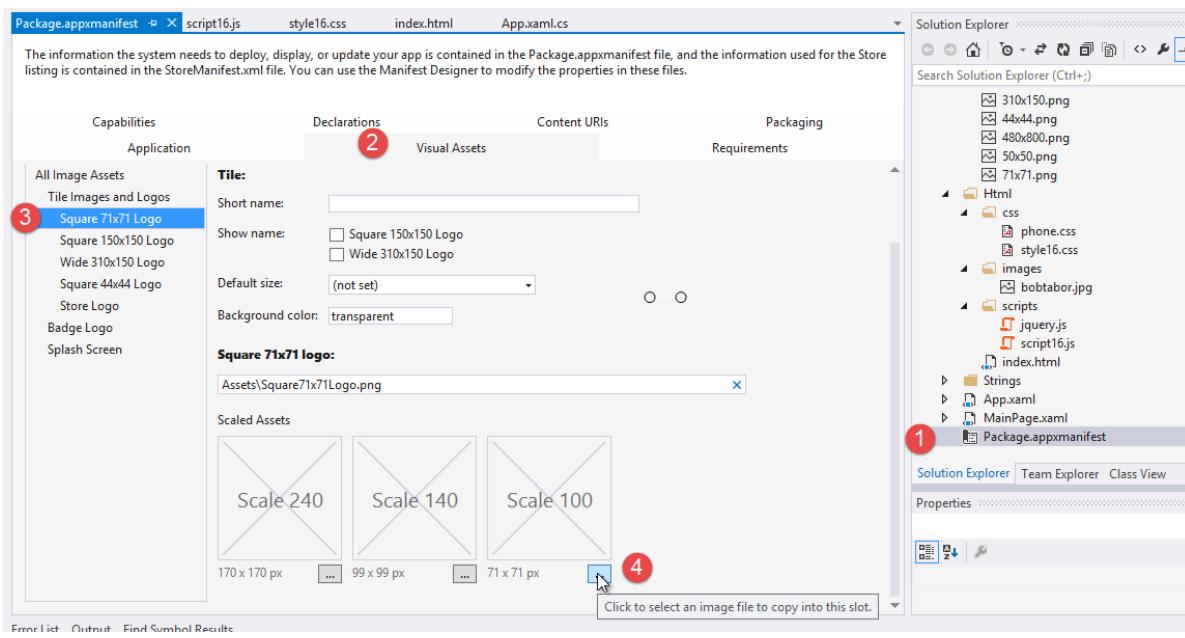
<script src="scripts/jquery.js"></script>
<script type="text/javascript" src="scripts/script16.js"></script>
<link rel="stylesheet" type="text/css" href="css/style16.css" />

    <title>Windows Phone</title>
</head>
<body>
    <div>
        quite possibly the dumbest game ever!
    </div>
    <div id="page-title">
        whack-a-bob
    </div>

    <div id="game">
        
        <p>SCORE: <span id="score">0</span></p>
        <p id="success">You whacked-a-bob!</p>
    </div>

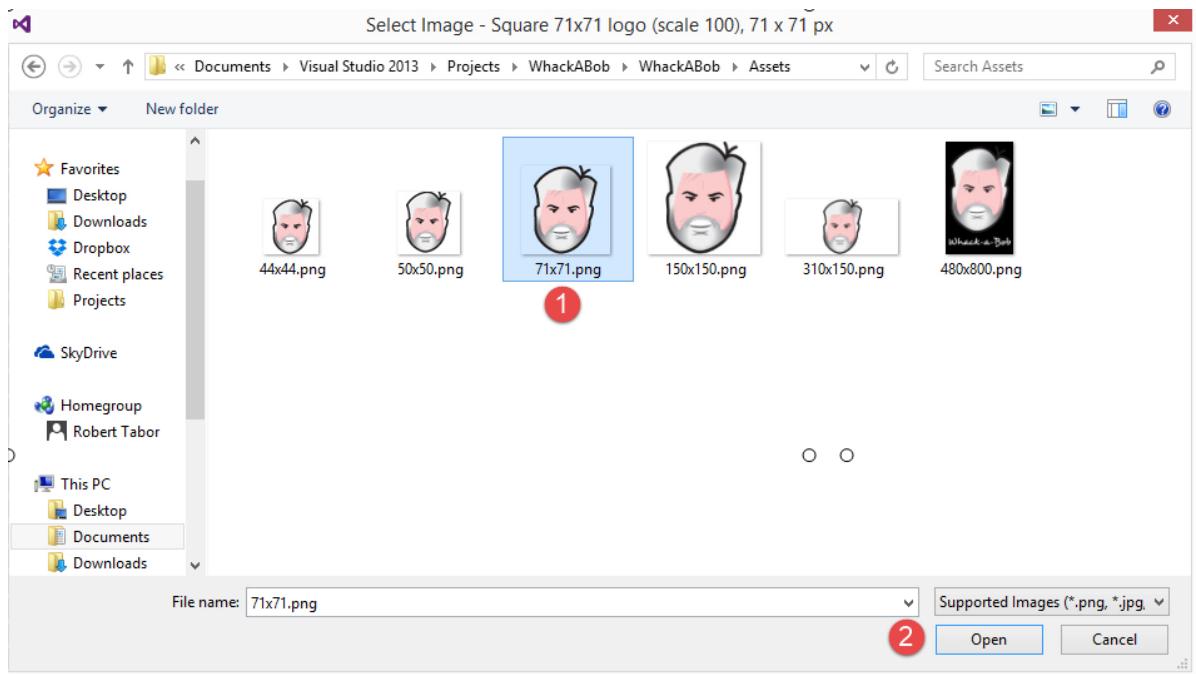
    <div id="startover">Start over</div>
```

Before we take our app for a test run, I'll modify the branding elements. You'll recall earlier we copied the sized "cartoon bob head" images into the Assets folder. To utilize these ...



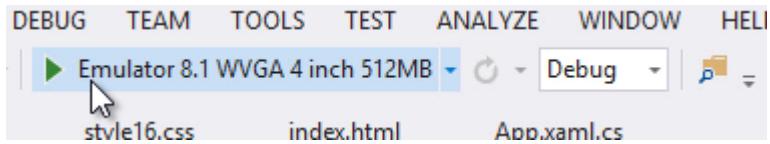
- (1) Open the package.appxmanifest
- (2) Select the Visual Assets tab
- (3) Select the Square 71x71 Logo page on the left
- (4) Under the Scale 100 image, click the ellipsis button ...

This will allow you to choose an image from your hard drive. Navigate to the Assets folder for the project and (1) select the 71x71.png file, then (2) select the Open button.

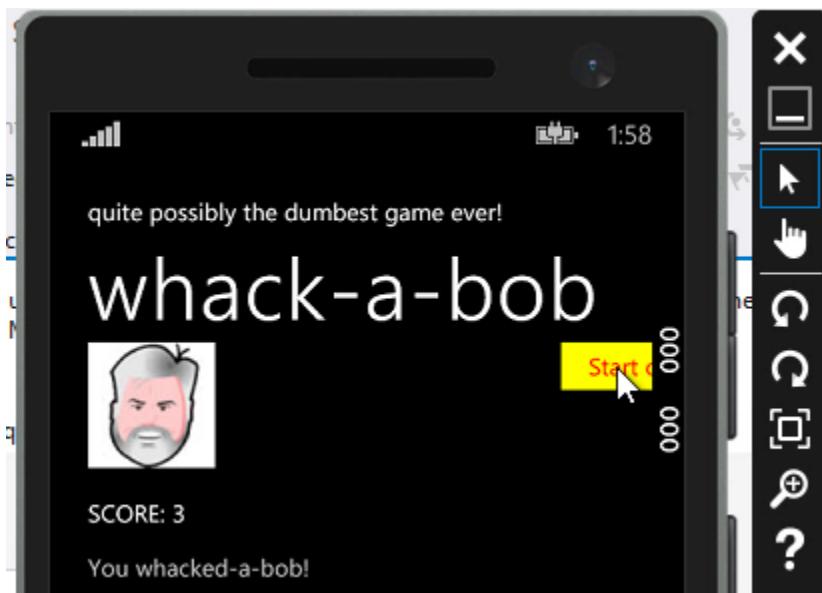


Repeat those basic steps for (1) the 150x150 Logo, (2) the Wide 310x150 Logo, (3) the Square 44x44 Logo, and (4) the Splash Screen. In each case, make sure to set the “Scale 100” image.

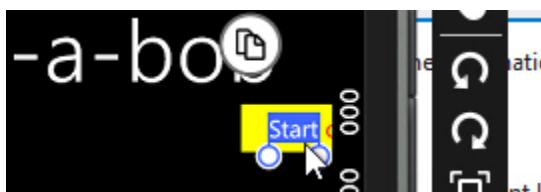
Now we’re ready to test what we have so far. Select to debug the app in the Emulator:



And while it's not perfect (yet) we can see that with minimal changes, we're able to re-create the majority of the app complete with jquery enabled interactive functionality.



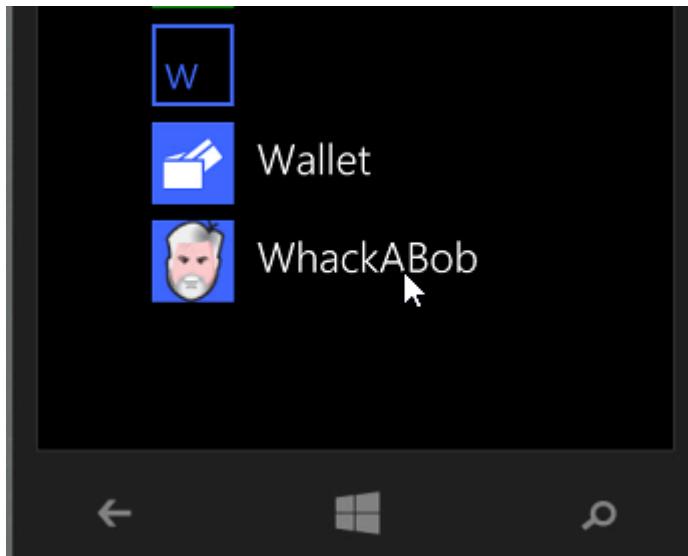
However, there are a few annoyances. For example, I accidentally am selecting text on the Start Over button:



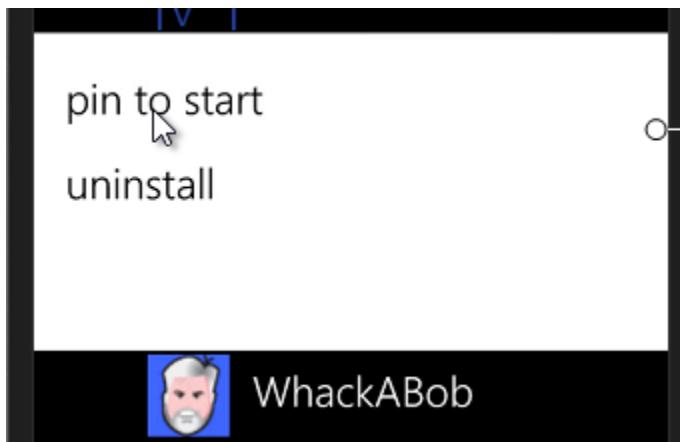
... ideally, I would not be able to do that. Furthermore, if I attempt to drag the entire web page down, I can pull the page away from the edges. That's something I want to disable as well.



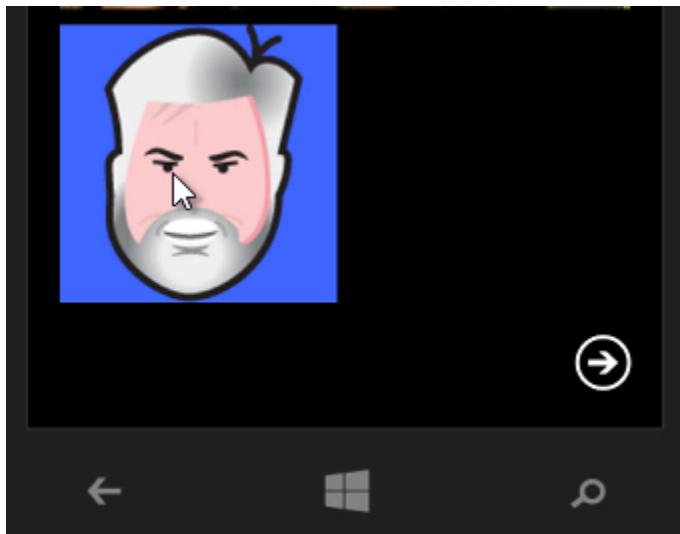
Clicking the emulator's Start button, then navigating to the Apps page, I can see that my logo appears next to the name of the app. However, the name is not quite right. I would like to change the text to: Whack-a-Bob



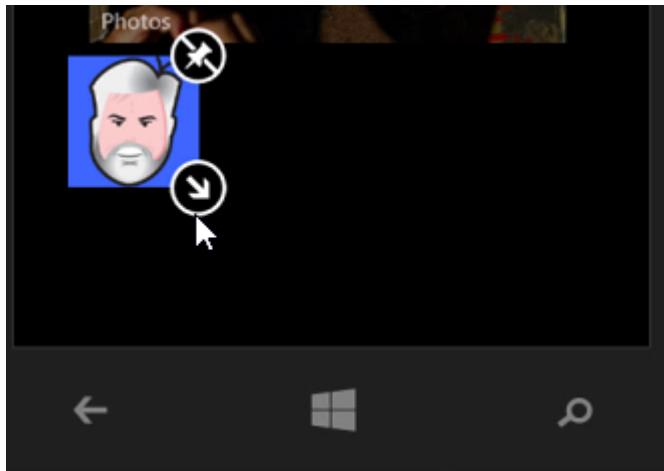
If I hold down on the app's entry, the context menu appears allowing me to test how the tiles will look when pinned to the start menu:



By default, it looks good however I would prefer if the app's title appeared in text under the logo (like the other apps on my phone).



If I hold down on the tile, I can modify the various sizes of the app tile to test those as well.



I stop debugging and begin to fix a few of the issues with my app.

To begin, I'll modify the display name that appears on the app and the start page tiles of the phone by going to the package.appxmanifest and modifying the Application tab's Display name property:

The information the system needs to deploy, display, or update your app is contained in the StoreManifest.xml file. You can use the Manifest Designer

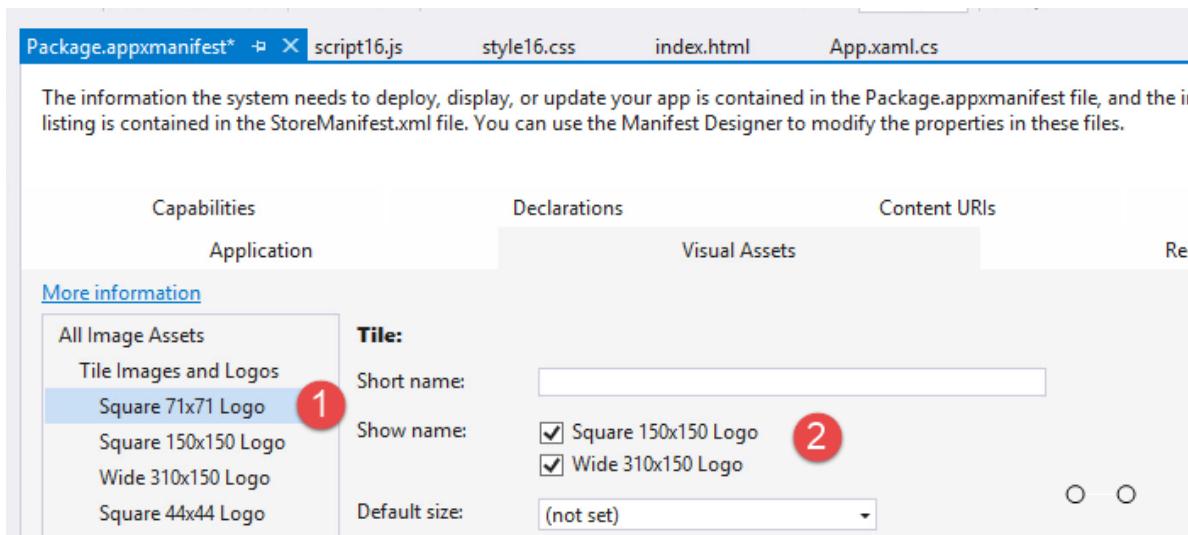
Capabilities	Declarations
Application	Visual Assets

Use this page to set the properties that identify and describe your app.

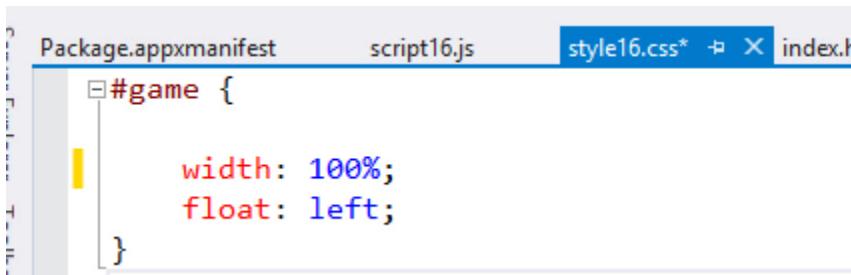
Display name: Whack-a-Bob

Entry point: WhackABob.App

To add the display name under the pinned tiles on the Phone's Start page, on the Visual Assets tab, I'll (1) select the Square 71x71 Logo page, and (2) put a check mark next to both the Show Name check boxes:



Next, I'll want to modify the index.html's layout to properly fit everything on the page, modify the text size, etc. Appropriate for a Phone app (as opposed to it's original purpose as a simple web page). I'll modify the #game id, removing the specific width (200px) replacing it with 100%, and floating to the left instead of the right:



The Start Over button requires some major changes. I change virtually every CSS property like so (in the #startover id's css definition):

The screenshot shows a code editor window with the tab bar at the top containing "phone.css", "Package.appxmanifest", "script16.js", "style16.css*", and "index.html". The "style16.css*" tab is active. The CSS code in the editor is:

```
#game {  
    width: 100%;  
    float: left;  
}  
  
#startover {  
    width: 200px;  
    height: 50px;  
    text-align: center;  
    padding-top: 10px;  
    margin-bottom: 20px;  
    float: left;  
    font-size: x-large;  
}
```

Next, in the index.html, I want to move the Start Over button (1) from the bottom, to (2) above the game div:

The screenshot shows the Visual Studio code editor with the file `index.html` open. The code is a simple HTML page for a game. Annotations are present:

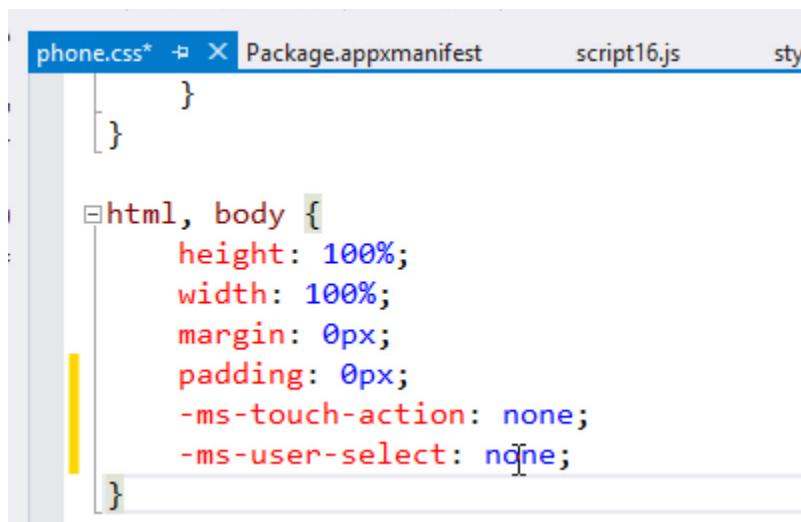
- Annotation 1:** A red circle with the number "1" is placed near the closing tag `</body>`.
- Annotation 2:** A red circle with the number "2" is placed near the text "Start over" inside the `<div id='startover'>` element.

```
<title>Windows Phone</title>
</head>
<body>
    <div>
        quite possibly the dumbest game ever!
    </div>
    <div id="page-title">
        whack-a-bob
    </div>

    <div id="startover">Start over</div> 2

    <div id="game">
        
        <p>SCORE: <span id="score">0</span></p>
        <p id="success">You whacked-a-bob!</p>
    </div>
1
</body>
</html>
```

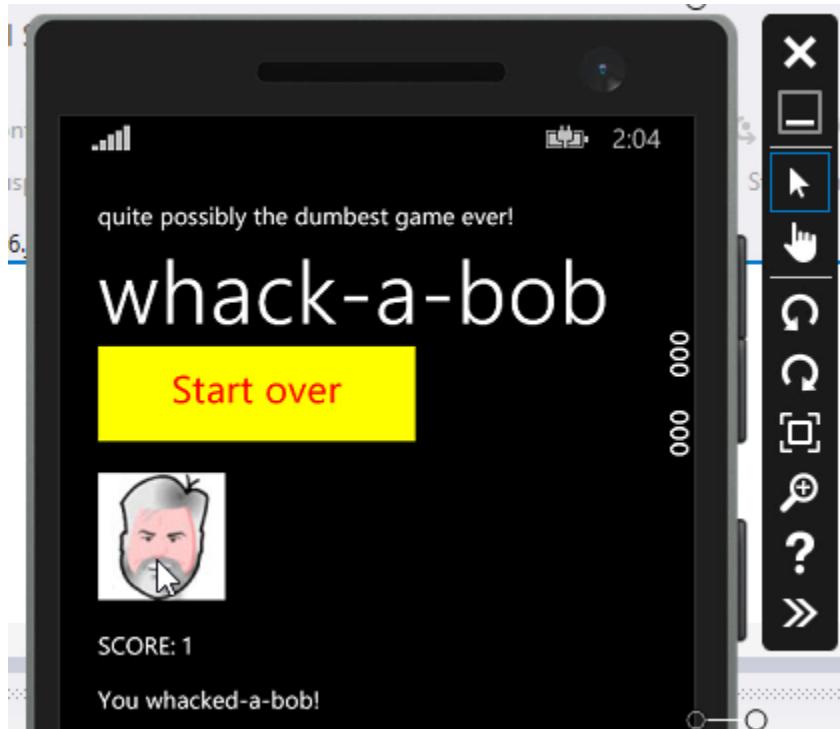
To fix the text selection issue and the drag / movement issue, I'll modify the phone.css page's html / body definition adding the `-ms-touch-action: none;` and `-ms-user-select: none;` settings like so:



```
phone.css* ✘ X Package.appxmanifest script16.js sty
}
}

html, body {
    height: 100%;
    width: 100%;
    margin: 0px;
    padding: 0px;
    -ms-touch-action: none;
    -ms-user-select: none;
}
```

This time when I re-run the app, I get the results I desire:



Before we conclude, Matthias wanted me to point out that if you're going to do a lot of work with jQuery inside of Visual Studio, you might prefer to install using Nuget package "jQuery vsdoc". This will give you IntelliSense for jQuery, syntax highlighting, and other helpful additions inside of Visual Studio. You'd get the same experience with Javascript and with all the jQuery methods and selectors and things of that nature as you get with XAML and C#.

<https://www.nuget.org/packages/jQuery/2.1.0>

The screenshot shows a web browser displaying the NuGet Gallery at <http://www.nuget.org/packages/jQu...>. The page title is "jQuery Events - 16 | Jav...". The main content area shows the package "jQuery vsdoc 2.1.0". The package icon is a blue circle with white dots. The download count is listed as "19,953 Downloads" and "547 Downloads of v 2.1.0". The package description states "jQuery IntelliSense for Visual Studio." and provides a command to install it via the Package Manager Console: "PM> Install-Package jquery-vsdoc". Below the description are social sharing links for Twitter and Facebook.

Lesson 15: Understanding the Hub App Template Overview and Navigation

In this lesson, we'll learn about the Hub App template. If you're familiar with the Windows Phone, you'll have seen instances of the Hub in use. There is so much here that we'll spend the majority of the remaining series talking about this template, the technologies it uses, and how to modify it and extend it to get the results we want. That's no small matter.

- (1) Create a new Hub App Template project called "HubAppTemplate"
- (2) Run the app. Note the navigation. Tie the navigation to pages in the project.
- (3) Open the HubPage.xaml and roll up the major sections ... they will primarily break down into: DataTemplates and HubSections.

Begin by creating a "throw away" project using the Hub App Template.

So, let's talk about a high level about how this app works and then we'll drill down into each of the pieces.

At the highest level is the Hub control which allows you to create a series of panels, or rather, HubSections. You can see there are five HubSections that correspond to the five panels in our app.

Each HubSection is comprised of (a) data and (b) a template that defines how that data should be displayed. So, two parts, data and a data template.

I'll talk about the source of the data in just a moment. However, let's take a look at the data templates. If we drill into a given HubSection like "SECTION 2":

```
<HubSection x:Uid="Section2Header"  
    Header="SECTION 2"  
    Width="Auto"  
    DataContext="{Binding Section2Items}"  
    d:DataContext="{Binding Groups[1],  
        Source={d:DesignData  
            Source=../DataModel/SampleData.json,Type=data:SampleDataSource}}">  
<DataTemplate>  
    <GridView ItemsSource="{Binding Items}"  
        AutomationProperties.AutomationId="ItemGridView"  
        AutomationProperties.Name="Items In Group"  
        ItemTemplate="{StaticResource Standard200x180TileItemTemplate}"  
        SelectionMode="None"
```

```

    IsItemClickEnabled="True"
    ItemClick="ItemView_ItemClick">
<GridView.ItemsPanel>
    <ItemsPanelTemplate>
        <ItemsWrapGrid />
    </ItemsPanelTemplate>
</GridView.ItemsPanel>
</GridView>
</DataTemplate>
</HubSection>

```

There's a lot going on here. That's because this is employing a whole framework of templating in order to make it extremely flexible. The unfortunate trade off to making a technology very flexible is often complexity, at least at first. However, it's all logical, so if you can just understand the logic behind it, if you can build a mental model, you can at a minimum be able to command it to do what you need it to by force with a lot of trial and error.

Besides the properties of the HubSection itself which I'll ignore for now because it has to do with the data, the HubSection has a child called DataTemplate. The DataTemplate is a GridView ... similar to a Grid control that we've used for layout, but it supports binding to data. The GridView represents this entire area in "SECTION 2". The GridView's ItemsPanel controls the flow of each individual item in the GridView. So, you have a collection of items. Will they flow horizontally or vertically? In this case, the ItemsWrapGrid by default will position each child elements sequentially from top to bottom. When elements extend beyond the container edge (the bottom, in this case), elements are positioned in the next column.

So, The ItemsPanel uses the ItemsWrapGrid to layout each item in our data in a certain flow direction, from top to bottom, then left to right.

But what about the layout of each individual item in our data? The GridView's **ItemTemplate="{StaticResource Standard200x180TileItemTemplate}"** property dictates that. As you can see, it is binding to a StaticResource called Standard200x180TileItemTemplate which is defined in the Page.Resources section:

```

<Page.Resources>
    <!-- Grid-appropriate item template as seen in section 2 -->
    <DataTemplate x:Key="Standard200x180TileItemTemplate">
        <Grid Width="180">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
            </Grid.RowDefinitions>

```

```

<Border Background="{ThemeResource ListViewPlaceholderBackgroundThemeBrush}"
    Height="173"
    Width="173"
    Grid.Row="0"
    HorizontalAlignment="Left">
    <Image Source="{Binding ImagePath}"
        Stretch="UniformToFill"
        AutomationProperties.Name="{Binding Title}"
        Height="173"
        Width="173"/>
</Border>
<TextBlock Text="{Binding Title}"
    Style="{ThemeResource BaseTextBlockStyle}"
    Typography.Capitals="SmallCaps"
    Grid.Row="1"
    Margin="0,12,0,0"
    IsTextScaleFactorEnabled="False"/>
</Grid>
</DataTemplate>

```

Each individual item is made up of a Grid that is 180 pixels wide. The grid defines two rows. The first row is filled with a border that contains an image, these gray images we see in the designer. I think the Border is just there in case the image cannot be displayed, or perhaps it provides a themed background color for transparent images. I'm not really sure of the purpose, but I think that will be the result.

The second row is filled with a TextBlock, the “Item Title: 1” and so forth.

So, now we understand how each item in our data source will be rendered to screen for “Section 2”. Admittedly, there are many tiny details I’m overlooking, but I can perform a search online for their meaning. I highly encourage you to do the research when you come across something that is not obvious by just reading its name or setting.

For example, I refrained from talking about the `<TextBlock Typography.Capitals="SmallCaps" />` because I give you enough credit to be able to look at that and understand what it is doing when you see the results on screen:



However, the <TextBlock IsTextScaleFactorEnabled="False" /> is not, at least to me, quite as obvious. So, I'll perform a search online to learn more about it.

Back in the GridView definition, there are three properties that affect navigation:

```
<GridView
  ...
  SelectionMode="None"
  IsItemClickEnabled="True"
  ItemClick="ItemView_ItemClick">
</GridView>
```

Setting SelectionMode to none means that we're not allowing individual items to be selected, as if there were a list of items that we're selecting for inclusion in some operation.

Instead, we want to allow the user to click a given item and perform a navigation operation. Setting IsItemClickEnabled="True" instructs the GridView how we want a tap to be interpreted. We want a tap on a given item to trigger an event called ItemClick, not a selection event called ItemSelected. The semantic difference of those two interactions is very important.

The ItemClick="ItemView_ItemClick" wires up an event handler to the ItemClick event.

If we navigate to that event handler in our code behind, we find the following definition:

```
void ItemView_ItemClick(object sender, ItemClickEventArgs e)
{
    // Navigate to the appropriate destination page, configuring the new page
    // by passing required information as a navigation parameter
    var itemId = ((SampleDataItem)e.ClickedItem).UniqueId;
    this.Frame.Navigate(typeof(ItemPage), itemId);
}
```

The first line of code is retrieving the clicked item from the ItemClickEventArgs and then casting it to an instance of the class that we're binding to. This SampleDataItem class is just sample data provided by the Hub App Template so that you can see some data when you run the application. We'll investigate that class in the next lesson. For now, just understand that it has a property called UniqueId. We're retrieving that UniqueId that presumably uniquely identifies the specific item that was tapped amongst the dozen or more items that the user could have tapped.

The next step is simple ... navigate to the ItemPage.xaml page passing in the id of the item that we want displayed on that page.

Based on our understanding of the Navigation model and how we can retrieve the object that was passed from one page to the next, we might think we know what happens on the ItemPage.xaml. We would expect to handle the OnNavigatedTo() method like we've done before.

However, if we were to look at the ItemPage.xaml.cs file, you'll find no such method declared. Instead, we see that the Hub App Template relies on a class called NavigationHelper to perform some repetitive tasks related to navigation and passing data between pages.

The NavigationHelper.cs file is located in the \Common folder. If you're following along, take a moment to read the lengthy explanation in the comments about what this class does and how to use it. In a nut shell, it handles state management for you, allowing users to return to the exact page in your app where they left off in their previous session. I would recommend that you do not make any changes to this class for now.

So, I'll just pluck out the lines of code that are used to enable the use of the NavigationHelper class from within the ItemPage.xaml.cs (in other words, I'm removing all the code that doesn't deal with navigation in the following code just so you can easily isolate what has been added to work with the NavigationHelper class):

```
public sealed partial class ItemPage : Page
{
    private NavigationHelper navigationHelper;

    public NavigationHelper NavigationHelper
    {
        get { return this.navigationHelper; }
    }

    public ItemPage()
    {
        this.navigationHelper = new NavigationHelper(this);
        this.navigationHelper.LoadState += navigationHelper_LoadState;
    }

    private async void navigationHelper_LoadState(object sender, LoadStateEventArgs e)
    {
        var item = await SampleDataSource.GetItemAsync((String)e.NavigationParameter);
        this.DefaultViewModel["Item"] = item;
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
```

```

{
    navigationHelper.OnNavigatedTo(e);
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    navigationHelper.OnNavigatedFrom(e);
}

#endregion
}

```

In a nut shell:

- (a) A new instance of NavigationHelper is created and its reference is available throughout the ItemPage class.
- (b) In order to keep the NavigationHelper class in the loop about navigation to and from this page, the page's OnNavigatedTo and OnNavigatedFrom methods turn around and call similar methods on the NavigationHelper.
- (c) In the constructor, the navigationHelper_LoadState method is wired up. It will be triggered by the NavigationHelper class when the page is navigated to. In the navigationHelper_LoadState event handler the data associated with the item that was clicked on will be retrieved and set as the source of data for the current page.

Recap

I want to stop there. There are many other features of the Hub App Template that we'll learn about in the following lessons. In this lesson, I focused on the overall layout and templating of the Hub control and the navigation technique that was employed. In the next lesson, we'll learn about the sample data and how the various templates bind to it for display.

Lesson 16: Understanding the Hub App Template's Sample Data Model

In this lesson, we'll pick back up with the default Hub App Template prior to making any changes to it. Previously we learned about the Hub control and its Hub Sections, and how each panel of content in the Hub control is arranged using a series of data templates. In this lesson, we'll look at where the actual data is coming from. In the next lesson, we'll look at how we use the data model to bind the user interface to the underlying data.

First, at a high level, the idea of binding is that you have a collection of objects, for example, let's say a collection of Cars like so:

```
List<Car> cars = new List<Car>();
```

Let's suppose that the Car class is simple, like the classes we worked with in the C# Fundamentals for Absolute Beginners series.

```
public class Car
{
    public int Id { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    public string ImagePath { get; set; }
}
```

You can add cars to that collection by retrieving data from a text file, a database, etc. Suppose you read from a comma delimited file or a JSON (JavaScript Object Notation) file and create a new instance of the Car class for each row of data in that file.

At this point, you would have a complete (albeit, a very simple) data model. In other words, the classes and the code that opens the data file, reads it, and turns it into instances of objects, and adds those objects to a collection, is essentially the data model.

And once you have that data model in place, essentially a collection of Cars in memory, you can bind that collection to a GridView or some other list-style control that will allow you to display them in a tabular or grid-like manner. You could use the ImagePath as the data source for a thumbnail picture of the given car. You could display the Make and Model in text that appears on top of the image. Clicking a given item in the GridView would allow you to see more information about the given car in the ItemsDetail.xaml page. More about that in the next lesson.

Hopefully you understand what we're trying to achieve. The Hub App Template supplies you with a generic data model including sample data, classes that represent data structures,

and classes that provide access to the instances of those data structures. Ideally you can just modify what they already have and use it for whatever type of data you want to work with. In fact later in this series, we'll do exactly that — we'll modify the Hub App Template to create a complete app that displays images and recipes for cupcakes.

The data model created by the Hub App Template is self contained in the \DataModel folder. There are two files:

SampleDataSource.cs - This single file has multiple parts, but it defines a class for sample data items, as well as a class that is used to group the items together. This gives you a simple hierarchical structure you can wrap your mind around. If you need to further add another grouping on top of that, you can extend this. Furthermore, you can scrap this all together. All you really need is a collection of objects, or some other object graph. The SampleDataSource.cs also knows how to open a data file, retrieve the results, and create instances of the Item and Group objects. So, unfortunately, this serves several purposes and as a result, the code may be a bit obscure for beginners.

Where does the actual data come from?

SampleData.json - JSON stands for JavaScript Object Notation. I created a lesson in the JavaScript Fundamentals for Absolute Beginner's series about JSON. I recommend you watch that for a thorough explanation of the syntax.

Now, I get asked the following questions often:

(1) Why did the developers of this template choose the JSON format over, say, a comma delimited file format?

Well, I don't have any special knowledge about this, but I suspect two reasons. First, JSON files can easily represent object graphs, or rather, collections that contain objects (in this case, "Groups") that have a property ("Items") which is a collection of an object ("Item"). You don't get that structure from a flat comma-delimited file. Secondly, it's relatively easy to read and write JSON using built-in classes as we'll learn throughout the remainder of this series.

I realize at first glance, JSON can be a bit intimidating. However, the good news is that it is quickly becoming a very popular file format and has the added benefit of being easily utilized by JavaScript and some commercial and open source document databases.

(2) Can I use some sort of structured relational database instead of JSON?

I have not personally done this, but I know some have tried to use a SQL Server Compact or LocalDb. Others have had success with a local file-based database called SQLite. I've not tried either of these techniques but I suspect it is possible. Also, you could use some external service such as those available on Windows Azure or Azure Mobile Services. We don't cover

those in this series, but these are all options depending on the requirements of your application.

One final thing I want to address conceptually / generically: what is a “data model”? This is a term that deals with your application’s architecture, or in other words, how you separate and layer your application’s code. Typically you want to keep logical layers separated and in most applications that means you keep the persistence code separate from the business rules, or rather, the domain logic, and you keep all of that separate from the presentation logic that displays data to the user. The benefit of abstracting the various responsibilities of your application into software layers is that you can manage change more effectively. I’ll resist the urge to talk about this in more detail.

The “data model” is a layer that encapsulates all of the logic required to retrieve data from an underlying data source, such as a JSON file or a database, and put it into an object graph that can be used by the other layers of the software. In our case, the result of calling into the data model is an object graph that contains the data that we’ll display in our application.

As I said earlier, the data model — more specifically the `SampleDataSource.cs` — has several classes all collected into one file.

The `SampleDataItem` class is the child. It is what you see in “SECTION 2” of the Hub.

The `SampleDataGroup` class is the parent. It has an `Items` property which is a collection of `SampleDataItem`. (Actually, it is a special type of collection called an `ObservableCollection<T>` which we’ll talk about in another video. For now, just think of it as a simple `List<T>` and I’ll explain the difference later.)

For the most part, `SampleDataItem` and `SampleDataGroup` are just data structures ... no methods outside of the constructor, just a set of properties.

All of the real heavy lifting in regards to retrieving data and creating instances of the `SampleDataItem` and `SampleDataGroup` classes is in the `SampleDataSource` class.

It looks like there’s a lot going on here, so let’s make sure we understand what it is doing.

First this line of code:

```
private static SampleDataSource _sampleDataSource = new SampleDataSource();
```

... is a coding pattern called “Singleton”. It forces just one instance of the class to be created, never more. That way you can always make sure you’re working with the same instance of the class.

Next:

```
private ObservableCollection<SampleDataGroup> _groups = new  
ObservableCollection<SampleDataGroup>();  
public ObservableCollection<SampleDataGroup> Groups  
{  
    get { return this._groups; }  
}
```

This exposes a Groups property which is a collection of SampleDataGroup. Exposing this property makes it easy to bind to the Groups and Items using the binding syntax in XAML. More about that in the next lesson.

The next three methods will return ALL groups, one specific group, or one specific item by either returning the _groups, or by performing some LINQ to Objects query on _groups:

```
public static async Task<IEnumerable<SampleDataGroup>> GetGroupsAsync() { ... }  
  
public static async Task<SampleDataGroup> GetGroupAsync(string uniqueId) { ... }  
  
public static async Task<SampleDataItem> GetItemAsync(string uniqueId) { ... }
```

Notice that they each call the EnsureSampleDataLoaded() method:

```
private static async Task EnsureSampleDataLoaded()  
{  
    if (_sampleDataSource == null || _sampleDataSource.Groups.Count == 0)  
        await _sampleDataSource.GetSampleDataAsync();  
  
    return;  
}
```

In other words, make sure we've loaded up the data before continuing. If Groups is empty, go load that data!

That leads us to the final method in the class, GetSampleDataAsync(). Before I talk about that
...

I've taken a great liberty by ignoring keywords that embellish these methods like "async" and "Task<T>" and even the "await" keyword. We'll devote an entire lesson to a high level explanation of these ideas. Let's just ignore they exist for now.

In pseudo code, the `GetSampleDataAsync()` method does the following:

- Make sure there's no data in Groups. If there is, this isn't necessary.
- Open the data file.
- Read the file's contents into a string, then parse it to JSON (or actually, into an array of JSON)
- Loop through the array, creating a Group object
- One element of the array contains another JSON array (Items).
- Loop through that array, creating an Item object, add it to the Items collection.
- Add the group (and all of its children) to the Groups collection.

Now, when the `GetGroupsAsync()`, `GroupGroupAsync()`, `GetItemAsync()` or just the `Groups` property are accessed, we'll have data available.

So, that's how the *sample* Data Model works. However I think it is important to emphasize that you don't have to follow this exact technique / approach. You can replace some or all of this. It's just a template, not a requirement. Your application may require a less complex class hierarchy, or perhaps a more complex one. You may replace the JSON file with some other data source. You can even retrieve data from over the Internet like in the case with Windows Azure and Azure Mobile Services. At the end of the day, you'll want to make this yours. As long as the result of your data model is collection you can bind to from within XAML, you're in good shape. That's what we'll talk about in the next two lessons.

Lesson 17: Understanding Data Binding, Data Sources and Data Contexts

Continuing our examination of the Hub App Template, we'll learn how the HubPage.xaml page and the ItemPage.xaml binds to the data model we learned about in the previous lesson.

Binding, also known as data binding, is common in the programming world. It is a convenient way of tying the presentation layer of your application to the data model. You can declaratively define the data source (most likely a collection of objects) for a given control, say, a grid or a list, and then tell the control to bind to the data source. It then takes care of creating a visual representation of each item in the data source as a row, a list item, etc.

When building Phone apps in XAML, you use the Binding syntax we learned about previously. We saw how to bind to a static resource for the purpose of styling.

You bind to data in a similar way using a similar syntax.

```
<TextBlock Text="{Binding Title}" ... />
```

Let's start from the top.

The Page class, in this case, the HubPage class defines private field called "defaultViewModel". This will be set to an object graph of Groups and related Items. That private field is exposed publicly through a read-only property called "DefaultViewModel". The important thing is that we're creating an instance of an ObservableDictionary<string, object>(). This class is something the template developers added to the template — it is not part of the Phone API. You can see its definition in the \Common folder. It's not all that important to understand how it works under the hood. All I really care about is what the name implies.

We've not talked about the term Observable yet, but we will soon. For now, just focus on the fact that we're working with a dictionary style collection. Recall from the C# Fundamentals for Absolute Beginners series that a dictionary is a specialized collection that allows you to create a key and an associated value. This will be important in just a moment.

As we learned about in an earlier lesson regarding the NavigationHelper class, the navigateionHelper_LoadState event handler is triggered when the page is navigated to. Let's look at how it's defined in the HubPage.xaml.cs:

```
private async void NavigationHelper_LoadState(object sender, LoadStateEventArgs e)
{
    // TODO: Create an appropriate data model for your problem domain to replace the
    sample data
```

```
    var sampleDataGroups = await SampleDataSource.GetGroupsAsync();
    this.DefaultViewModel["Groups"] = sampleDataGroups;
}
```

This code:

```
this.DefaultViewModel["Groups"]
```

... adds a new item to the dictionary with the key “Groups”.

What is going into the dictionary? The Groups” that are retrieved using the SampleDataSource.GetGroupsAsync().

You may wonder, why are they using a Dictionary if all they need is one item? Again, I would remind you this is a template. You could add more things to the Dictionary, then pull out what you need in the binding expression in XAML. But you can only set the data context for the page to one object, so you would use this as the container for as much data for the various hub sections.

What will we do with this dictionary once it is filled up with data? Our goal is to get the various GridViews in our XAML to bind to this data.

Look at the top of the HubPage.xaml page:

```
<Page
x:Class="HubAppTemplate.HubPage"
...
DataContext="{Binding DefaultViewModel, RelativeSource={RelativeSource Self}}"
```

Here we’re setting the DataContext of the entire Page, binding it to the DefaultViewModel of the current class (i.e., the {RelativeSource Self} means “you can find this class relative to myself, in my own class definition”).

A DataContext is defined by a parent and then utilized by the children. In this example, the parent is the Page class, and the children are the various controls that will take the data from the DefaultViewModel and bind to it.

So, given that the DataContext for the Page is set to the DefaultViewModel property, the ObservableDictionary already full of data, we can now revisit the XAML we looked at a couple of lessons ago (I’ll remove all of the XAML that is not important):

```
<HubSection Header="SECTION 2" DataContext="{Binding Groups[0]}" ...>
```

```

<DataTemplate>
<GridView ItemsSource="{Binding Items}"
    ItemTemplate="{StaticResource Standard200x180TileItemTemplate}">
    ..
</GridView>
</DataTemplate>
</HubSection>

```

As you can see, I've removed A LOT of the XAML in order to clarify this example. Again, we're working with the second panel, or rather, HubSection. The DataContext for the HubSection is Section2Items, the ItemsSource is the Items property. If you compare this to DefaultViewModel, it has a dictionary element called Section2Items which is of type SampleDataGroup. So, Section2Items (an instance of SampleDataGroup) has an Items property of type ObservableCollection<SampleDataItem>.

Now, the ItemTemplate comes into play. The ItemTemplate of the GridView is used to render each item that is displayed in the GridView. So, in our case, each item in the Items property collection of SampleDataItem will be rendered using the Standard200x180TileItemTemplate defined near the top of the file. Again, I've removed almost all the XAML except for those parts that are important for this discussion:

```

<DataTemplate x:Key="Standard200x180TileItemTemplate">
<Grid>
    ...
<Image Source="{Binding ImagePath}"
AutomationProperties.Name="{Binding Title}" />

<TextBlock Text="{Binding Title}" />
</Grid>
</DataTemplate>

```

For each SampleDataItem in the DefaultViewModel["Section2Items"]'s Items property, The Image's Source property is set to the SampleDataItem's ImagePath. The TextBlock's Text property is set to the SampleDataItem's Title.

A couple of take aways from this first example:

(1) Keep in mind that the DataContext keeps flowing downward. In other words, the Page's data context flows into the HubSection. The HubSection's data context flows into the GridView, etc. Understanding the hierarchy of the binding will allow you to keep things straight in your mind.

(2) When you're first getting started, it's useful to stay close to examples like the ones in this template. If you can merely modify the existing example until you're comfortable with the

hierarchy of controls and settings, you'll be more productive up front. You'll see how I am able to utilize this Hub App Template to create an entire app in just a few lessons. Make small changes to these templates — you may even keep the original file in tact and use it for reference. Create a second page and copy parts from the HubPage.xaml into your new page until you're comfortable with the relationships between the various parts.

Let's take a look at a couple of more examples.

I'll back up to the first HubSection which displays the Groups in a list:

Here's the XAML that accomplishes this, removing all code that doesn't deal with data binding:

```
<HubSection x:Uid="Section1Header"
DataContext="{Binding Groups}>
<DataTemplate>
...
</DataTemplate>
</HubSection>
```

The HubSection's DataContext is set to the DefaultViewModel's "SectionGroups" property.

```
private async void navigationHelper_LoadState(object sender, LoadStateEventArgs e)
{
    var sampleDataGroups = await SampleDataSource.GetGroupsAsync();
    this.DefaultViewModel["Groups"] = sampleDataGroups;
```

GetGroupsAsync() merely returns the list of Groups. Easy enough.

What about the DataTemplate?

```
<DataTemplate>
<ListView ItemsSource="{Binding}"
IsItemClickEnabled="True"
ItemClick="GroupSection_ItemClick"
ContinuumNavigationTransitionInfo.ExitElementContainer="True">
<ListView.ItemTemplate>
<DataTemplate>
<StackPanel>
<TextBlock Text="{Binding Title}"
Style="{ThemeResource ListViewItemTextBlockStyle}" />
```

```
</StackPanel>
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
</DataTemplate>
```

The big differences between the Section 2 and Section 1:

- (a) Section 1 uses a ListView instead of a GridView, so all items flow vertically in one list, not in rows and columns like we saw in the GridView.
- (b) In Section 1, each individual item is styled in the body of the DataTemplate, it doesn't use a StaticResource binding expression.
- (c) It uses the binding expression {Binding} because it is binding to the same data context as the entire page — the list of Group objects, whereas Section 2 is binding to a subset of the page's data context, just the first Group object, Groups[0]. More about that in a moment.

The rest of the HubSections use similar techniques. Before I leave the HubPage.xaml, I do want to explain one last curiosity. How is it that the designer view displays the actual data? How it is loading what seems to be the actual data from the SampleData.json? The key is in the Page's declaration:

```
<Page
...
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:data="using:HubAppTemplate.Data"
DataContext="{Binding DefaultViewModel, RelativeSource={RelativeSource Self}}"
d:DataContext="{Binding Source={d:DesignData Source=/DataModel/SampleData.json,
Type=data:SampleDataSource}}"
mc:Ignorable="d">
```

Any time you see the d: prefix applied, think “design time”. For reasons I don’t want to explain in detail at the moment, that last half of the <HubSection /> snippet (above) loads sample data (from the SampleData.json file) and sets it to the data source. It’s a convoluted string of binding statements and parameters, but in a nut shell that is what’s happening. In other words, when you see data in Visual Studio (or another tool like Blend), that is being loaded as a result of that command. Remember, XAML is used to create instances of data. At compilation, that XAML is converted into Intermediate Language code and at run time, it is executed just like the other C# code we wrote. But at DESIGN TIME, Visual Studio (and Blend)

need a special set of commands to take the data from our JSON file (or other data source) and use it for display.

Why do we need a design time experience that displays data visually at all? We don't. If you were creating your own app, you could skip this step. However, it is convenient when you want to use the Visual Studio designer to preview what the content will look like when properly styled. Just know this ... if you change from the SampleData.json file and use some other technique or some other data source for your app, you may get weird results at design time. The same general principles apply here that we've been learning about in this lesson.

Before we conclude our examination of Binding, Data Contexts and Data Sources, let's move on to the ItemPage.xaml. When you click on a single item in the HubSections (all except the first HubSection which will open SectionPage.xaml) you will navigate to the ItemPage.xaml. It, too, has a DefaultViewModel that is populated when the page is navigated to:

```
private ObservableDictionary defaultViewModel = new ObservableDictionary();  
  
private async void NavigationHelper_LoadState(object sender, LoadStateEventArgs e)  
{  
    // TODO: Create an appropriate data model for your problem domain to replace the sample  
    // data  
    var item = await SampleDataSource.GetItemAsync((string)e.NavigationParameter);  
    this.DefaultViewModel["Item"] = item;  
}
```

The most important part of this is (1) it calls GetItemAsync to load the data for a single SampleDataItem, and (2) the DefaultViewModel only has one dictionary key called "Item".

In the page's XAML, the Page's DataContext is set to the DefaultViewModel as we would expect:

```
<Page  
    x:Name="pageRoot"  
    x:Class="HubAppTemplate.ItemPage"  
    DataContext="{Binding DefaultViewModel.Item, RelativeSource={RelativeSource Self}}"
```

... and the only element on the templated page that binds to the data source is the TextBlock used for the title of the page:

```
<Grid x:Name="LayoutRoot" >  
    ...  
    <StackPanel />
```

```
...
<TextBlock Text="{Binding Title}" />
```

Obviously, the intent here is to merely provide the framework that you would use to add more detail to this page. The most important aspect of this is the binding syntax, in this case used to navigate from Item (the only key in the DefaultViewModel, which is of type SampleDataItem) to its Title property.

Let me take a moment to talk about a few popular binding expressions. There are so many permutations of this that I couldn't possibly talk about them all. Let me just review a few of the basics.

Basic Binding

{Binding} - Bind to the current DataContext. You see this when the current control, say, a ListView control, binds to the parent Page's DataContext. You would then expect the ListView's ItemTemplate to bind to properties of the DataContext.

{Binding MyProperty} - Bind to the MyProperty property of the current DataContext.

{Binding MyObject.MyProperty} - Bind to the MyProperty property of MyObject. MyObject should be a property of the current DataContext.

{Binding ElementName=MyTextBox, Path=Text} - Bind to the Text property of another XAML element named MyTextBox.

{Binding MyObject RelativeSource={RelativeSource Self}} - Bind to MyObject which should be a member of the current class. So, I would expect MyObject to be an object or object graph defined as a partial class for this Page.

In addition to these there are many more advanced scenarios. If you want to see more examples and see them in context, check out:

<http://code.msdn.microsoft.com/windowsapps/Data-Binding-7b1d67b5>

Admittedly, it is a Windows App Store project, however most of the ideas transfer over to binding with the Phone.

I'd also recommend that you read all the articles here:

[http://msdn.microsoft.com/en-us/library/ms750612\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms750612(v=vs.110).aspx)

Admittedly, this is for Windows Presentation Foundation, however many of the ideas transfer over to binding with the Phone.

My final word on binding; in our examples so far, we've only seen one side of data binding — from a data source to the XAML. However, you can also bind in a two-way fashion allowing changes to the data via input controls like a TextBox, ComboBox, CheckBox, RadioButton, etc. to update the underlying data source and anything binding to it, in turn. In fact, this two-way binding is a basic tenet of a pattern known as MVVM, or rather, Model-View-ViewModel which we'll examine in the next lesson.

Recap

We covered a lot of ground in this lesson. Binding, and specifically data binding, is a key concept in building data driven applications based on a data model. It is how you marry the data stored in a data file (like the SampleData.json) to the user interface. My recommendation is to keep your structure as flat as possible as you're getting started to ensure that you don't confuse yourself as you set the DataContext for your app's object graph.

Lesson 18: Understanding MVVM: ObservableCollection<T> and INotifyPropertyChanged

In the previous lessons we learned about the Data Model and how it is used as the Data Context for Pages and Controls. And when we talked about the Data Model, we're talking specifically the classes in the SampleDataSource.cs file. We learned about the binding syntax used by XAML to bind to the Data Model that was exposed by the code behind.

The Hub App Template employs a design pattern in software called MVVM, or Model-View-ViewModel. At a high level, MVVM seeks to “separate concerns”, extracting code that is specific to presentation (the View, or rather, the XAML that defines the user interface), the code specific to the domain (the Model, or rather, the SampleDataGroup and SampleDataItem classes), and the code that coordinates the activities of the model and provides the view with access to the model, (the ViewModel, or rather, the SampleDataSource class).

MVVM is a slightly advanced topic, and frankly, I'm not sure I'm qualified to provide an exhaustive explanation. Unfortunately, even some authors and books that have attempted to clarify this topic have not always done a good job doing it either. But what I can do is show you the basics and how and why they are employed in the Hub App Template, so that you can begin to understand the thought process that influenced the organization and the design of the code in the template. Obviously, my goal is to help you better make changes to this template, and a fundamental understanding of MVVM should help you build a mental model of the responsibilities for each class, each piece of the puzzle.

Let's start with the Model. In software architecture, a Model is a representation of a given business or domain problem. If you were building an Accounting system, your Model might have classes for things like Account, Customer, Credit, Debit, and so on. If you were building a game, your Model might have classes for things like Planet, Spaceship, Soldier, Weapon and so on. If you were building a media app, your Model might have classes for Sound of Video, Categories of Sounds and videos, and so on. In most MVVM implementations I've seen, the Model is little more than a data structure ... a class defining properties and basically the “state” of the object, but almost no methods to implement business logic. (Just to provide a little guidance here, many believe this is an anti-pattern, or rather, “a bad idea”, called an Anemic Domain Model in which business logic is typically implemented in separate classes which transform the state of the domain objects. But we'll ignore that for now.) You can see examples of a typical model that utilizes MVVM in our Hub App Template: the SampleDataItem and SampleDataGroup classes. These have properties, but no real methods the implement logic.

Next is the View. The view has the responsibility of presentation of the Model. It does this through Binding, a core component of XAML through the binding syntax we've seen. What does the View bind to? We've seen examples of binding to data via the ViewModel, but later in this series we'll see how the View binds to commands that are implemented in the ViewModel.

Commands are essentially methods that perform logic and respond to the user's interaction with the View.

The ViewModel is the key to making MVVM work. The ViewModel provides a layer of abstraction between the View and the Model. We can see this in the SampleDataSource class. It provides a data source to the View. It must load data into instances and collections of the Model classes to create an object graph. When I use the term "object graph" I'm referring to instances of classes that are related. We see this at work in the GetSampleDataAsync() method. Then, once the object graph is loaded into memory, there are various ways of returning parts of it based on the needs of a given view. So, that is the role of the public property Groups, as well as the methods GetGroupsAsync(), GetGroupAsync() and GetItemAsync().

You might wonder whether the code behind (i.e., HubApp.xaml.cs) for the XAML is part of the View or the ViewModel. After all, (1) it retrieves the appropriate data for the given View in the navigationHelper_LoadState() method, and (2) it exposes the data retrieved from the ViewModel as a property to the XAML (i.e., defaultViewModel). I think of these are merely helpers to the main job of the View, and therefore I would classify them as part of the View itself.

I say all of that to say this ... the View should merely observe the data exposed by the ViewModel. In other words, it should watch the collections (or instances of objects) that were delivered by the ViewModel for changes and react to those changes. This is called "observability". If an instance of an object that is part of a collection managed by the ViewModel is added, removed or updated, then the View should be notified of the change and the View should update itself with that change. Fortunately, all of the "plumbing" required to make this work is baked into XAML. But from your perspective, as the developer

In order to make this work, your classes must implement the INotifyPropertyChanged interface. By implementing INotifyPropertyChanged, your class says "You can observe me, I'll let you know when I change." I don't have a complete example to show you, but let me show the simplest example I can think of to show how such an implementation might work:

```
class Sample : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public int ID { get; set; }

    private string name = String.Empty;

    public string Name
    {
```

```

get
{
    return this.name;
}

set
{
    if (value != this.name)
    {
        this.name = value;
        NotifyPropertyChanged("Name");
    }
}
}

private void NotifyPropertyChanged([CallerMemberName] String propertyName = "")
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
}

```

By implementing `INotifyPropertyChanged` we must agree to raise an event called `PropertyChanged`. This is the fundamental wiring that allows other classes (like user interface classes) to be notified that changes to an instance of this class have occurred (so now go update yourself to reflect those changes in your user interface). This event is triggered in the `NotifyPropertyChanged()` method, which is called whenever the `Set` operation is invoked on the `Name` property. So, in other words, when a line of code like this is executed:

```

Sample mySample = new Sample();
mySample.Name = "Bob";

```

... the `PropertyChanged` event is fired and any class that listens to this event (like certain user interface classes / controls in the Phone API) will have an opportunity to refresh the data they display.

Having said all of that, the Hub App Template does not have any classes that directly implement `INotifyPropertyChanged`. Instead, it uses `ObservableCollection<T>` (as well as a custom class, `ObservableDictionary`) which implements the `INotifyPropertyChanged` interface internally. So, the moral to the story is that, as long as we're using `ObservableCollection<T>` to expose collections of the objects from the `ViewModel` to the `View`, we'll get observability at the

COLLECTION level. `ObservableCollection<T>` is essentially a `List<T>` with the added superpower of observability.

A couple of important aspects about this.

The `SampleDataGroup` has an `Items` property which is an `ObservableCollection<SampleDataItem>`. Therefore, whenever new `SampleDataItems` are added or removed from the `Items` property, the `PropertyChanged` event will be fired by the collection on any user interface elements that bind to the `Items` collection. Unfortunately, this is a bit anticlimactic because we do not see this happen in the template as it stands right now. In other words, there's no functionality that requires we remove items from the collection, and therefore we don't see it in play.

The second important aspect I want to point out is that if we want observability at the `SampleDataItem` level — in other words, if we want user interface elements that bind to the `Items` collection to update when the properties of an item in that collection are changed, we would need to manually write the code to implement `INotifyPropertyChanged` in the `SampleDataItem` class. Since that is not how the class is currently implemented, if you were to change the value of a property, such as the `Title` or `Subtitle`, while the app is running, you would NOT see the user interface update to reflect that change. Why? Because the `SampleDataItem` class doesn't have the ability (again, as it is currently implemented) to fire the `PropertyChanged` event and notify any observers of the change. However, I believe I'll demonstrate this when we build an application later in this series that works with observable data.

Recap

In this lesson we talked about two very important ideas. Hopefully you can see the larger role of MVVM and its influence on the Hub App Template. That design pattern shaped and guided the developers of the template to provide an implementation that separated the concerns of the app and delegate responsibilities to certain classes. Furthermore, hopefully you now understand, at least at a high level, why the Hub App Template is using `ObservableCollection<T>`. It provides a convenient means of updating user interface controls the notification to update itself when the underlying data source has changed.

There are more aspects of MVVM that we'll cover in this series. I'll talk Commands later in this series which allows the user interface to bind events like a button click to a method in the `ViewModel`.

Lesson 19: Understanding async and Awaitable Tasks

We've spent a lot of time examining the Hub App Template, and for good reason — it illustrates many important concepts. We'll turn our focus back to the SampleDataSource class to talk about several keywords that I've ignored until now. The keywords I'm referring to are:

- (1) `async`
- (2) `await`
- (3) `Task<T>`

These keywords are newly added to C# 5.0 and are generally referred to as the new "async" feature, which is short for "asynchronous" or "asynchrony". In a nut shell, this new feature is a simplified way of improving the performance of the app and making it more responsive to the user without the complexity of writing code to use multiple threads. If you call a method of the Windows Phone API or some other library supporting `async` that could potentially take a "long time", the method will say "I promise to get those results to you as soon as possible, so go on about your business and I'll let you know when I'm done". The app can then continue executing, and can even exit out of method contexts. Another word for this in computer programming terminology is a "promise".

Under the hood, when you compile this source code (i.e., the `SampleDataSource` class), the Compiler picks apart the source code and implements it as a complex series of statements in the Intermediate Language to allow this to happen flawlessly and it does it without the use of multiple threads in most cases.

`Async` is best used for operations that have a high degree of latency, but are not compute intensive. So, for example, the `GetSampleDataAsync()` method retrieves data from the `SampleData.json` file and loads it into an object graph of `SampleDataGroup` and `SampleDataItem` instances. That could potentially take a second or two which is an eternity in computing. Another example would be calling a web API to collect data. Relying on communications over the Internet could potentially take a long time. But the Phone's PROCESSOR is not busy at all. It's just sitting there, waiting for a reply from the web service. This is known as "I/O Bound" ... I/O means "In / Out" ... so things like the file system, the network, the camera, etc. involve "I/O bound operations" and are good candidates for `async`. In fact, the Windows Phone API designers decided to bake `async` into all I/O Bound operations forcing you to use this to keep the phone responsive to the end user.

Contrast that to a compute-intensive operation such as a photo filter app that must take a large image from the camera and run complex mathematical algorithms to change the colors or the position of each pixel in the image. That could take a long time, too, but in that case, the Phone's processor is hard at work. This type of operation is known as "CPU Bound". This is NOT a good use of `async`. In this case, you would want to consider a Background Worker which helps to manage threads on the Windows Phone platform. If you are developing in .NET, you

may prefer to work with threading via the Task Parallel Library instead, or revert back to managing threads manually which has been an option since the very earliest versions of .NET.

Understanding multi-threading, parallel programming, the Task Parallel Library, even Background Workers on the Windows Phone API are WAY beyond the scope of this series and this lesson. It's a large topic and I'll not lie to you, it makes my head spin. If you want to learn a little more about async and how it applies to the Windows RunTime, check out:

Working with Async Methods in the Windows Runtime

<http://channel9.msdn.com/Series/Windows-Store-apps-for-Absolute-Beginners-with-C-/Part-12-Working-with-Async-Methods-in-the-Windows-Runtime>

... and be sure to read the comments where I further explain this idea.

But back to the topic at hand ... async is for those common occasions when you have a blocking operation that is not compute intensive, such as our case where we are waiting for these lines of code to complete:

```
StorageFile file = await StorageFile.GetFileFromApplicationUriAsync(dataUri);

string jsonText;
using (var stream = await file.OpenAsync(Windows.Storage.FileAccessMode.Read)) {
    ...
}
```

In this case, we use the await keyword, which says "I'll get back to you when I'm finished". The thread of execution can continue on through the other lines of code until it absolutely must have the results of that operation. Now, in our case, we attempt to open and work with the contents of the file in the very next line of code, so little advantage is gained. However, we must still use the await keyword because the `StorageFile.GetFileFromApplicationUriAsync()` and `file.OpenAsync()` methods both return `IAsyncOperation<TResult>`. `IAsyncOperation<TResult>` is just a specialized version of `Task<TResult>` that is used by the Windows and Phone API. By virtue of the fact that they return a `Task<T>`, we must await the results. You'll see await-able methods with a common convention ... they all end with the suffix `Async`.

Furthermore, any method that uses an await-able method must be marked with the `async` keyword in their method signature, AND if it is supposed to return a value it must return the value wrapped with a `Task<T>`. If the method is void, it will merely be marked void in the method signature. Since both the `StorageFile.GetFileFromApplicationUriAsync()` and `file.OpenAsync()` methods are marked with await, and our method, `GetSampleDataAsync()` returns a `Task` we must mark our method with `async`.

```
private async Task GetSampleDataAsync() { }
```

In this case, Task is the return value. What is returned exactly? Just a promise that the GetSampleDataAsync() will finish running and when it does, any code calling GetSampleDataAsync() will be notified. Here again, a lot of compilation magic happens at this point, the likes of which I don't pretend to fully understand.

How about in the case of GetItemAsync():

```
public static async Task<SampleDataItem> GetItemAsync(string uniqueId)
{ . . . }
```

In this case, Task<SampleDataItem> is returned. But what is returned exactly? Just a promise that the GetItemAsync() will finish running and when it does, any code calling GetItemAsync will be notified and will be rewarded with an instance of SampleDataItem. Again, at this point a lot of compilation magic happens.

In both of these cases, the presence of an awaitable task essentially means that any code that calls our methods can continue executing until it absolutely needs the result of those methods.

Again, remind me, why are we doing all of this? In hopes of making our app more responsive. This operation should not be blocking the Phone's processor from taking on other tasks like answering a phone call or running background tasks on behalf of other apps. It won't prevent the user from selecting any of the hardware buttons on the phone and getting an instant response.

Finally, even if you didn't understand much of what I just said, let me make it really simple:

(1) Certain methods in the Phone API use awaitable tasks to make ensure that they return control back to your code thus making your app hopefully as responsive as possible.

(2) Therefore, when you need to use one of these methods in your app, you need to do the following:

```
private void myExample() {
    Uri dataUri = new Uri("ms-appx:///DataModel/SampleData.json");
    StorageFile file = StorageFile.GetFileFromApplicationUriAsync(dataUri);
}
```

(2a) Add await before the call to the Async method

(2b) Remove void, add: async Task

```
private async Task myExample() {
    Uri dataUri = new Uri("ms-appx:///DataModel/SampleData.json");
    StorageFile file = await StorageFile.GetFileFromApplicationUriAsync(dataUri);
}
```

(2c) Call your new method using await

... Or, if you intend to return a value ...

```
private StorageFile myExample() {  
    Uri dataUri = new Uri("ms-appx:///DataModel/SampleData.json");  
    StorageFile file = StorageFile.GetFileFromApplicationUriAsync(dataUri);  
}
```

Follow the previous steps, but do this instead:

(2d) Rework the return type to: `async Task<StorageFile>`

So:

```
private async Task<StorageFile> myExample() {  
    Uri dataUri = new Uri("ms-appx:///DataModel/SampleData.json");  
    StorageFile file = await StorageFile.GetFileFromApplicationUriAsync(dataUri);  
}
```

... and the compiler will take care of the rest.

Recap

To recap, the big take away in this lesson is what `async` is, how it works, what scenarios it was designed to address, and its overall purpose -- to keep the phone and your apps responsive during long running I/O bound operations.

Lesson 20: Playing Video and Audio in a MediaElement Control

You may think it an odd transition from talking about MVVM and awaitable tasks to something rather fun and mundane like playing video and audio in your phone app, however this is the last lesson before we go off and build a full app based on the ideas we've learned thus far. I won't spoil the surprise by foreshadowing what it is we'll be building, but it will involve playing video and the Hub App Template just to show how flexible the Hub App Template is with a little imagination.

To keep the example as simple as possible, I'm going to create a new Blank App Template project called "SoundAndVideo".

I will illustrate some very simple scenarios in this lesson ... just a bit more than we'll need for our full featured app in the next lesson, but not enough to go off and create some great audio player app. You'll want to consult the documentation if you have a more advanced scenario.

I'll begin by adding the XAML for my project:

```
<StackPanel>
  <MediaElement x:Name="myMediaElement"
    Height="10"
    Width="10"
    Source="/Assets/Duck.wav"
  />

</StackPanel>
```

Also, I'll copy the Duck.wav file from the assets I've included with this series to the \Assets folder in my project by dragging-and-dropping from Windows Explorer to Visual Studio.

When I run the app, I immediately hear the Duck quacking. Admittedly, the sound is briefly interrupted during playback, but I've marked that up to the fact that there's a lot going on during the deployment from Visual Studio to the emulator.

What if I do not want the sound to play immediately, but only want it to play when I click a button?

```
<MediaElement x:Name="myMediaElement"
  Height="10"
  Width="10"
  Source="/Assets/Duck.wav"
```

```

        AutoPlay="False"
    />

<Button x:Name="playSoundButton"
    Height="80"
    Width="200"
    Content="Play Sound"
    Click="playSoundButton_Click" />

```

I added the AutoPlay="False" to stop the sound from playing automatically. In the button's click event:

```

private void playSoundButton_Click(object sender, RoutedEventArgs e)
{
    myMediaElement.Play();
}

```

What if I want to use this same MediaElement for multiple sounds or videos? First, I would remove the Source and AutoPlay properties:

```

<MediaElement x:Name="myMediaElement"
    Height="10"
    Width="10"
    />

```

And I would set the Source property in code:

```

private void playSoundButton_Click(object sender, RoutedEventArgs e)
{
    myMediaElement.Source = new Uri("ms-appx:///Assets/Duck.wav",
UriKind.RelativeOrAbsolute);
    myMediaElement.Play();
}

```

The ms-appx:/// means "look for this in the current app package once it is deployed to the Phone."

What if I want to use the same MediaElement to play both an audio file AND a video file? I'll make the size of the MediaElement larger to accommodate the video:

```
<MediaElement x:Name="myMediaElement"
```

```
Margin="0,40,0,40"  
Height="400"  
Width="240" />
```

I'll add a button:

```
<Button x:Name="playVideoButton"  
Height="80"  
Width="200"  
Content="Play Video"  
Click="playVideoButton_Click"/>
```

I'll copy the video named coffee.mp4 file from the assets I've included with this series to the \Assets folder in my project by dragging-and-dropping from Windows Explorer to Visual Studio.

And I'll handle the button's click event:

```
private void playVideoButton_Click(object sender, RoutedEventArgs e)  
{  
    myMediaElement.Source = new Uri("ms-appx:///Assets/coffee.mp4",  
UriKind.RelativeOrAbsolute);  
    myMediaElement.Play();  
}
```

What if I want to pause the video during playback? I would need to keep track of the current state of the MediaElement. I suppose there are many ways to do that, but I can imagine several states: Playing, Paused, Stopped. So, I'll add a new class file to create an enum called MediaState:

```
public enum MediaState  
{  
    Stopped,  
    Playing,  
    Paused  
}
```

Then I'll use that to perform the following logic:

```
private MediaState state = MediaState.Stopped;
```

```

private void playVideoButton_Click(object sender, RoutedEventArgs e)
{
    if (state==MediaState.Stopped)
    {
        myMediaElement.Source = new Uri("ms-appx:///Assets/coffee.mp4",
UriKind.RelativeOrAbsolute);
        state = MediaState.Playing;
        myMediaElement.Play();
    }
    else if (state==MediaState.Playing)
    {
        state = MediaState.Paused;
        myMediaElement.Pause();
    }
    else if (state==MediaState.Paused)
    {
        state = MediaState.Playing;
        myMediaElement.Play();
    }
}

```

I'll also modify the MediaElement to handle the MediaEnded event in order to set the MediaState back to Stopped.

```

<MediaElement x:Name="myMediaElement"
    Margin="0,40,0,40"
    Height="400"
    Width="240"
    MediaEnded="myMediaElement_MediaEnded" />

```

Finally, I'll handle the MediaEnded event:

```

private void myMediaElement_MediaEnded(object sender, RoutedEventArgs e)
{
    state=MediaState.Stopped;
}

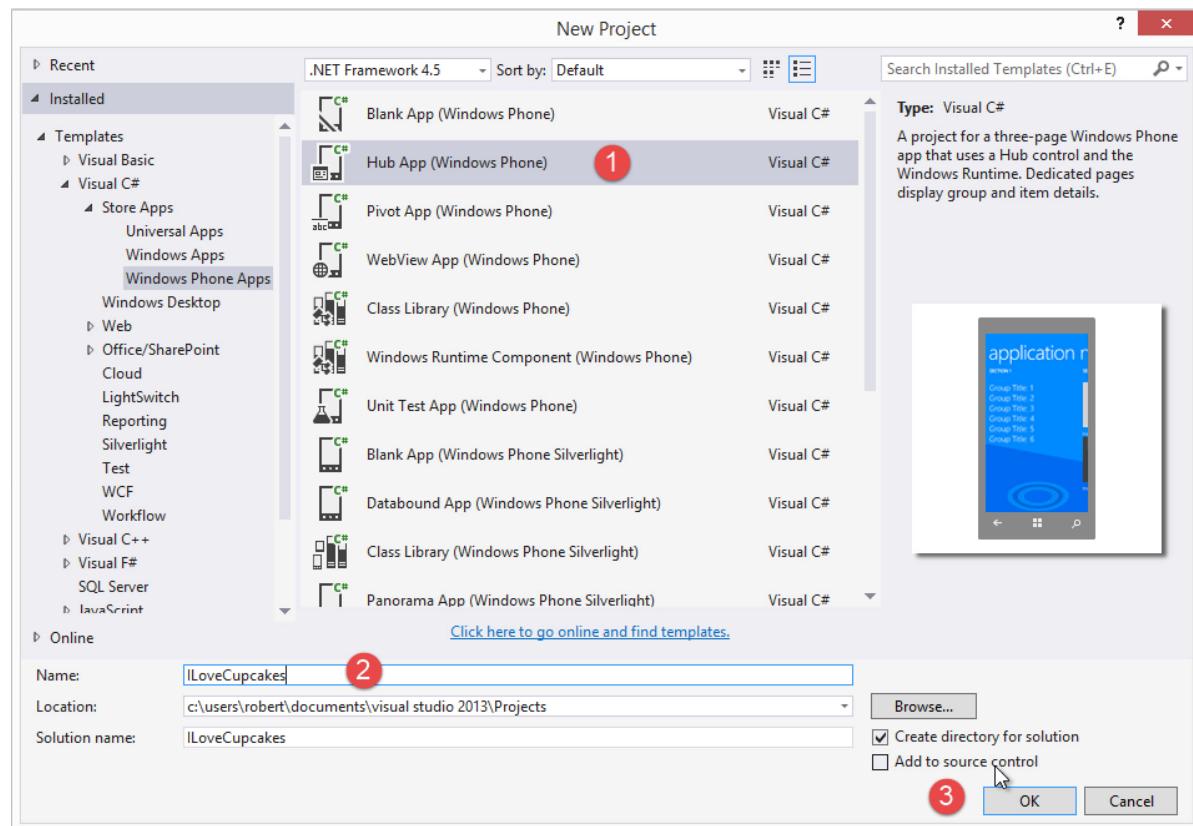
```

As I said at the outset, there are more features of the MediaElement than we'll discuss. However, this will at least get you started thinking about what's possible. In the next lesson, we'll put many of these ideas together to create an entire app.

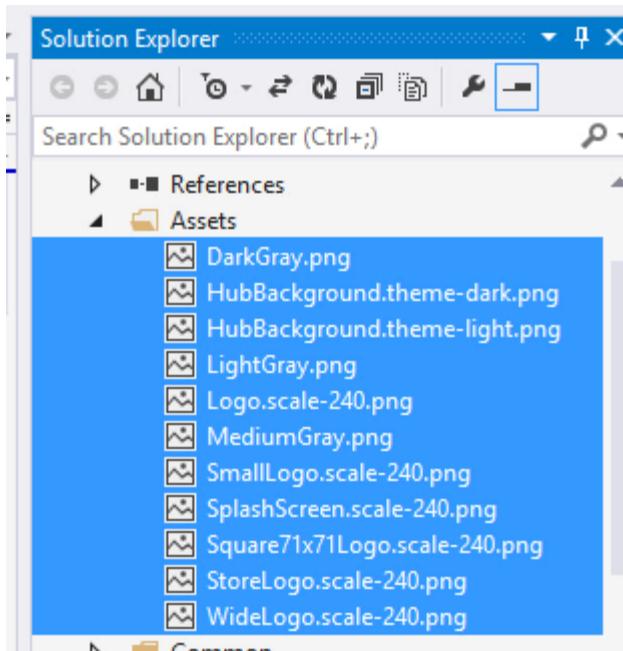
Lesson 21: Exercise: I Love Cupcakes App

In this exercise, I want to build an entire app based on the lessons we have learned about the Hub App project template. I want to show you how easy it is to transform that default Hub App project template into something completely different. In this case we will create a media application that will display images and recipes of cupcakes as well as instructional videos of how to make cupcakes.

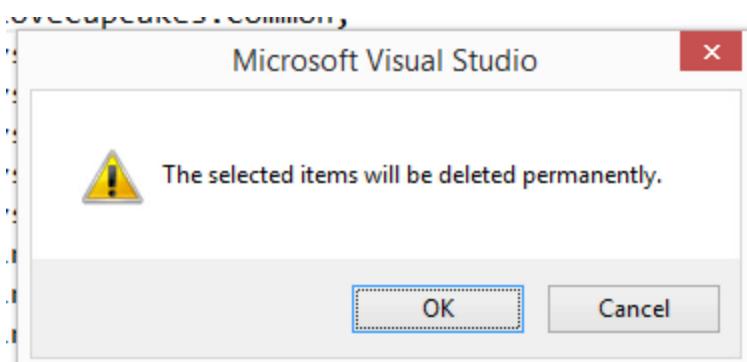
Start by creating a new Project. In the New Project dialog, (1) select the Hub App template, (2) rename to: ILoveCupcakes, (3) Click OK:



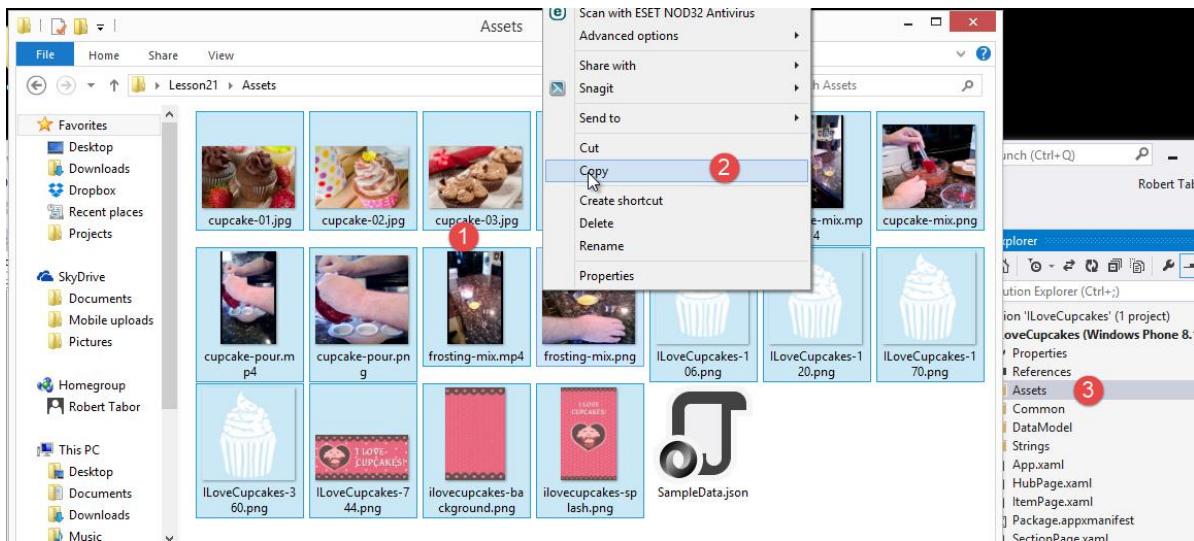
First, we'll remove all of the assets in order to replace them with our own tiles, backgrounds, etc. In the Assets folder, select all files:



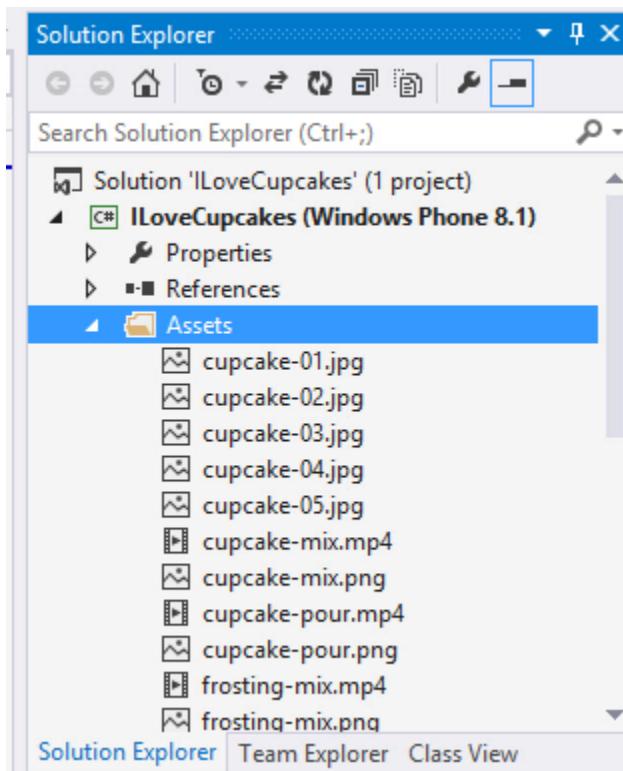
... then use the Delete keyboard key. When asked to confirm, select OK:



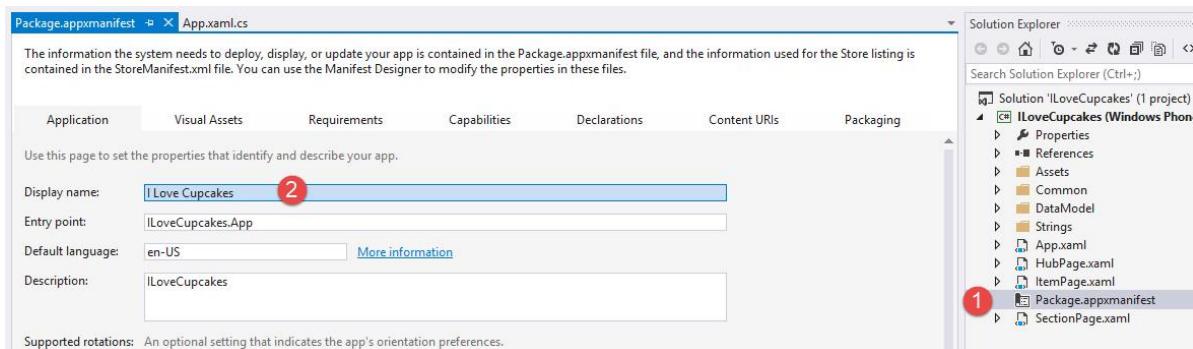
Lesson21.zip contains an Assets folder with all of the images for app tiles, hub images, background images, etc. In Windows Explorer, (1) select everything EXCEPT SampleData.json, and (2) right-click and select Copy from the context menu.



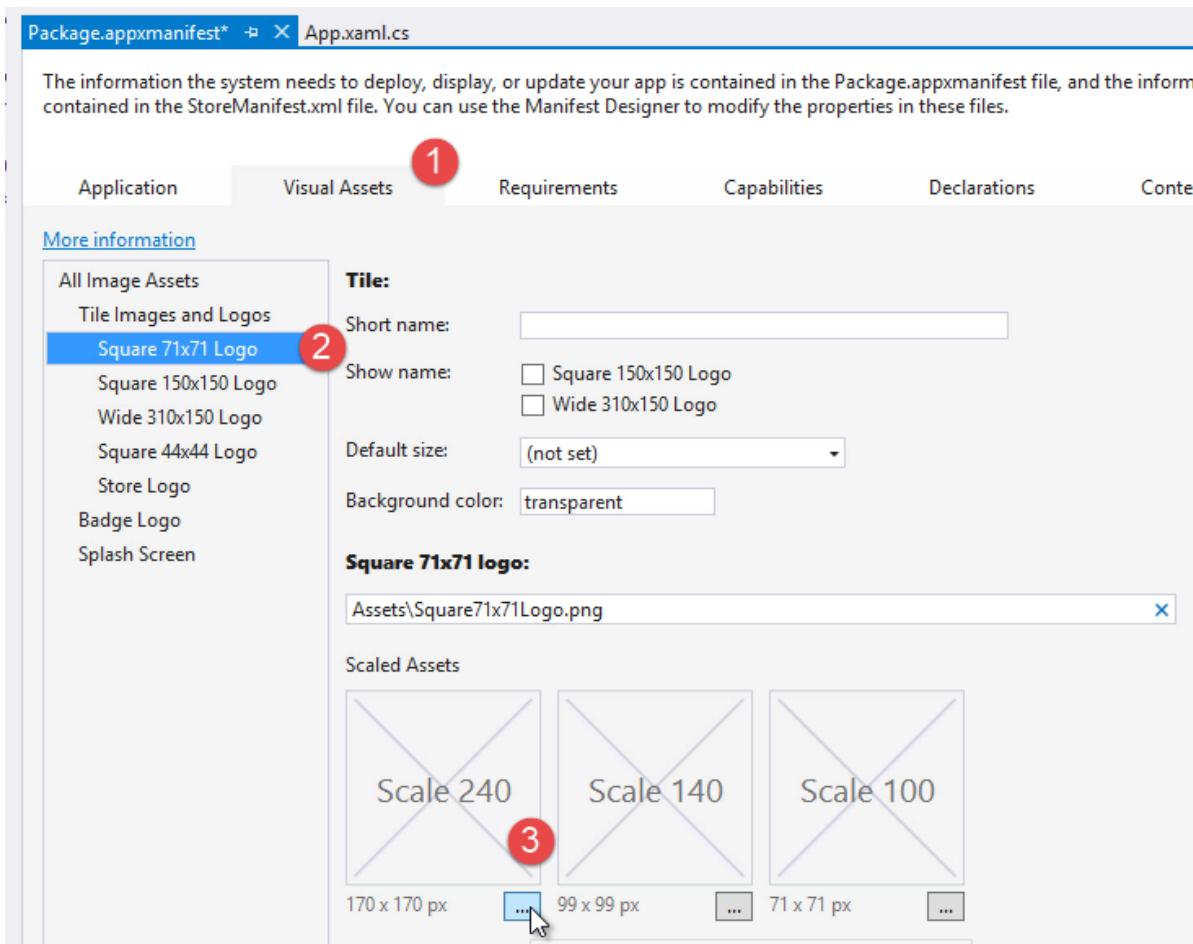
Right-click the Assets folder in the Solution Explorer and select Paste. Your file structure should look similar to this:



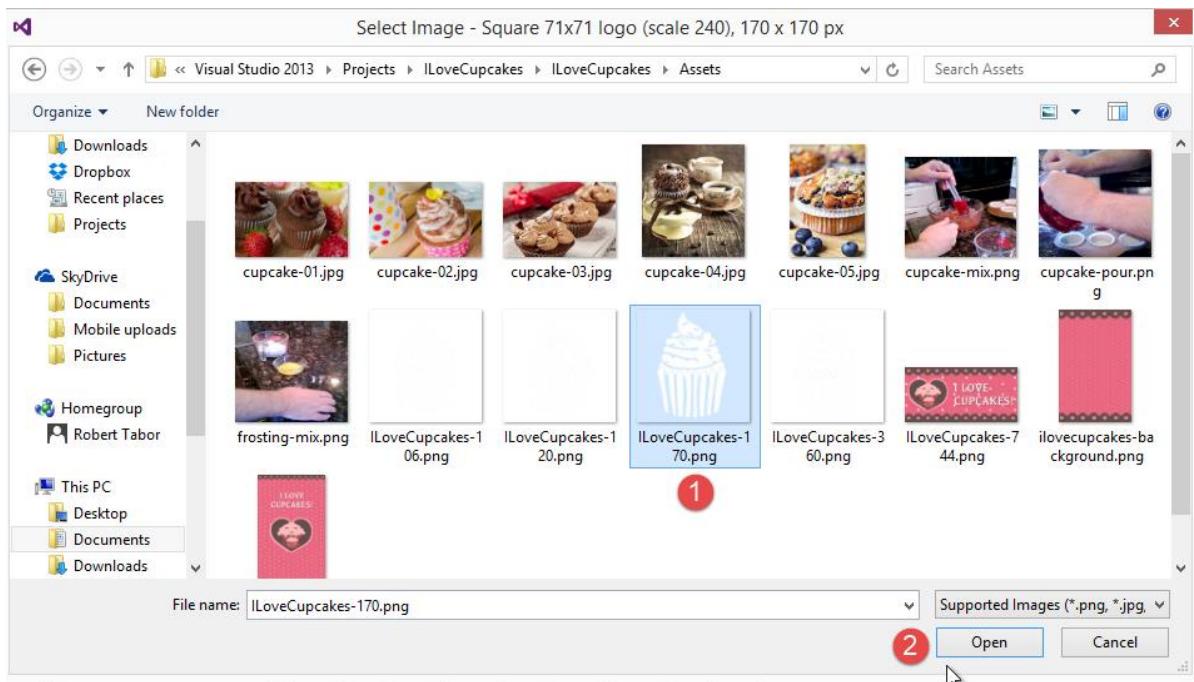
Next, (1) open the package.appxmanifest, and (2) in the Application tab, modify the Display name property to: I Love Cupcakes ...



Next, (1) select the Visual Assets tab, (2) select the Square 71x71 Logo, (3) select the ellipsis button beneath the Scaled 240 asset ...

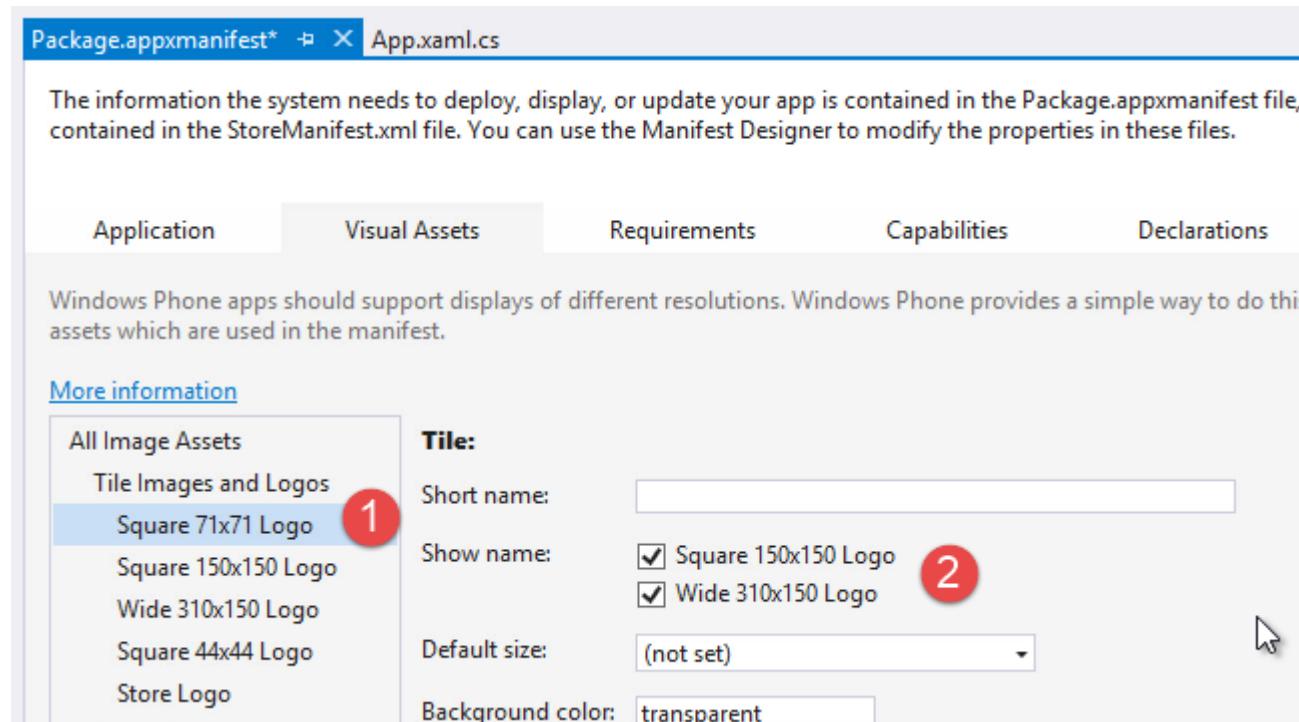


... in the Select Image dialog, (1) select the ILoveCupcakes-170.png, and (2) click Open.

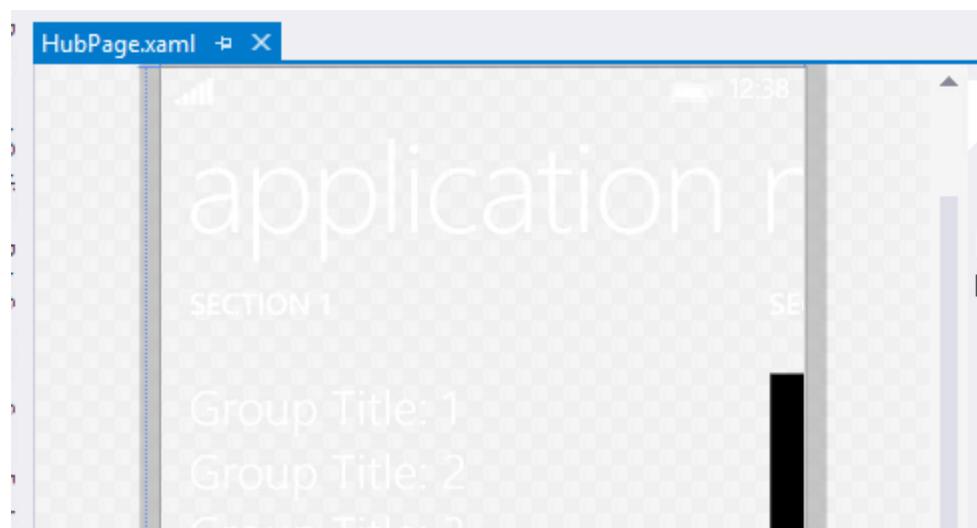


Repeat that technique to select the proper scaled image for (1) the Square 150x150 Logo, (2) the Wide 310x150 Logo, (3) the Square 44x44 Logo, (4) the Splash Screen, each time (5) selecting the appropriate image from the Assets folder.

(1) Select the Square 71x71 Logo page again, and this time (2) in the show name check boxes, place a check next to both check boxes.



If you open the HubPage.xaml, you'll see that the background image is no longer visible. In fact, the background is transparent. This is because the style used for the background is no longer referencing an image in our project. We'll fix that in a moment.



First, on the HubPage.xaml we'll want to remove the Hub Sections we no longer need. I'll comment out the definition for HubSection1 ...



... as well as HubSection4 and HubSection5:

```

<!--
<HubSection x:Uid="HubSection4" Header="SECTION 4"
             DataContext="{Binding Groups[2]}>
<DataTemplate>
    <ListView
        AutomationProperties.AutomationId="ItemListViews"
        AutomationProperties.Name="Items In Group"
        SelectionMode="None"
        IsItemClickEnabled="True"
        ItemsSource="{Binding Items}"
        ItemClick="ItemView_ItemClick"
        ContinuumNavigationTransitionInfo.ExitElementContainer="True">
        <ListView.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding Title}" Style="{StaticResource ListViewItemTextBlockStyle}"/>
                    <TextBlock Text="{Binding Subtitle}" Style="{StaticResource ListViewItemSubTitleTextBlockStyle}"/>
                </StackPanel>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</DataTemplate>
</HubSection>

<HubSection x:Uid="HubSection5" Header="SECTION 5"
             DataContext="{Binding Groups[3]}>

```

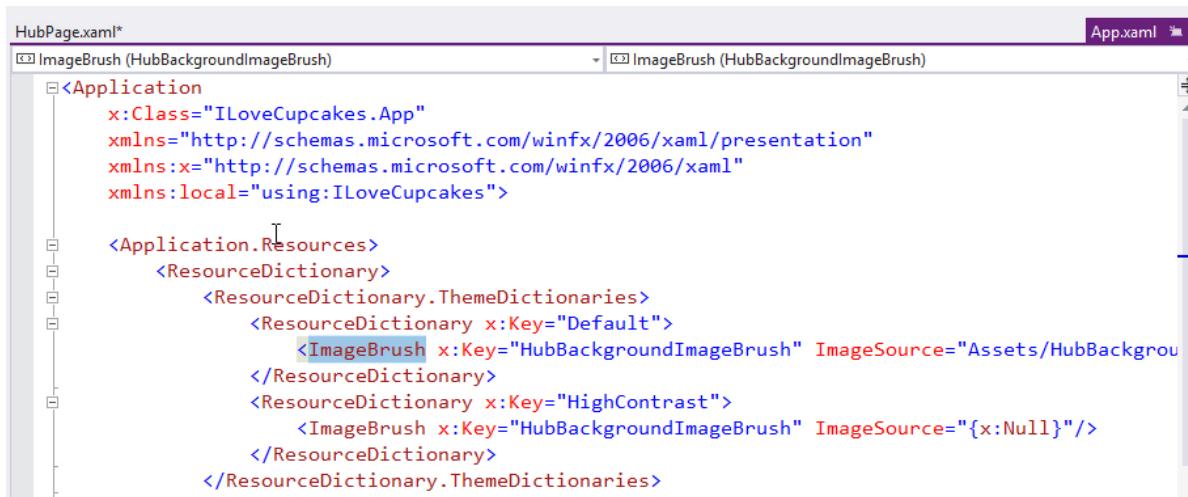
Next, I'll locate the parent Hub control. Notice that the Background property for the control is bound to a ThemeResource called HubBackgroundImageBrush. I'll place my mouse cursor inside of that identifier and select the F12 key on my keyboard:

```

<Grid x:Name="LayoutRoot">
    <Hub x:Name="Hub" x:Uid="Hub"
         Header="i love cupcakes"
         Background="{ThemeResource HubBackgroundImageBrush}">

```

... to view the definition for that ThemeResource. The ImageSource property of the ImageBrush is set to an image that no longer exists in our project:

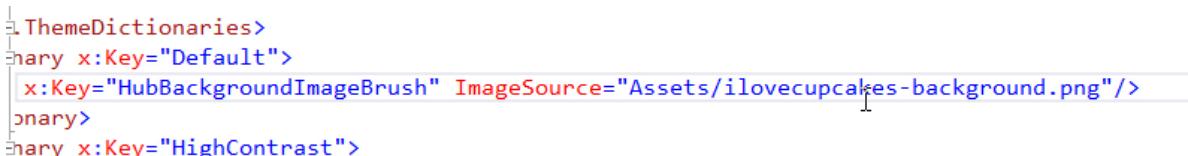


```
<Application>
    <x:Class>ILoveCupcakes.App</x:Class>
    <xmlns>http://schemas.microsoft.com/winfx/2006/xaml/presentation</xmlns>
    <xmlns:x>http://schemas.microsoft.com/winfx/2006/xaml</xmlns:x>
    <xmlns:local>using:ILoveCupcakes</xmlns:local>

    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.ThemeDictionaries>
                <ResourceDictionary x:Key="Default">
                    <ImageBrush x:Key="HubBackgroundImageBrush" ImageSource="Assets/HubBackgroundImageBrush" />
                </ResourceDictionary>
                <ResourceDictionary x:Key="HighContrast">
                    <ImageBrush x:Key="HubBackgroundImageBrush" ImageSource="{x:Null}" />
                </ResourceDictionary>
            </ResourceDictionary.ThemeDictionaries>
        </ResourceDictionary>
    </Application.Resources>

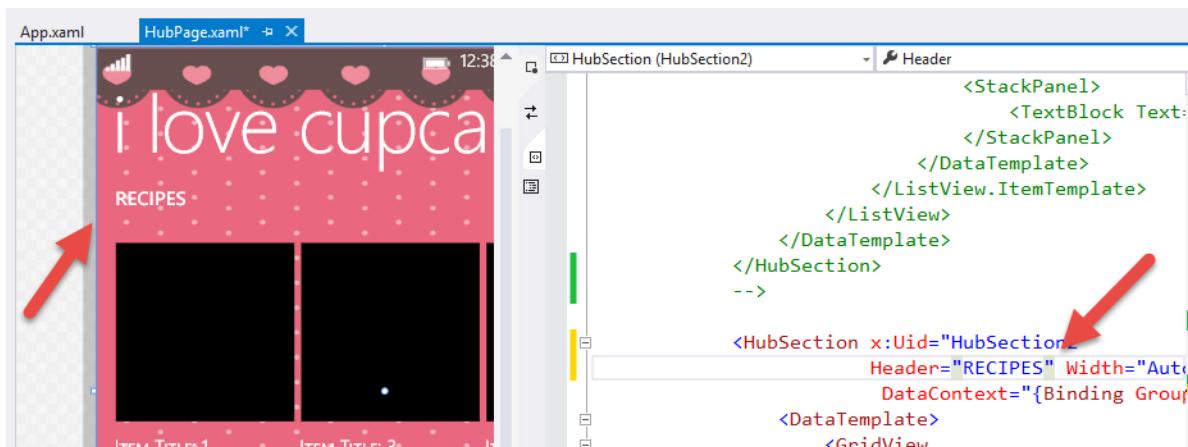
```

I'll update the reference to the new background image called: ilovecupcakes-background.png ...

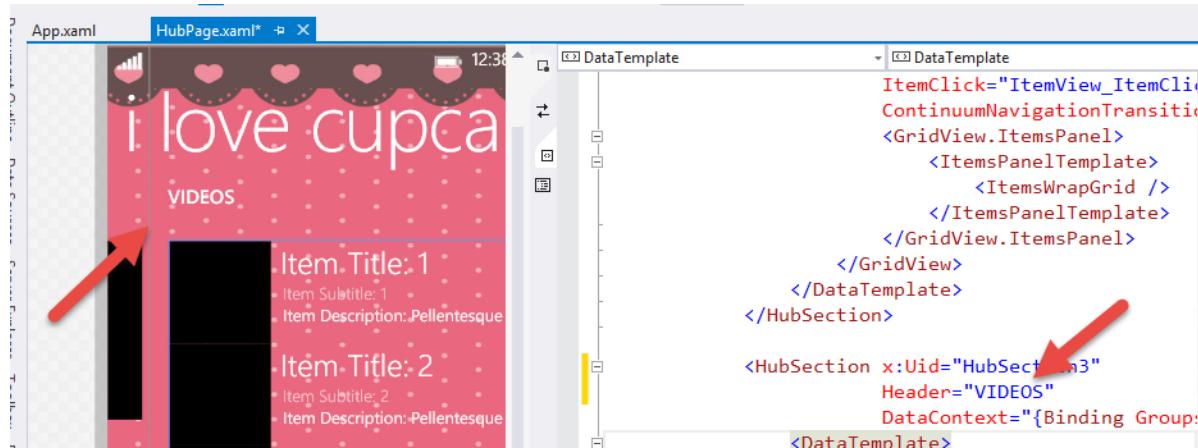


```
<ResourceDictionary x:Key="Default">
    <ImageBrush x:Key="HubBackgroundImageBrush" ImageSource="Assets/ilovecupcakes-background.png" />
</ResourceDictionary>
<ResourceDictionary x:Key="HighContrast">
```

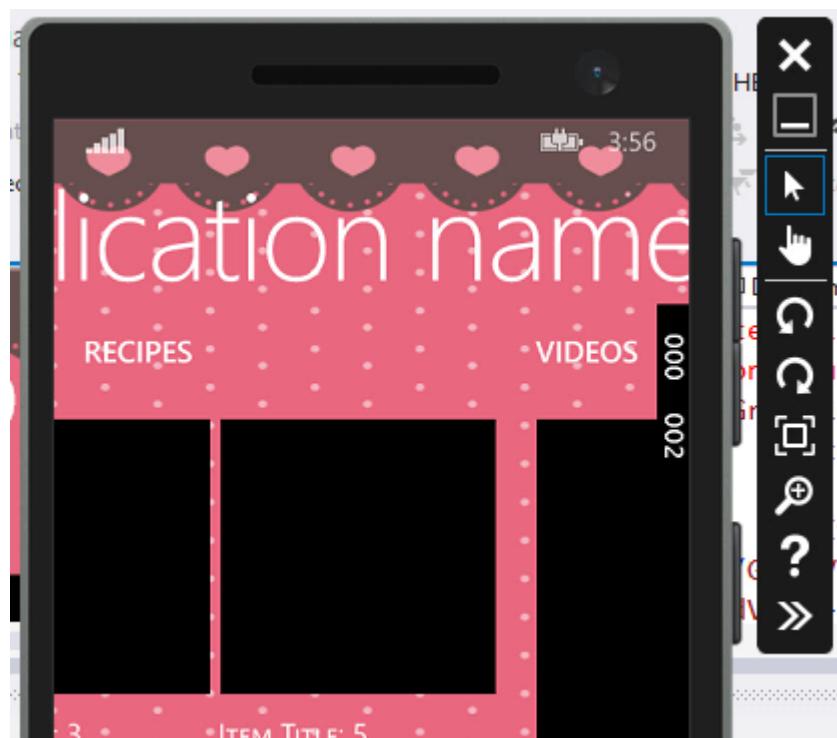
Now, when I return to the HubPage.xaml, I can see the pink background. Furthermore, I'll modify the Header property of the HubSection1 to "RECIPES" and see it reflected in the preview pane:



Likewise, I'll update HubSection3's Header to "VIDEOS" and see that change in the preview pane as well:

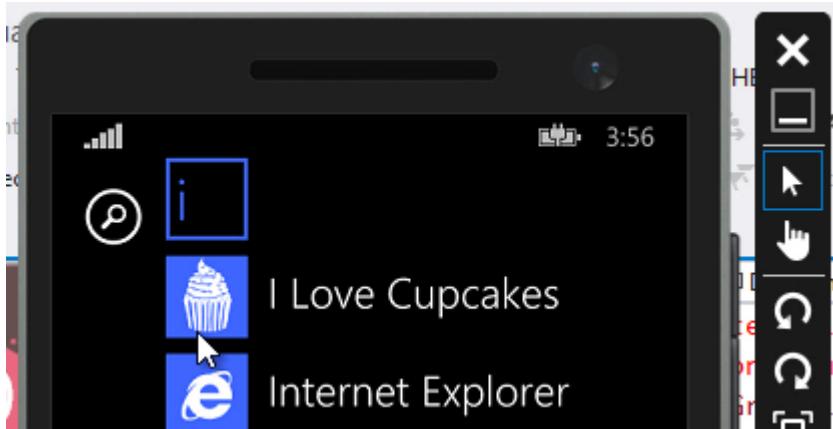


At this point, we've made significant changes to the app (although we haven't re-written much code per se). Most of the changes have been "configuration" changes up to now.

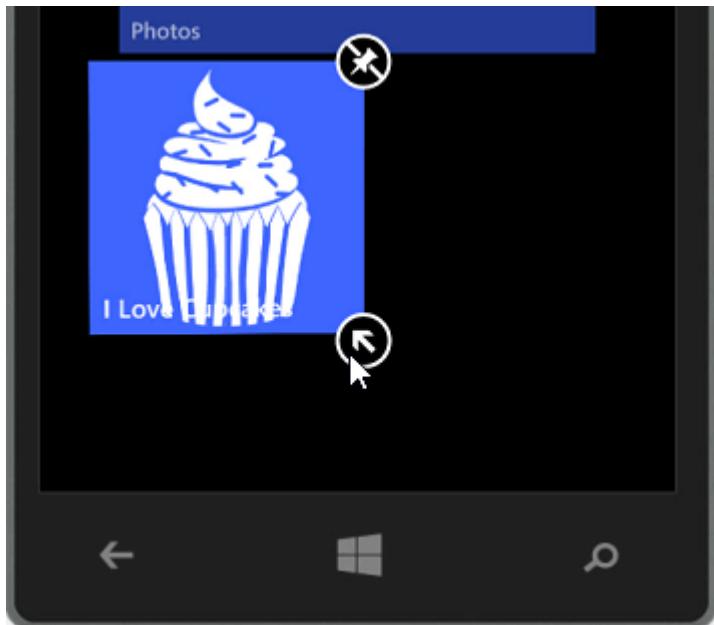


The app is styled as we would expect. However, we're missing data. We'll fix that soon.

If you select the Windows button and navigate to the Apps page, you can see the app's icon and title appears as we would expect:



Furthermore, if we pin the app to the Start page and modify the tile sizes, we can see the titles appear ...



... And the large branded tile appears as well:



Now it's time to make more significant changes to the app, specifically the data model. In the `SampleDataSource.cs`, we'll modify the `SampleDataItem` class adding a property called `Type`. We'll use the `Type` property to determine whether to navigate to a page that can display recipes and images or a page that can display video.

I'll use the `prop [tab] [tab]` code snippet to create the new `Type` property of type string:

```
SampleDataSource.cs* App.xaml HubPage.xaml
I Love Cupcakes I Love Cupcakes.Data.SampleDataItem To

{
    public SampleDataItem(String uniqueId, String title,
    {
        this.UniqueId = uniqueId;
        this.Title = title;
        this.Subtitle = subtitle;
        this.Description = description;
        this.ImagePath = imagePath;
        this.Content = content;
    }

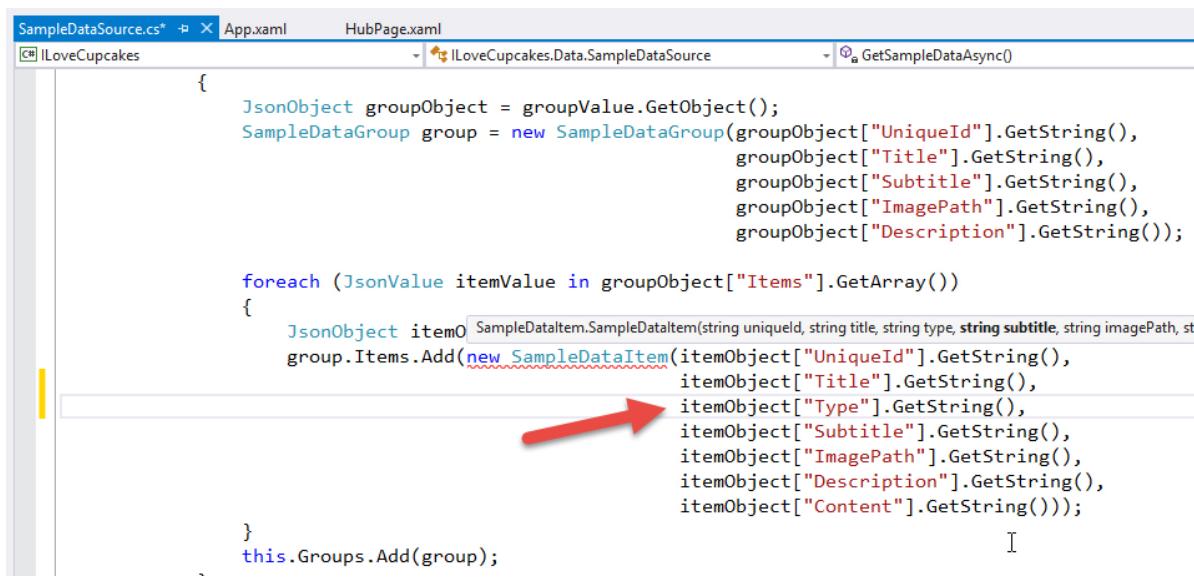
    public string UniqueId { get; private set; }
    public string Title { get; private set; }
    public string Type { get; set; }
    public string Subtitle { get; private set; }
    public string Description { get; private set; }
    public string ImagePath { get; private set; }
    public string Content { get; private set; }
}
```

I'll also change the constructor to allow the new type property to be passed in at construction, and will set the property in the same manner as the other properties are set:

```
SampleDataSource.cs* App.xaml HubPage.xaml
I Love Cupcakes I Love Cupcakes.Data.SampleDataItem To SampleDataItem(String uniqueId, String title, string type)

/// Generic item data model.
/// </summary>
public class SampleDataItem
{
    public SampleDataItem(String uniqueId, String title, string type, String subtitle, String description, String imagePath, String content)
    {
        this.UniqueId = uniqueId;
        this.Title = title;
        this.Type = type; // Red arrow here
        this.Subtitle = subtitle;
        this.Description = description;
        this.ImagePath = imagePath;
        this.Content = content;
    }
}
```

The other change comes when we're loading the JSON into memory. Therefore, we have to update the creation of new SampleDataItems in the GetSampleDataAsync() to include the new input parameter to the constructor:



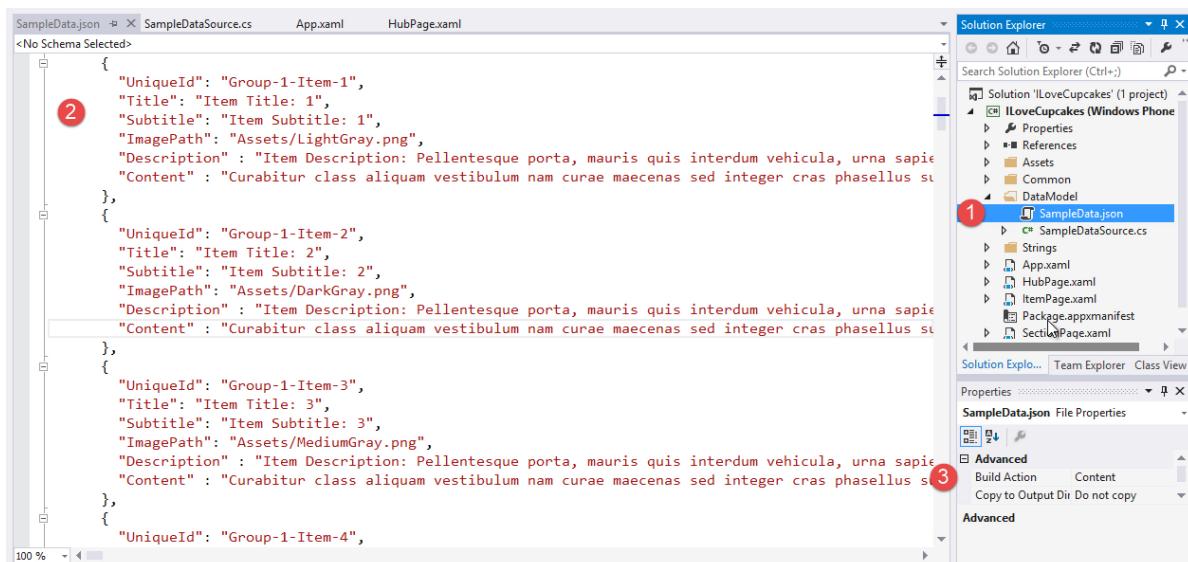
```

    JsonObject groupObject = groupValue.GetObject();
    SampleDataGroup group = new SampleDataGroup(groupObject["UniqueId"].GetString(),
                                                groupObject["Title"].GetString(),
                                                groupObject["Subtitle"].GetString(),
                                                groupObject["ImagePath"].GetString(),
                                                groupObject["Description"].GetString());

    foreach (JsonValue itemValue in groupObject["Items"].GetArray())
    {
        JsonObject itemObject = itemValue.GetObject();
        SampleDataItem item = new SampleDataItem(itemObject["UniqueId"].GetString(),
                                                itemObject["Title"].GetString(),
                                                itemObject["Type"].GetString(), // Red arrow points here
                                                itemObject["Subtitle"].GetString(),
                                                itemObject["ImagePath"].GetString(),
                                                itemObject["Description"].GetString(),
                                                itemObject["Content"].GetString());
        group.Items.Add(item);
    }
    this.Groups.Add(group);
}

```

(1) Open the the SampleData.json file. (2) Currently doesn't have a "Type" property. Furthermore, there's a lot of data to input including Group and Item titles, subtitles, image paths, etc. To keep this lesson short and reduce the possibility of fat-fingering the data in the file (causing runtime errors), I decided to create an alternative SampleData.json that we'll copy into our project. Before we delete this file, take note that (3) the Build Action property of the file is set to "Content". When I replace this file, I'll need to make sure to set that property correctly to ensure the compilation process treats our new SampleData.json correctly.



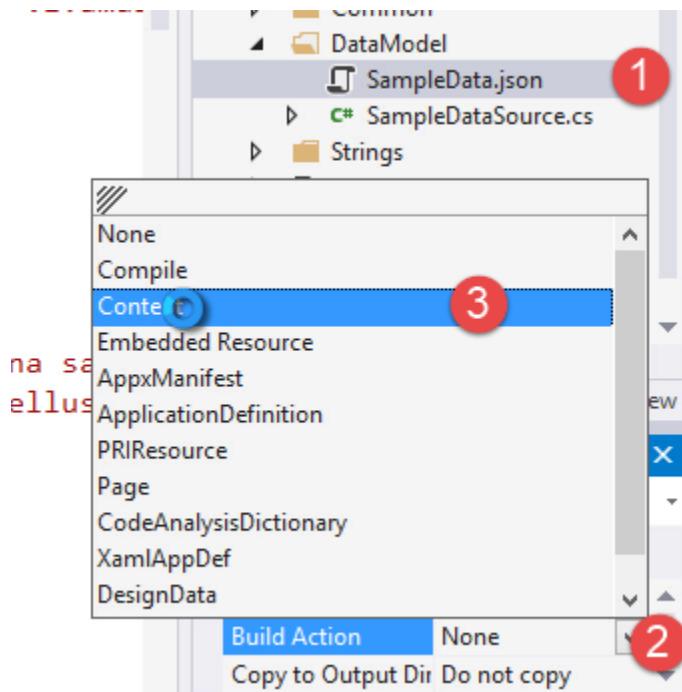
```

{
    "UniqueId": "Group-1-Item-1",
    "Title": "Item Title: 1",
    "Subtitle": "Item Subtitle: 1",
    "ImagePath": "Assets/LightGray.png",
    "Description": "Item Description: Pellentesque porta, mauris quis interdum vehicula, urna sapien",
    "Content": "Curabitur class aliquam vestibulum nam curae maecenas sed integer cras phasellus si"
},
{
    "UniqueId": "Group-1-Item-2",
    "Title": "Item Title: 2",
    "Subtitle": "Item Subtitle: 2",
    "ImagePath": "Assets/DarkGray.png",
    "Description": "Item Description: Pellentesque porta, mauris quis interdum vehicula, urna sapien",
    "Content": "Curabitur class aliquam vestibulum nam curae maecenas sed integer cras phasellus si"
},
{
    "UniqueId": "Group-1-Item-3",
    "Title": "Item Title: 3",
    "Subtitle": "Item Subtitle: 3",
    "ImagePath": "Assets/MediumGray.png",
    "Description": "Item Description: Pellentesque porta, mauris quis interdum vehicula, urna sapien",
    "Content": "Curabitur class aliquam vestibulum nam curae maecenas sed integer cras phasellus si"
},
{
    "UniqueId": "Group-1-Item-4",
}

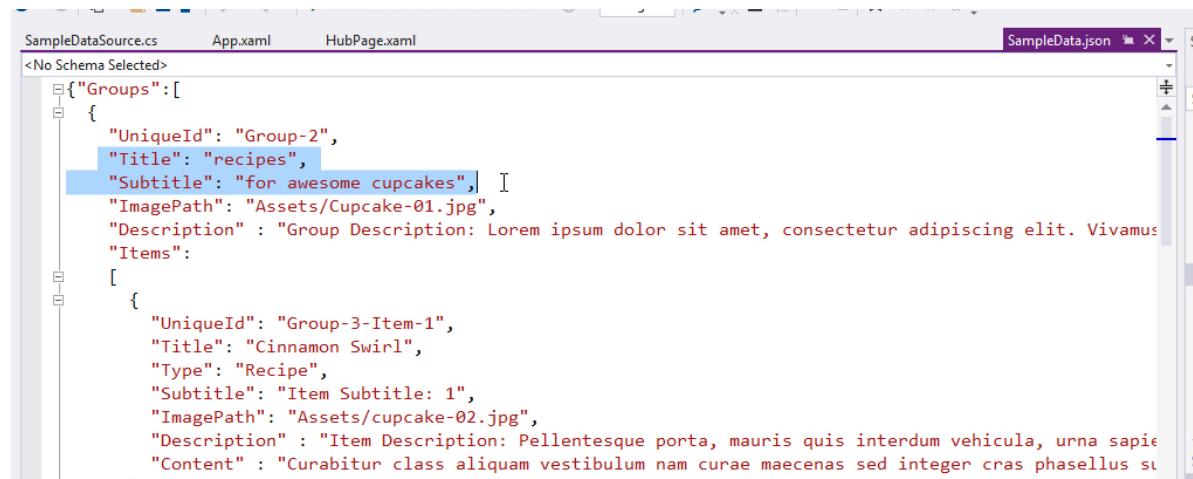
```

First, I'll delete the current SampleData.json. In Windows Explorer, in Lesson21.zip's Assets folder, drag and drop the new SampleData.json into the DataModel folder of the project in the Solution Explorer.

(1) Select the new SampleData.json file in the Solution Explorer, then (2) in the Build Action, change the value from None (the default) to (3) Content.



If you open the new SampleData.json, you can see the shape of the data has changed to reflect our new Type property and the data also is specific to our app.

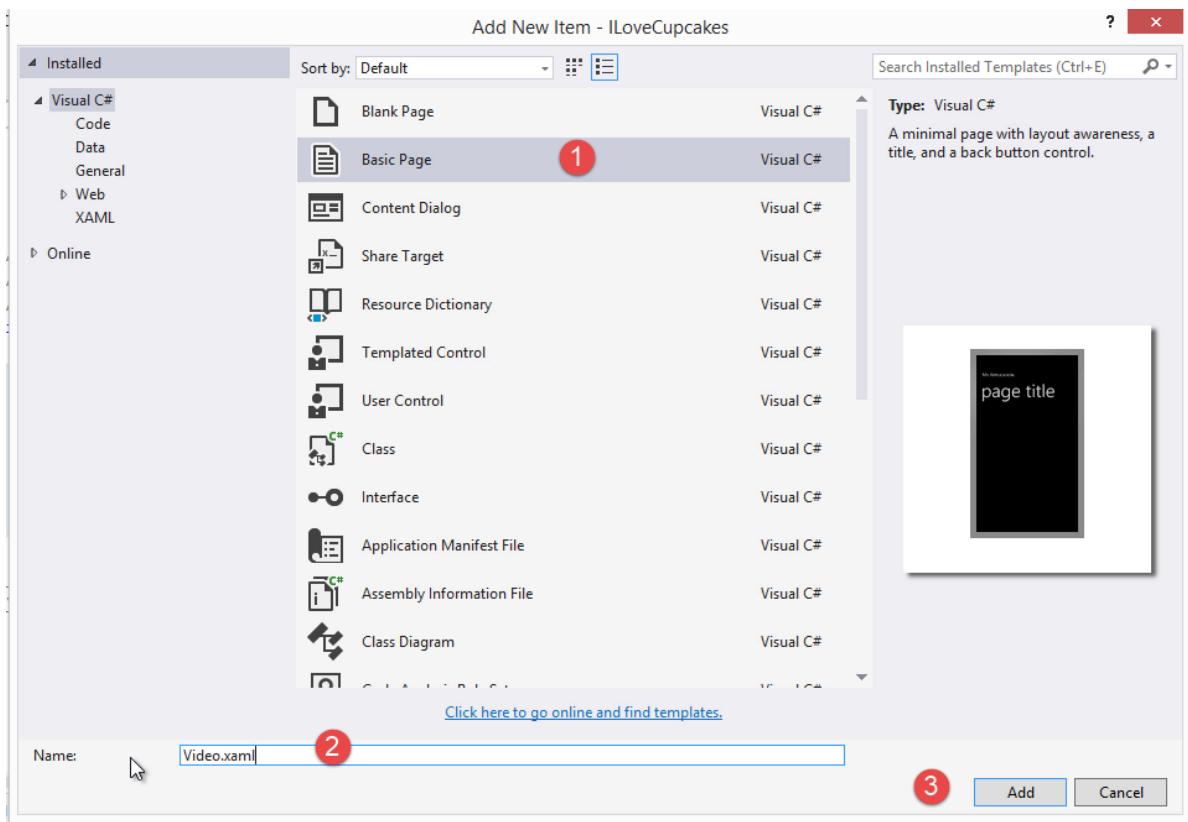


If we re-run the app, we should now see images and cupcake-specific titles for the tiles in the Hub control.



The next step is to set up the navigation from these tiles in the Hub control to pages that can display the pictures and recipe data or the “How To” videos for making cupcakes. For the former requirement, I’ll simply re-use the ItemPage.xaml. For the latter, I’ll add a new video.xaml page.

Right-click the project name in the Solution Explorer, select Add > New Item ... from the context menu. In the Add New Item dialog, (1) select Basic Page, (2) change the name to Video.xaml, (3) click the Add button:



In the new Video.xaml page, I'll add a MediaElement control telling it to stretch to the dimensions of the parent grid, and setting the AutoPlay property to true, play as soon as the media source is determined:



In the Video.xaml.cs, we need to determine which video to load into the MediaElement. To do this, we'll need to accept a parameter from the HubPage.xaml. We'll use the UniqueID of the SampleDataItem to determine which video should be displayed. The UniqueID should be the same as the name of the video, sans the file extension ".mp4". Therefore, when a user taps a thumbnail image on HubSection3, we'll determine the UniqueID associated with that item, and pass that UniqueID as a navigation parameter to the Video.xaml page. In the NavigationHelper_LoadState() method, we'll retrieve the SampleDataItem associated with the UniqueID (via the LoadStateEventArgs.NavigationParameter), we'll construct a URI to the video file in the Assets folder by appending the file extension ".mp4" to the UniqueID. Finally, we'll set the MediaElement's Source property to the newly constructed URI.

```

Video.xaml.cs*  X  Video.xaml      SampleDataSource.cs      App.xaml      HubPage.xaml      HubPage.xa
C#| ILoveCupcakes      I LoveCupcakes.Video      NavigationHelper_LoadState(object sender
{
    /// provided when recreating a page from a prior session.
    /// </summary>
    /// <param name="sender">
    /// The source of the event; typically <see cref="NavigationHelper"/>
    /// </param>
    /// <param name="e">Event data that provides both the navigation parameter passed to
    /// <see cref="Frame.Navigate(Type, Object)"> when this page was initially requested and
    /// a dictionary of state preserved by this page during an earlier
    /// session. The state will be null the first time a page is visited.</param>
    private void NavigationHelper_LoadState(object sender, LoadStateEventArgs e)
    {
        var item = await SampleDataSource.GetItemAsync((String)e.NavigationParameter);
        this.DefaultViewModel["Item"] = item;

        var uri = "ms-appx:///Assets/" + item.UniqueId + ".mp4";
        MyMediaElement.Source = new Uri(uri, UriKind.Absolute);
        PageTitle.Text = item.Title;
    }
}

```

There are a few problems with this code that need to be resolved. First, we need to add a using statement for the ILoveCupcakes.Data namespace. Second, we'll need to add the async keyword to the method signature to accommodate the await keyword when calling SampleDataSource.GetItemAsync.

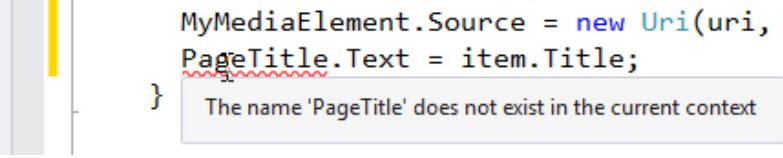
```

    /// session. The state will be null the first time a page is visited.</param>
private async void NavigationHelper_LoadState(object sender, LoadStateEventArgs e)
{
    var item = await SampleDataSource.GetItemAsync((String)e.NavigationParameter);
    this.DefaultViewModel["Item"] = item;

    var uri = "ms-appx:///Assets/" + item.UniqueId + ".mp4";
    MyMediaElement.Source = new Uri(uri, UriKind.Absolute);
}

```

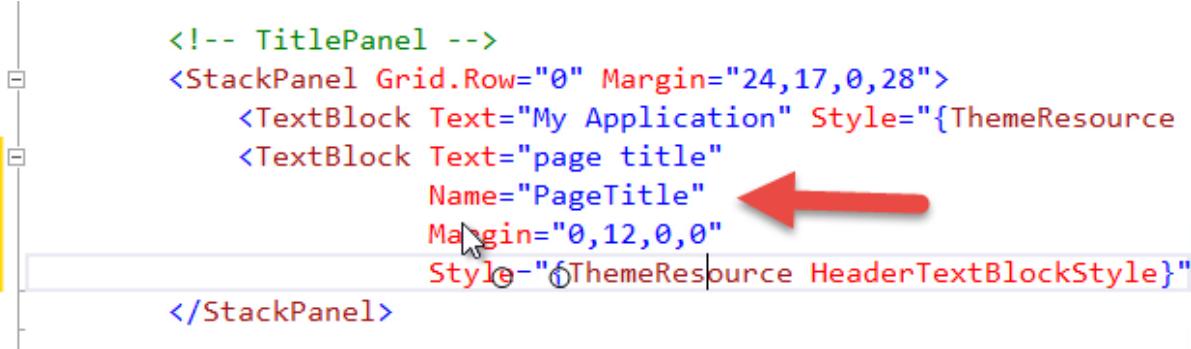
Also, if we want to set the title of the page, we'll need to give the TextBlock that contains the page title a name.



```
    MyMediaElement.Source = new Uri(uri, UriKind.Absolute);
    PageTitle.Text = item.Title;
}
```

The name 'PageTitle' does not exist in the current context

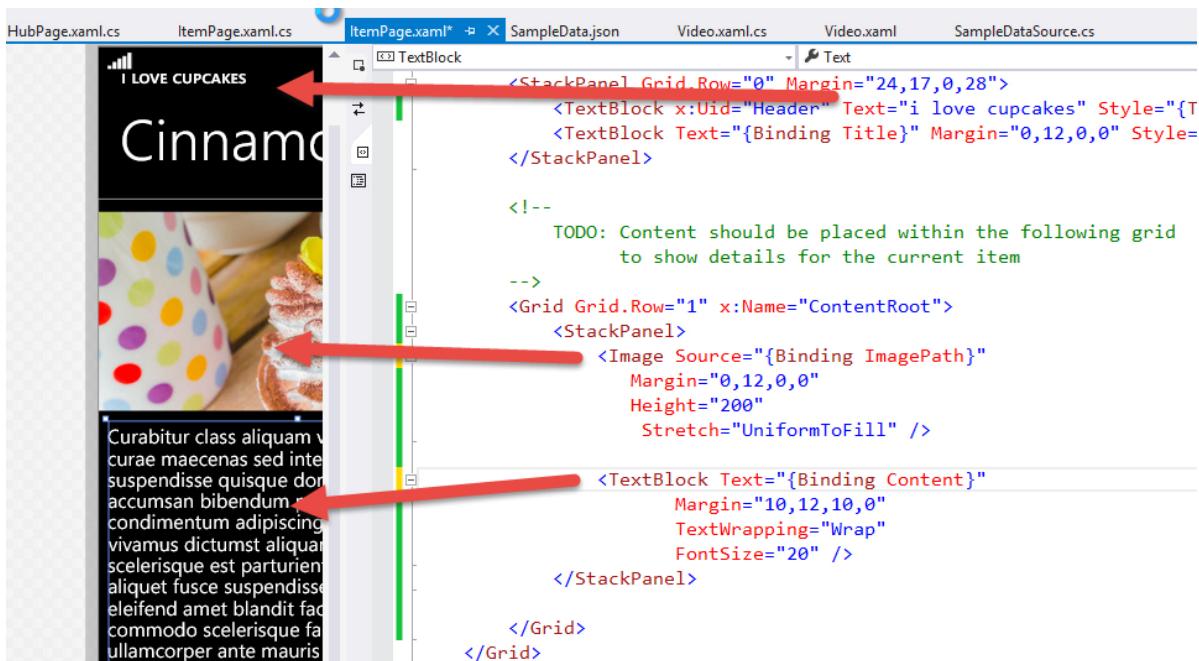
Therefore, back in Video.xaml, I'll modify the page title in the "TitlePanel" by giving it a programmatic name "PageTitle" so that our C# code that sets it will work:



```
<!-- TitlePanel -->
<StackPanel Grid.Row="0" Margin="24,17,0,28">
    <TextBlock Text="My Application" Style="{ThemeResource
    <TextBlock Text="page title"
        Name="PageTitle" ←
        Margin="0,12,0,0"
        Style={ThemeResource HeaderTextBlockStyle}">
    </TextBlock>
</StackPanel>
```

That concludes the changes required to the Video.xaml.

Next, we'll need to modify the ItemPage.xaml which will display the app's title, and we'll add a StackPanel in the Grid named "ContentRoot" to display an Image and a TextBlock. After adding the XAML (below) you can see the changes to the ItemPage.xaml's preview:



Since the ItemPage.xaml was already “wired up” by the project template, no further changes will be necessary.

The last significant change will be in the logic that determines whether to send the user to the ItemPage.xaml or the Video.xaml page based on the type of the item in the Hub’s ItemView controls that the user clicked on. Note that both ItemView controls (defined in HubSection2 and HubSection3) use the same event handler method, ItemView_ItemClick. Therefore, we’ll retrieve the ItemClickEventArgs.ClickedItem input parameter which lets us know which item was clicked and cast it to SampleDataItem. Once we have the item, we can determine both the Type and UniqueID properties for the selected item.

If the Type is “Recipe”, then we’ll send the user to the ItemPage. If the Type is “Video”, we’ll send them to the Video.xaml page. In both cases, we’ll send the UniqueID so that the proper SampleDataItem is displayed to the user.

```

    /// <summary>
    /// Shows the details of an item clicked on in the <see cref="ItemPage"/>
    /// </summary>
    private void ItemView_ItemClick(object sender, ItemClickEventArgs e)
    {
        // Navigate to the appropriate destination page, configuring the new page
        // by passing required information as a navigation parameter

        var type = ((SampleDataItem)e.ClickedItem).Type;
        var itemId = ((SampleDataItem)e.ClickedItem).UniqueId;

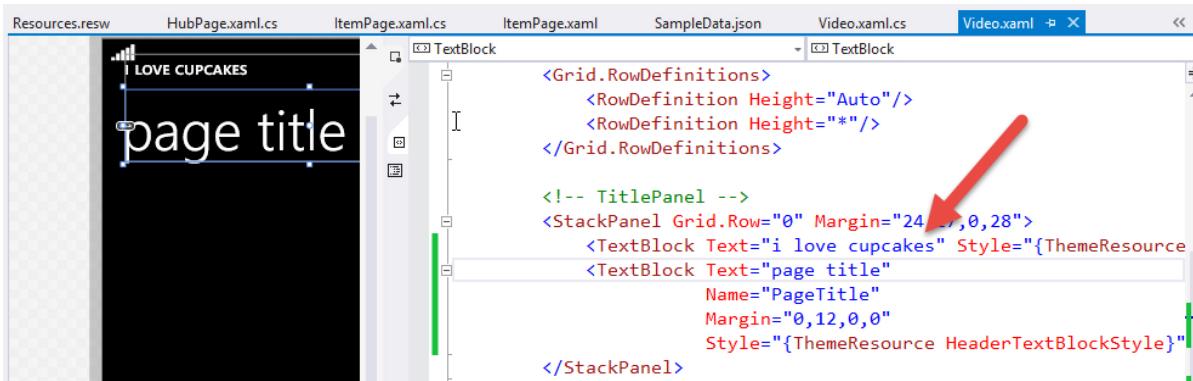
        if (type == "Recipe")
        {
            this.Frame.Navigate(typeof(ItemPage), itemId);
        }
        else
        {
            this.Frame.Navigate(typeof(Video), itemId);
        }
    }
}

```

One final issue remains. While we attempted to change the app name in several places in our XAML, it seems to ignore those changes. The answer is that these settings were localized. Localization is a topic we don't cover in this series, however it allows you to create multi-language versions of your app based on the user's language and region preferences. In this case, we'll open the Strings\en-US\Resources.resw file in the Solution Explorer and modify the top two properties setting their values to "i love cupcakes".

Name	Value	Comment
Header.Text	i love cupcakes	
Hub.Header	i love cupcakes	
HubSection1	SECTION 1	
HubSection2	SECTION 2	
HubSection3	SECTION 3	
HubSection4	SECTION 4	
HubSection5	SECTION 5	
NavigationFailedExceptionMessage	Navigation failed.	

Furthermore, in the Video.xaml page, I'll make sure to update the title of the app (since I forgot to do it earlier):



Now, when I run the app, it looks complete and the app's name is consistent throughout.

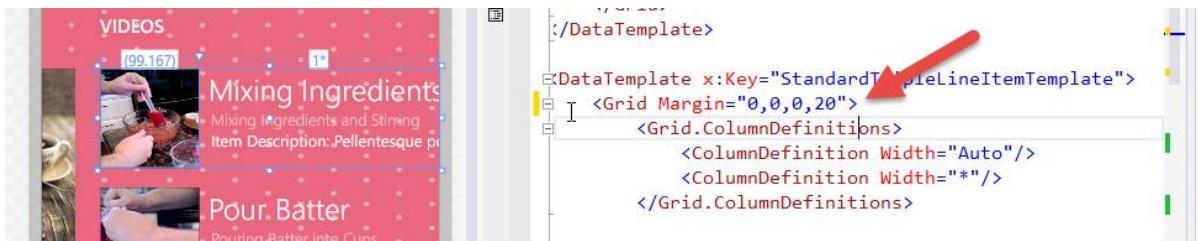


I see one last tweak to the default layout of the ItemTemplate for my videos ... there's no space between each of the video thumbnail images.

To fix this, I'll return to HubSection3 and locate the ListView's ItemTemplate property. It is set to a page resource called StandardTripleLineItemTemplate. I'll put my mouse cursor on that name and press the F12 key on my keyboard to find its definition near the top of the page.



In the DataTemplate called StandardTripleLineItemTemplate, I'll add a Margin="0,0,0,20" to put 20 pixels of space below each item.



There are many other small tweaks we could make to the styling and layout, but we'll consider this exercise complete.

Lesson 22: Storing and Retrieving Serialized Data

I want to continue to build on the knowledge we've already gained from building the I Love Cupcakes app. I want to focus on some slightly more advanced ideas so that we can create an even more advanced app.

In this lesson we'll discuss the technique required to save a text file to the Phone's storage, then open and read that file from the Phone's storage. There are probably several ways to write and read data from a phone. For example, if we were working with binary data — like an image file — we would go about this differently. However, for our purposes, we want to merely save application data.

In our case, we will have an object graph that we want to save to the Phone's storage. This will be input the user created when working with our app. If it were a contact management app, it would store the names, addresses, phone numbers and email addresses for our friends and acquaintances. We will want to store that data on the phone so that we can retrieve it the next time the user wants to look up a contact. So, we need to figure out how to take an object graph and persist it to storage.

To do this, we will need to serialize the data. Serialization is the process of representing an object graph in memory into a stream that can be written to disk using some format that keeps each instance of an object in the object graph distinct. It keeps all of its state — its property values — as well as its relationship to other objects in the object graph intact.

The content of the stream is based on which class in the Phone API you choose to perform the serialization. In this lesson I'll look at the `DataContractSerializer` which will convert our object graph into XML, and I'll look at the `DataContractJsonSerializer` which will convert our object graph into Json. These classes also can `DE-SERIALIZE`, which performs the opposite task of taking a stream and turning it back into an object graph for use in our app.

Once we have a serialized stream of objects, we can persist it to storage. On the Phone, each app has a designated area where data for the app should be written and read from called the Local Folder. The local folder is the root folder of your app's data store. Use this folder to persist data on the phone. In addition to general data storage, there are sub folders that should be used for specific scenarios.

[http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402541\(v=vs.105\).aspx](http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402541(v=vs.105).aspx)

I found this Quickstart very helpful when I was first trying to understand how the Phone stores application data and other media.

The way you access the Local Folder is through the:

`Windows.Storage.ApplicationData.Current.LocalFolder` object

... like so:

```
ApplicationData.Current.LocalFolder.OpenStreamForReadAsync(fileName);
```

... and ...

```
ApplicationData.Current.LocalFolder.OpenStreamForWriteAsync(fileName,  
CreationCollisionOption.ReplaceExisting)
```

These will open a stream that will read contents from the file or write the contents of a stream to a file, respectively. As you can see, when opening a stream for writing, you have some options regarding file name collisions. `CreationCollisionOption` enum has the following values:

`OpenIfExists` - Instead of writing, open the file into the current stream.

`FailIfExists` - Throw an exception if a file by the same name already exists.

`GenerateUniqueName` - Generate a new name for the file if a file by the same name already exists.

`ReplaceExisting` - Overwrite the existing file with the same name no matter what.

Ok, so there are two basic components. You need a stream to read a file into, or a stream with data to write to a file. The stream is the in memory representation of the file.

Then you have the Phone API methods that operate on the physical storage of the Phone.

But there's also one final (optional) part ... the Phone API classes and methods that perform serialization and deserialization. They write the object graph to the stream using a particular serialization scheme - XML, Json, Binary, etc. AND they perform the corollary operation of deserializing the stream from XML, Json, Binary, etc. BACK into an object graph.

To demonstrate these pieces, we'll create a simple example using the Blank App Template called "StoringRetrievingSerializing".

We'll start by building the User Interface:

```
<StackPanel Margin="0,50,0,0">
    <Button x:Name="writeButton"
        Content="Write"
        Click="writeButton_Click"
    />

    <Button x:Name="readButton"
        Content="Read"
        Click="readButton_Click" />

    <TextBlock x:Name="resultTextBlock"
        FontSize="24"
        Height="400"
        TextWrapping="Wrap"
    />

</StackPanel>
```

Next, I'll create an object graph. I'll define a class (Car, for simplicity's sake) and a helper method that will create instances of the Car class and save them in a List<Car>:

```
public class Car
{
    public int Id { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
}

... and ...

private List<Car> buildObjectGraph()
{
    var myCars = new List<Car>();

    myCars.Add(new Car() { Id = 1, Make = "Oldsmobile", Model = "Cutlas Supreme", Year = 1985 });
    myCars.Add(new Car() { Id = 2, Make = "Geo", Model = "Prism", Year = 1993 });
    myCars.Add(new Car() { Id = 3, Make = "Ford", Model = "Escape", Year = 2005 });

    return myCars;
}
```

In the first example, the Write button will serialize the object graph to XML, then save it as a file to the Local Folder. The Read button will read the file from the Local Folder and simply display the contents to screen. We will NOT deserialize the XML back to an object graph in this example because I want you to see the XML that was created. Therefore, we'll simply use a StreamReader class to read the content from the stream into a String that we can then display in the resultTextBlock element.

```
private const string XMLFILENAME = "data.xml";

private async Task writeXMLAsync()
{
    var myCars = buildObjectGraph();

    var serializer = new DataContractSerializer(typeof(List<Car>));
    using (var stream = await ApplicationData.Current.LocalFolder.OpenStreamForWriteAsync(
        XMLFILENAME,
        CreationCollisionOption.ReplaceExisting))
    {
        serializer.WriteObject(stream, myCars);
    }

    resultTextBlock.Text = "Write succeeded";
}

private async Task readXMLAsync()
{
    string content = String.Empty;

    var myStream = await
ApplicationData.Current.LocalFolder.OpenStreamForReadAsync(XMLFILENAME);
    using (StreamReader reader = new StreamReader(myStream))
    {
        content = await reader.ReadToEndAsync();
    }

    resultTextBlock.Text = content;
}
```

... Now, we'll call these two helper methods from the appropriate button click event handler methods:

```

private async void readButton_Click(object sender, RoutedEventArgs e)
{
    await readXMLAsync();
}

private async void writeButton_Click(object sender, RoutedEventArgs e)
{
    await writeXMLAsync();
}

```

Run the app, click the Write button, then the Read button to see the results:



In this series of lessons we'll use Json as our serialization format of choice. I suppose there's nothing wrong with XML other than it being verbose. It has gotten a bad rap through the years as being difficult to work with. There's a joke: "I had a problem, and to solve it I used XML. Now I have two problems."

Json seems to be the preferred data format de jour. It's less verbose, and it is the native format for JavaScript objects, and JavaScript is all the rage right now. But other than that, I see no technical reason to prefer one over the other. Nonetheless, we'll be using Json from here on out.

Let's create two new helper methods that work with Json instead of XML. As you'll see, the code is almost identical:

```
private const string JSONFILENAME = "data.json";  
  
private async Task writeJsonAsync()  
{  
    // Notice that the write is ALMOST identical ... except for the serializer.  
  
    var myCars = buildObjectGraph();  
  
    var serializer = new DataContractJsonSerializer(typeof(List<Car>));  
    using (var stream = await ApplicationData.Current.LocalFolder.OpenStreamForWriteAsync(  
        JSONFILENAME,  
        CreationCollisionOption.ReplaceExisting))  
    {  
        serializer.WriteObject(stream, myCars);  
    }  
  
    resultTextBlock.Text = "Write succeeded";  
}
```

```
private async Task readJsonAsync()  
{  
    // Notice that the write **IS** identical ... except for the serializer.  
  
    string content = String.Empty;  
  
    var myStream = await  
ApplicationData.Current.LocalFolder.OpenStreamForReadAsync(JSONFILENAME);  
    using (StreamReader reader = new StreamReader(myStream))  
    {  
        content = await reader.ReadToEndAsync();  
    }  
  
    resultTextBlock.Text = content;  
}
```

... And I'll modify the button click events ...

```
private async void readButton_Click(object sender, RoutedEventArgs e)  
{
```

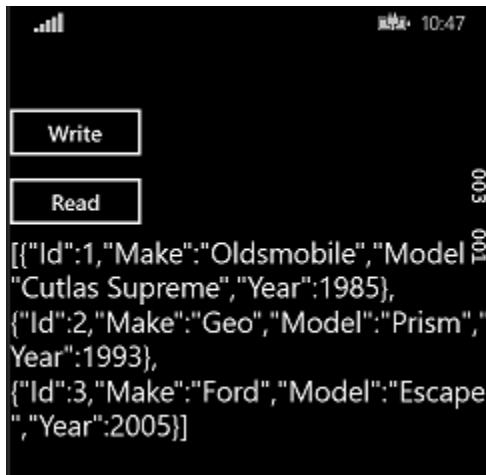
```

//await readXMLAsync();
await readJsonAsync();
}

private async void writeButton_Click(object sender, RoutedEventArgs e)
{
    //await writeXMLAsync();
    await writeJsonAsync();
}

```

When I run the app ...



But ultimately we do want to work with an object graph, not just Json. So, I'll create one final helper method:

```

private async Task deserializeJsonAsync()
{
    string content = String.Empty;

    List<Car> myCars;
    var jsonSerializer = newDataContractJsonSerializer(typeof(List<Car>));

    var myStream = await
    ApplicationData.Current.LocalFolder.OpenStreamForReadAsync(JSONFILENAME);

    myCars = (List<Car>)jsonSerializer.ReadObject(myStream);

    foreach (var car in myCars)
    {

```

```

        content += String.Format("ID: {0}, Make: {1}, Model: {2} ... ", car.Id, car.Make, car.Model);
    }

    resultTextBlock.Text = content;
}

```

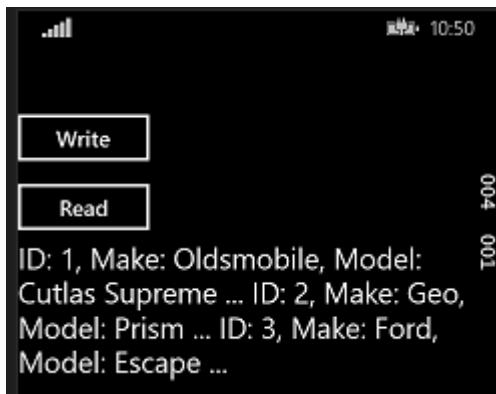
And change the Read button's event handler method to call it:

```

private async void readButton_Click(object sender, RoutedEventArgs e)
{
    //await readXMLAsync();
    //await readJsonAsync();
    await deserializeJsonAsync();
}

```

... and then run the app ...



Recap

The biggest challenge is understanding the role each of the pieces play. If you don't fundamentally understand that there are four parts:

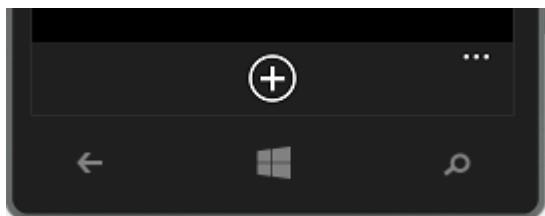
- The object graph
- The serialization classes / methods
- The stream
- The file in the Local Folder

... then it will be a mystery when you ask "What do I need to do next?" Hopefully you can remember at a high level the notion of serialization because it plays a big role whenever transferring data between layers or formats. A good example is serializing and deserializing data for transfer via web services.

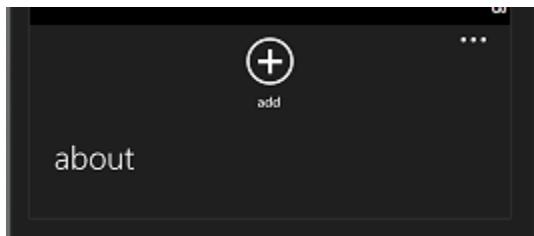
Lesson 23: Working with the Command Bar

In this short lesson we'll learn about the CommandBar, another user interface element we can use on the Phone to provide more options in the form of icons and menu options to the user than what they see displayed by default.

Here's an example of a CommandBar with a single PrimaryCommand AppBarButton:



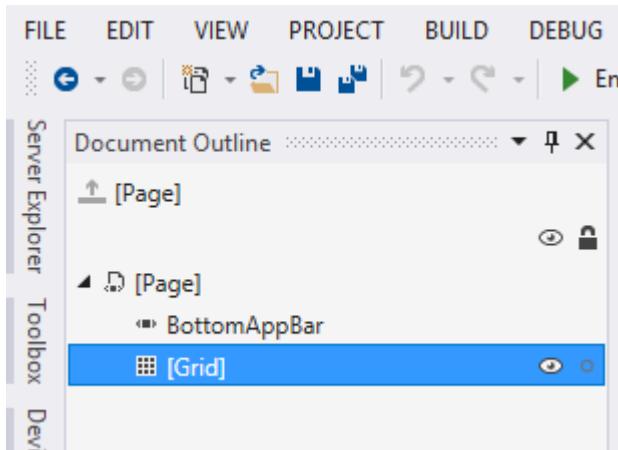
Clicking the ellipsis on the right will reveal the label for the PrimaryCommand AppBarButton and reveals a SecondaryCommand AppBarButton for an “about” feature:



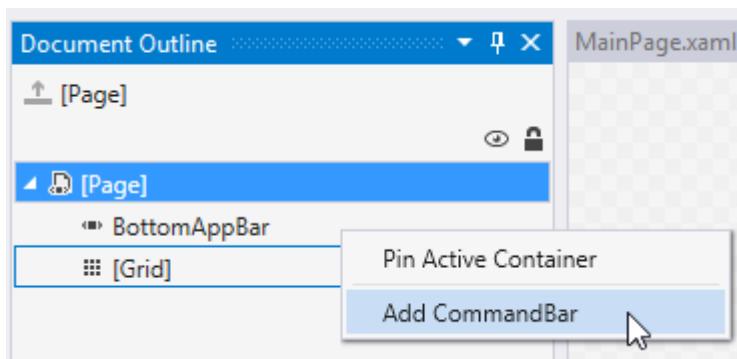
How do you add these? Obviously, with XAML. However, Visual Studio does have a tool that makes quick work of this.

Create a new Blank App project called CommandBarExample. Open the MainPage.xaml into the designer.

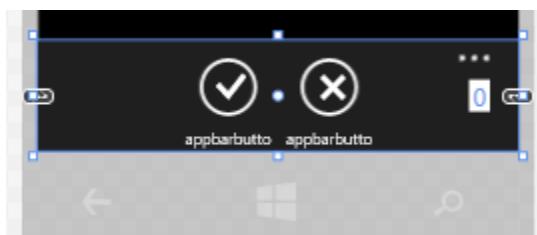
On the left side of Visual Studio, hover over the Document Outline tab and pin it down. It will reveal each of the elements of the Page including a BottomAppBar and a Grid:



Right-click the BottomAppBar to reveal a context menu. Select “Add CommandBar”:



The following will be generated for you:

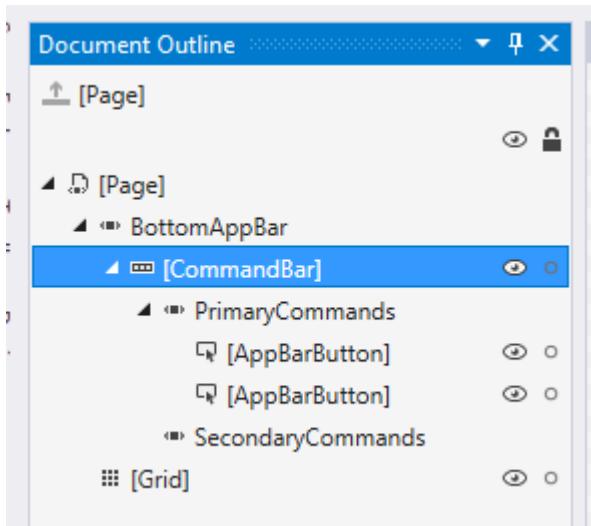


As you can see, several items were created in XAML. First, a BottomAppBar is created for the Page. Inside the BottomAppBar is a CommandBar element containing two AppBarButton controls:

```
<Page.BottomAppBar>
  <CommandBar>
    <AppBarButton Icon="Accept" Label="appbarbutton"/>
    <AppBarButton Icon="Cancel" Label="appbarbutton"/>
```

```
</CommandBar>  
</Page.BottomAppBar>
```

If you peek back at the Document Outline window, it will display the new elements in the outline:



We'll right-click the SecondaryCommands item and select the menu option from the context menu to add a new AppBarButton:

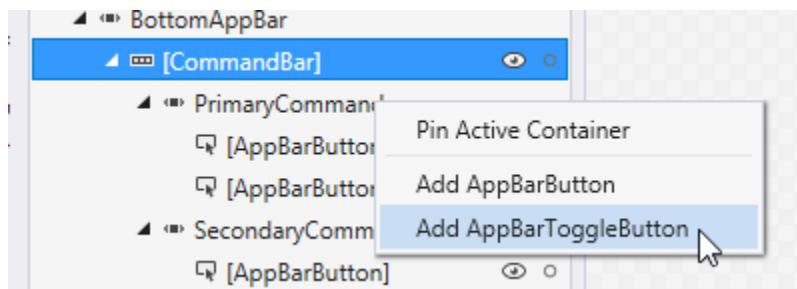


Now, the XAML looks like this:

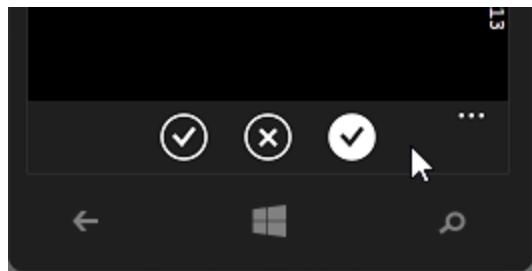
```
<Page.BottomAppBar>  
  <CommandBar>  
    <CommandBar.SecondaryCommands>  
      <AppBarButton Label="appbarbutton"/>  
    </CommandBar.SecondaryCommands>  
    <AppBarButton Icon="Accept" Label="appbarbutton"/>  
    <AppBarButton Icon="Cancel" Label="appbarbutton"/>
```

```
</CommandBar>  
</Page.BottomAppBar>
```

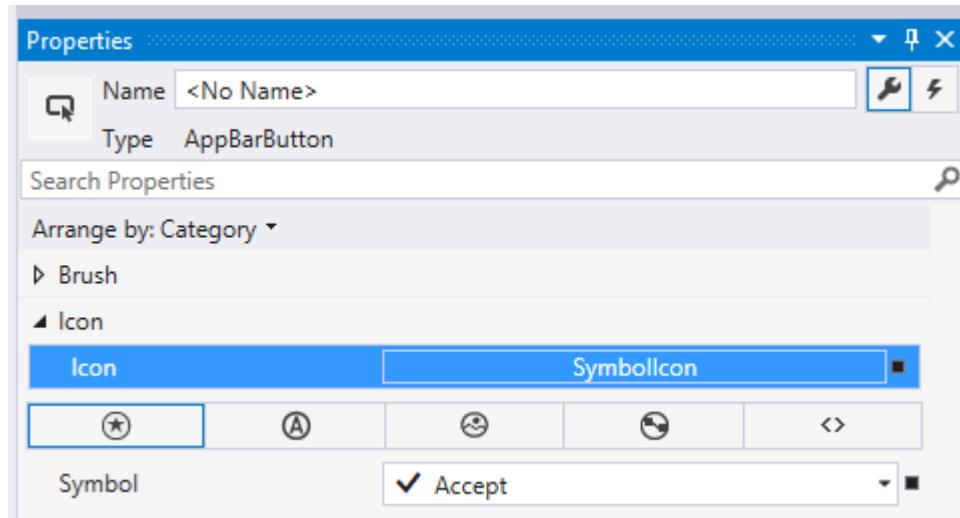
In addition to the PrimaryCommands and SecondaryCommands, we can also add an AppBarToggleButton:



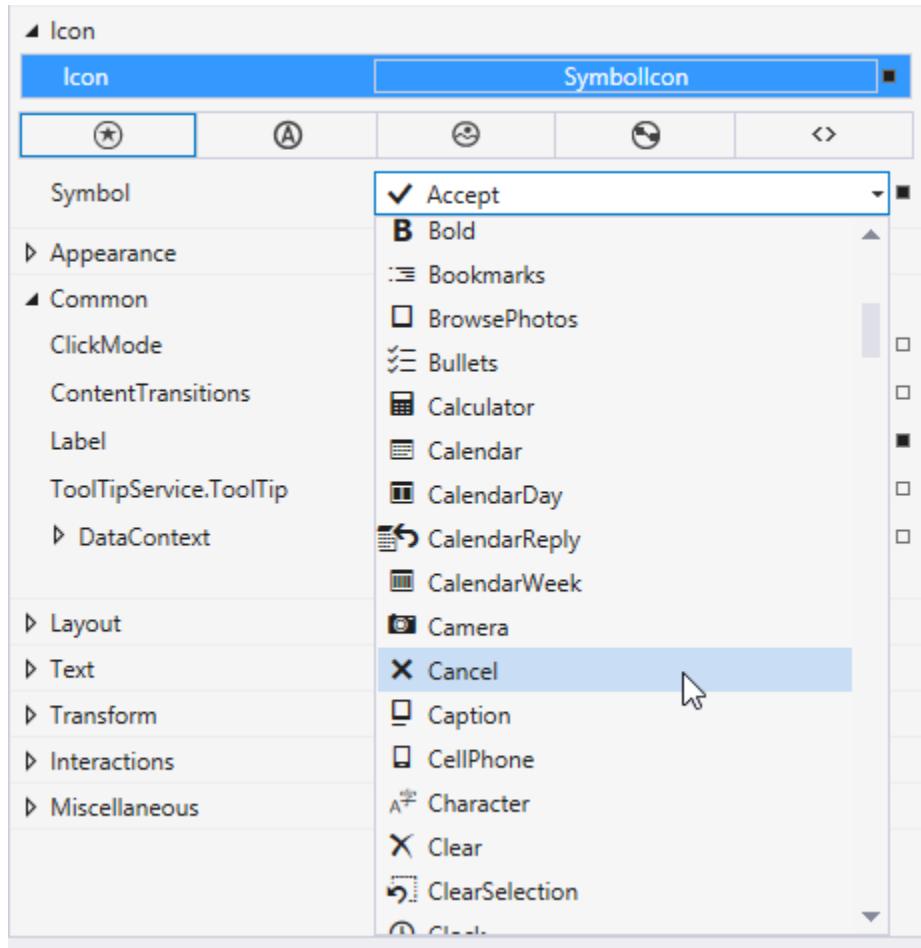
This will provide two states ... on and off, as designated by the “chrome” around the symbol:



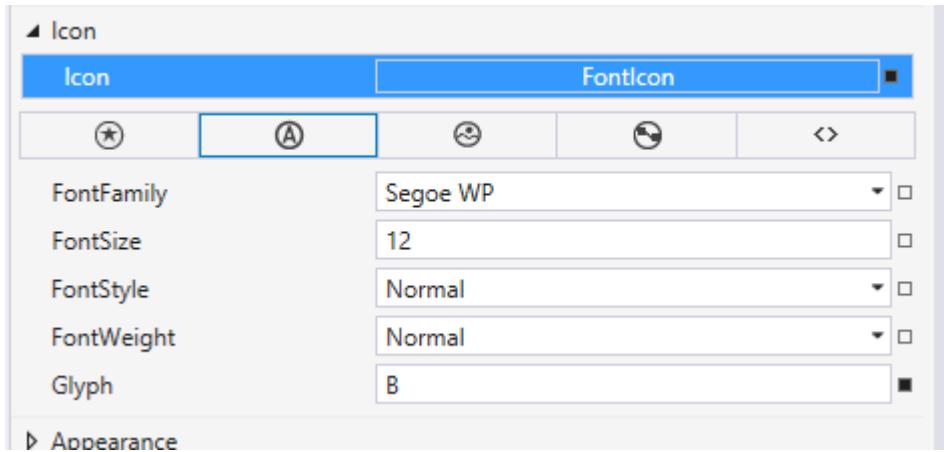
The Phone API has many common symbols available. Put your mouse cursor on a given AppBarButton in the XAML view, then look at the Properties window. If you scroll down to the Icon section, the Symbol Icons are visible by default:



You can select one of the available symbols for use in your app:



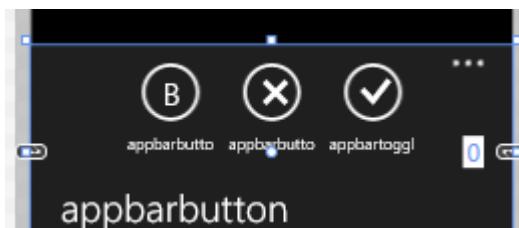
You can also use Fonts to choose a Glyph (letter) for use as an icon:



This selection produces the following XAML:

```
<AppBarButton Label="appbarbutton">
    <AppBarButton.Icon>
        <FontIcon Glyph="B"/>
    </AppBarButton.Icon>
</AppBarButton>
```

And produces the following preview.



There are other options that allow you to use your own images as well. Final note: the AppBarButton operates much like a regular button insomuch that you can handle the Click event and other related options.

Lesson 24: Binding to Commands and CommandParameters

We've already discussed several facets of the Model-View-ViewModel design pattern, or rather MVVM. In this lesson I want to add one more piece to the puzzle so that we can employ it in our upcoming project.

MVVM provides a clean separation of concerns between the data model and the view. This is accomplished through the ViewModel class which is responsible for publishing methods that return collections of our data model to the view so that the view bind to it and display it.

That covers the *display* of data.

What if, for example, we want to not only bind data in our view, but also bind the Click event of a button to a method defined on our data model?

Commands allow us to do that.

It is easier to show you what I mean than to explain it in theory without a concrete example in front of you.

I'll create an app that might be used to check in cars at a Car rental company. Customers drop off their cars in an airport, the Car rental employee sees a list of cars scheduled to be returned today, they click a button next to the car that they receive and see a message with the check in time. This is far from a complete example, but it will allow us to see a simple example of Commands in action.

I'll start by creating a new project called "BindingToCommands". I'll immediate add a Car class:

```
public class Car
{
    public int ID { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public string CheckedInDateTime { get; set; }
}
```

Note: Ideally we would store CheckedInDateTime as a DateTime, not a string. I did this because I want to format the string nicely. As you'll learn in the next lesson, we can create a ValueConverter for the purpose of formatting the date nicely while keeping the value as a DateTime.

Next I create a simple CarDataSource like so:

```

public class CarDataSource
{
    private static ObservableCollection<Car> _cars
        = new ObservableCollection<Car>();

    public static ObservableCollection<Car> GetCars()
    {
        if (_cars.Count == 0)
        {
            _cars.Add(new Car() { ID = 1, Make = "Olds", Model = "Cutlas" });
            _cars.Add(new Car() { ID = 2, Make = "Geo", Model = "Prism" });
            _cars.Add(new Car() { ID = 3, Make = "Ford", Model = "Pinto" });
        }
        return _cars;
    }
}

```

In the MainPage.xaml, I'll call the static GetCars() method to wire up the Default View Model for the page:

```

private ObservableCollection<Car> _carsViewModel =
    CarDataSource.GetCars();

public ObservableCollection<Car> CarsViewModel {
    get { return this._carsViewModel; }
}

```

... and then binding to the view model in XAML like so:

```

<Page
    ...
    DataContext="{Binding CarsViewModel,
        RelativeSource={RelativeSource Self}}">

```

Then I'll implement the layout and make sure the binding works.

```

<StackPanel Margin="20,40,20,40">
    <ListView x:Name="myListView" ItemsSource="{Binding}" >
        <ListView.ItemTemplate>
            <DataTemplate x:Name="dataTemplate">

```

```

<StackPanel Orientation="Horizontal">
    <StackPanel Orientation="Vertical" Margin="0,0,20,0">
        <TextBlock Text="{Binding Make}" FontSize="24" />
        <TextBlock Text="{Binding Model}" FontSize="24" />
    </StackPanel>

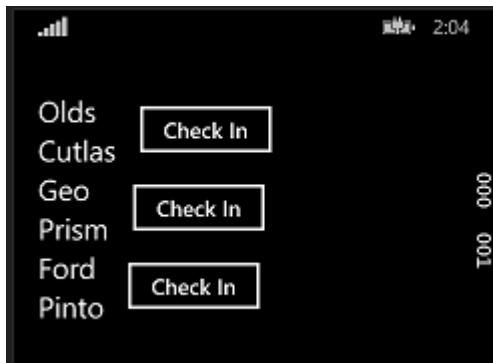
    <Button Content="Check In"
        Width="100"
        Height="50"
        Margin="0,0,20,0" />

    <TextBlock Text="{Binding CheckedInDateTime}" FontSize="24" />
</StackPanel>
</DataTemplate>
</ListView.ItemTemplate>

</ListView>
</StackPanel>

```

This produces the following visual result:



With the preliminary work out of the way, we can turn our focus to implementing the Command. The first order of business is to implement the command that represents a “check in” for a car. I’ll create a class called CheckInButtonClick that implements ICommand. Then, I’ll use the Control + [dot] keyboard combination to implement the ICommand interface:



This provides a stubbed out set of methods and the event declaration like so:

```
public class CheckInButtonClick : ICommand
{
    public bool CanExecute(object parameter)
    {
        throw new NotImplementedException();
    }

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        throw new NotImplementedException();
    }
}
```

First, I need to merely answer the question “can this be executed”? Here I could write logic to decide whether or not this command can be executed based on the current state of the Car object that is passed in as a parameter. Ultimately, I’ll return a bool. In our case, since I’m going to keep this simple, I’ll merely return true. This command can always be executed regardless of the state of the object.

```
public class CheckInButtonClick : ICommand
{
    public bool CanExecute(object parameter)
    {
        return true;
    }

    ...
}
```

In actuality, I would probably want to check to see whether if this instance of Car has already been checked in prior to allowing it a second time. I’ll leave the implementation of that business rule to you for experimentation.

Now, I need to associate the command with my Car class:

```
public ICommand CheckedInCommand { get; set; }
```

```

public Car()
{
    CheckedInCommand = new CheckInButtonClick();
}

```

I create an auto-implemented property of type ICommand, then in the Constructor, set the property to an instance of the CheckInButtonClick() command class. Now I have a class that can be Commanded, in other words, that I can bind a Command to.

Next, I'll want to tell the command what to do when this command is executed? That's what we'll do in the Execute method. In this case I want to set the CheckedInDateTime property to DateTime.Now. Also, per my note, above, I want to format that for proper display.

I have two options. I can either implement this in my model (i.e., a public method on my Car class) or the classes I use to generate by View Model (i.e., the CarDataSource class). Which one you choose depends on how accessible you want to make your View Model and how flexible you want to make your Command. If you were to either make your Data Source a property of the App OR make all the methods of your Data Source class static, AND you don't mind your Command being tied specifically to your CarDataSource class, then you could implement the Execute command like so:

```

public async void Execute(object parameter)
{
    CarDataSource.CheckInCar((Car)parameter);
}

```

And in your CarDataSource, you would implement the CheckInCar() method like so:

```

public class CarDataSource
{
    ...

    public static void CheckInCar(Car car) {
        _cars.FirstOrDefault(p => p.ID == car.ID)
            .CheckedInDateTime = String.Format("{0:t}", DateTime.Now);
    }
}

```

However, this approach has one obvious downside. The Command is now coupled to the CarDataSource. It could not be used as a more general purpose command. That might be fine for your purpose, however another option is to make the CheckInCar() method a public property of the Car class itself:

```

public class Car
{
    ...

    public void CheckInCar()
    {
        this.CheckedInDateTime = String.Format("{0:t}", DateTime.Now);
    }
}

```

And I can call this method from the Command's Execute() method like so:

```

public async void Execute(object parameter)
{
    //CarDataSource.CheckInCar((Car)parameter);
    ((Car)parameter).CheckInCar();
}

```

I prefer this.

Next, I'll need to wire up the binding correctly. I will want to fire the Command when I click the button, AND I'll want to pass in the current instance of the Car to the Command's Execute() method. To do this, I'll add the following XAML in the Button's declaration:

```

<Button Content="Check In"
    ...
    Command="{Binding CheckedInCommand}"
    CommandParameter="{Binding}"
    .../>

```

In this case, the CommandParameter is bound to it's parent binding which is defined in the ListView's ItemSource:

```
<ListView x:Name="myListView" ItemsSource="{Binding}" >
```

... which in turn is bound to:

```

<Page
    ...
    DataContext="{Binding CarsViewModel, RelativeSource={RelativeSource Self}}">

```

To me, this is a bit confusing. It looks like I'm binding the ENTIRE CarViewModel to the CommandParameter. However, the correct way to interpret this ... the ListView's ItemSource expects a collection. Then each item is then bound to just one instance of that collection,

therefore the CommandParameter="{}{Binding}" means "bind to the current instance of the Car class", not to the entire collection. That was a little tricky for me to understand at first. The key is understanding how the DataContext flows down from the page, into the ListView and then into the ItemTemplate.

Unfortunately, when I run the app, I may not get the results I was hoping for visually. I would expect the value of CheckedInDateTime to change and then be presented on the screen when I click the Check In button. It does not. I assure you, the value is being changed in the data model, but because Car is not notifying the user interface of this property change, the user interface does not know it should update its presentation of the data.

Do you recall what we need to do to tell the user interface that a property changed?

Yes, we need to implement the INotifyPropertyChanged interface on the Car class, then in the CheckedInDateTime setter, call the NotifyPropertyChanged() method that the interface forces us to implement. Let's implement that now using the Control + [dot] technique from earlier.

Note: Keep in mind that this is not part of Commands per se, however if your Commands will be updating properties, you very well may need to implement INotifyPropertyChanged to enable some user interface update like we'll do from this point on in the lesson.

```
public class Car : INotifyPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;

    private void NotifyPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

In this case, I really only care to call the NotifyPropertyChanged() method on the setter of CheckedInDateTime. So, I'll change the current implementation from an auto implemented property:

```
public DateTime CheckedInDateTime { get; set; }
```

To full property using the snippet: propfull [tab] [tab]

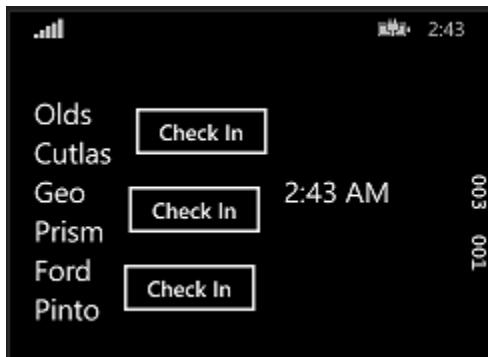
The full implementation after making the appropriate changes to the full property:

```
//public DateTime CheckedInDateTime { get; set; }

private string checkedInDateTime;

public string CheckedInDateTime
{
    get { return checkedInDateTime; }
    set {
        checkedInDateTime = value;
        NotifyPropertyChanged("CheckedInDateTime");
    }
}
```

Now when I click the Check In button, you can see the update in the user interface:



Recap

The beauty of a Command is that it keeps the separation of concerns in tact while allowing you to handle events triggered on the user interface that affect an instance of your view model. We talked about the implementation of the ICommand interface and walked through a complete example. We also looked at a practical implementation of the INotifyPropertyChanged interface.

Lesson 25: Advanced Binding with Value Converters

The last piece of the puzzle we'll need before building our next complete exercise app is to understand Value Converters. In short, sometimes you want to use data in your View Model that is not in the correct format or must be in some way modified before you can bind to it. A Value Converter allows you to perform a conversion on the data to format the data, change the value of the data, perform calculations on the data and more.

In fact, in this lesson, I'll demonstrate how diverse your value converters can be.

I've created a "Before" folder. Please make sure you download this and unzip it to a directory called `BindingWithValueConverters`.

Here I have the beginnings of a weather app. In this example, I'm randomly generating weather, however in a real app you would obviously be pulling weather from an online weather service.

Notice the structure of the classes that will form the data model. First, there is a `DayForecast` defined as so:

```
public class DayForecast
{
    public DateTime Date { get; set; }
    public Weather Weather { get; set; }
    public ObservableCollection<TimeForecast> HourlyForecast { get; set; }
}
```

For a given day (the `Date` property) the `Weather` property will indicate the defining attribute of the day (whether sunny, cloudy, rainy or snowy ... I've reduced the number of options for simplicity's sake). I've used an enum called `Weather` to denote the various values of `Weather`. Finally, the `HourlyForecast` property is made up of an `ObservableCollection<TimeForecast>` ... precisely 24 `TimeForecast` objects, to be exact, one for each hour of the day from 0 (midnight) to 11pm (hour 23) as indicated by the `Hour` property. `TimeForecast` is defined as:

```
public class TimeForecast
{
    public int Hour { get; set; }
    public int Temperature { get; set; }
}
```

How will we create data for the `Temperature` of each `TimeForecast` object? I've created a `FiveDayForecast` class which has a single method, `GetForecast()`. Its job is to deliver an

ObservableCollection<DayForecast> to the MainPage.xaml. This class also has a helper method called generateRandomTimeForecast that creates 24 hours with a random temperature and returns those as an ObservableCollection<TimeForecast> for use in the DayForecast.HourlyForecast property:

```
private static ObservableCollection<TimeForecast> generateRandomTimeForecast(int seed)
{
    var random = new System.Random(seed);
    var forecast = new ObservableCollection<TimeForecast>();

    for (int i = 0; i < 24; i++)
    {
        var timeForecast = new TimeForecast();
        timeForecast.Hour = i;
        timeForecast.Temperature = random.Next(105);
        forecast.Add(timeForecast);
    }
    return forecast;
}
```

How will this data be displayed in our app? If you run the Before app as is, you'll see the layout of the User Interface.



Admittedly, most of the data is faked right now. The data exists, however we've not bound to it because it's not in the correct format for our purposes.

Let's look at the XAML to understand what is happening and how we'll need to plug our converted values into the existing app.

```
<Page  
...  
    DataContext="{Binding WeatherViewModel, RelativeSource={RelativeSource Self}}">  
  
<ScrollViewer>  
<StackPanel>  
    <ListView ItemsSource="{Binding}">  
        <ListView.ItemTemplate>  
            <DataTemplate>  
                <StackPanel Orientation="Vertical" Margin="20,0,0,30">  
                    <TextBlock Text="{Binding Date}" FontSize="24" />  
                    <StackPanel Orientation="Horizontal" Margin="0,10,0,0">  
                        <Image Source="assets/snowy.png" Width="100" Height="100" />  
                    <StackPanel Orientation="Vertical" Margin="20,0,0,0">  
                        <TextBlock Text="High: " FontSize="14" />
```

```

<TextBlock Text="55" FontSize="18" />
<TextBlock Text="Low: " FontSize="14" />
<TextBlock Text="32" FontSize="18" />
</StackPanel>
</StackPanel>
</StackPanel>
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
</StackPanel>
</ScrollViewer>

```

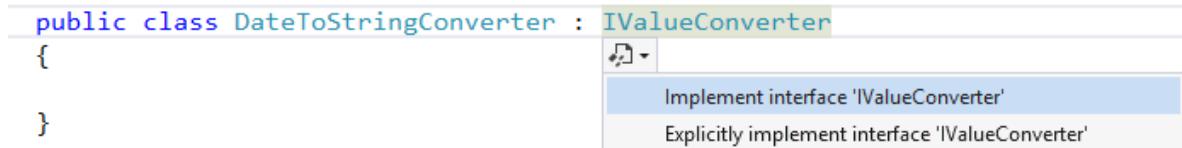
As you can see, we're employing a ScrollViewer to allow us to scroll past the bottom of the viewable area. It wraps a StackPanel that contains a ListView. In retrospect, a GridView may have worked as well if not better in this situation. The ListView is bound to the ViewModel. The ListView's DataTemplate is made up of a series of StackPanels with TextBlocks and an Image. The only data item we're binding to at the moment is the Date, but even that is in need of a little massaging to make it look correct. Ultimately, we'll want to bind the source of the Image based on the Weather enum, and we'll determine the high and low temperatures for a given day all using Value Converters.

I'll add a new class to the project called DateToStringConverter.

First, I'll make the class public and I'll require it to implement `IValueConverter`. Once I've added that code,

```
public class DateToStringConverter : IValueConverter
{
}
```

Next, with my text cursor on the term `IValueConverter` I'll use the Command [dot] shortcut to display the Intellisense menu:



... which will allow me to choose to Implement the interface by hitting the [Enter] key on my keyboard. My class now looks like this:

```
public class DateToStringConverter : IValueConverter
{
```

```

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

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

```

As you can see, two method stubs are added to the class by implementing the `IValueConverter`. `Convert` will perform the conversion to your target display value, `ConvertBack` will do the opposite. In our case, we merely need to format the Date into a string with the format I desire, so I implement like so:

```

public class DateToStringConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        DateTime date = (DateTime)value;

        return String.Format("{0:dddd} - {0:d}", date);
    }

    public object ConvertBack(object value, Type targetType, object parameter, string language)
    {
        // For "display only", I don't bother implementing ConvertBack
        // HOWEVER if your planning on two-way binding, then you must.
        throw new NotImplementedException();
    }
}

```

Note that a better / more complete example of a `ValueConverter` that takes into account the language / culture as well as the converter parameter is found here:

<http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.data.ivalueconverter>

Next, I'll need to tell my MainPage.xaml to use this new converter (I'll remove everything except what will be added or changed):

```
<Page
...
    xmlns:valueconverter="using:ValueConverterExample"
...
<Page.Resources>
    <valueconverter:DateToStringConverter x:Key="DateToStringConverter" />
</Page.Resources>

...
<TextBlock Text="{Binding Date,
    Converter={StaticResource DateToStringConverter}}"
    FontSize="24" />
```

In the Page declaration, I create a new namespace. In this specific case I didn't have to do this because the Value Converter I created is in the same namespace as the local namespace that's already defined in the page. However, at some point I plan on moving my Value Converters into a folder and will change their namespace accordingly. Therefore, I've added this as a reminder / placeholder to myself.

Next, I add the Value Converter key "DateToStringConverter" and point it to the class named DateToStringConverter (inside the namespace that is aliased to "valueconverter").

Third, I use the value converter in my TextBlock, inside the binding expression.

Testing the result provides the format we desire.

Next, I'll add a value converter that will be used to change the Image's source. I'll use the same basic procedure as before to create a new value converter called WeatherEnumToImagePathConverter. A bit wordy, admittedly, but very descriptive. I'll implement it like so:

```
public class WeatherEnumToImagePathConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        var weather = (Weather)value;
```

```

var path = String.Empty;

switch (weather)
{
    case Weather.Sunny:
        path = "assets/sunny.png";
        break;
    case Weather.Cloudy:
        path = "assets/cloudy.png";
        break;
    case Weather.Rainy:
        path = "assets/rainy.png";
        break;
    case Weather.Snowy:
        path = "assets/snowy.png";
        break;
    default:
        Break;
}

return path;
}

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

```

Hopefully you see a pattern. I am merely taking the value, casting it to the original type I am expecting, then performing some logic or formatting on the value to determine the end result. In this case, I'm returning the path to the image I'll use in the Image's source.

I'll now add the code to the XAML to implement this:

```

...
<Page.Resources>
    ...
    <valueconverter:WeatherEnumToImagePathConverter
        x:Key="WeatherEnumToImagePathConverter" />
</Page.Resources>

```

```
<Image Source="{Binding Weather,
    Converter={StaticResource WeatherEnumToImagePathConverter}}"
    Width="100"
    Height="100" />
```

Running the app will display each of the icons at least once.

Next, I'll determine the high and the low for each day. Recall that each day's forecast contains an `ObservableCollection<TimeForecast>` that represent the temperature for each hour. All we need to do is look at that collection and find the highest number (or lowest number) and return those from our Value Converter. A simple LINQ query will do the trick in each case.

I'll add two more classes,

`HourlyCollectionToDailyHighConverter`
`HourlyCollectionToDailyLowConverter`

Here's the `HourlyCollectionToDailyHighConverter`:

```
public class HourlyCollectionToDailyHighConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        var hourly = (ObservableCollection<TimeForecast>)value;

        return hourly
            .OrderByDescending(p => p.Temperature)
            .FirstOrDefault()
            .Temperature;
    }
    ...
}
```

And the `HourlyCollectionToDailyLowConverter`:

```
public class HourlyCollectionToDailyLowConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
```

```

var hourly = (ObservableCollection<TimeForecast>)value;

return hourly
    .OrderBy(p => p.Temperature)
    .FirstOrDefault()
    .Temperature;
}

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

```

And now I'll add them to the XAML:

```

<Page.Resources>
...
<valueconverter:HourlyCollectionToDailyHighConverter
x:Key="HourlyCollectionToDailyHighConverter" />
<valueconverter:HourlyCollectionToDailyLowConverter
x:Key="HourlyCollectionToDailyLowConverter" />
</Page.Resources>

<TextBlock Text="High: " FontSize="14" />
<TextBlock Text="{Binding HourlyForecast,
    Converter={StaticResource HourlyCollectionToDailyHighConverter}}"
    FontSize="18" />
<TextBlock Text="Low: " FontSize="14" />
<TextBlock Text="{Binding HourlyForecast,
    Converter={StaticResource HourlyCollectionToDailyLowConverter}}"
    FontSize="18" />

```

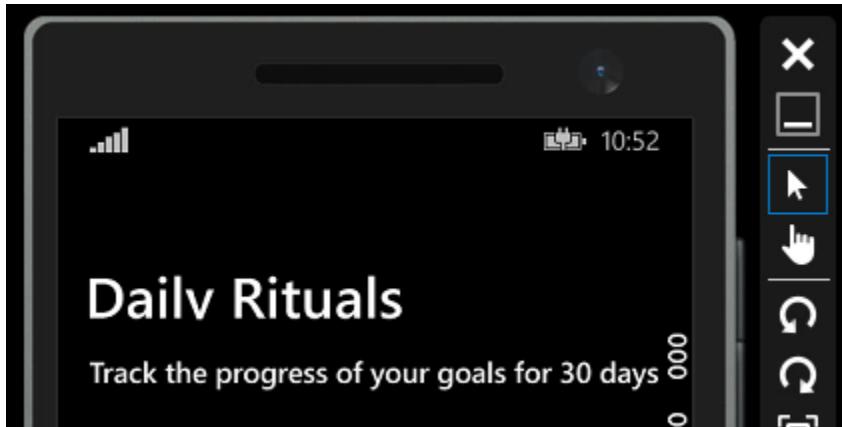
Now, when you run the completed app, the results are essentially the same (from an aesthetic perspective) however the data is being pulled from our View Model and shaped appropriately into what we need for our View.

Recap

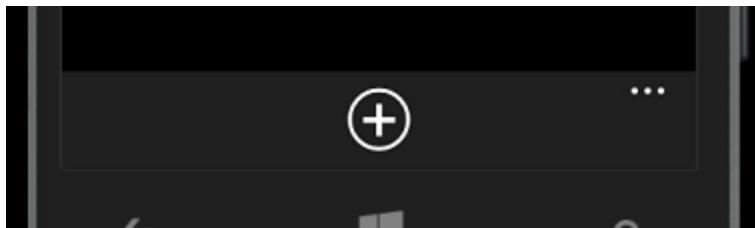
Value Converters allow us to shape the data items of our View Model into the format need to correctly bind to our View. You can merely format the data item, or change it completely. We'll use this technique as we build a complete example in the next lesson.

Lesson 26: Exercise: The Daily Rituals App

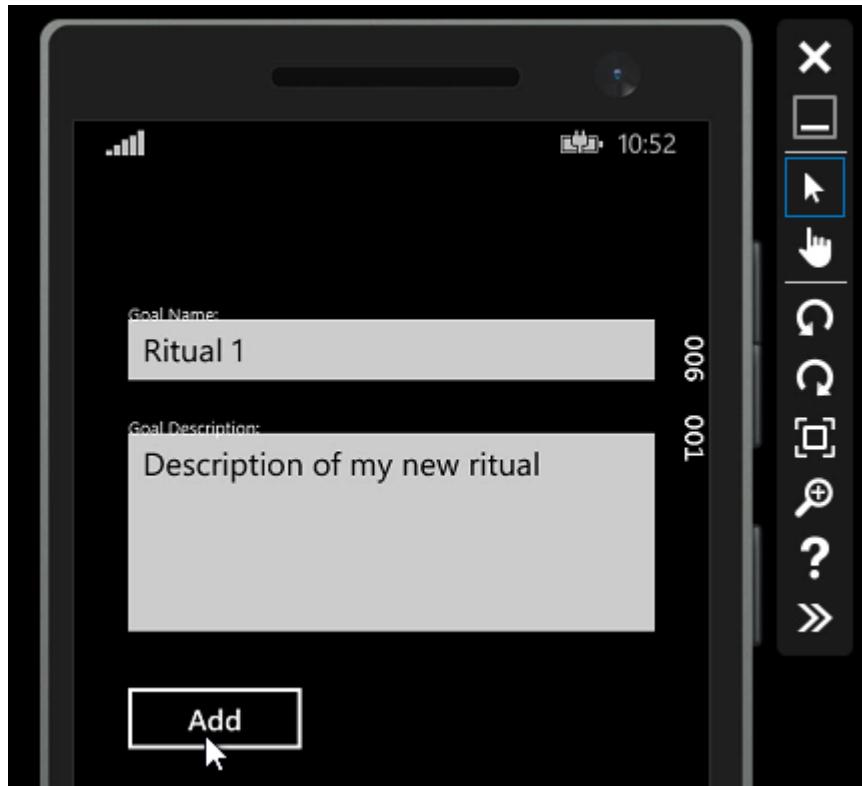
In this exercise we'll build a project that combines many of the things we talked about previously into a single example. The Daily Rituals project is essentially it's a goal-tracking app. The idea is, for 30 days, you'll track your progress against all the goals you've created.



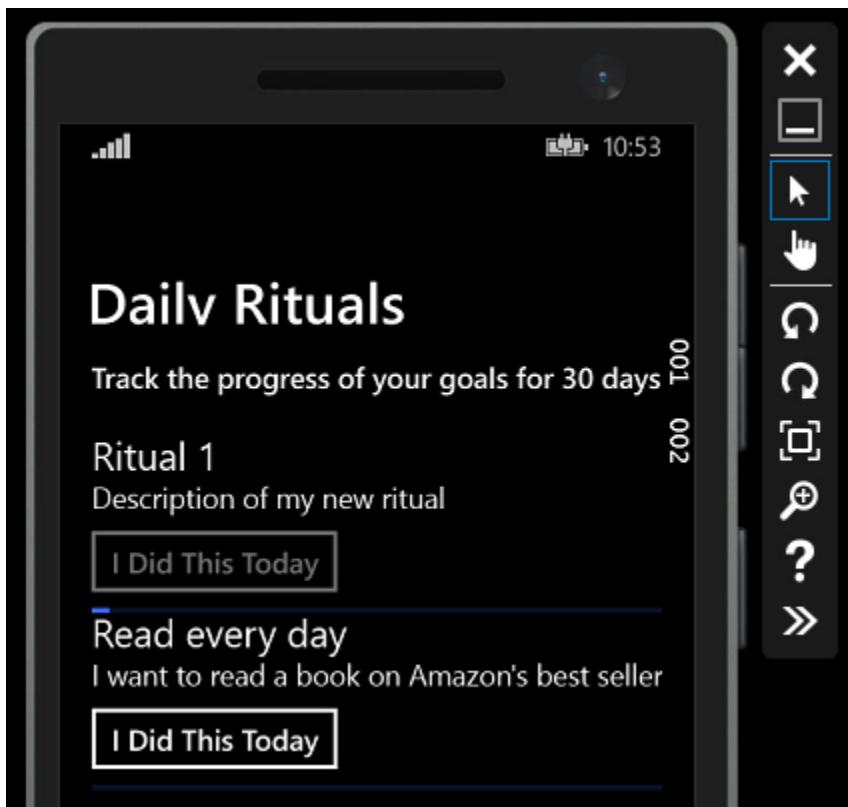
For example, I'll create a new ritual or goal for myself. I'll click the + Add icon in the Command Bar ...



Which allows me to add a new Ritual. I'll create dummy data for this demo, the Goal Name "Ritual 1" and the Goal Description: "Description of my new ritual", and I will click the Add button ...

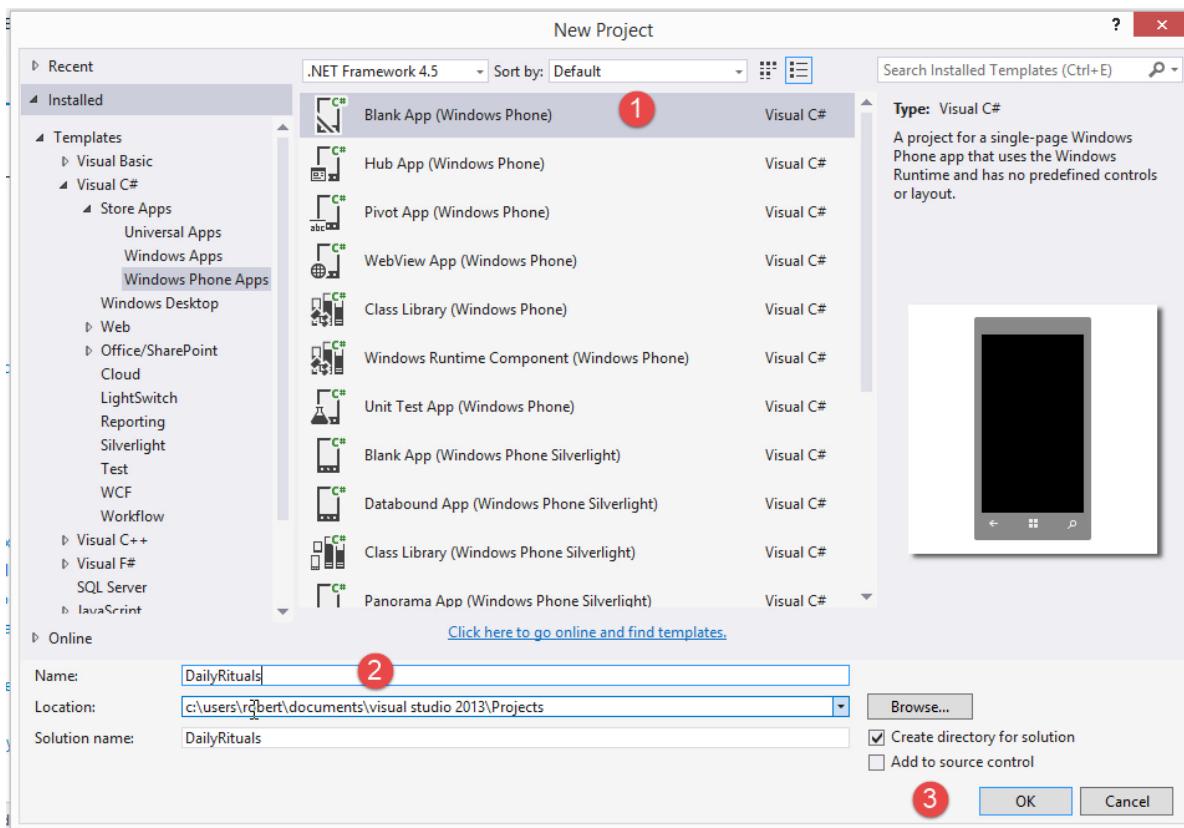


I'll return to the main screen where I can see a list of rituals. (I've added a second one in the screen shot, below):

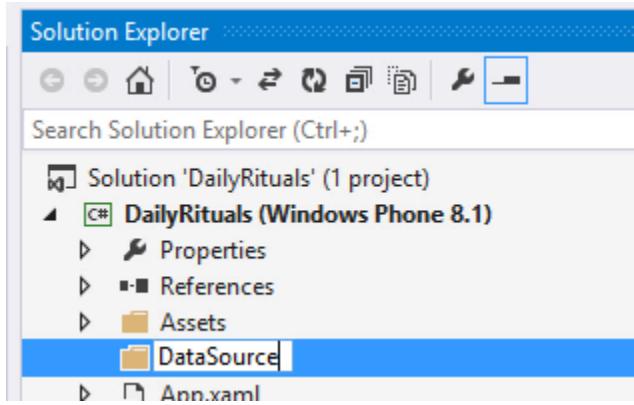


I can click the “I Did This Today” button which will keep track of the days I completed this goal and it will show progress on a Progress Control below the Ritual. The progress bar will fill up over 30 days of establishing these daily rituals.

I'll begin by (1) creating a new Blank App project (2) called Daily Rituals. (3) I'll click the OK button on the New Project dialog to create the project.

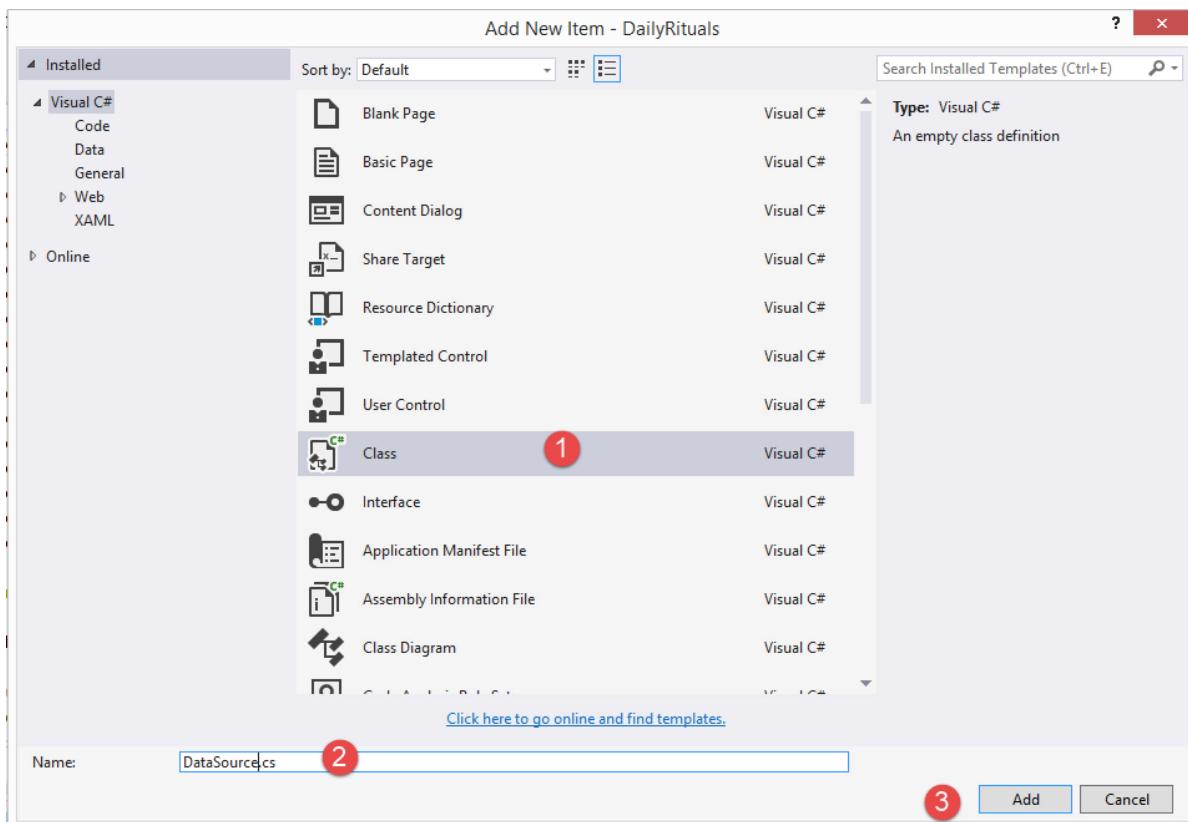


First, I'll create a new folder called **DataSource**:



Then right-click to Add > New Item ... in that folder.

In the Add New Item dialog, (1) I'll add a Class (2) named **DataSource.cs** then (3) click the Add button:



First, I'll create a simple Ritual class using the prop code snippet to create the auto-implemented properties. I'll also add the proper using statement for the ObservableCollection<T> using the Control + [dot] keyboard shortcut.

```

namespace DailyRituals.DataModel
{
    public class Ritual
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public ObservableCollection<DateTime> Dates { get; set; }
    }
}

```

Second, I'll flesh out the DataSource class adding a private field of type ObservableCollection<Ritual> called _rituals. I'll create a new instance of ObservableCollection<Ritual> in the constructor. As you may anticipate, the rest of the DataSource class will define methods that operate on this private field.

```
public class DataSource
{
    private ObservableCollection<Ritual> _rituals;

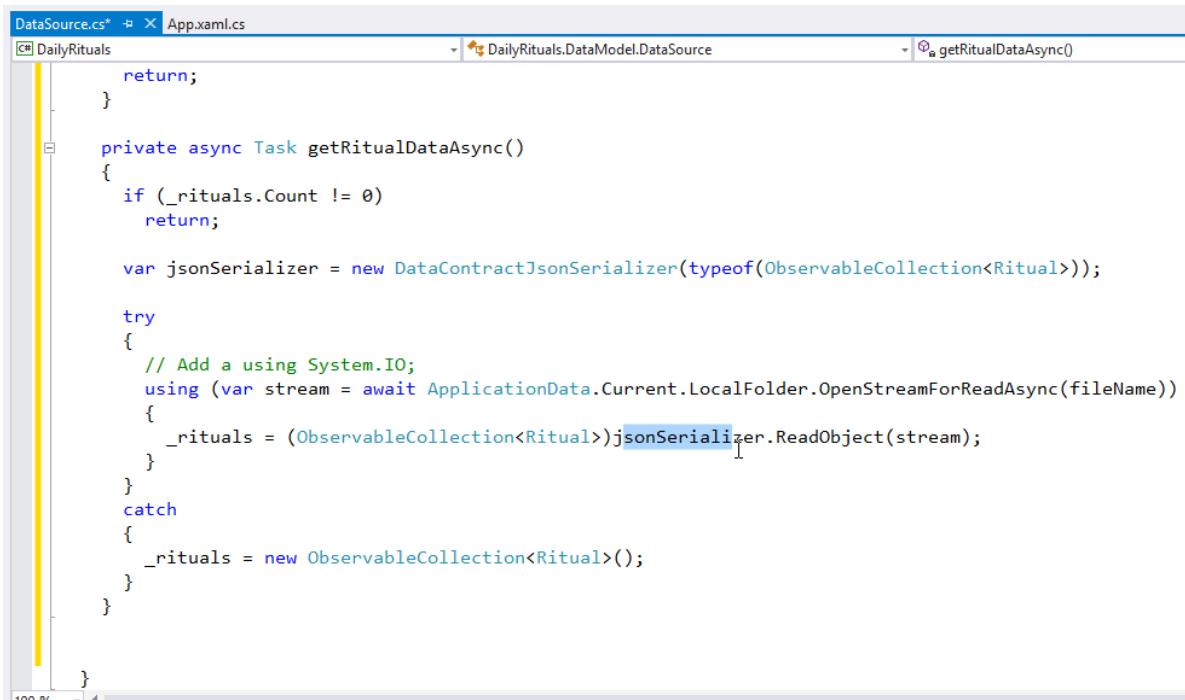
    public DataSource()
    {
        _rituals = new ObservableCollection<Ritual>();
    }
}
```

We'll follow the pattern we've seen several times before now, creating a helper method called `ensureDataLoaded()` that will determine whether or not the `_rituals` collection contains instances of `Ritual`. If not, then we'll call the (yet to be implemented) `getRitualDataAsync()` method.

```
private async Task ensureDataLoaded()
{
    if (_rituals.Count == 0)
        await getRitualDataAsync();

    return;
}
```

We delegate the responsibility of retrieving data from a serialized file back into an object graph stored in the `_rituals` collection to the `getRitualDataAsync()` method. We've seen this code before when working with serialized types that we need to "re-hydrate" back into an object graph. Most of the heavy lifting is performed by the `DataContractJsonSerializer` and the `ApplicationData.Current.LocalFolder.OpenStreamForReadAsync()` method.



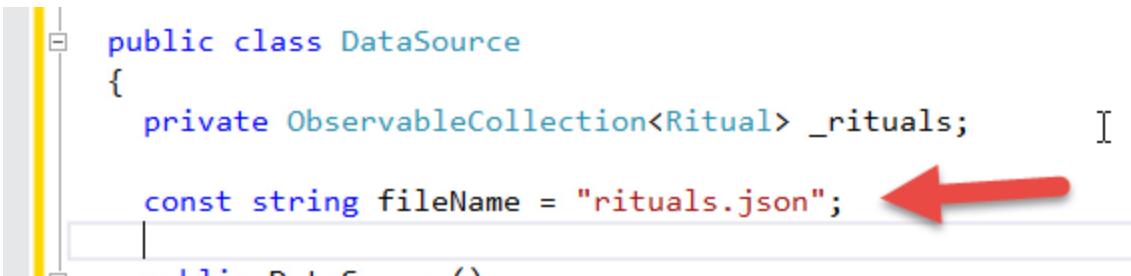
```
DataSource.cs*  App.xaml.cs
C# DailyRituals  DailyRituals.DataModel.DataSource  getRitualDataAsync()
    return;
}

private async Task getRitualDataAsync()
{
    if (_rituals.Count != 0)
        return;

    var jsonSerializer = new DataContractJsonSerializer(typeof(ObservableCollection<Ritual>));

    try
    {
        // Add a using System.IO;
        using (var stream = await ApplicationData.Current.LocalFolder.OpenStreamForReadAsync(fileName))
        {
            _rituals = (ObservableCollection<Ritual>)jsonSerializer.ReadObject(stream);
        }
    }
    catch
    {
        _rituals = new ObservableCollection<Ritual>();
    }
}
```

There are several things we'll have to fix with this code, including the creation of a constant for the fileName which I'll define near the top of the DataSource class:



```
public class DataSource
{
    private ObservableCollection<Ritual> _rituals;

    const string fileName = "rituals.json";
}
```

... and I'll also need to add various using statements to satisfy all of the class references:

```
DataSource.cs* X App.xaml.cs
C# DailyRituals DailyRituals.DataModel.Ritual
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Runtime.Serialization.Json;
using System.Text;
using System.Threading.Tasks;
using Windows.Storage;
using System.IO;
```

Next, I'll implement the corollary to the `getRitualDataAsync()`, the `saveRitualDataAsync()` method which will be delegated the responsibility of persisting / "de-hydrating" the object graph stored in the `_rituals` field to disk. Here again, most of the heavy lifting is performed by the `DataContractJsonSerializer` and the `Application.Current.LocalFolder.OpenStreamForWriteAsync()` method.

```
private async Task saveRitualDataAsync()
{
    var jsonSerializer = new DataContractJsonSerializer(typeof(ObservableCollection<Ritual>));
    using (var stream = await ApplicationData.Current.LocalFolder.OpenStreamForWriteAsync(fileName,
        CreationCollisionOption.ReplaceExisting))
    {
        jsonSerializer.WriteObject(stream, _rituals);
    }
}
```

We will want to allow one of our pages to add new Rituals to the `_rituals` collection. Therefore, we'll implement a public method that will do that:

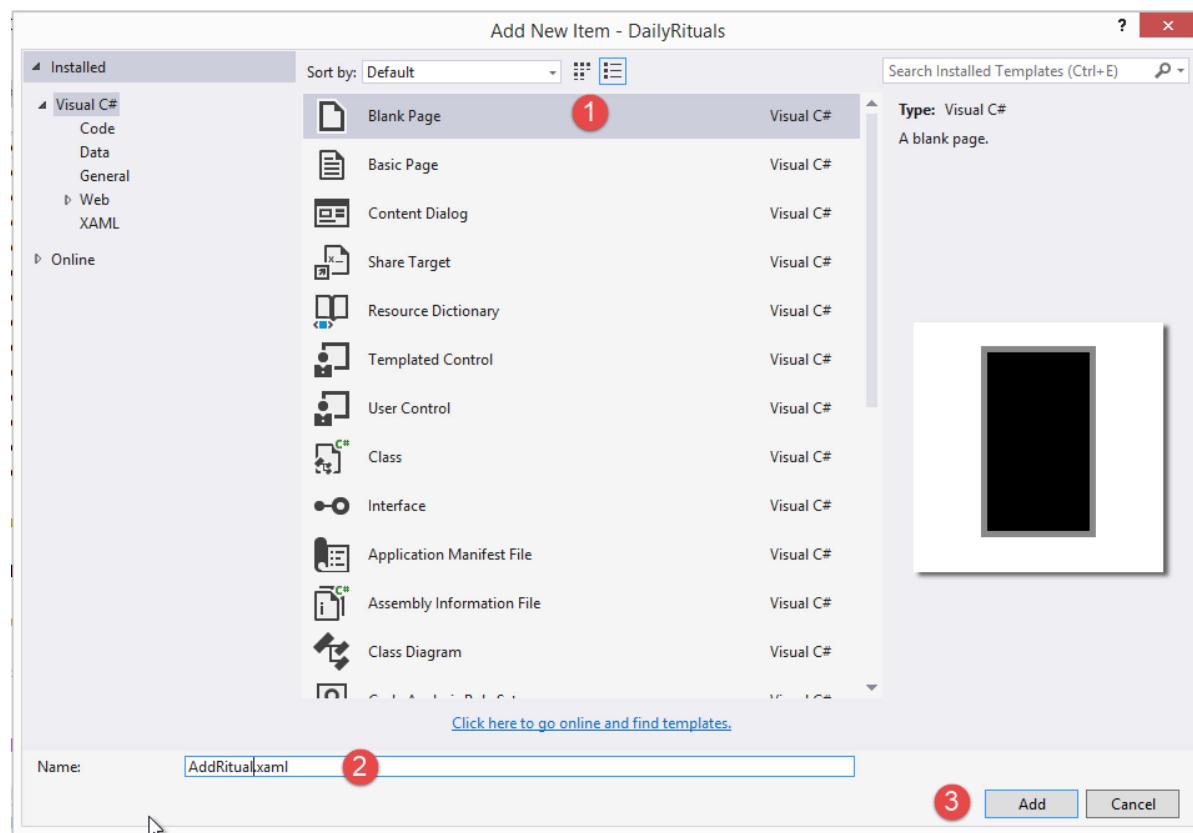
```
public async void AddRitual(string name, string description)
{
    var ritual = new Ritual();
    ritual.Name = name;
    ritual.Description = description;
    ritual.Dates = new ObservableCollection<DateTime>();

    _rituals.Add(ritual);
    await saveRitualDataAsync();
}
```

Furthermore, we'll want to retrieve the `_rituals` collection:

```
public async Task<ObservableCollection<Ritual>> GetRituals()
{
    [
        await ensureDataLoaded();
        return _rituals;
}
```

We have enough of the data model in place. Next we can focus on the `AddRitual.xaml` page. I'll right-click the project name and select `Add > New Item ...` to display the Add New Item dialog. I'll (1) select a Blank Page, (2) name it `AddRitual.xaml`, and (3) click the Add button:



Before we begin work on the new page, we'll need to be able to reference the `DataSource` class from the entire project. To do so, (1) in the `App.xaml.cs`, (2) I'll create a public static field called `DataModel` (3) which will necessitate the need to add a `using` statement to the `DataModel` namespace.

```

AddRitual.xaml MainPage.xaml.cs MainPage.xaml DataSource.cs App.xaml.cs ① DailyRituals DailyRituals.App DataModel
using Windows.UI.Xaml.Media.Animation;
using Windows.UI.Xaml.Navigation;

// The Blank Application template is documented at http://go.microsoft.com/fwlink/?LinkId=391641

namespace DailyRituals
{
    /// <summary>
    /// Provides application-specific behavior to supplement the default Application class.
    /// </summary>
    public sealed partial class App : Application
    {
        public static DataSource DataModel; ②

        private Tran ③
        using DailyRituals.DataSource;
        DailyRituals.DataSource.DataSource
        Generate class for 'DataSource'
        Generate new type...
        /// <summary>
        /// Initiali
        /// executed
        /// </summary>
    }
}

```

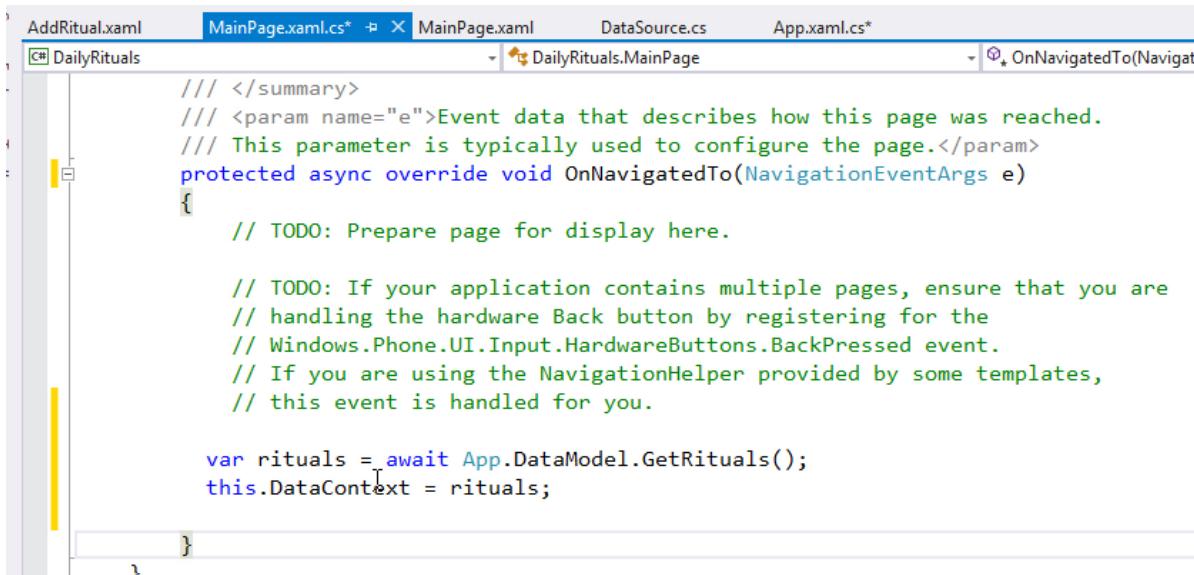
In the App() class' constructor, I'll create a new instance of DataSource().

```

    /// <summary>
    public App()
    {
        this.InitializeComponent();
        this.Suspending += this.OnSuspending;
        DataModel = new DataSource(); ④
    }
}

```

Now, that I have a way to get at the DataModel, I'll wire up the data context for the MainPage.xaml. Up to now, we've done this by creating an instance of the DataSource class and holding a reference to it in a public read only property that we bind to declaratively. However, this time I decided to do something different. Here I'm calling App.DataModel.GetRituals() to get an ObservableCollection<Ritual> and set the DataContext property of the page. Using this technique, there's no additional work to be done in XAML.

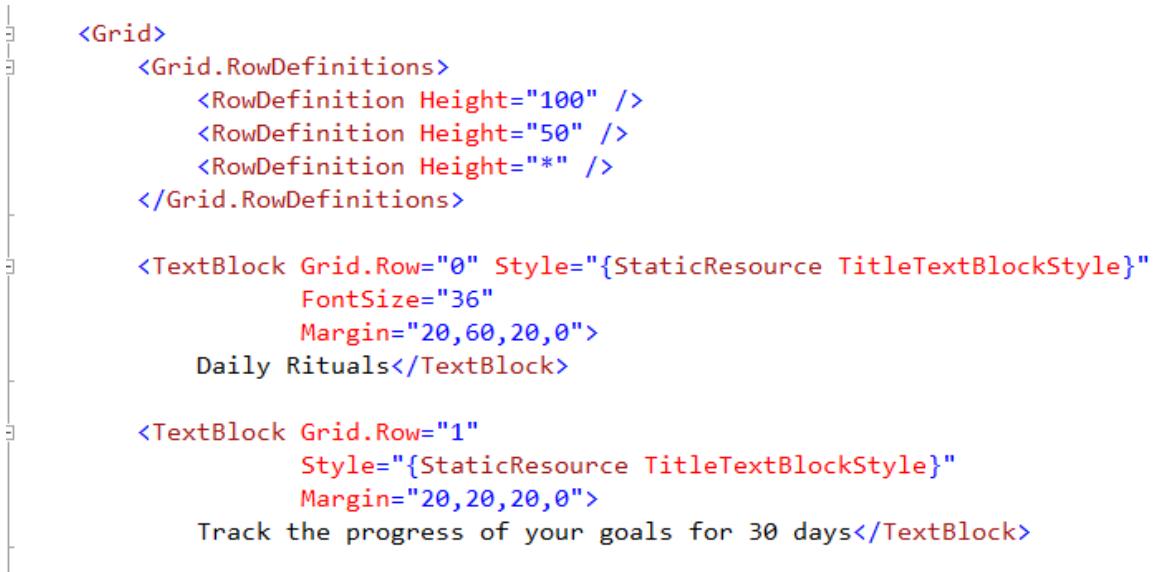


```
/// </summary>
/// <param name="e">Event data that describes how this page was reached.
/// This parameter is typically used to configure the page.</param>
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    // TODO: Prepare page for display here.

    // TODO: If your application contains multiple pages, ensure that you are
    // handling the hardware Back button by registering for the
    // Windows.Phone.UI.Input.HardwareButtons.BackPressed event.
    // If you are using the NavigationHelper provided by some templates,
    // this event is handled for you.

    var rituals = await App.DataModel.GetRituals();
    this.DataContext = rituals;
}
```

In the MainPage.xaml, I add RowDefinitions, then the title TextBlocks:



```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="50" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <TextBlock Grid.Row="0" Style="{StaticResource TitleTextBlockStyle}"
        FontSize="36"
        Margin="20,60,20,0">
        Daily Rituals</TextBlock>

    <TextBlock Grid.Row="1"
        Style="{StaticResource TitleTextBlockStyle}"
        Margin="20,20,20,0">
        Track the progress of your goals for 30 days</TextBlock>
```

Below that, I add an inner Grid containing an ItemsControl. This will give the Grid the binding behavior of a ListView. Notice that the ItemTemplate has a blue-squiggly line beneath it ... that's because we haven't created the ItemTemplate yet:



```
<Grid Grid.Row="2" Margin="20,20,20,0" >

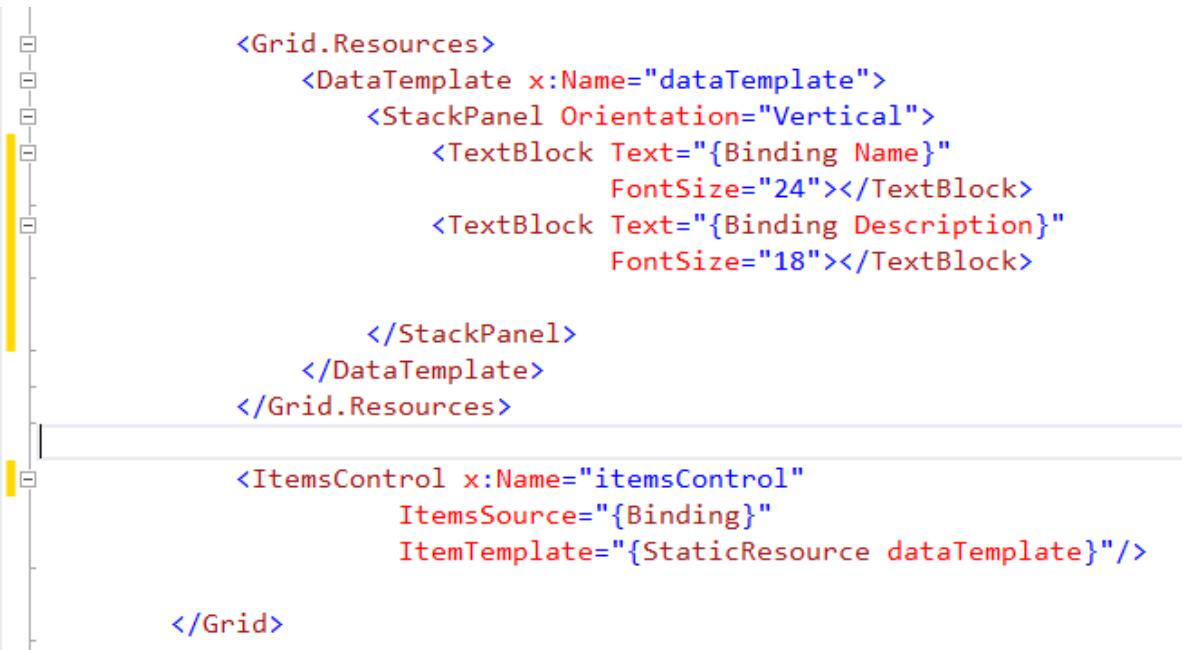
    <ItemsControl x:Name="itemsControl"
        ItemsSource="{Binding}"
        ItemTemplate="{StaticResource dataTemplate}"/>

```

I

```
</Grid>
```

Next, we'll add the ItemTemplate called dataTemplate. We implement it as a child to the Grid in the Grid's Resources property. It consists of a StackPanel that arranged two TextBlocks vertically. The TextBlocks are bound to the Name and Description of a given Ritual object:



```
<Grid.Resources>
    <DataTemplate x:Name="dataTemplate">
        <StackPanel Orientation="Vertical">
            <TextBlock Text="{Binding Name}"
                FontSize="24"></TextBlock>
            <TextBlock Text="{Binding Description}"
                FontSize="18"></TextBlock>
        </StackPanel>
    </DataTemplate>
</Grid.Resources>

    <ItemsControl x:Name="itemsControl"
        ItemsSource="{Binding}"
        ItemTemplate="{StaticResource dataTemplate}"/>

</Grid>
```

We'll add a Button control to the DataTemplate. We'll utilize the Button later, but for now it's just for layout purposes:

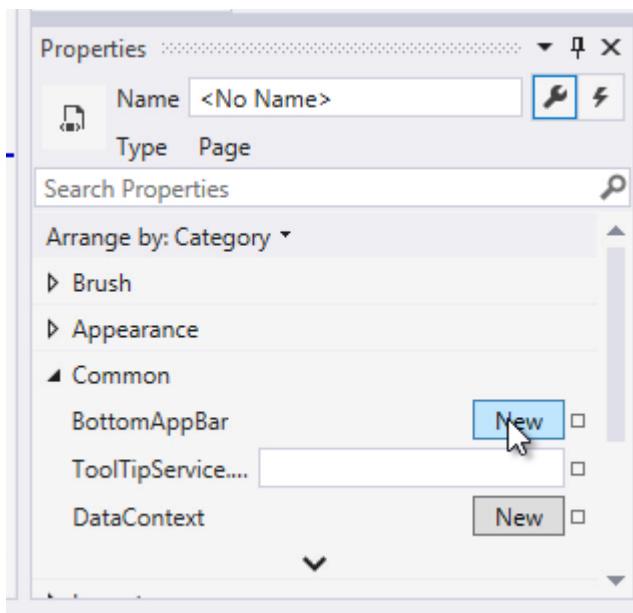
```

<Grid.Resources>
    <DataTemplate x:Name="dataTemplate">
        <StackPanel Orientation="Vertical">
            <TextBlock Text="{Binding Name}"
                       FontSize="24"></TextBlock>
            <TextBlock Text="{Binding Description}"
                       FontSize="18"></TextBlock>

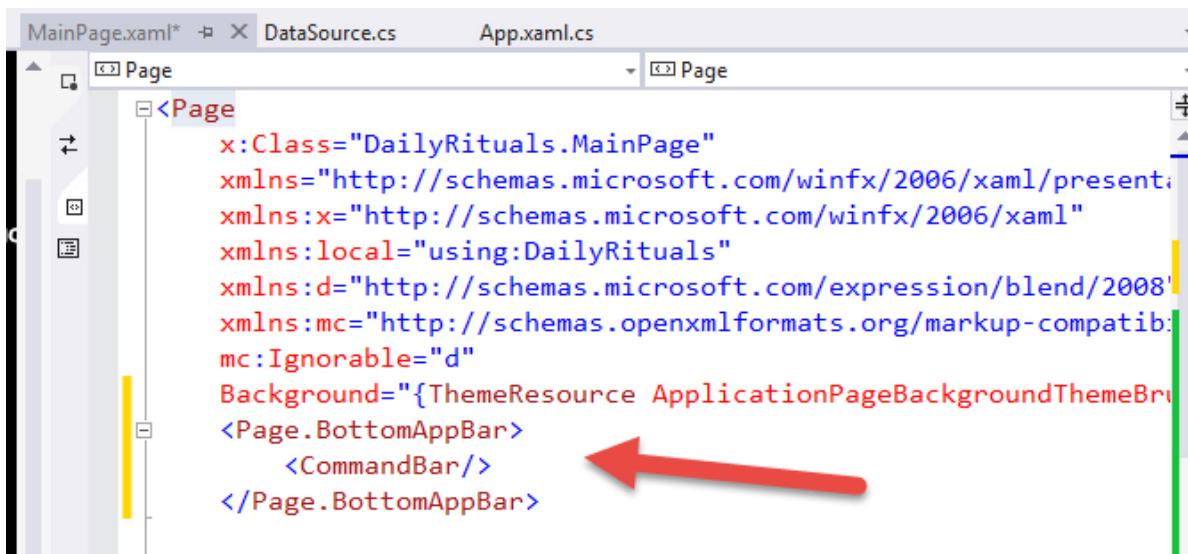
            <Button Name="CompletedButton"
                   Content="I Did this Today!"
                   />
        </StackPanel>
    </DataTemplate>
</Grid.Resources>

```

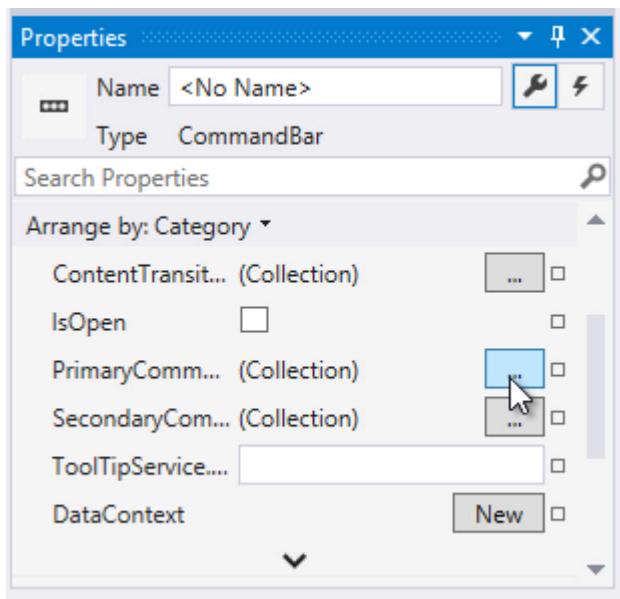
In order to add a new Ritual, we'll need a CommandBar with a Primary Command Button. To begin, put your mouse cursor in the Page element (top or bottom) and in the Properties window, select the New button next to BottomAppBar:



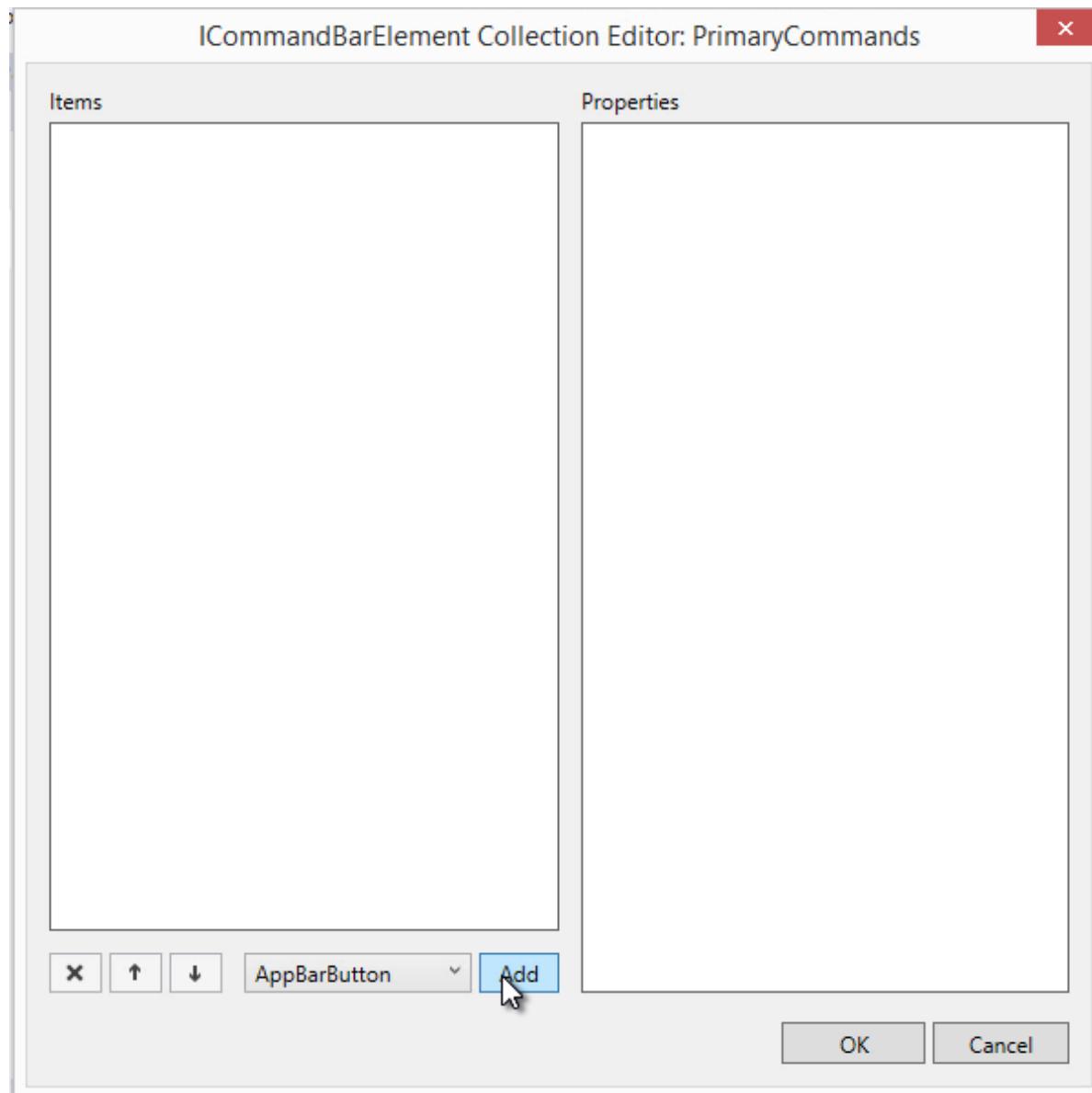
This will create a CommandBar object as a child to the Page's BottomAppBar property:



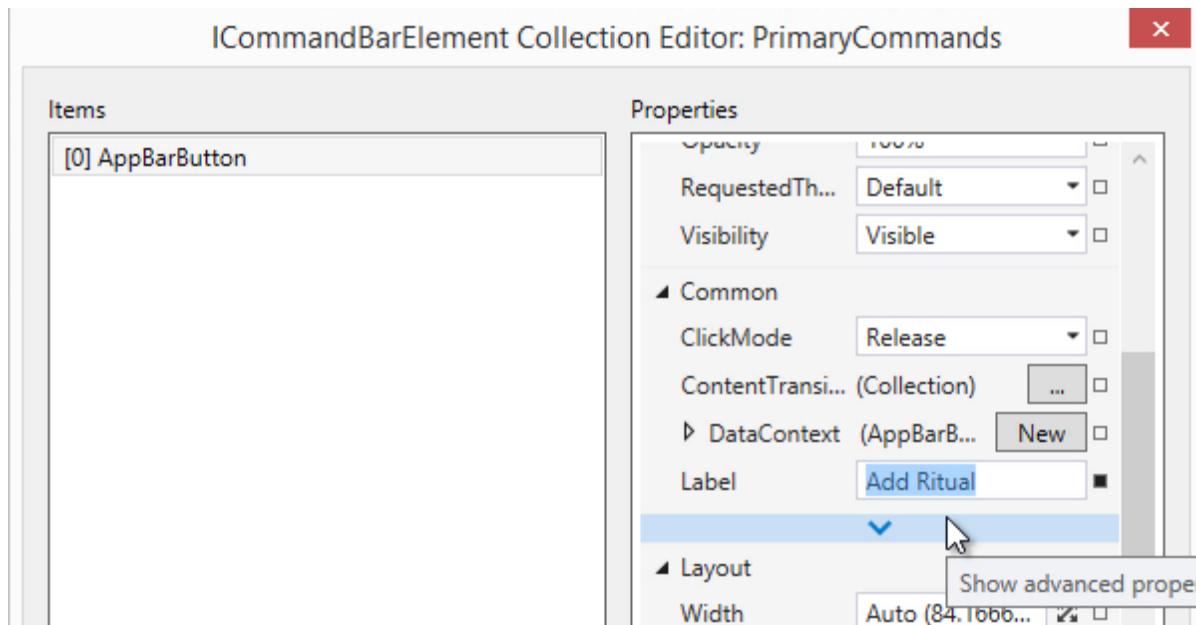
Put your mouse cursor in the CommandBar element and in the Properties pane click the ellipsis button next to the PrimaryCommands property:



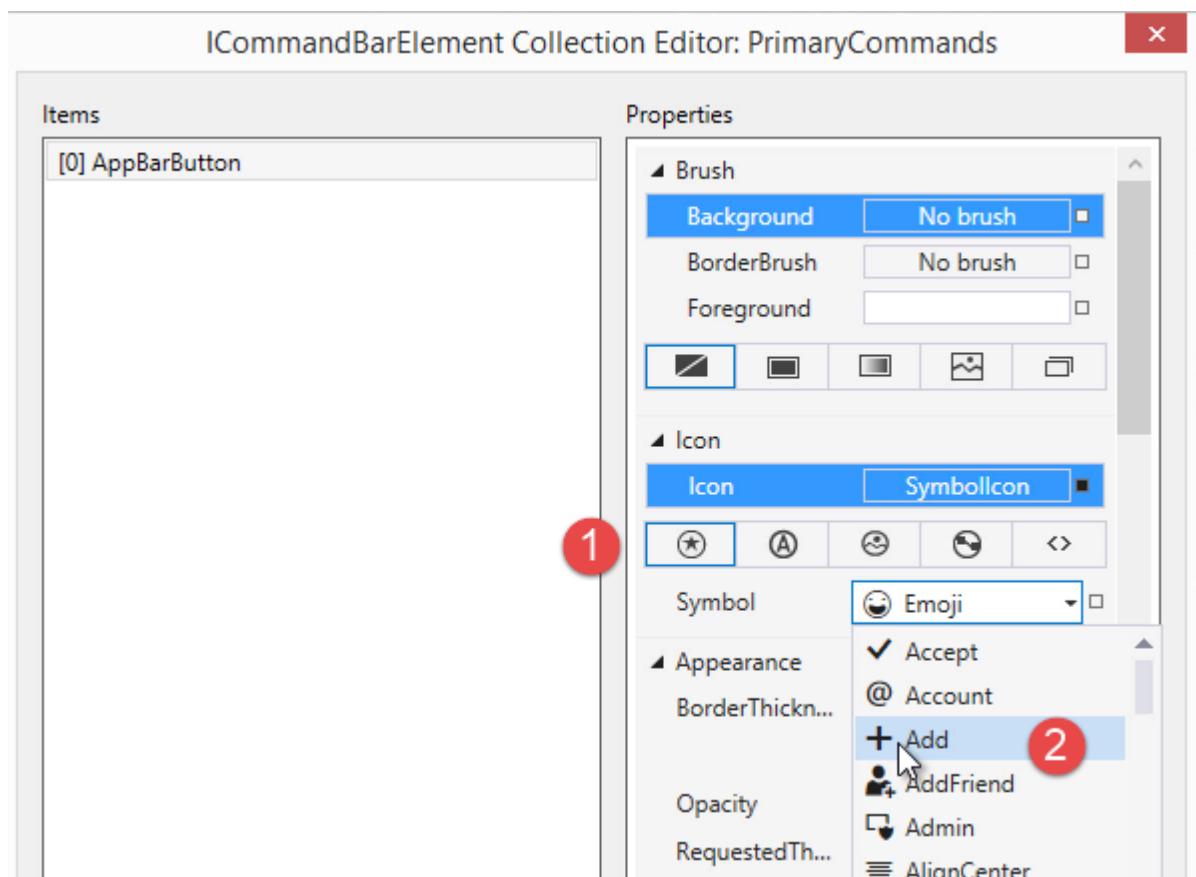
This will display an editor allowing you to add app bar buttons. Click the Add button to create your first app bar button:



The Properties pane on the right allows you to modify properties of the selected app bar button on the left. Here we'll change the label to: Add Ritual ...



... and the Icon to (1) an Icon (2) set to the + Add symbol:



To close the editor dialog, click the OK button at the bottom. This will have created an AppBarButton element. We'll add a Name and Click attribute like so:

```
<AppBarButton Icon="Add" Label="Add Ritual" Name="AddRitual" Click="AddRitual_Click"/>
```

Put your mouse cursor in the AddRitual_Click event handler name and select the F12 keyboard key to create a method stub. When a user clicks the Add button in the command bar, we want to navigate to our new AddRitual.xaml page, so we'll add the following code:

```
private void AddRitual_Click(object sender, RoutedEventArgs e)
{
    Frame.Navigate(typeof(AddRitual));
}
```

Next, in the AddRitual.xaml page, I'll paste in the following XAML inside of the Grid element:

```
<TextBlock HorizontalAlignment="Left"
    Margin="34,161,0,0"
    TextWrapping="Wrap"
    Text="Goal Description:"
    VerticalAlignment="Top"/>

<TextBox x:Name="goalDescriptionTextBox"
    HorizontalAlignment="Left"
    Margin="34,170,0,0"
    TextWrapping="Wrap"
    VerticalAlignment="Top"
    Height="125"
    Width="332"/>

<TextBox x:Name="goalNameTextBox"
    HorizontalAlignment="Left"
    Margin="34,98,0,0"
    TextWrapping="Wrap"
    VerticalAlignment="Top"
    Width="332"/>

<TextBlock HorizontalAlignment="Left"
    Margin="34,89,0,0"
    TextWrapping="Wrap"
    Text="Goal Name:"
```

```

        VerticalAlignment="Top"/>

<Button x:Name="addButton"
        Content="Add"
        HorizontalAlignment="Left"
        Margin="34,322,0,0"
        VerticalAlignment="Top"
        Click="addButton_Click" />

```

Which creates the following preview:



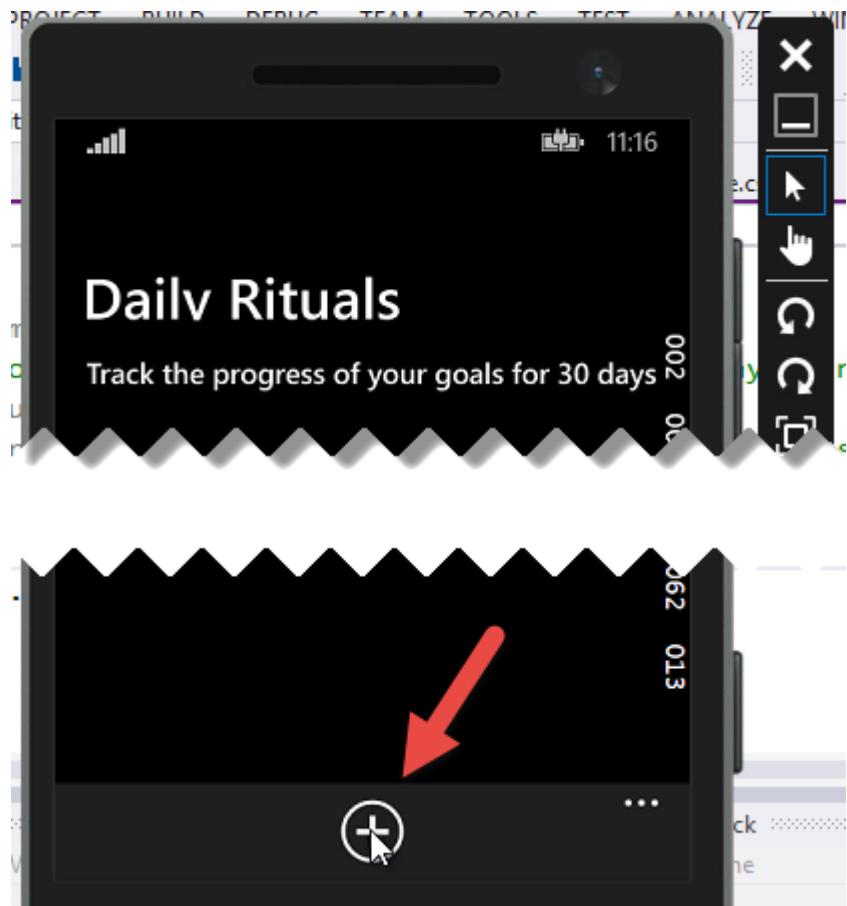
Next, we'll need to write code to handle the click event of the Add button. We'll want to (a) add the new Ritual to the collection of rituals in memory, and save it to the Phone's storage, then (b) navigate back to the MainPage.xaml. Put your mouse cursor on the addButton_Click event handler name and press the F12 key on your keyboard to create the method stub. I'll add the following two lines of code to the method:

```

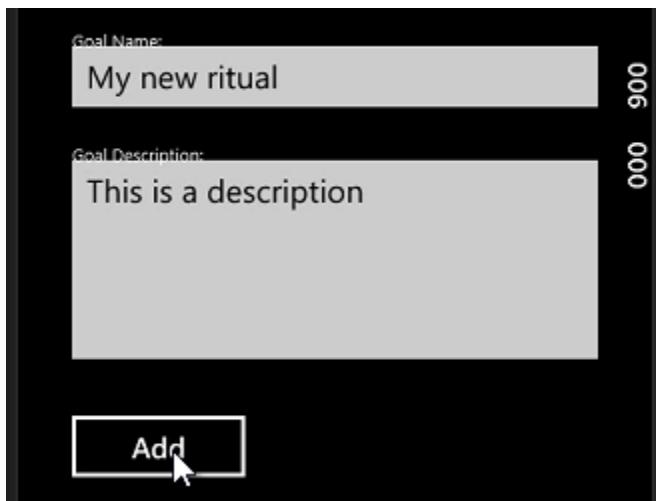
private void addButton_Click(object sender, RoutedEventArgs e)
{
    // Save this new ritual
    App.DataModel.AddRitual(goalNameTextBox.Text, goalDescriptionTextBox.Text);
    Frame.Navigate(typeof(MainPage));
}

```

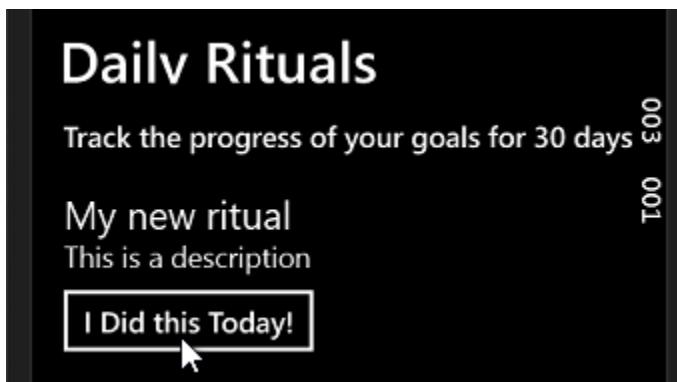
Now when I run the app, I can click the Add button in the command bar:



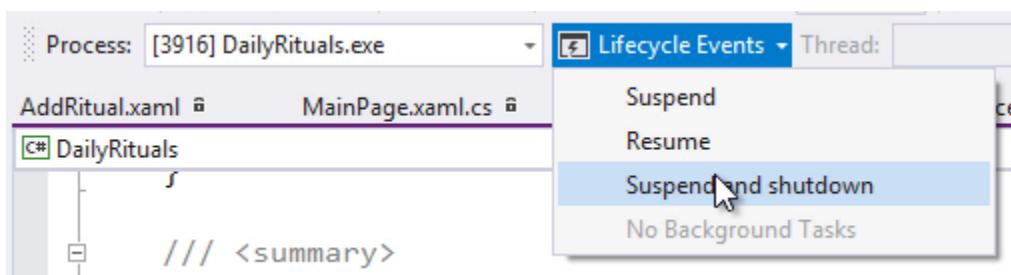
... I can enter a Goal Name and Goal Description, then click the Add button:



And when I return to the MainPage.xaml, I should be able to see the ritual I just added.



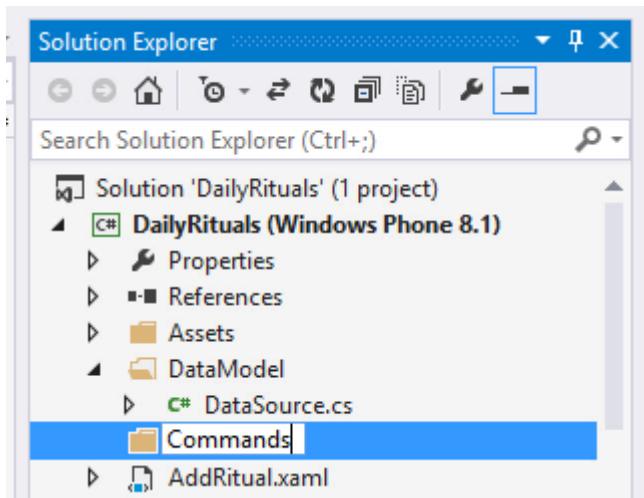
To ensure that we're saving the data to the Phone's local storage, I'll go to Visual Studio while the app is running in the emulator and select the drop down list of Lifecycle Events and choose "Suspend and shutdown":



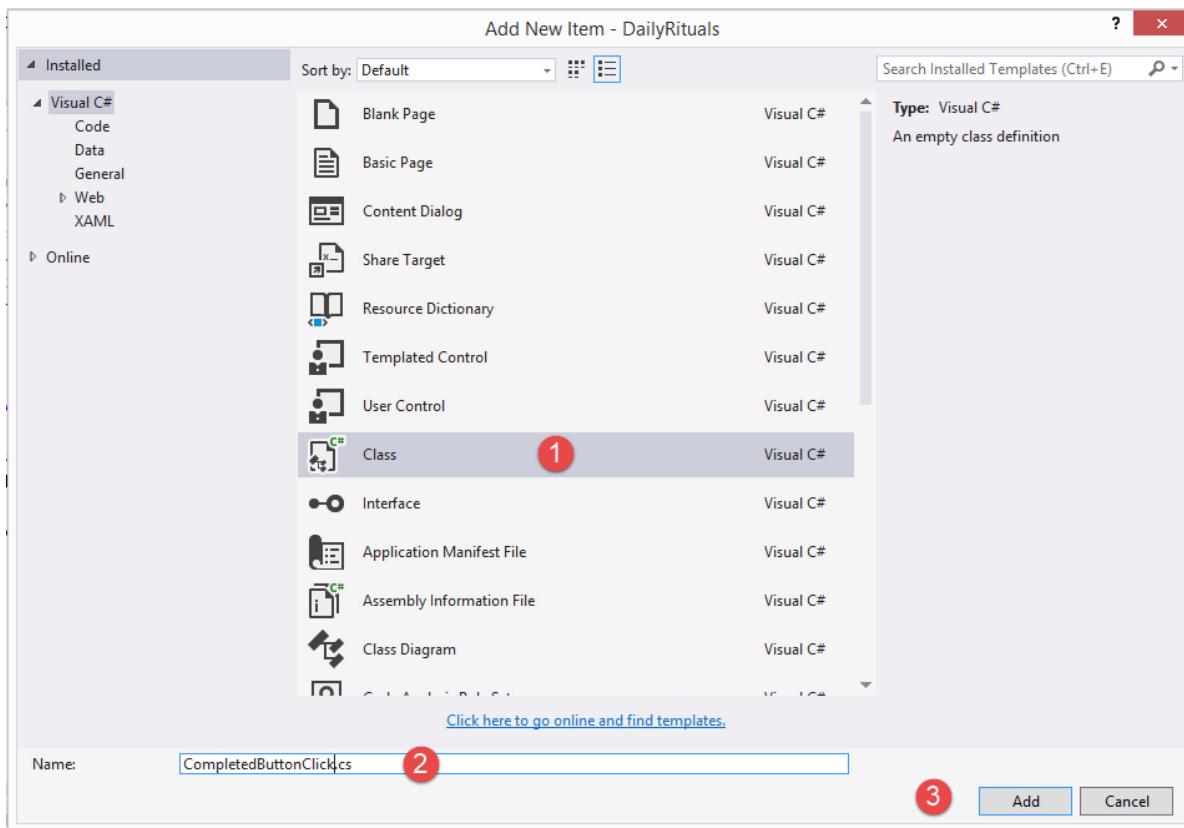
When I re-start the app, the new ritual should still be present.

Next, we'll want to wire up the "I Did This Today" button so that, when a user clicks that button, it will record today's date for the associated Ritual.

To begin, I'll add a new Command folder:



Then I'll add a new class. In the Add New Item dialog, (1) select Class, (2) rename to CompletedButtonClick, then (3) select the Add button:



In the new CompletedButtonClick class, you'll make it implement the ICommand interface like we learned in a previous lesson. To accomplish this, first you'll need to add a using statement using the Control + [dot] technique, then selecting the first option from the Intellisense menu:

The screenshot shows a code editor in Visual Studio with the file `CompletedButtonClick.cs` open. The code defines a class `CompletedButtonClick` that implements the `ICommand` interface. The interface implementation is currently empty, consisting of two curly braces. A tooltip menu is displayed over the closing brace of the implementation block, listing options to implement the interface: "using System.Windows.Input;" (selected), "System.Windows.Input.ICommand", "Generate class for 'ICommand'", and "Generate new type...".

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DailyRituals.Commands
{
    class CompletedButtonClick : ICommand
    {
    }
}
```

Then you'll need to use the Control + [dot] technique a second time to display the Intellisense option to "Implement interface 'ICommand'".

The screenshot shows the same code editor with the tooltip menu still open. The option "Implement interface 'ICommand'" has been selected, highlighted in blue. This selection triggers the generation of stubbed-out implementation code for the `CanExecuteChanged` event and the `CanExecute()` and `Execute()` methods.

```
namespace DailyRituals.Commands
{
    class CompletedButtonClick : ICommand
    {
    }
}
```

Once you've selected this, the `CanExecuteChanged` event, as well as the `CanExecute()` and `Execute()` methods will be stubbed out for you:

```

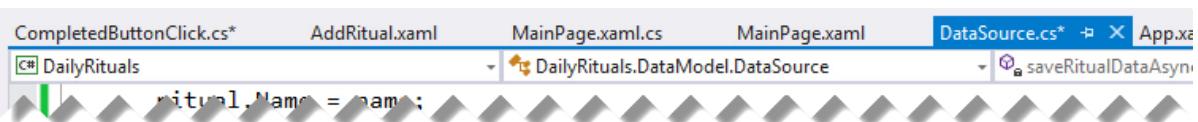
namespace DailyRituals.Commands
{
    class CompletedButtonClick : ICommand
    {
        public bool CanExecute(object parameter)
        {
            throw new NotImplementedException();
        }

        public event EventHandler CanExecuteChanged;
        [

        public void Execute(object parameter)
        {
            throw new NotImplementedException();
        }
    }
}

```

Before we develop our own implementation of the Execute() method, we'll need to create a method that our Execute() method will call in our DataSource. To that end, I want to create a method that will accept an instance of Ritual, add today's date to the Dates property (an ObservableCollection<Date>) and then save that change to the Phone's storage. So, I implement the CompleteRitualToday() method as follows:



```

CompletedButtonClick.cs*      AddRitual.xaml      MainPage.xaml.cs      MainPage.xaml      DataSource.cs*      App.xaml
[C#] DailyRituals           DailyRituals.DataModel.DataSource      saveRitualDataAsync

        ritual.Name = name;
        }

        [
public async void CompleteRitualToday(Ritual ritual)
{
    int index = _rituals.IndexOf(ritual);
    _rituals[index].AddDate();
    await saveRitualDataAsync();
}

```

We'll need to implement the AddDate() method for a given Ritual. I'll add the AddDate() method to the Ritual class definition:

```

namespace DailyRituals.DataModel
{
    public class Ritual
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public ObservableCollection<DateTime> Dates { get; set; }

        public void AddDate()
        {
            Dates.Add(DateTime.Today);           ←
        }
    }
}

```

Now, back in the CompletedButtonClick class, I can (1) comment out the stubbed out code and simply return true always when asked whether this command can be executed, and (2) call the CompleteRitualToday() method I just implemented a moment ago in the DataSource class, passing in the input parameter and casting it to a Ritual. (3) I will have to make sure that I add a using statement for the DataModel namespace by using the Control + [dot] keyboard shortcut to display the Intellisense menu option:

```

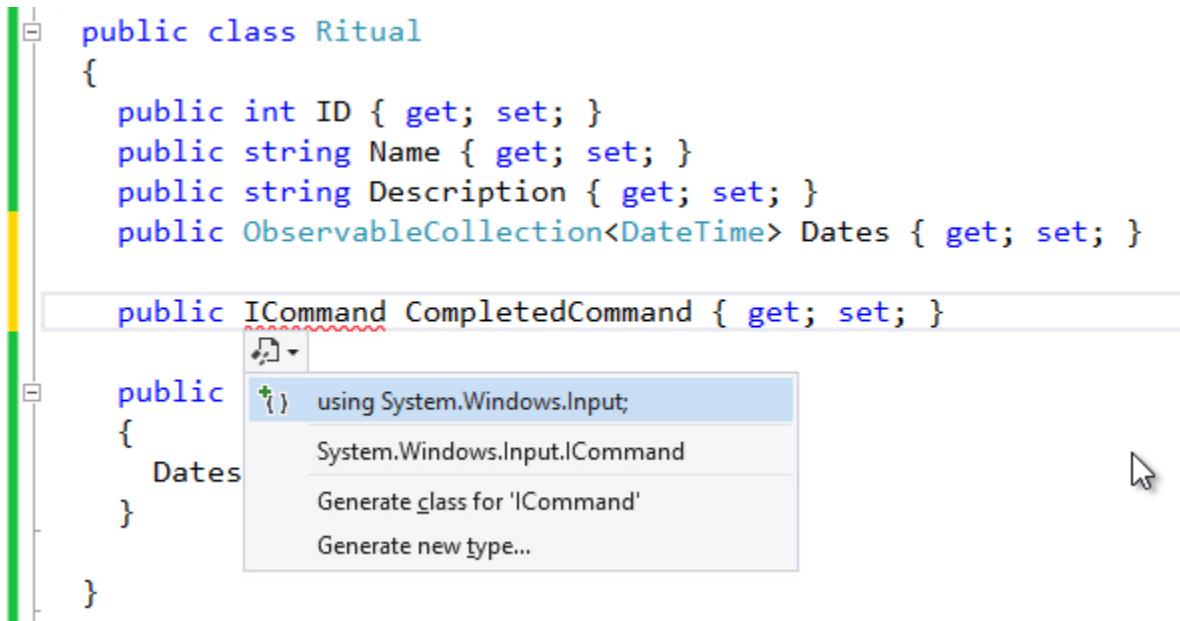
class CompletedButtonClick : ICommand
{
    public bool CanExecute(object parameter)
    {
        //throw new NotImplementedException();
        return true;
    }

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        //throw new NotImplementedException();
        App.DataModel.CompleteRitualToday((Ritual)parameter);
    }
}

```

Now that we've implemented the CompletedButtonClick command, we'll need to add it as a property to our Ritual class. Therefore, any user interface property that binds to Ritual can invoke the command. We'll call our property CompletedCommand which is of type ICommand. Naturally, we'll need to add the using statement for System.Windows.Input using the Control + [dot] keyboard shortcut to display the Intellisense menu option:



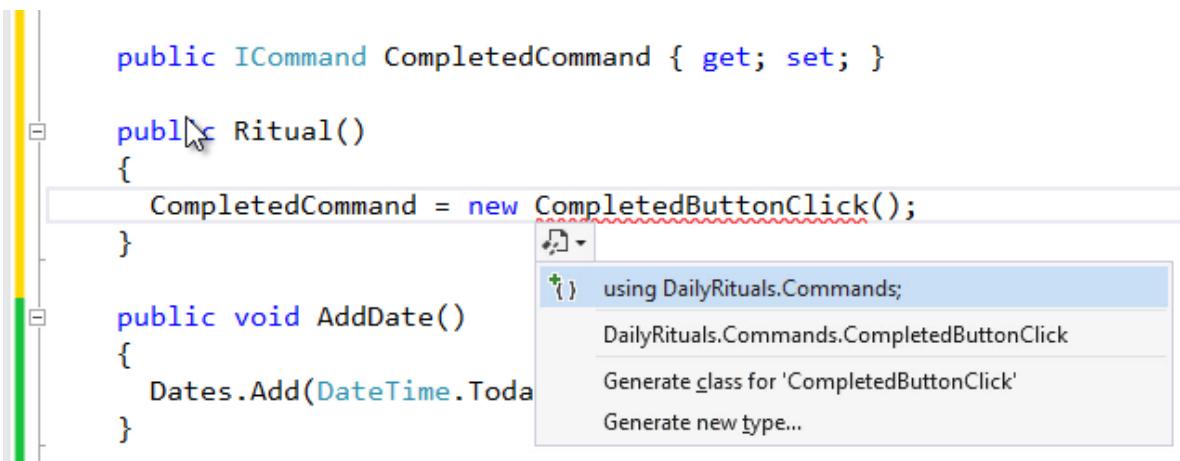
```
public class Ritual
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public ObservableCollection<DateTime> Dates { get; set; }

    public ICommand CompletedCommand { get; set; }
}
```

The Intellisense menu is open at the line `public ICommand CompletedCommand { get; set; }`. The menu shows the following options:

- { } using System.Windows.Input;
- System.Windows.Input.ICommand
- Generate class for 'ICommand'
- Generate new type...

In the Ritual class' constructor, we'll need to create a new instance of CompletedbuttonClick and set it to our new CompletedCommand property. In this way, we wire up the property called by the user interface to the CompletedButtonClick class we created. Again, we'll need to add a using statement to our DailyRitual.Commands namespace using the technique Control + [dot] keyboard shortcut.



```
public ICommand CompletedCommand { get; set; }

public Ritual()
{
    CompletedCommand = new CompletedButtonClick();
}

public void AddDate()
{
    Dates.Add(DateTime.Today)
}
```

The Intellisense menu is open at the line `CompletedCommand = new CompletedButtonClick();`. The menu shows the following options:

- { } using DailyRituals.Commands;
- DailyRituals.Commands.CompletedButtonClick
- Generate class for 'CompletedButtonClick'
- Generate new type...

Finally, in the constructor I'll need to create a new instance of ObservableCollection<DateTime> and set it equal to Dates to properly initialize that property as well:

```
public Ritual()
{
    CompletedCommand = new CompletedButtonClick();
    Dates = new ObservableCollection<DateTime>();
}
```

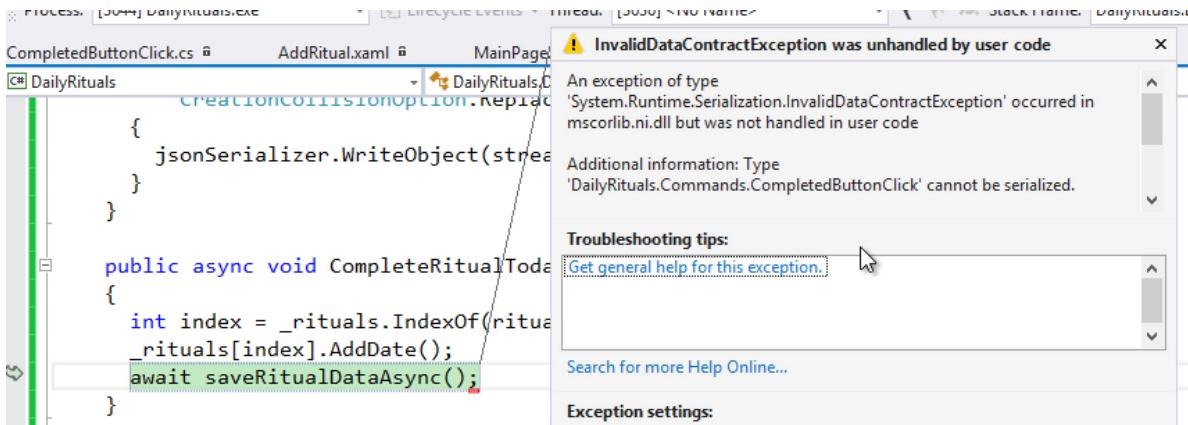
Now that I have everything in my data model wired up correctly for the new Command, I can use it in my XAML. I'll update the MainPage's CompletedButton by binding the Command to the CompletedCommand of the Ritual and binding the CommandParameter to the Ritual itself so that it gets properly sent into the Command and we know which Ritual the invoked command belongs to:

```
<Grid.Resources>
    <DataTemplate x:Name="dataTemplate">
        <StackPanel Orientation="Vertical">
            <TextBlock Text="{Binding Name}"
                       FontSize="24"></TextBlock>
            <TextBlock Text="{Binding Description}"
                       FontSize="18"></TextBlock>

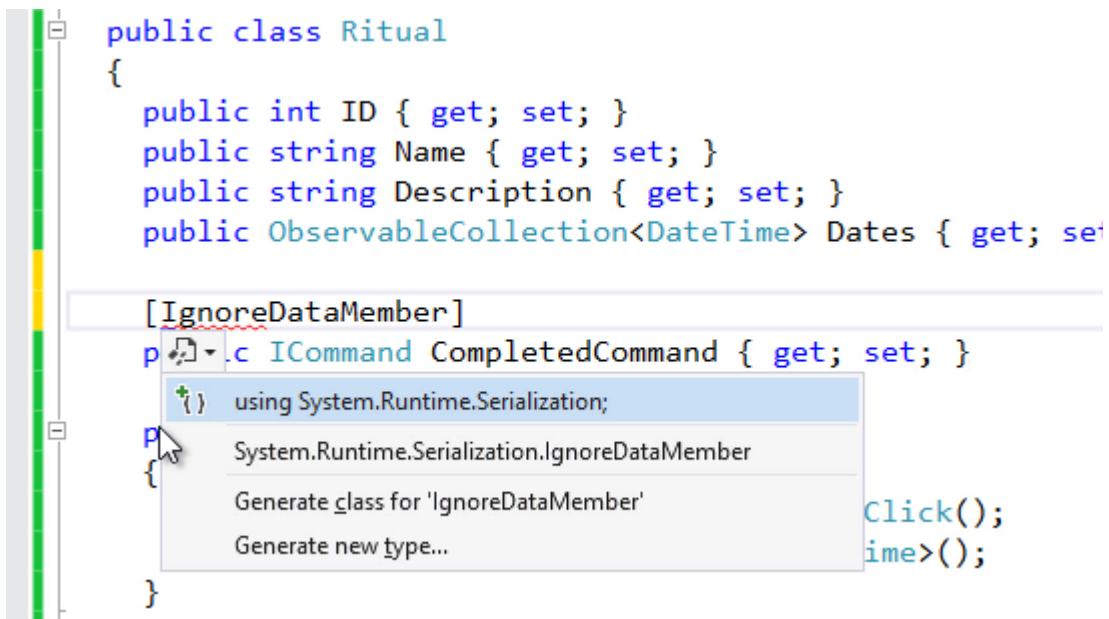
            <Button Name="CompletedButton"
                   Content="I Did this Today!"
                   Command="{Binding CompletedCommand}"
                   CommandParameter="{Binding}"
                   />
        </StackPanel>
    </DataTemplate>
</Grid.Resources>
```



However, when we run the application, we'll get an error clicking the "I Did This Today" button for a given Ritual. Why? Because theDataContractJsonSerializer does not know what to do with our new CompletedCommand property ... it tried to persist it to JSON format, but it could not:



Therefore, we'll need to adorn the CompletedCommand property with an attribute that will tell the DataContractJsonSerializer to ignore this property when serializing an instance of Ritual to JSON. We'll add the [IgnoreDataMember] attribute above the CompletedCommand property. This will require we add the proper using statement (using the technique you should now be familiar with):

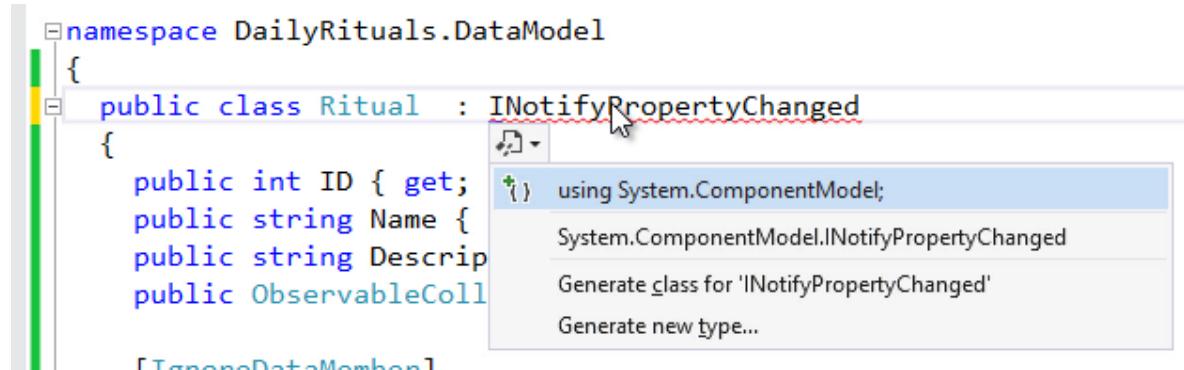


Important: Once you make this change, you will need to reset the data stored for the app because it corrupted. To do this, simply shut down the emulator completely. The next time you run your application in debug mode, the emulator will be refreshed and your old data will be removed. Your app should now work correctly.

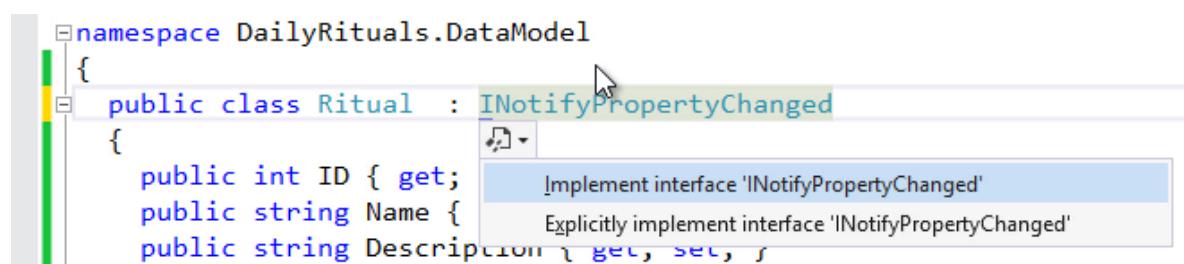
Next, we'll want to disable the "I Did This Today" button once it has been clicked. Furthermore, it should be disabled when we re-run the application a second and subsequent time on the

same day. Therefore, we'll need to (a) implement `INotifyPropertyChanged` on the `Ritual` class to notify the user interface when the `Dates` property, an `ObservableCollection` of `DateTime` instances, contains today's date, and (b) we'll need to create a value converter that will be used to tell the user interface whether or not to enable the button based on whether or not the `Dates` property contains today's date.

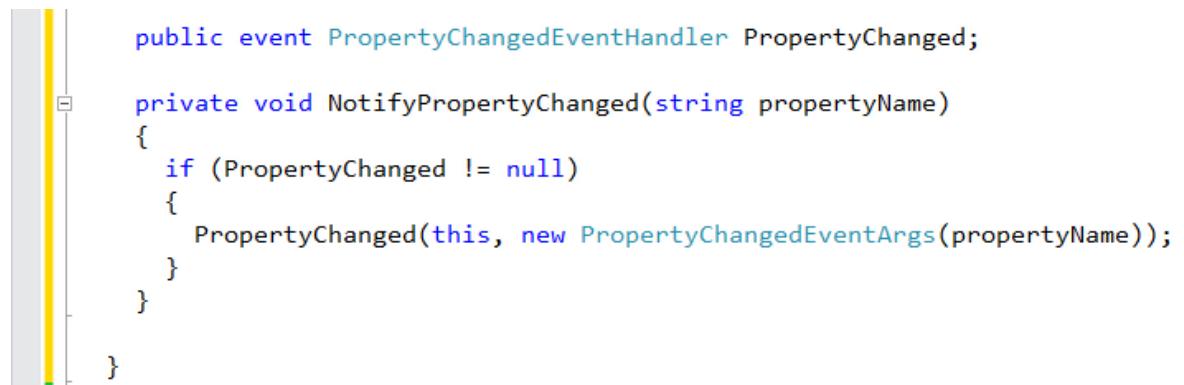
We'll start by implementing the `INotifyPropertyChanged` interface on the `Ritual` class:



And I'll choose the keyboard shortcut to display Intellisense and implement the `INotifyPropertyChanged` interface:



Once implemented, we'll want to also add our helper method `NotifyPropertyChanged()` like so:



Finally, we'll want to fire the event by calling NotifyPropertyChanged() from the AddDate() method. We'll pass in the name of the property that caused the event to fire (i.e., "Dates"):

```
public void AddDate()
{
    Dates.Add(DateTime.Today);
    NotifyPropertyChanged("Dates");
}
```

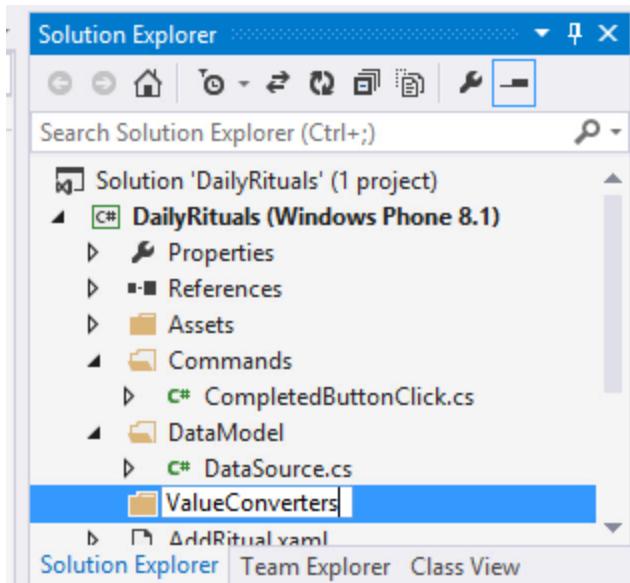


To prepare for the value converter, I'll add the IsEnabled="False" attribute / value setting. Soon, we'll bind the value to a new Value Converter that will determine whether this should be true or false, depending on whether today's date is in the Dates ObservableCollection<Date>:

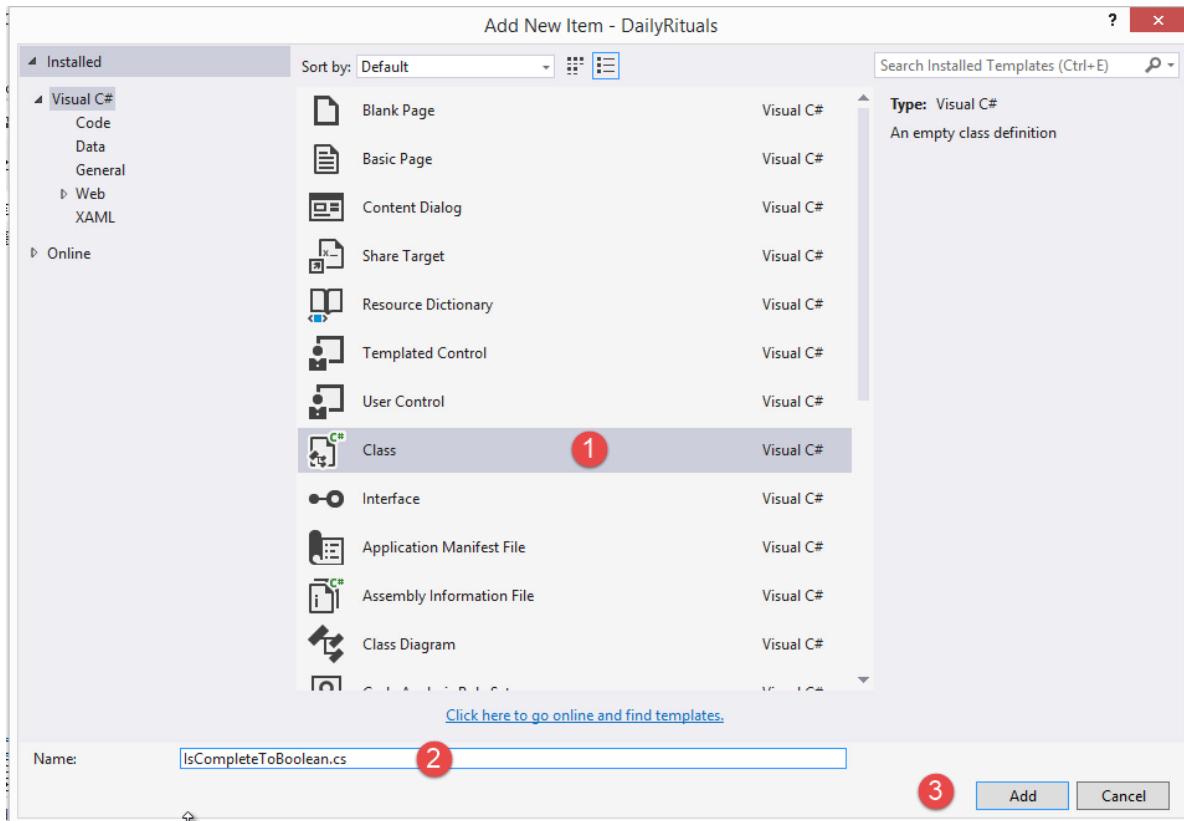
```
<DataTemplate x:Name="dataTemplate">
    <StackPanel Orientation="Vertical">
        <TextBlock Text="{Binding Name}" FontSize="24"></TextBlock>
        <TextBlock Text="{Binding Description}" FontSize="16"></TextBlock>
        <Button Name="CompletedButton"
               Content="I Did this Today!"
               Command="{Binding CompletedCommand}"
               CommandParameter="{Binding}"
               IsEnabled="False"
               />
    </StackPanel>
</DataTemplate>
```



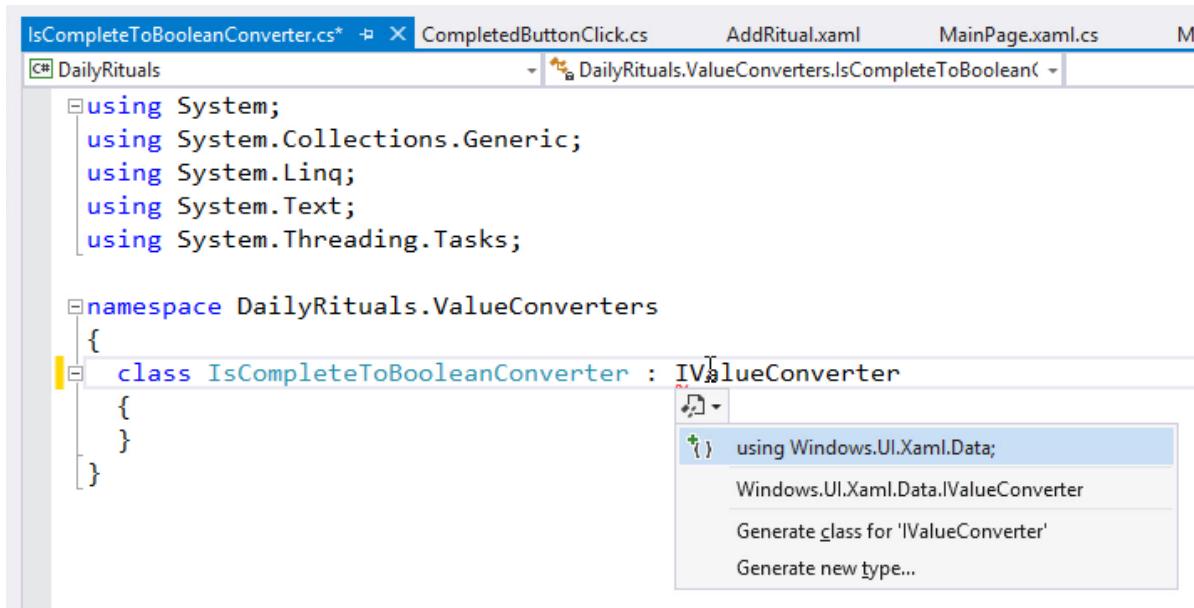
I'll create a new folder called ValueConverters:



... and I'll add a new item into that folder, specifically (1) a Class (2) named IsComplete.ToBoolean.cs, and then I'll (3) click the OK button in the Add New Item dialog to complete this operation:



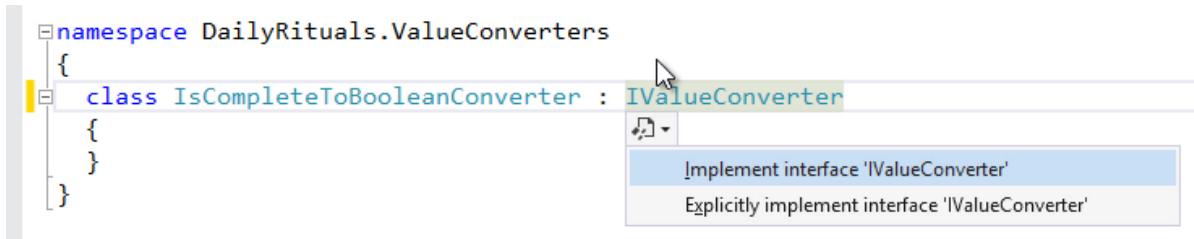
First, I'll need to implement the `IValueConverter` interface for this new class. I'll use the usual technique to first add a using statement:



A screenshot of the Visual Studio IDE showing the code editor for `IsComplete.ToBooleanConverter.cs`. The cursor is at the end of the line `: IValueConverter`. A context menu is open with the following options:

- + using Windows.UI.Xaml.Data;
- Windows.UI.Xaml.Data.IValueConverter
- Generate class for 'IValueConverter'
- Generate new type...

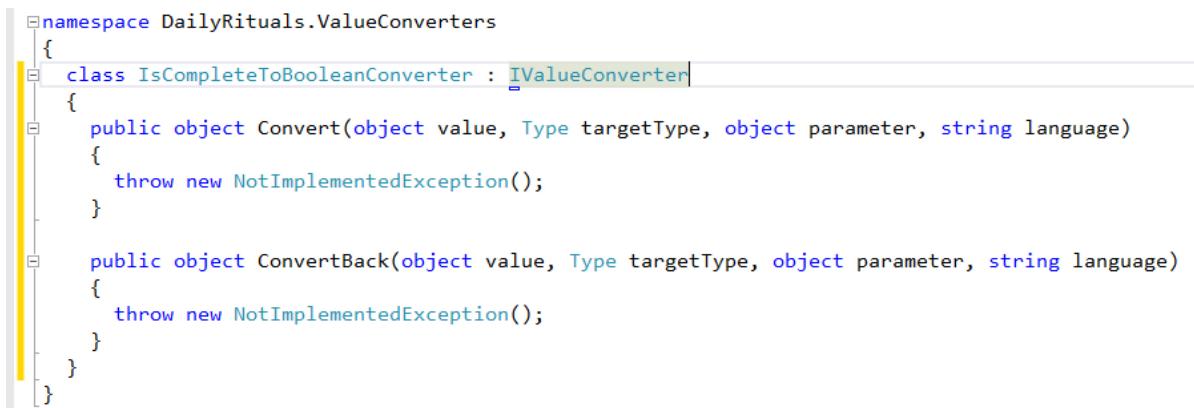
... and the same Control + [dot] technique to implement the interface:



A screenshot of the Visual Studio IDE showing the code editor for `IsComplete.ToBooleanConverter.cs`. The cursor is at the end of the line `: IValueConverter`. A context menu is open with the following options:

- Implement interface 'IValueConverter'
- Explicitly implement interface 'IValueConverter'

... which should produce the following stubbed out implementation:



A screenshot of the Visual Studio IDE showing the code editor for `IsComplete.ToBooleanConverter.cs`. The class definition is as follows:

```
namespace DailyRituals.ValueConverters
{
    class IsComplete.ToBooleanConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, string language)
        {
            throw new NotImplementedException();
        }

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

We'll only implement the Convert() method. Here we expect the Dates property to be passed in as value. I'll first need to cast the value parameter to an ObservableCollection<DateTime>. Then, I'll merely call the Contains() extension method to determine whether DateTime.Today is part of that collection. If it is, return false (because we want to set the Button's IsEnabled property to False, since we already clicked it once today) and if it is not present, then return true (because we want to set the Button's IsEnabled property true since we have NOT clicked the button today).

```
class IsComplete.ToBooleanConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        ObservableCollection<DateTime> dates = (ObservableCollection<DateTime>)value;
        if (dates.Contains(DateTime.Today))
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

Now, back in the MainPage.xaml, we'll add a new prefix for the ValueConverters namespace we created when we added the ValueConverters folder, and we'll add a Page.Resources section, and a new <valueconverter> instance, giving it a key of the same name.

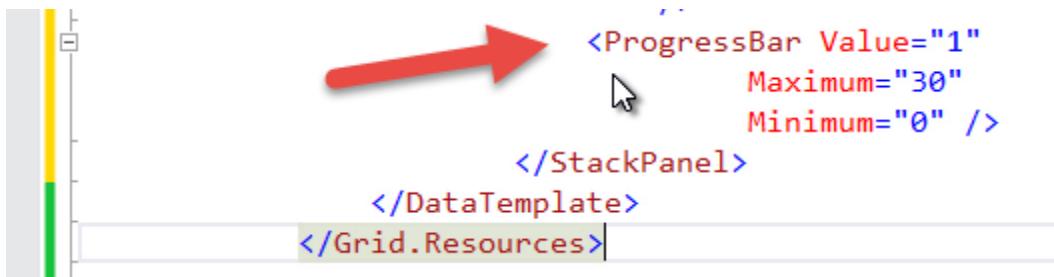
```
<Page
    x:Class="DailyRituals.MainPage"
    ...
    xmlns:valueconverter="using:DailyRituals.ValueConverters"
    ...
    >
    <Page.Resources>
        <valueconverter:IsComplete.ToBooleanConverter x:Key="IsComplete.ToBooleanConverter" />
        <valueconverter:CompletedDatesToIntegerConverter
x:Key="CompletedDatesToIntegerConverter" />
    </Page.Resources>
```

Below that, we'll modify our Button like so:

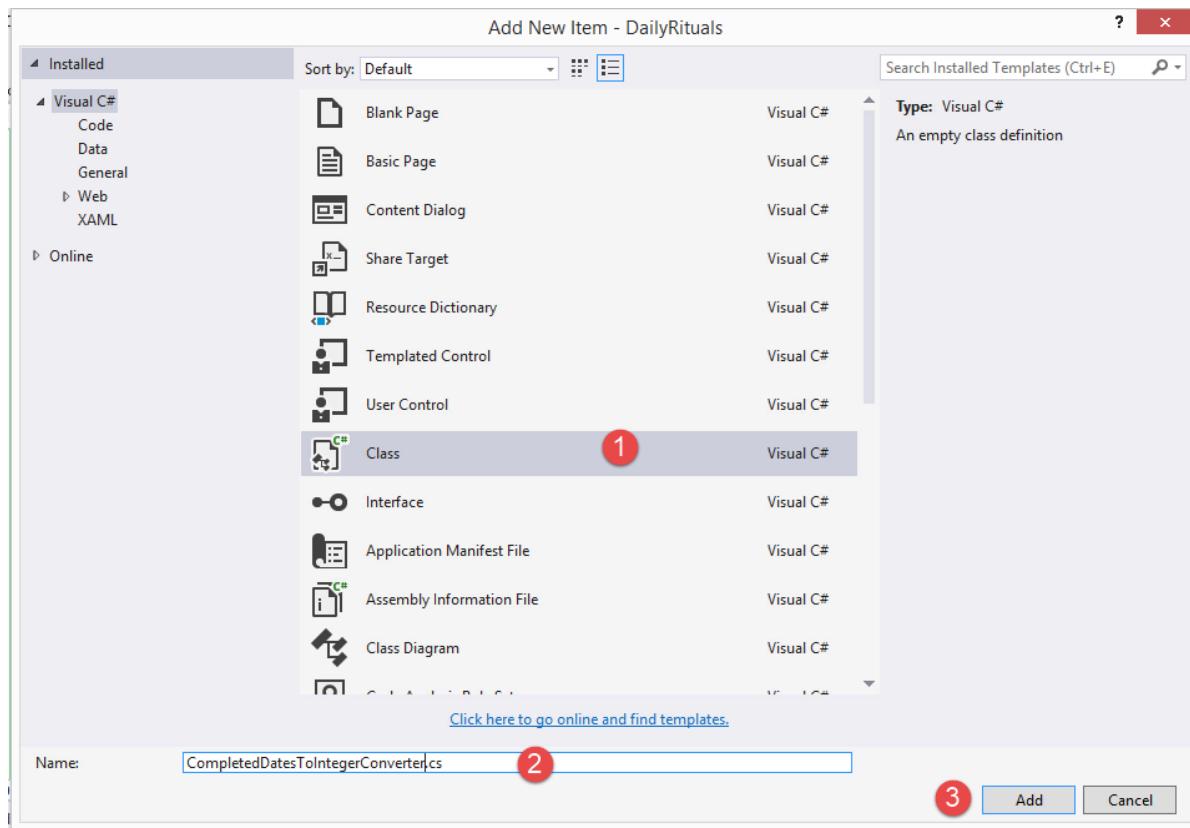
```
<Button Name="CompletedButton"
    Content="I Did this Today!"
    Command="{Binding CompletedCommand}"
    CommandParameter="{Binding}"
    IsEnabled="{Binding Dates, Converter={StaticResource IsComplete.ToBooleanConverter}}"
    />
```

The key here is the `.IsEnabled` property which is bound to the `Dates` collection, but converted using our new Value Converter.

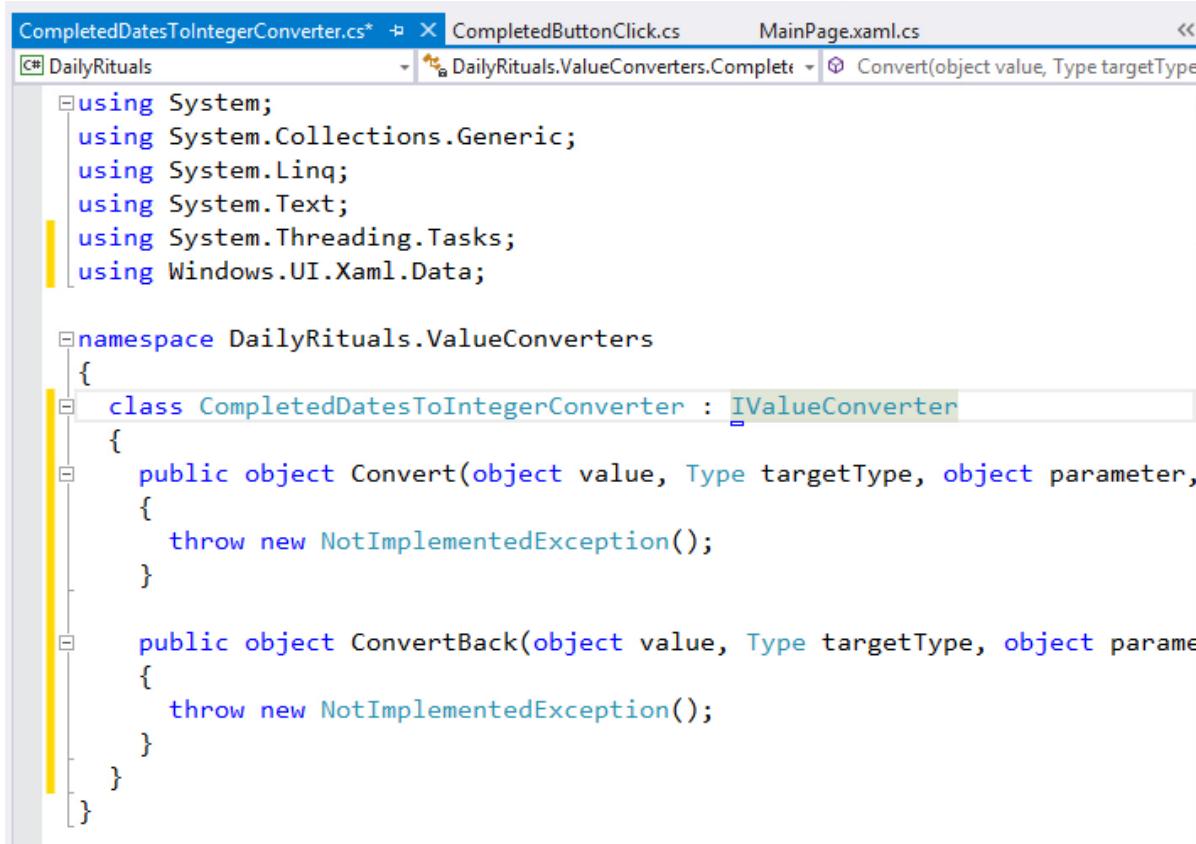
The final feature we'll add is the progress bar. We want to add a `ProgressBar` below the button. We'll hard wire the `Value` attribute to 1, but we'll work on a Value Converter that will bind the `Value` to a count of Dates for a given Ritual which will allow us to see the progress against our Goal:



In the `ValueConverter` folder, we'll Add a new Item. In the Add New Item dialog, we'll add (1) a new Class (2) named `CompletedDatesToIntegerConverter.cs` and (3) click the Add button:



We'll use the techniques described earlier to implement the `IValueConverter` interface:



The screenshot shows a Visual Studio code editor with the file 'CompletedDatesToIntegerConverter.cs' open. The code defines a class 'CompletedDatesToIntegerConverter' that implements the 'IValueConverter' interface. The 'Convert' method throws a 'NotImplementedException', and the 'ConvertBack' method also throws a 'NotImplementedException'. The code uses several namespaces from the .NET framework and Windows Phone libraries.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.UI.Xaml.Data;

namespace DailyRituals.ValueConverters
{
    class CompletedDatesToIntegerConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
        {
            throw new NotImplementedException();
        }

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

In the Convert() method, we'll expect the value to be passed in as an ObservableCollection<Date>. Then, we'll simply return the number of items in that collection:

```
namespace DailyRituals.ValueConverters
{
    class CompletedDatesToIntegerConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, string language)
        {
            ObservableCollection<DateTime> dates = (ObservableCollection<DateTime>)value;

            return dates.Count;
        }
    }
}
```

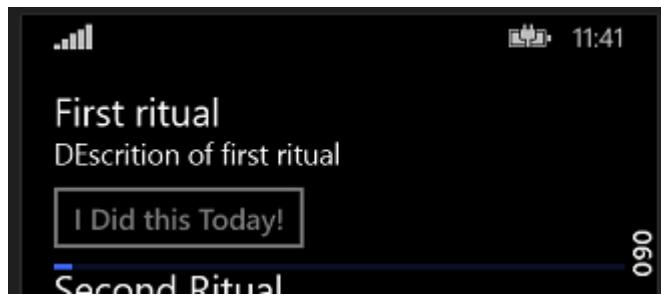
Once we've implemented our converter, we'll add a new reference to it in the Page.Resources section of MainPage:

```
<!-- Other resources -->
<Page.Resources>
    <valueconverter:IsCompleteToBooleanConverter x:Key="IsCompleteToBooleanConverter" />
    <valueconverter:CompletedDatesToIntegerConverter x:Key="CompletedDatesToIntegerConverter" />
</Page.Resources>
```

Then we'll change the Value attribute to bind to the Dates collection converting it to an integer (the count of dates) using our new CompletedDatesToIntegerConverter:

```
<ProgressBar Value="{Binding Dates, Converter={StaticResource CompletedDatesToIntegerConverter}}"  
    Maximum="30"  
    Minimum="0" />
```

Now when we run the app and click the “I Did This Today” button, we can see the button becomes disabled and the ProgressBar creeps up 1/30th of its width:

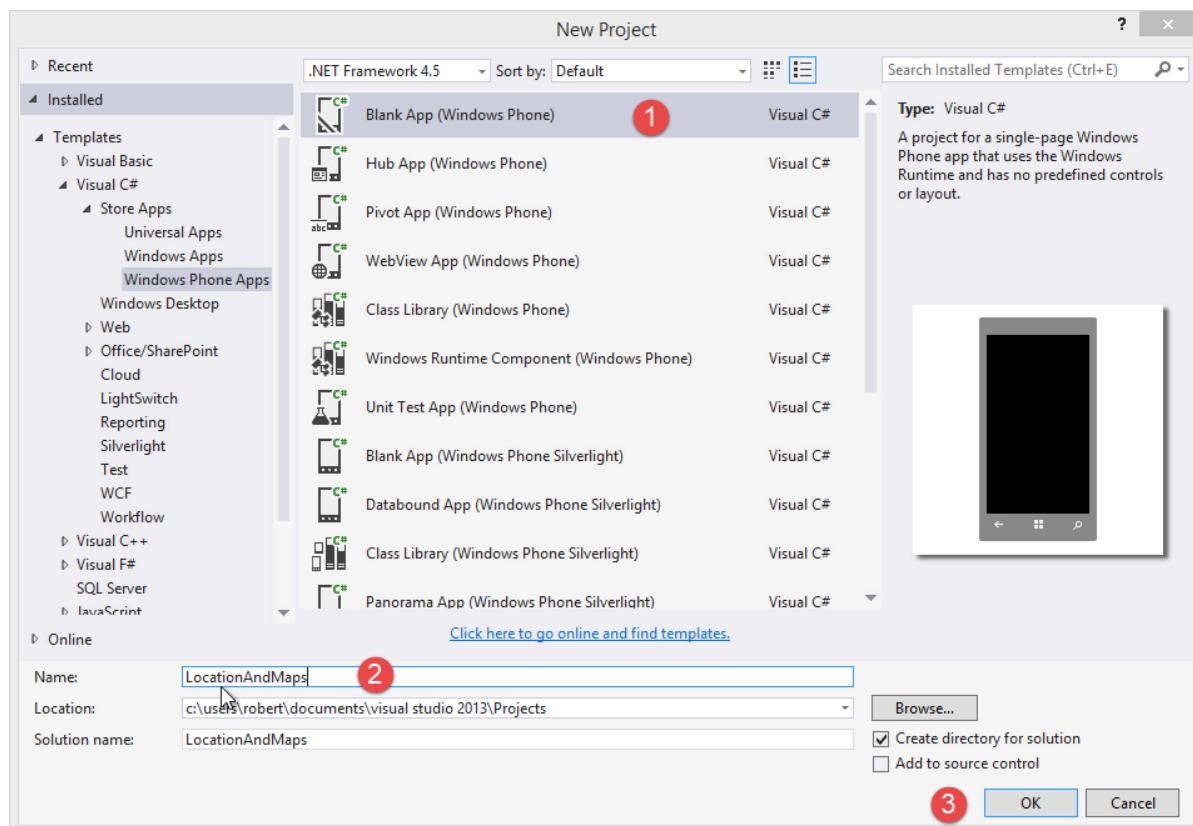


There are still some small improvements we could make to the layout and branding of the app, but we tackled the most challenging aspects from beginning to end. The great thing about this example is that we used many different pieces to we've learned about in previous lessons and you can see how they contributed to the overall functionality we desired.

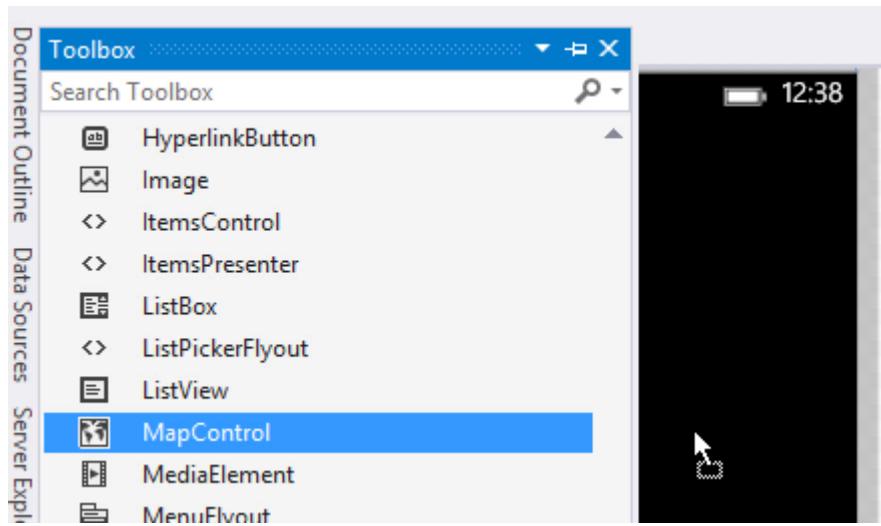
Lesson 27: Working with the Map Control and the Geolocation and GeoPosition Classes

In this lesson, we're going to talk about the map control and talk about the GPS sensor inside of the device. Besides the specifics of the GPS sensor, the more general principles of working with the Phone's sensors apply here as well. So, while we are examining a specific capability, you can apply the general knowledge to all the capabilities of the Phone and how to work with them. The Phone's API always supplies classes and methods representing the particular sensor and its functionality you want to work with. In the case of the Phone's location features, we'll be able to get the latitude and the longitude in a `GeoPosition` object that we can then work with. What can we do with the location, the `GeoPosition` object? For one thing, it works well with the Map Control. We can set current position of the Map Control to a specific location (`GeoPosition`) or retrieve the current location from the Map control and ask "Where in the world are you displaying?" We can zoom into the map, zoom out of the map, and things of that nature.

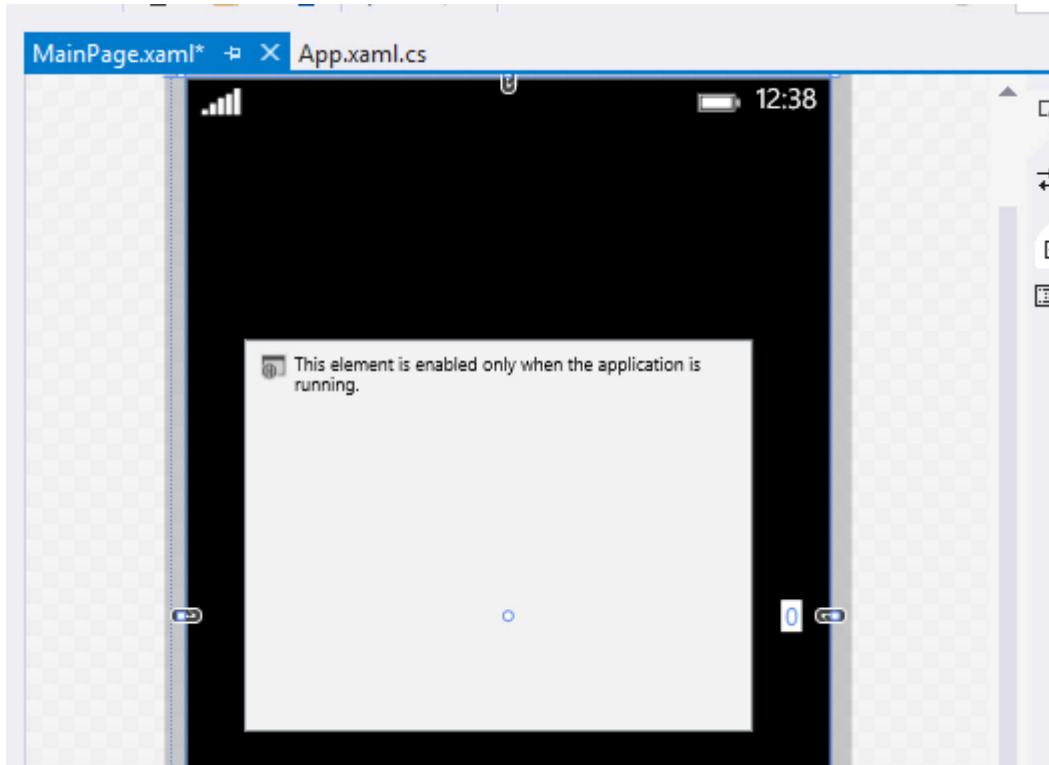
We'll start by opening the New Project dialog (1) creating a new Blank App project, (2) renaming it to `LocationAndMaps`, and (3) clicking the OK button.



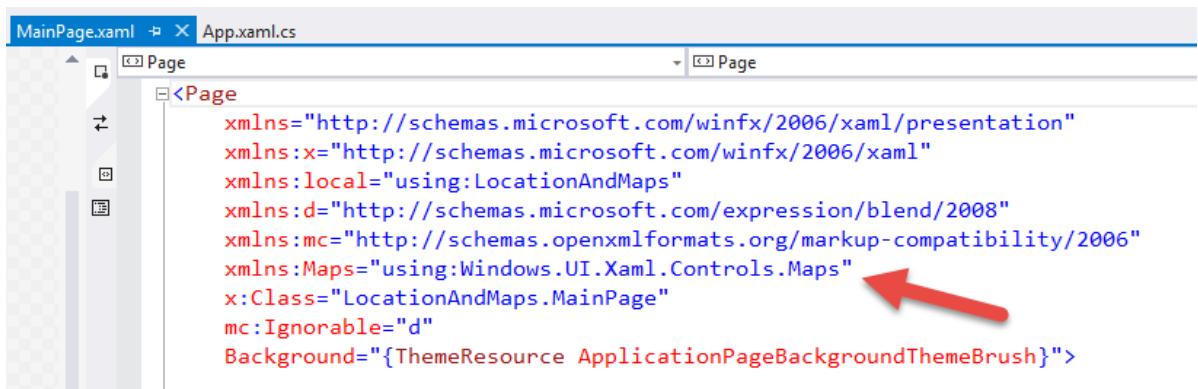
The easiest way to get started working with the Map Control is to use the Toolbox. Usually, I advocate directly typing in the XAML, however as we'll see in a moment, the Map Control requires a bit of extra work that the drag-and-drop action will take care of for us:



Here I resize the Map Control on the design surface so that we can see the message that it only renders when the app is running:



In addition to the XAML that was generated inside of the Grid, notice the namespace that was added to the Page's namespace declarations:



This is one reason why I prefer to drag-and-drop to the design surface when working with the MapControl.

Next, I'll edit the layout of the Grid and the Map. Ideally, it takes up the first row set to *:

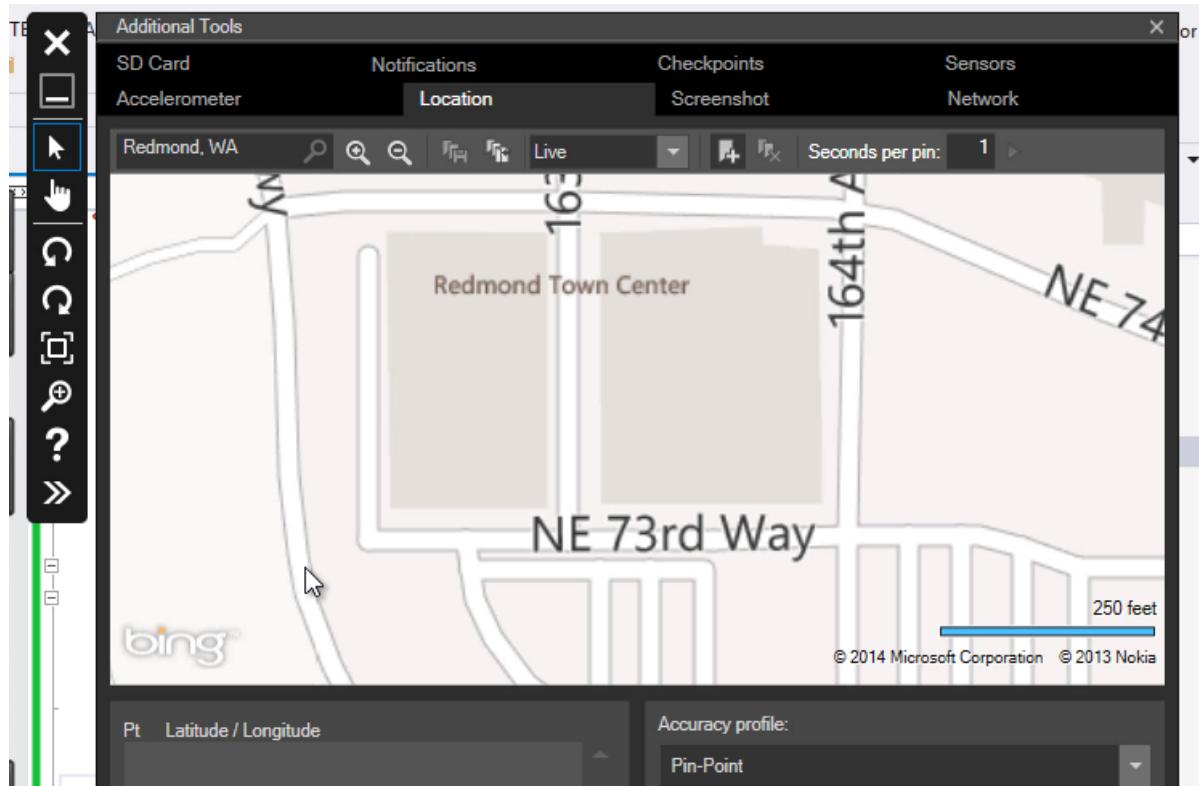
```
<Grid Margin="20,50,20,20">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="50" />
    </Grid.RowDefinitions>

    <Maps:MapControl Name="MyMap" />
```

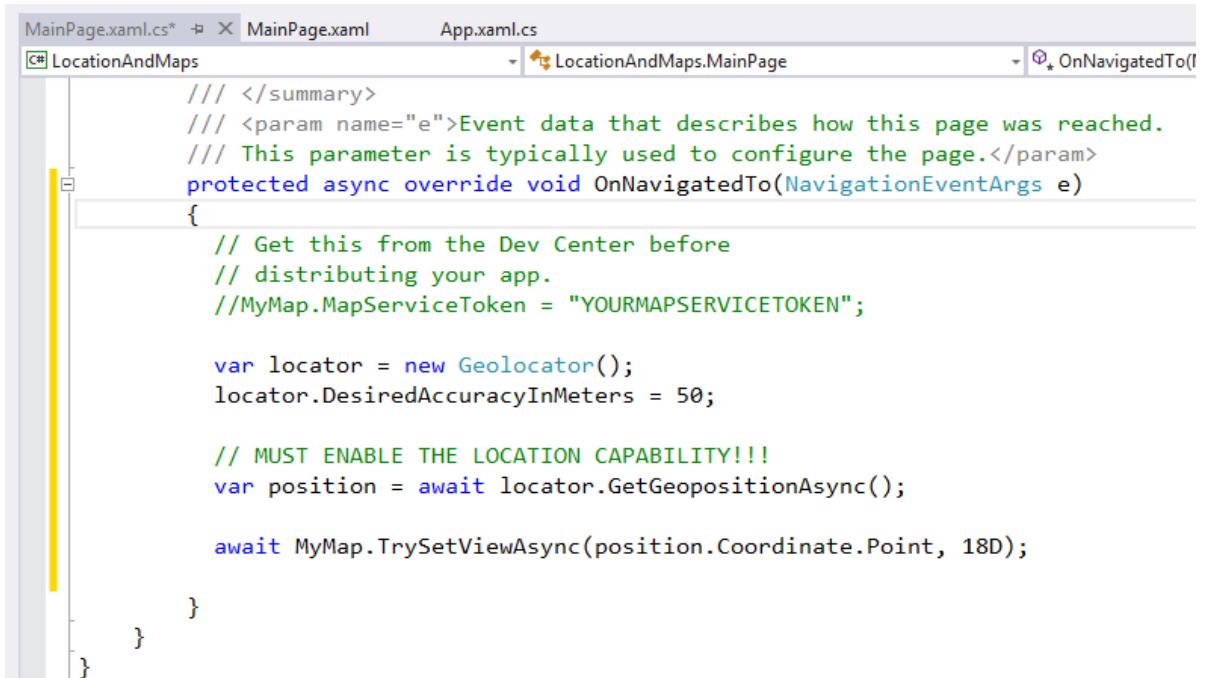
And when I run the app, notice that the Map is simply set to North America (at least, since I am in North America. I'm not sure what would happen if you tried this from other parts of the world).



The first thing I want to accomplish is the set the MapControl, by default, to a specific location. I'll choose an out door mall near the Microsoft campus in Redmond, Washington, as the location of the Phone using the Emulator's Additional Tools dialog:



Back in my project, in the `MainPage.xaml.cs` I'll modify the `OnNavigatedTo()` event handler method to work with the Geolocator. This object is your interface to the GPS / location features of the Phone. Since we're running in emulation, I would expect to retrieve the location in Redmond, Washington that I set. Once I have the Geolocator retrieving the location, I attempt to set the `MapControl`'s view to that location:



```
/// </summary>
/// <param name="e">Event data that describes how this page was reached.
/// This parameter is typically used to configure the page.</param>
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    // Get this from the Dev Center before
    // distributing your app.
    //MyMap.MapServiceToken = "YOURMAPSERVICETOKEN";

    var locator = new Geolocator();
    locator.DesiredAccuracyInMeters = 50;

    // MUST ENABLE THE LOCATION CAPABILITY!!!
    var position = await locator.GetGeopositionAsync();

    await MyMap.TrySetViewAsync(position.Coordinate.Point, 18D);
}
```

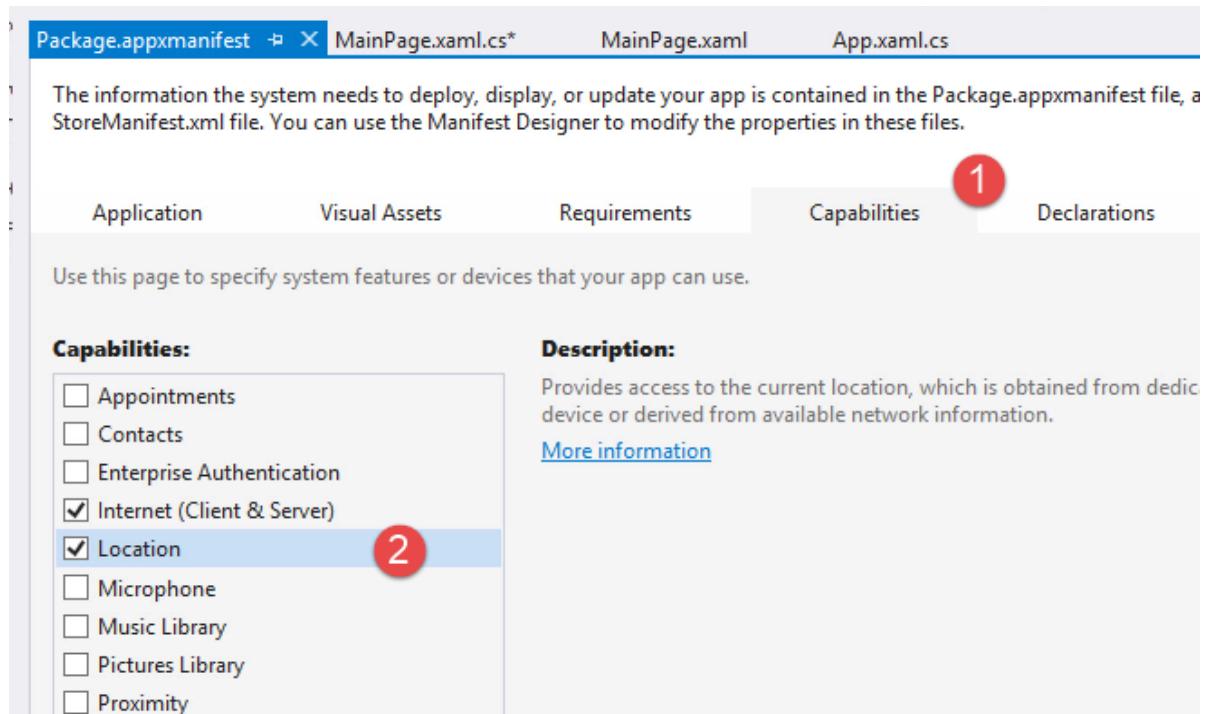
A few things I want to point out. Notice that we're setting the DesiredAccuracyInMeters to 50. Due to technological features of the phone and the GPS system, it is not always possible to get an exact location of the phone. It will try different techniques to get as close as possible. For example, it will not only use the GPS satellite, but it may also use wifi signals and cell tower signals to attempt to triangulate the location. Depending on how accurate you need this to be and what the resources are near the device at the given time that you're attempting to retrieve the position, it could take longer because it has to account for all of these possibilities, wifi signals that are nearby, cell towers, GPS in order to find the location. Honestly, I don't understand how all that works.

However, at some point, we're going to obtain a Geoposition object providing us with a Coordinate. A Coordinate is made up of positioning information. The most important for our purposes is a Point. A Point is simply just a combination of the latitude and longitude and we'll use that information to try and set the view for the map

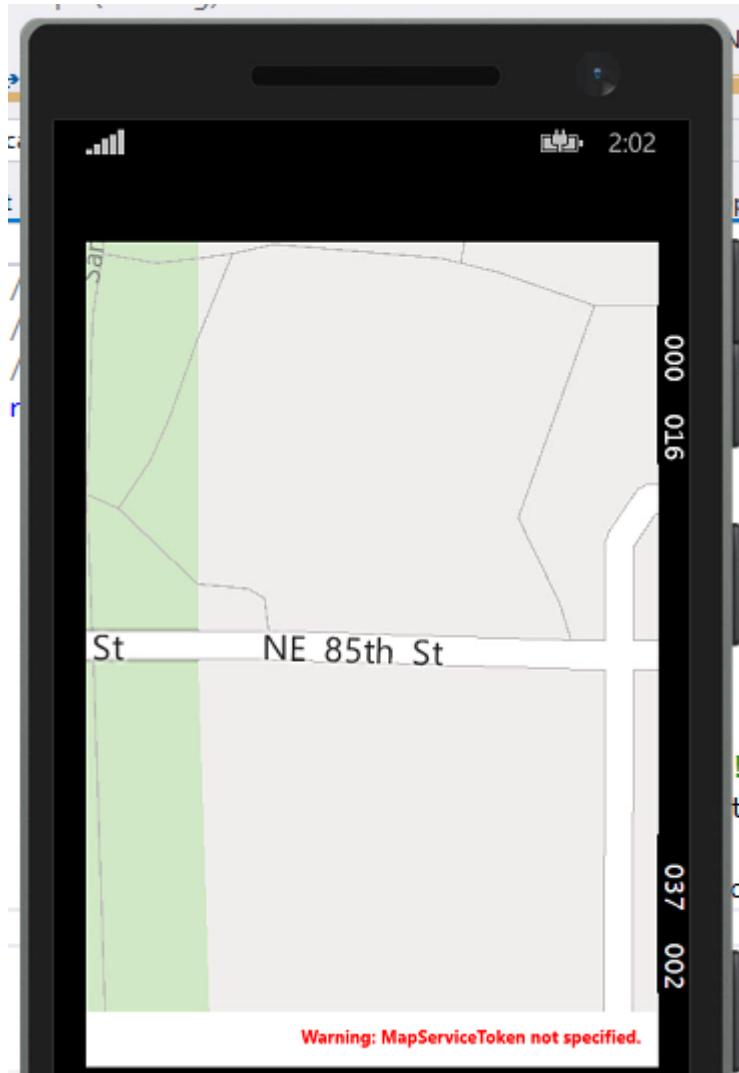
Whenever you see that key word “try” in a method name, that means it's going to return back to you a true or false: true if the operation was successful, false if it wasn't. For our purposes in this pithy example, we'll ignore the return value. In other words, if TrySetViewAsync() fails, then it just fails and it will be quiet about it.

In the TrySetViewAsync() we're passing a Point which is a Geocoordinate position which is basically what we get from position.Coordinate. We also pass in the zoom level. A setting of zero would zoom to see all of North and South America, and a zoom level of 20 would be where we can see in our own back yards. It is a double data type so I add the D next to it just is a way of making the value 18 into a double literal.

Before we run the app, we'll need to add the Location capability to our app. Open the package.appxmanifest, go to the (1) Capabilities tab, and (2) select the Location capability:



Now we should be able to run the application and then see our map control somewhere near this Redmond Town Center in Redmond, Washington. Unfortunately, I don't know the streets of Redmond, Washington well enough to verify how far the location we're given in the MapControl is to the location we set in the Emulator's Location tab, but I'll assume it is close.



Next, I'll implement two Button controls to set and retrieve the current position of the ma. I'll add two more row definitions and setting the height of each to 50:

```
<Grid Margin="20,50,20,20">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="50" />
        <RowDefinition Height="50" />
        <RowDefinition Height="50" />
    </Grid.RowDefinitions>
```



As a means of further exercising the Geolocation and Map Control, we'll write out the current position of the Map Control to a TextBlock and I'll programmatically set the position of the Map Control. Two Button controls will trigger these actions. So, I'll add the following XAML inside the Grid:

```
<StackPanel Orientation="Horizontal" Grid.Row="1">
    <Button Name="getPositionButton"
        Content="Get Position"
        Click="getPositionButton_Click"
        Margin="0,0,20,0" />
    <Button Name="setPositionButton"
        Content="Set Position"
        Click="setPositionButton_Click"
        Margin="0,0,20,0" />
</StackPanel>

<TextBlock Name="positionTextBlock"
    Grid.Row="2"
    FontSize="22" />
```

I'll create method stubs for the getPositionButton_Click and setPositionButton_Click by putting my mouse cursor on their names in the XAML and pressing F12.

In each of those two methods, I'll add the following code:

```
private void getPositionButton_Click(object sender, RoutedEventArgs e)
{
    positionTextBlock.Text = String.Format("{0}, {1}",
        MyMap.Center.Position.Latitude,
        MyMap.Center.Position.Longitude);
}

private async void setPositionButton_Click(object sender, RoutedEventArgs e)
{
    var myPosition = new Windows.Devices.Geolocation.BasicGeoposition();
    myPosition.Latitude = 41.7446;
    myPosition.Longitude = -087.7915;
```

```

var myPoint = new Windows.Devices.Geolocation.Geopoint(myPosition);
if (await MyMap.TrySetViewAsync(myPoint, 10D))
{
    // Haven't really thought that through!
}

}

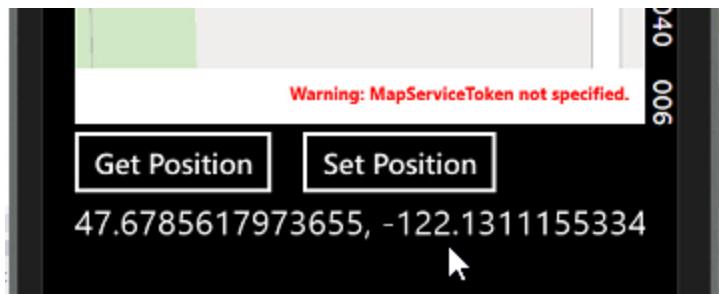
```

Hopefully all of this looks familiar, or at least, straight forward.

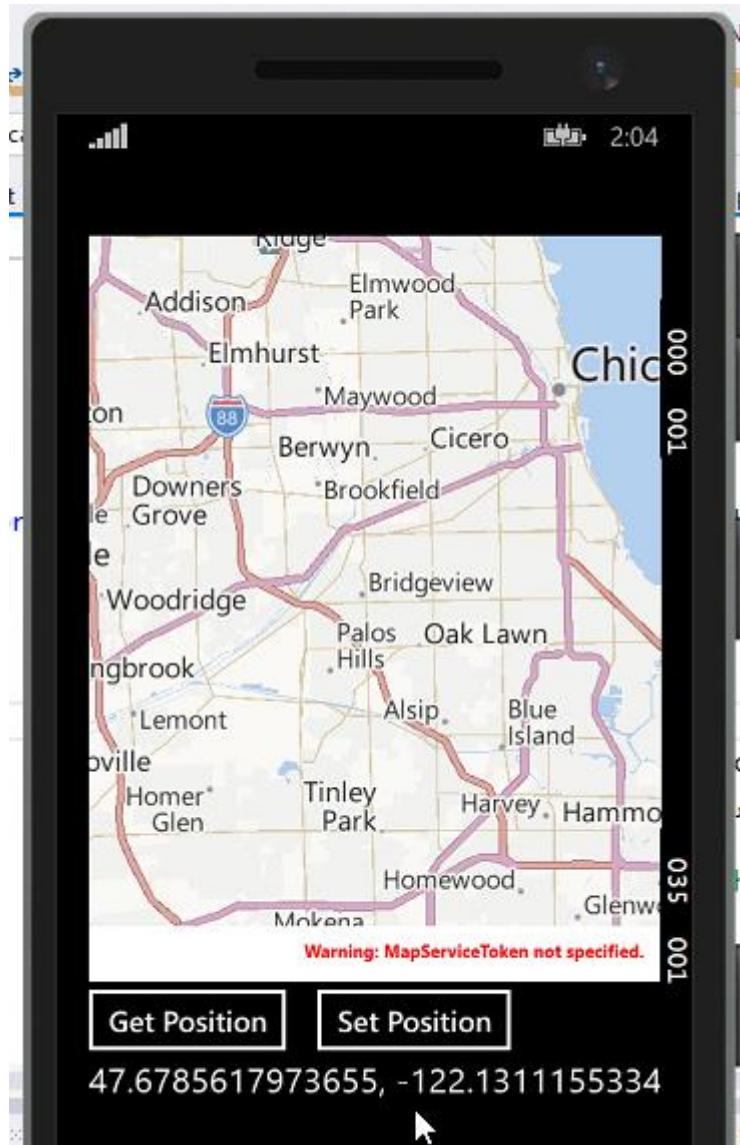
In the getPositionButton_Click I'm merely retrieving the Map's Center position, grabbing the latitude and longitude and printing to a TextBlock. This demonstrates how to retrieve all or part of the current location OF THE MAP CONTROL which can be useful when building Map-centric apps.

In the setPositionButton_Click, I'm hard-coding the position of the Map. Again, I'm using TrySetViewAsync() to do the heavy lifting. However, this time I'm creating a BasicGeoposition object to contain the latitude and longitude and then creating a Geopoint using the BasicGeoposition as a constructor argument.

When I run the app, I can retrieving the current position into a TextBlock by clicking the "Get Position" button ...



... and can set the position of the Map to a specific location near where I grew up by clicking the "Set Position" button:



Next, I'll add a Slider control to perform zooming. I'll add the following XAML to the Grid:

```
<Slider Name="mySlider"
    Maximum="20"
    Minimum="10"
    ValueChanged="Slider_ValueChanged"
    Grid.Row="3"
/>
```

And will use the F12 shortcut to create the Slider_ValueChanged event handler method, to which I add the following code:

```
private void Slider_ValueChanged(object sender, RangeBaseValueChangedEventArgs e)
{
    if (MyMap != null)
        MyMap.ZoomLevel = e.NewValue;
}
```

Simply put, the slider will allow me to move from 10 to 20 (extreme close up).

Finally, I'll initiate the Slider's value at start up by setting its value to the Map Control's current ZoomLevel in the OnNavigatedTo event handler method:

```
    /// This parameter is typically used to configure the page.</param>
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    // Get this from the Dev Center before
    // distributing your app.
    //MyMap.MapServiceToken = "YOURMAPSERVICETOKEN";

    var locator = new Geolocator();
    locator.DesiredAccuracyInMeters = 50;

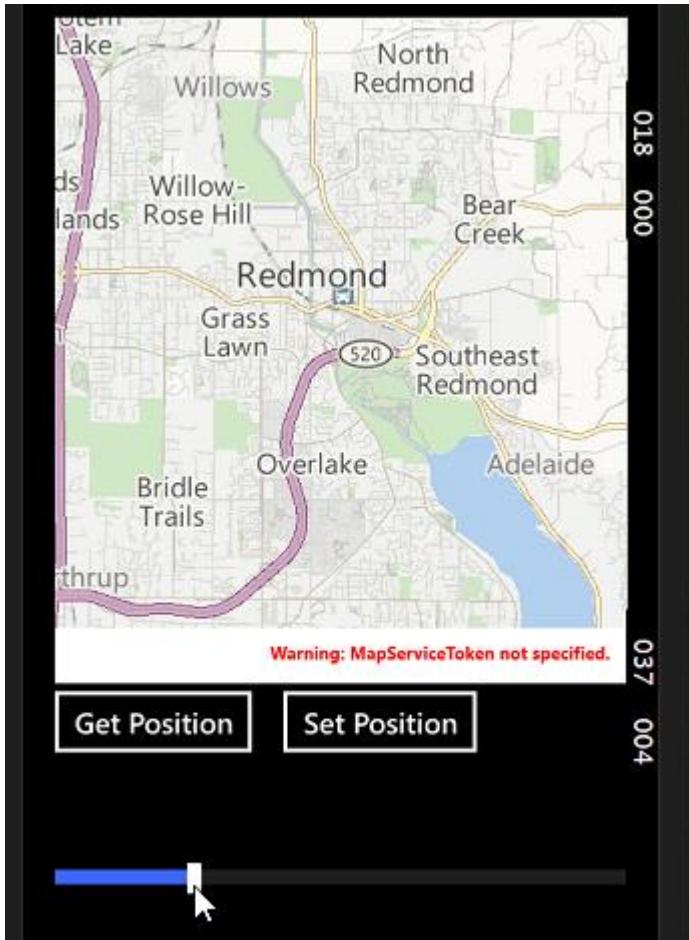
    // MUST ENABLE THE LOCATION CAPABILITY!!!
    var position = await locator.GetGeopositionAsync();

    await MyMap.TrySetViewAsync(position.Coordinate.Point, 18D);

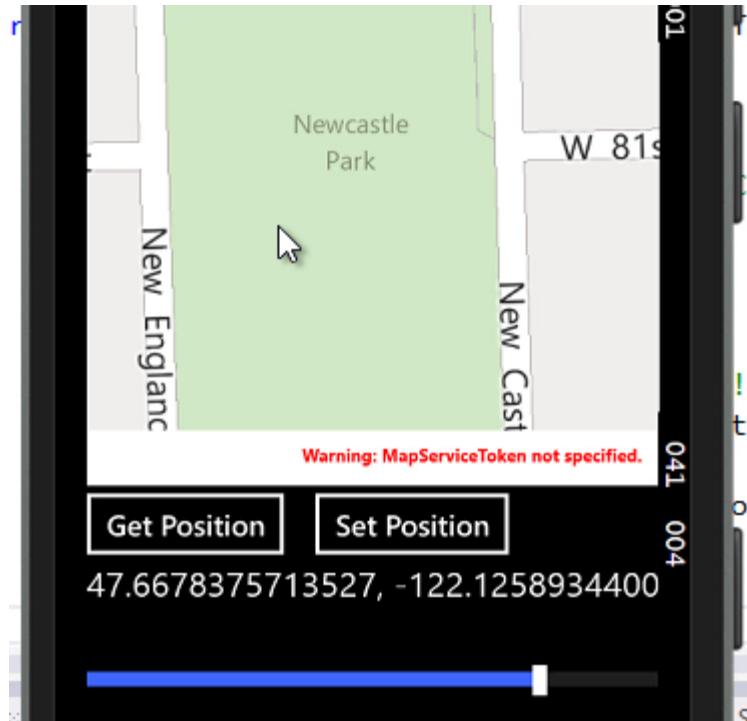
    mySlider.Value = MyMap.ZoomLevel; ←
}

T }
```

Now, when I run the app, I can adjust the zoom level:



I can reset the position using the “Set Position” button, and can zoom in dramatically:



Finally, if you want to build apps that include the Map Control, you'll need to provide a MapServiceToken. You've seen the warning message each during run time. You can obtain a MapServiceToken from the Windows Phone Dev Center after you register your app (which we do not cover in this series). You would set it like so:

```
    /// This parameter is typically used to configure the page.</param>
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    // Get this from the Dev Center before
    // distributing your app.
    //MyMap.MapServiceToken = "YOURMAPSERVICETOKEN";
```

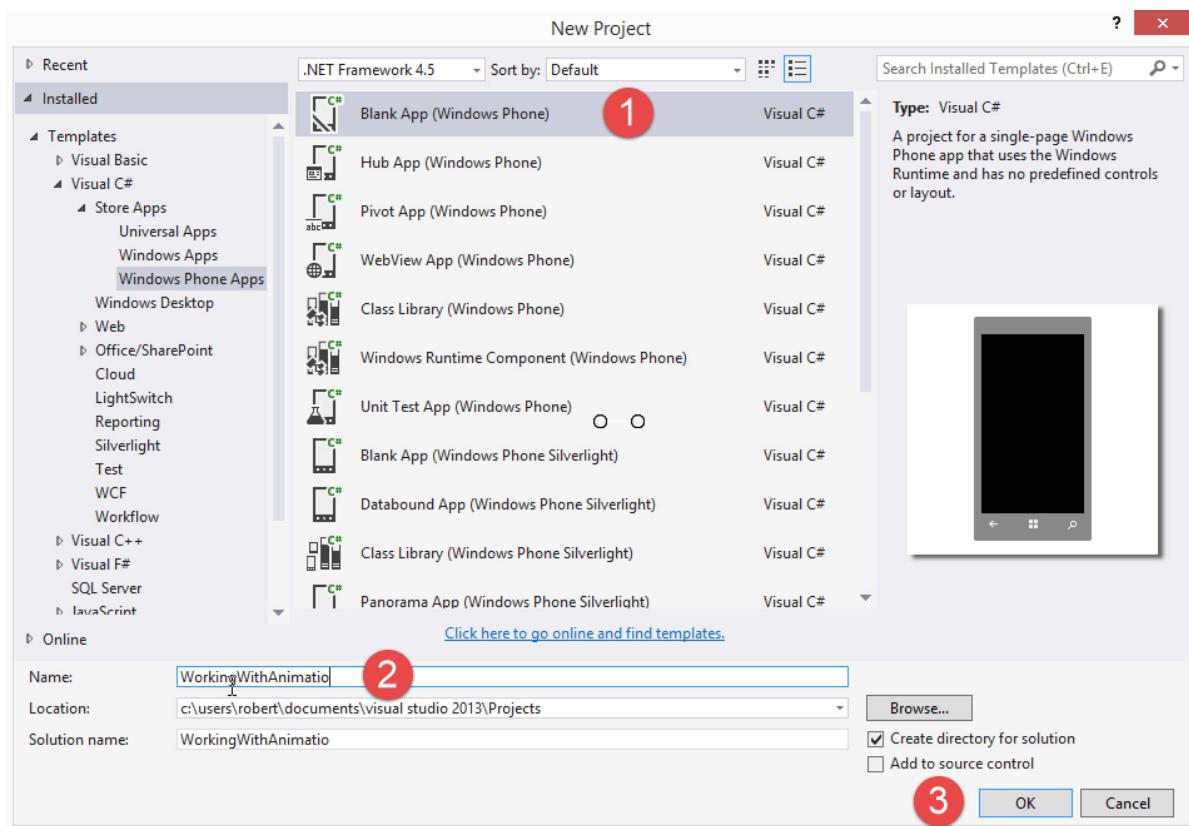

Lesson 28: Working with Animations in XAML

In this lesson we're going to talk about animation. To be clear, what we really do is change properties of the controls on our pages over time. This could take the form of moving an element from one place to another, changing its size from small to large, or changing the opacity of a control. In this lesson we're going to keep it very simple but you can combine many different effects together to create a complex sequence.

I originally hoped to incorporate animation into the last project in this series which we'll begin in the next video. However, as it turned out, I didn't really need animation. Still, I wanted to cover this topic, at least the very basics of it. If this is something you want to use in your apps, you'll want to learn more about Microsoft Blend which is a tool similar to Visual Studio that has a more powerful set of animation tools. However, this less should get you familiar with the concept of animation at a high level.

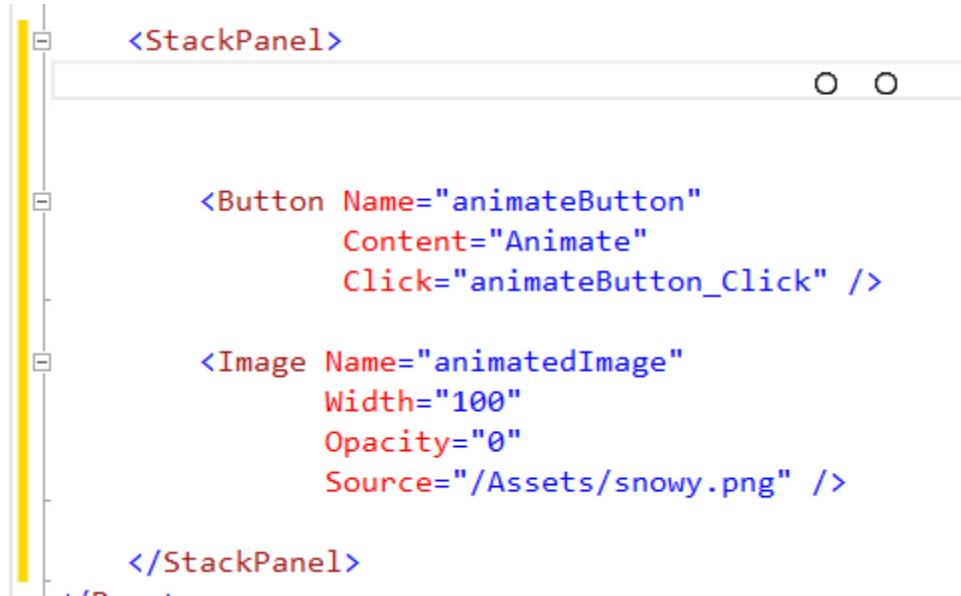
We'll talk about animation using both XAML and C#. If you watched the Windows Phone 8 for Absolute Beginner series, which I created about a year ago with Clint Rutkas, we used a simple animation to animate the images as we were pulling them down from Flikr. In that series I discussed why you might need to take a pure C# approach versus a pure XAML approach towards animating properties of controls. In this lesson I'll demonstrate both approaches and then let you choose which one you want to use for your specific purpose.

We'll start by opening the New Project dialog and (1) creating a new Black App project template called (2) WorkingWithAnimation, then (3) clicking the OK button.



On MainPage.xaml I'll replace the Grid with a StackPanel. We'll begin by animating an image's opacity to give it the appearance of fading the image in and out. To follow along, you'll need to add the two images from the Assets folder of the Lesson28 zip file into your project's Assets folder.

Inside the StackPanel I'll add a Button and an Image. I'll set the source of the Image control to the snowy.png file which we copied into the Assets folder. I'll also set the Opacity to 0 meaning we want to begin with the image being completely transparent.



My objective is to move the opacity property from 0.0 (completely transparent) to 1.0 (completely opaque) over two seconds. We'll get to that in just a moment. When the user clicks a button we'll kick off that animation.

Animations are defined inside of Storyboards. A Storyboard is simply a collection of animations that all happen at the same time. As a developer you decide which events trigger the Storyboard to play all of its animations.

As I said earlier, you can animate many properties of controls including the size, the opacity, the color of the control. There are different animation objects that know how to work with colors, numbers, etc.

Let's create a Storyboard. We'll create a Resources section of the StackPanel. In other words, here are the resources that'll be available to all the children of the StackPanel. Inside of the StackPanel.Resources, we'll create two Storyboards. The first I'll call "ShowStoryboard" and the second I'll call "HideStoryboard". Inside of the "ShowStoryboard" I'll set the target control, the target property of the target control, the original value and the target value, and finally the duration of the animation:



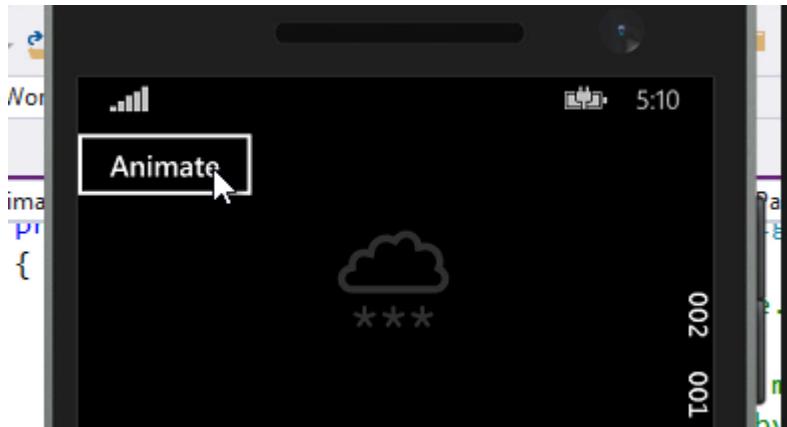
```
<StackPanel>
  <StackPanel.Resources>
    <Storyboard x:Name="ShowStoryboard">
      <DoubleAnimation Storyboard.TargetName="animatedImage"
        Storyboard.TargetProperty="Opacity"
        From="0"
        To="1"
        Duration="0:0:2" />
    </Storyboard>
    <Storyboard x:Name="HideStoryboard">
    </Storyboard>
  </StackPanel.Resources>
```

In this example, the ShowStoryboard has one animation defined, a double animation ("double" as in the double data type, with numeric values after the decimal point) which specializes in moving a numeric-based property from one number to another. When it is triggered, it will animate the animatedImage control's Opacity property from 0.0 to 1.0 over the course of two seconds.

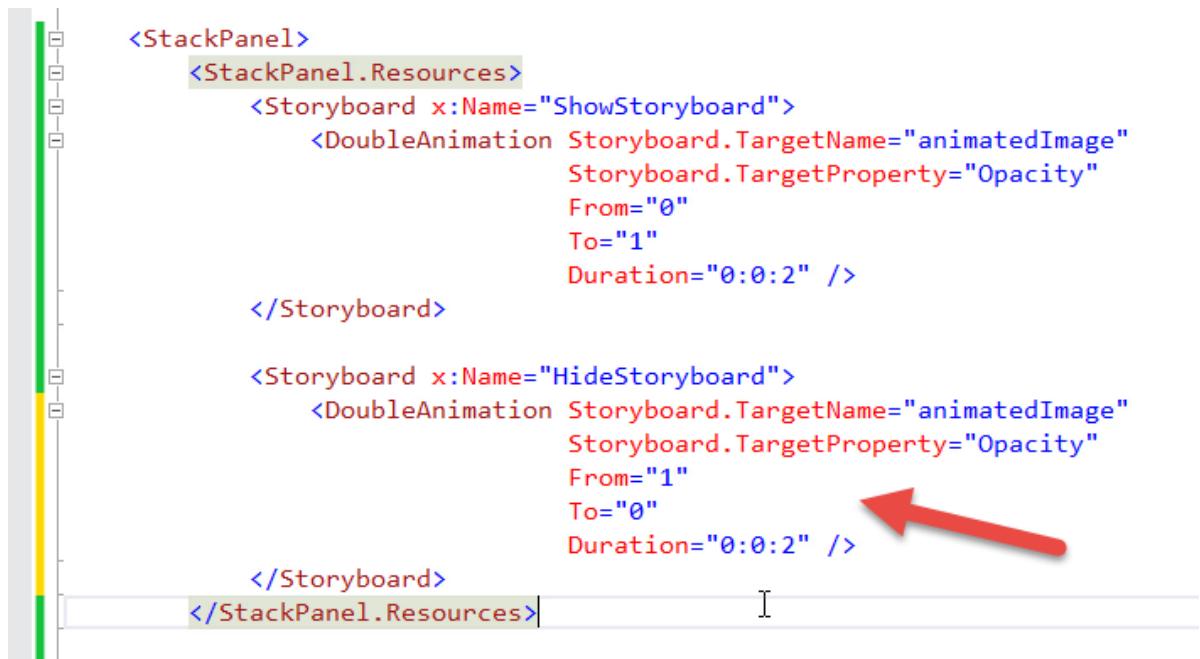
In our simple example, we'll trigger the ShowStoryboard's animations by calling the Begin() method in response to some event, such as the animateButton_Click event:

```
private void animateButton_Click(object sender, RoutedEventArgs e)
{
    ShowStoryboard.Begin();
}
```

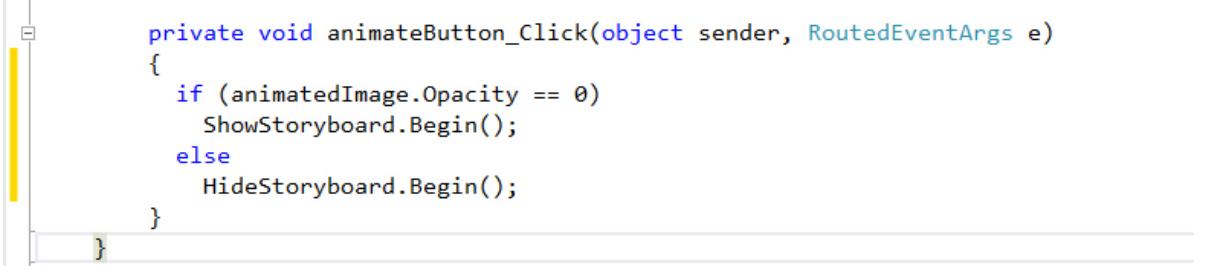
When we run the app, we can trigger the image to slowly appear by clicking the "Animate" button:



We can also cause the image to disappear slowly using a second Storyboard with a single DoubleAnimation object that does the opposite of the first: changing the Opacity property from 1.0 to 0.0 over the course of two seconds:



Now we'll add some logic to toggle the effect of showing and hiding the image by checking the current state of the `animatedImage`'s `Opacity` property and calling the appropriate storyboard's `Begin()` method:



```
private void animateButton_Click(object sender, RoutedEventArgs e)
{
    if (animatedImage.Opacity == 0)
        ShowStoryboard.Begin();
    else
        HideStoryboard.Begin();
}
```

So, we can define XAML Storyboards that contain DoubleAnimation objects then use C# to trigger those storyboard at key moments in the app.

We can take a second tact. This second tact is helpful when we want to programmatically add new controls and animate them as we add them. In this case, I'll re-work the Image control's XAML definition to include an ImageOpened event handler method called animatedImage_ImageOpened:



```
<Image Name="animatedImage"
       Width="100"
       Opacity="0"
       Source="/Assets/snowy.png"
       ImageOpened="animatedImage_ImageOpened"/>
```

The animatedImage_ImageOpened() method will fire whenever the Image control's image is changed via its Source property.

We'll also re-work the animateButton_Click() event handler method to dynamically load a different image into our Image control (the Assets/sunny.png file):

```
private void animateButton_Click(object sender, RoutedEventArgs e)
{
    /*
    if (animatedImage.Opacity == 0)
        ShowStoryboard.Begin();
    else
        HideStoryboard.Begin();
    */

    var image = new BitmapImage(new Uri("ms-appx:///Assets/sunny.png"));
    animatedImage.Source = image;
}

private void animatedImage_ImageOpened(object sender, RoutedEventArgs e)
{
}
```

Notice how we created the `BitmapImage` control ... by providing a new `Uri` object set to a location as designated by a special prefix:

`ms-appx://`

Similar to how we can use `http://` to indicate a URL, we can use the `ms-appx://` to denote a location that is relative to the current deployment package's root. Obviously in this case we're expecting our `sunny.png` file to be deployed to the `Assets` folder of the compiled package.

Inside the `animatedImage_ImageOpened()` event we have an opportunity to intercept the display of the image in the control with code we write. In this case, we'll intercept to create a new `Storyboard` object and child `DoubleAnimation` on the fly with the same definition as the ones we created earlier, except using just C# instead of XAML:

```

private void animatedImage_ImageOpened(object sender, RoutedEventArgs e)
{
    Image img = sender as Image;

    if (img == null)
        return;

    Storyboard s = new Storyboard();

    DoubleAnimation doubleAni = new DoubleAnimation();
    doubleAni.To = 1;
    doubleAni.Duration = new Duration(TimeSpan.FromMilliseconds(2000));

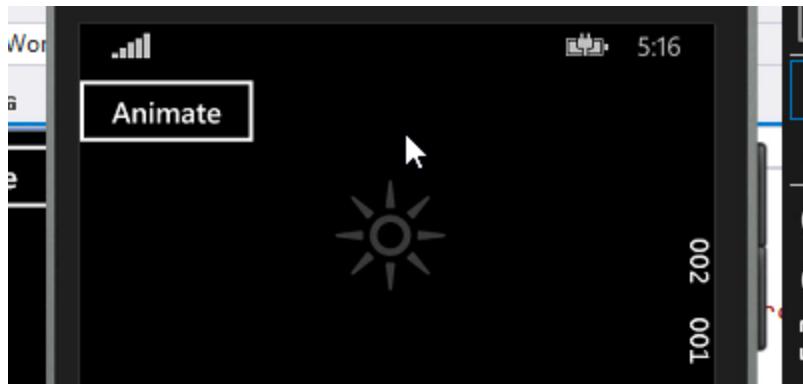
    Storyboard.SetTarget(doubleAni, img);
    Storyboard.SetTargetProperty(doubleAni, "Opacity");

    s.Children.Add(doubleAni);

    s.Begin();
}

```

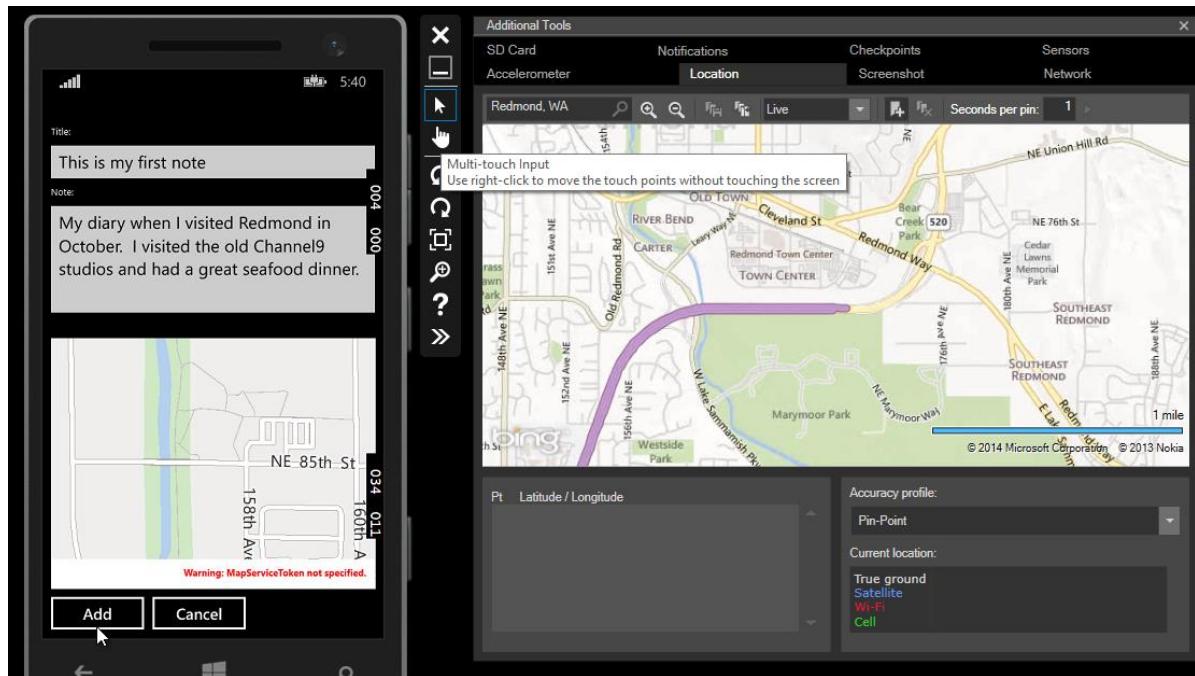
Now, when we load a new image into the Image control, the Image control will first create a Storyboard with a DoubleAnimation that animates the Opacity property, then kicks off the Storyboard:



As I said at the outset, these are two very simple examples of animation. However, that should get you at least some basic understanding of what animations are, how they're associated with storyboards, how to kickoff a storyboard with the storyboard's `Begin()` method.

Lesson 29: Exercise: The Map Notes App

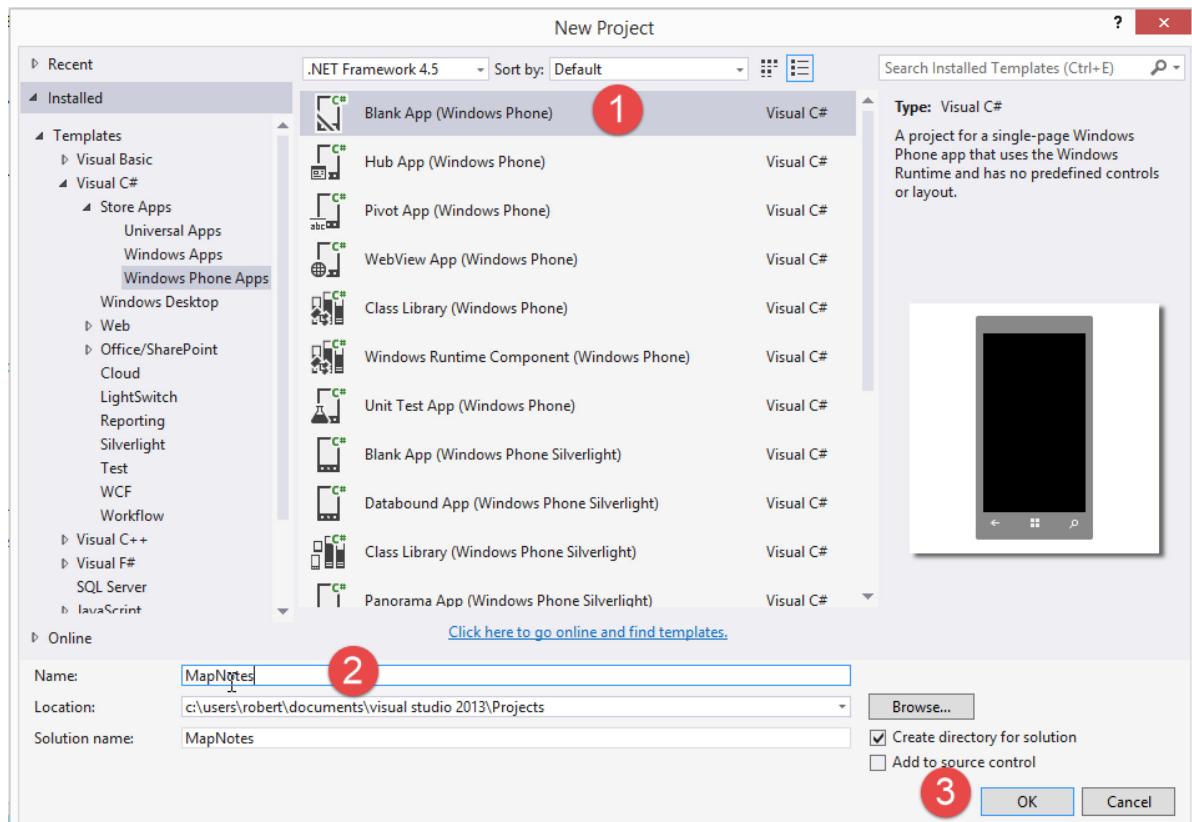
In this final lesson, we'll build the MapNotes app. It is a GPS-aware note app that records not just the note you take but also where you took that note and displays it on a map. Then you'll be able to read that note and see a map of where you took the note. You can then delete the note or continue on from there.



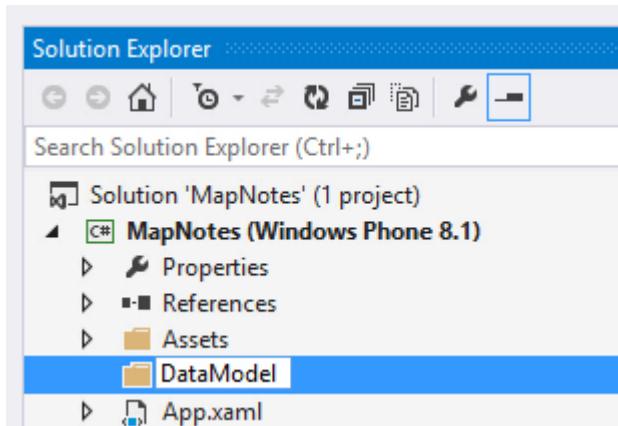
In addition to working with Phone storage, the Map control and Geolocation, I'll also add some nice little touches like truncating long note titles replacing the remainder of the title with an ellipsis and adding a MessageDialog to ask the user to confirm an operation:



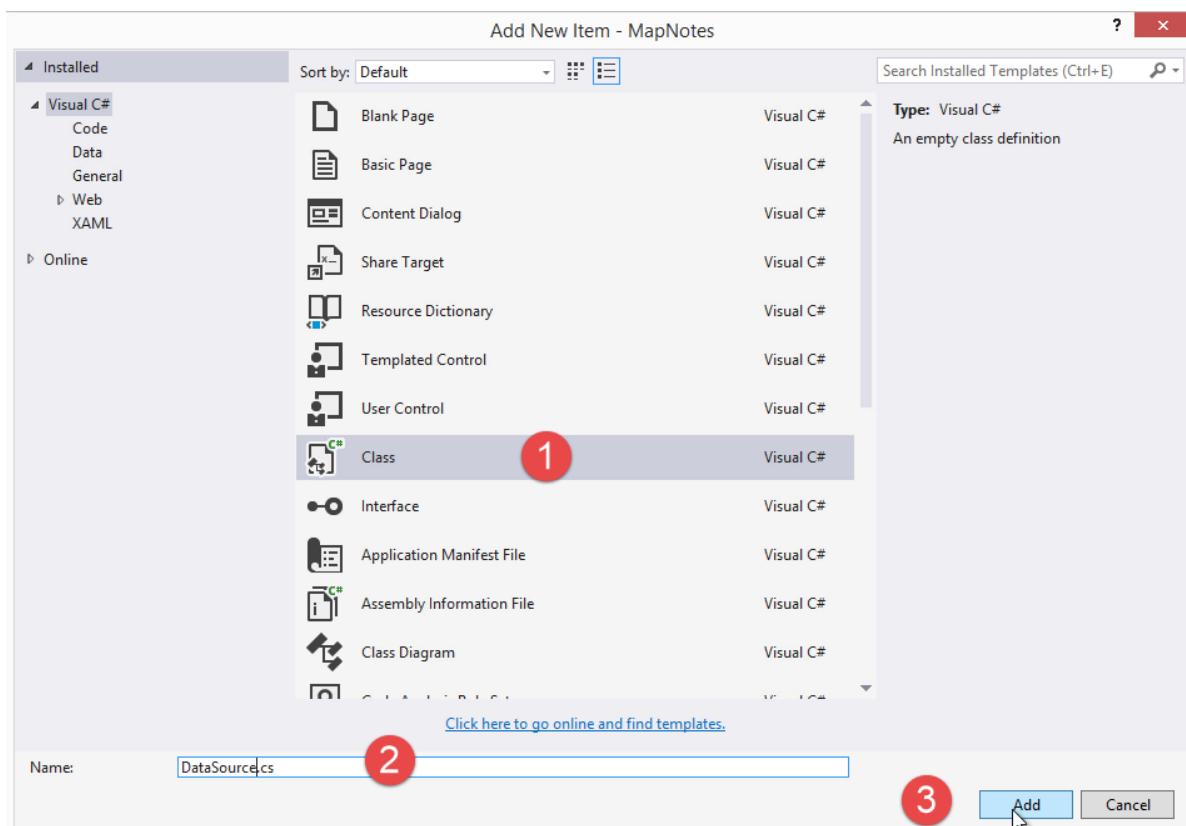
To begin, I'll open the New Project dialog, and (1) create a new Blank App project template (2) called MapNotes, then (3) click the OK button:



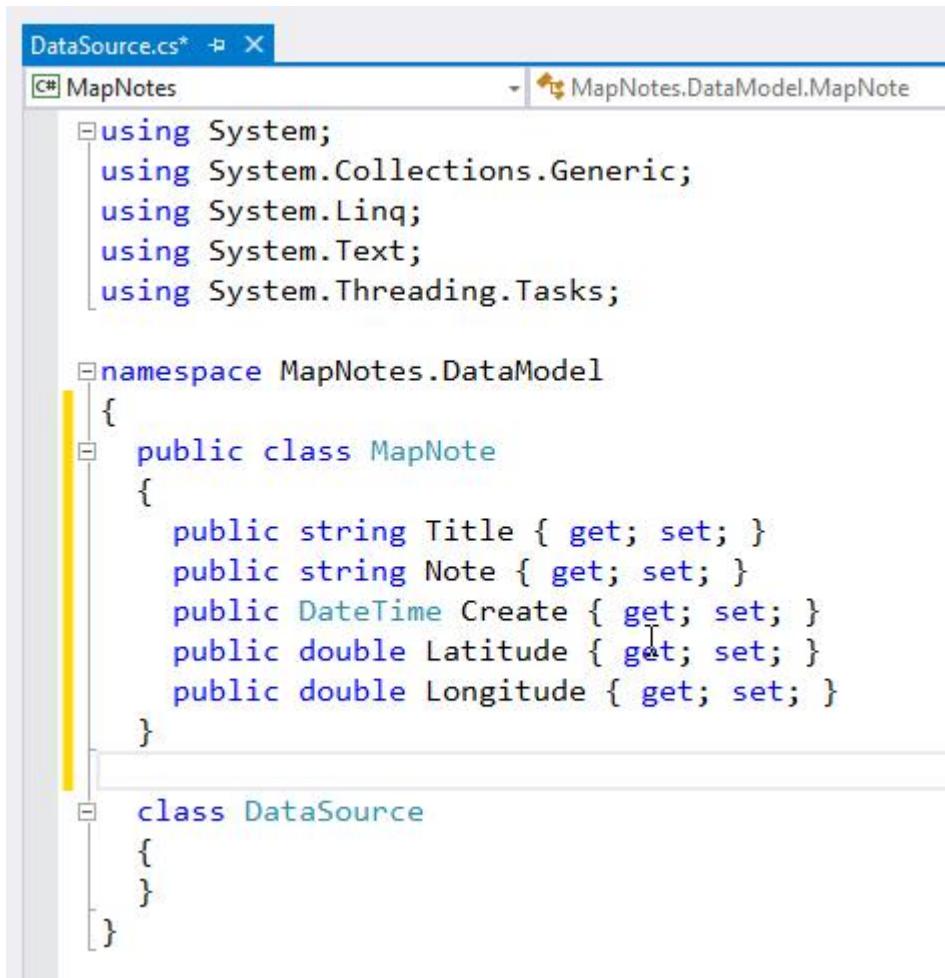
First, I'll add a DataModel folder:



Then will add a new file to that folder. In the Add New Item dialog, I'll add (1) a class (2) called DataSource.cs, then (3) I'll click the Add button:



In the new DataSource.cs file, I'll create a model called MapNote, adding a number of properties that represent the note, where the note was created, etc.:



```
DataSource.cs*  X
C# MapNotes  ↴ MapNotes.DataModel.MapNote
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MapNotes.DataModel
{
    public class MapNote
    {
        public string Title { get; set; }
        public string Note { get; set; }
        public DateTime Create { get; set; }
        public double Latitude { get; set; }
        public double Longitude { get; set; }
    }

    class DataSource
    {
    }
}
```

The DataSource class will be our primary focus. You'll notice how similar it is to other DataSource classes we created in the past. In fact, as I was developing this, I merely copied code and renamed the data class to MapNote (since that is the type of data we're working with in this app). While I don't advocate copy and paste always, as a means of templating some common functionality, it can be useful.

I'll start by creating a private ObservableCollection<MapNote> that the remainder of the methods in the DataSource class will manage... adding new items, deleting items, saving and retrieving from Phone storage, etc.:

```
public class DataSource
{
    private ObservableCollection<MapNote> _mapNotes;

    public DataSource()
    {
        _mapNotes = new ObservableCollection<MapNote>();
    }

    ...
}
```

We'll begin by providing access to the `_mapNotes` through the `GetMapNotes()` method. Again, note the templated nature of this method from what we've used earlier. `GetMapNotes()` called a helper method called `ensureDataLoaded()`. The `ensureDataLoaded()` method checks the count of items in the collection, and if it is empty will call the `getMapNoteDataAsync()` which has the responsibility of hydrating the object graph of `MapNotes` from Phone storage into memory.

```
public async Task<ObservableCollection<MapNote>> GetMapNotes()
{
    await ensureDataLoaded();
    return _mapNotes;
}

private async Task ensureDataLoaded()
{
    if (_mapNotes.Count == 0)
        await getMapNoteDataAsync();

    return;
}
```

We've talked about methods similar to `getMapNoteDataAsync()` in the past. In a nut shell, we'll use a `DataContractJsonSerializer` to open a file from the Phone's `LocalFolder` storage and de-serialize the file's contents into an in-memory collection of `MapNotes`.

```
private async Task getMapNoteDataAsync()
{
    if (_mapNotes.Count != 0)
        return;

    var jsonSerializer = new DataContractJsonSerializer(typeof(ObservableCollection<MapNote>));

    try
    {
        // Add a using System.IO;
        using (var stream = await ApplicationData.Current.LocalFolder.OpenStreamForReadAsync(fileName))
        {
            _mapNotes = (ObservableCollection<MapNote>)jsonSerializer.ReadObject(stream);
        }
    }
    catch
    {
        _mapNotes = new ObservableCollection<MapNote>();
    }
}
```

We'll need to create a constant with the fileName since we'll be using it to both save and retrieve data on disk.

```
public class DataSource
{
    private ObservableCollection<MapNote> _mapNotes;

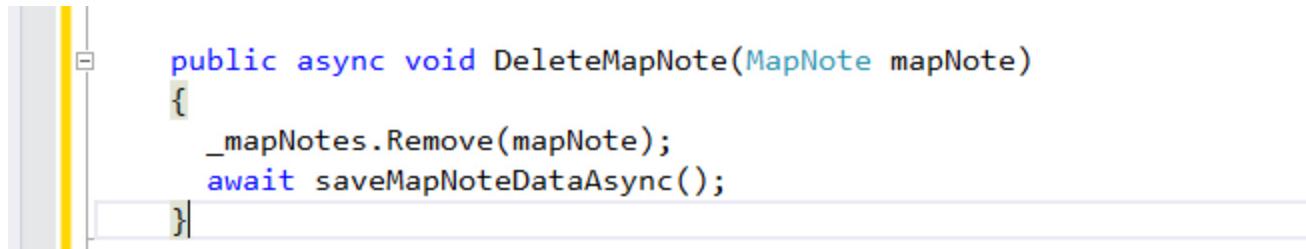
    const string fileName = "mapnotes.json";
    public DataSource()
```

We want to allow the user to create new MapNote objects and save them to the Phone's storage. The AddMapNote() method will add a new MapNote to the private ObservableCollection<MapNote>, then persist that to storage by calling saveMapNoteDataAsync(). The saveMapNoteDataAsync() is the corollary to getMapNoteDataAsync() ... it, too, uses a DataContractJsonSerializer to save a file to the Phone's Local Folder.

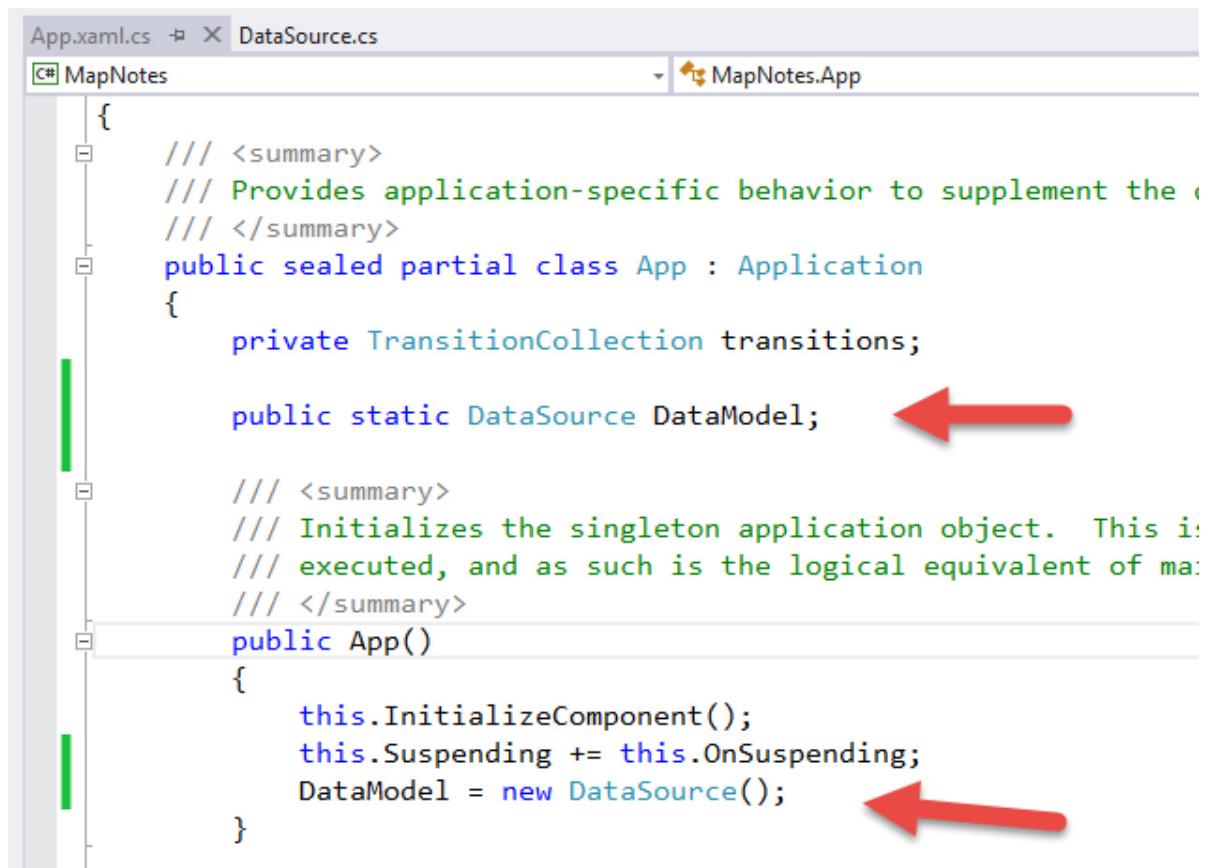
```
public async void AddMapNote(MapNote mapNote)
{
    _mapNotes.Add(mapNote);
    await saveMapNoteDataAsync();
}

private async Task saveMapNoteDataAsync()
{
    var jsonSerializer = new DataContractJsonSerializer(typeof(ObservableCollection<MapNote>));
    using (var stream = await ApplicationData.Current.LocalFolder.OpenStreamForWriteAsync(fileName,
        CreationCollisionOption.ReplaceExisting))
    {
        jsonSerializer.WriteObject(stream, _mapNotes);
    }
}
```

Finally, we want to allow a user to delete a note. We'll call Remove() on the collection of MapNotes, then again, call saveMapNoteDataAsync() to make sure that change is saved to the Phone's storage:

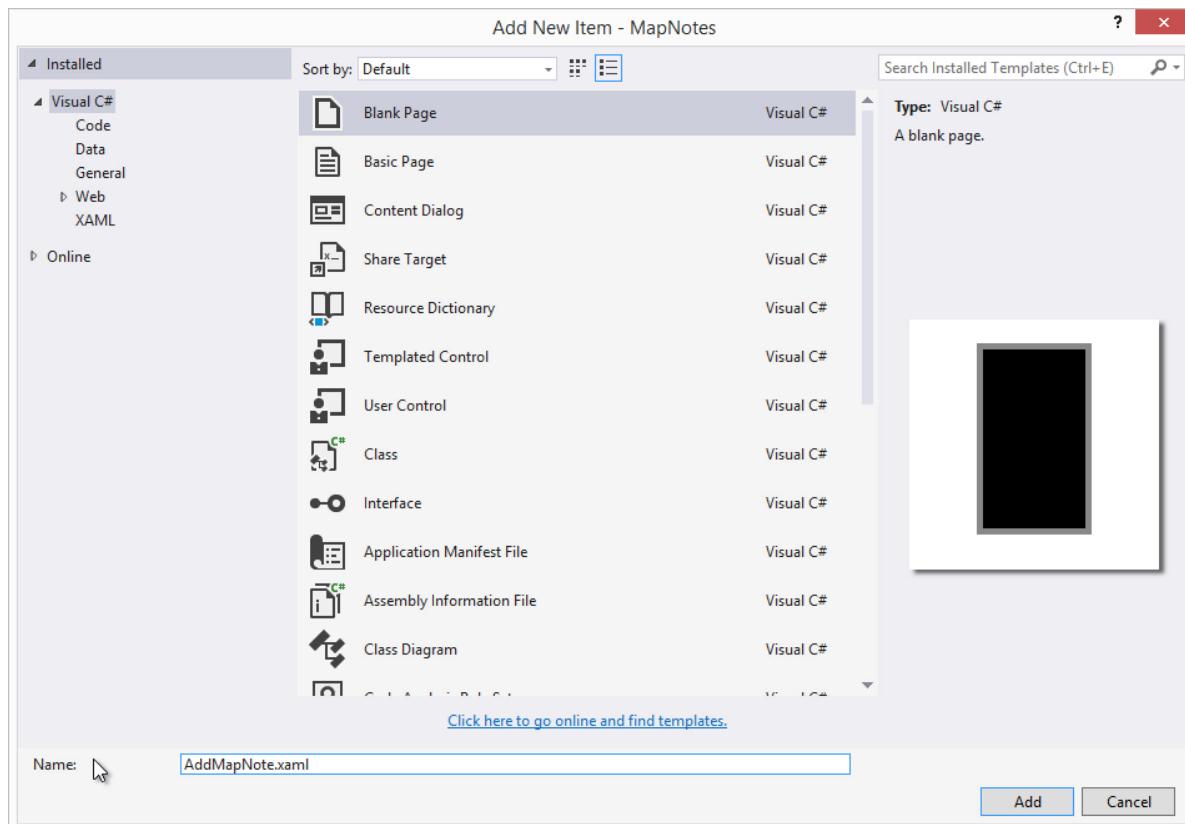


We want to make our DataSource class available throughout all pages in our app, so I decided to create an instance of DataSource as a public static field on the App class. In the App's constructor, I create an instance of DataSource and set it to the DataModel field:



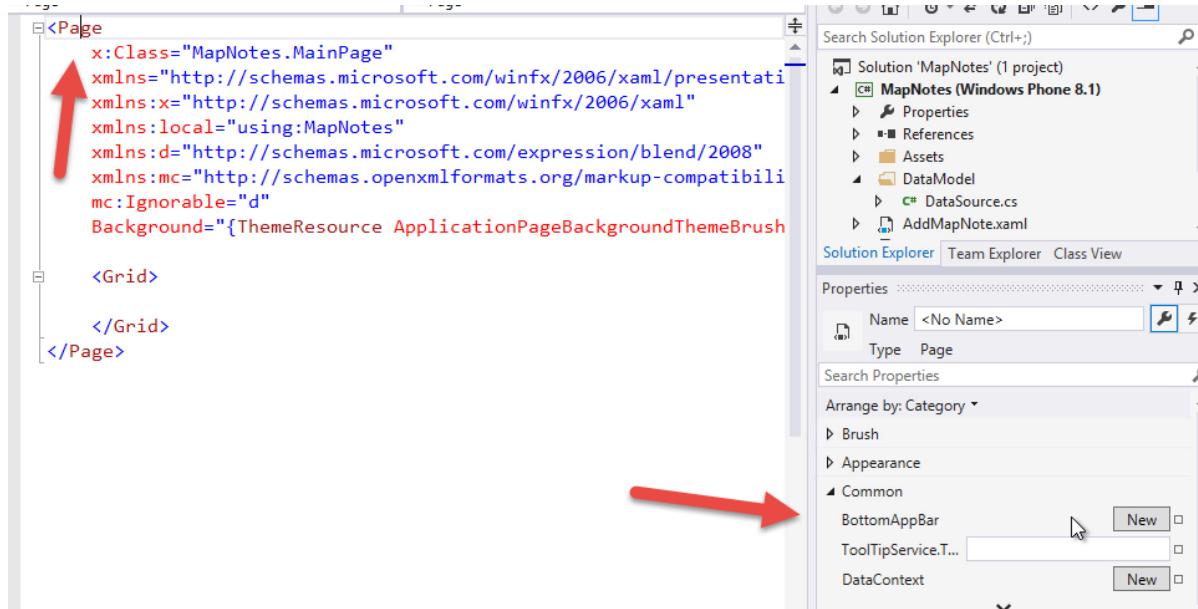
Now, we're ready to create a page that will allow a user to add a new note or look at an existing note. While looking at an existing note, we'll allow the user to delete the note as well. We'll use a single page for both purposes, changing out the names of the buttons and their operation depending on the current state of the current MapNote.

I'll add a new item, a new Blank Page called AddMapNote.xaml:

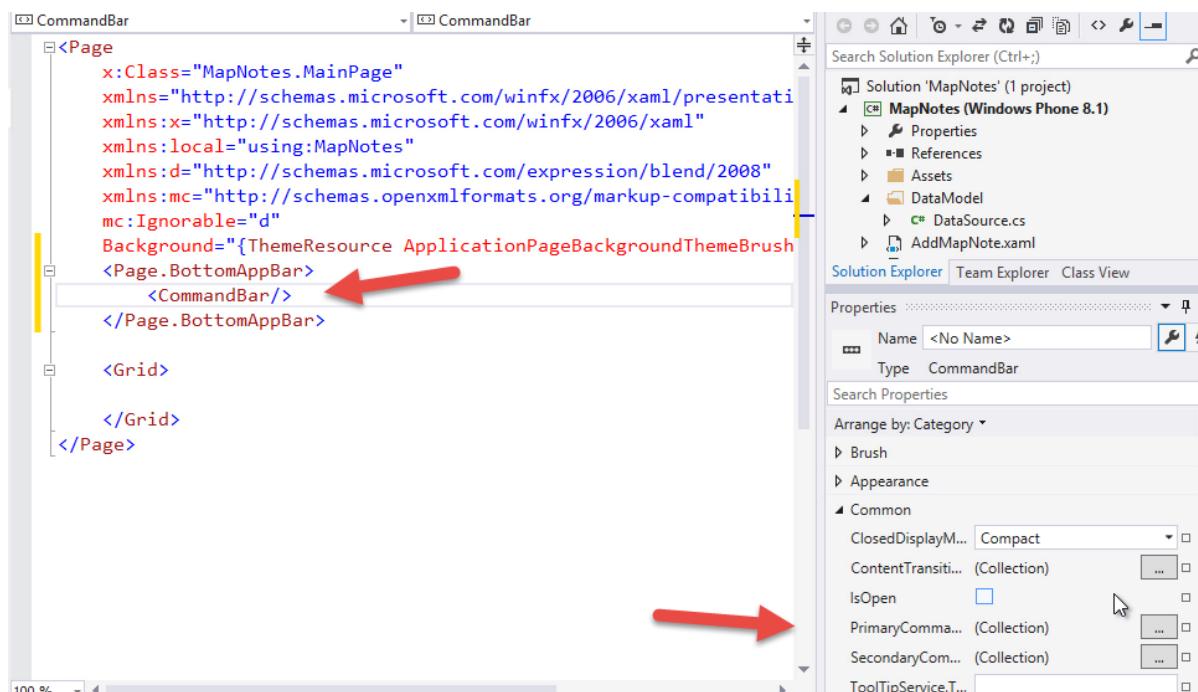


Before I begin to lay out the controls on the new AddMapNote.xaml page, I need a way to navigate from the MainPage.xaml to the AddMapNote.xaml page. There will be two ways to do this ... the first will be to click a button in the MainPage.xaml's CommandBar to add a new note. The second way will be to tap an existing note's title / entry in the list of notes on that page to view (and potentially delete) that note.

First, I want to add a CommandBar and CommandBarButton. I'll put my mouse cursor in the Page definition, and in the Properties window I'll click the New button next to BottomAppBar:

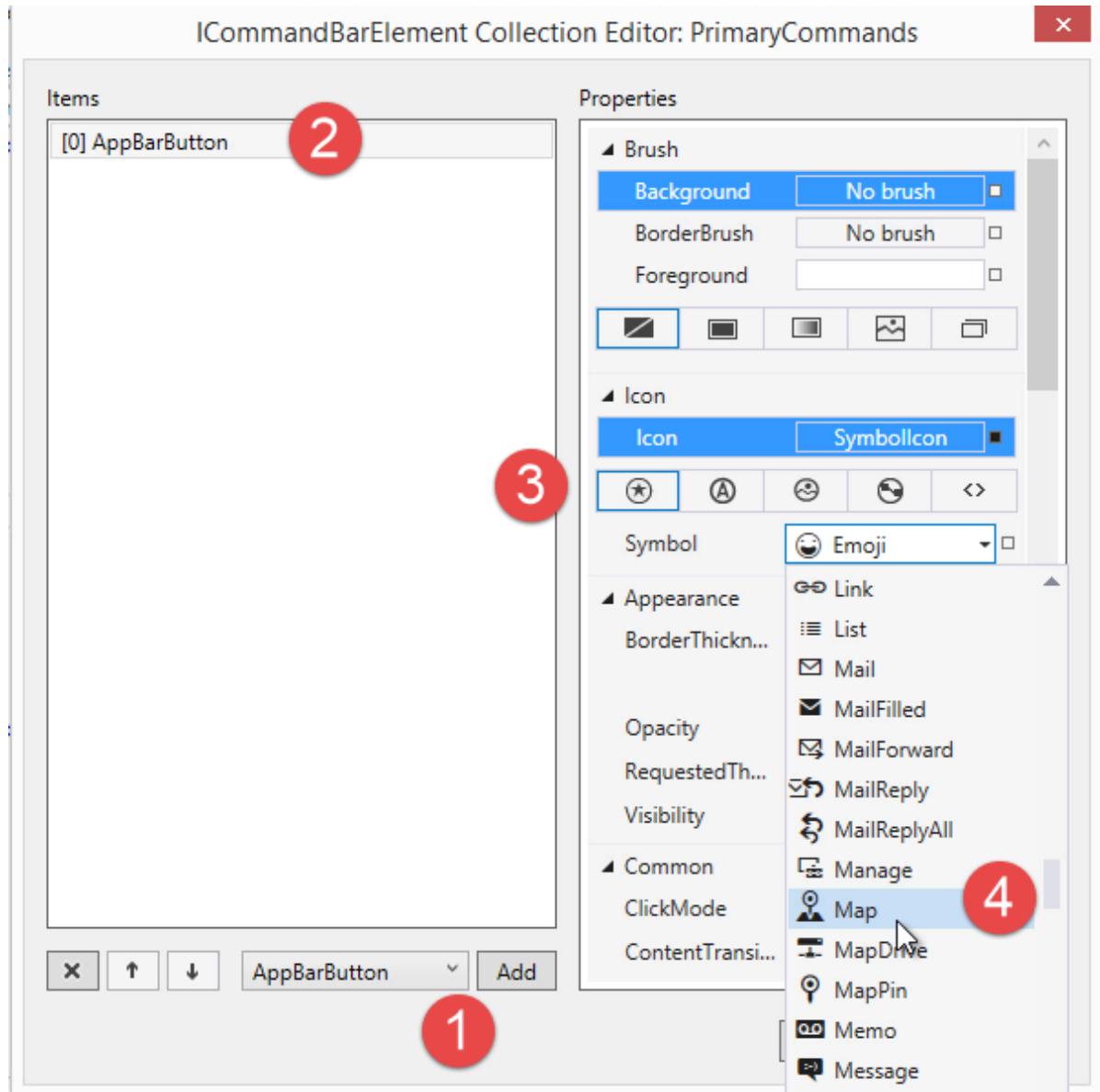


That action will create a Page.BottomAppBar and a CommandBar. Now, I'll put my mouse cursor in the CommandBar and then select the ellipsis button next to PrimaryCommandButton:

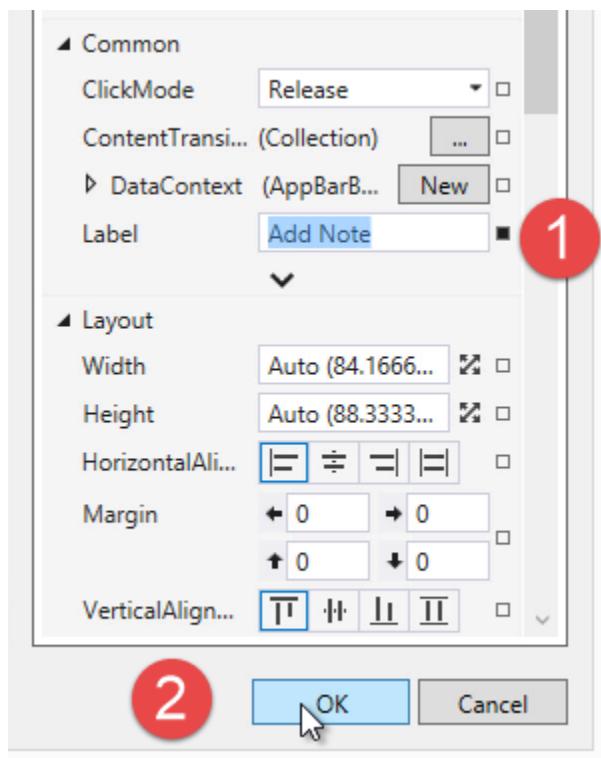


And the PrimaryCommandButton editor dialog appears.

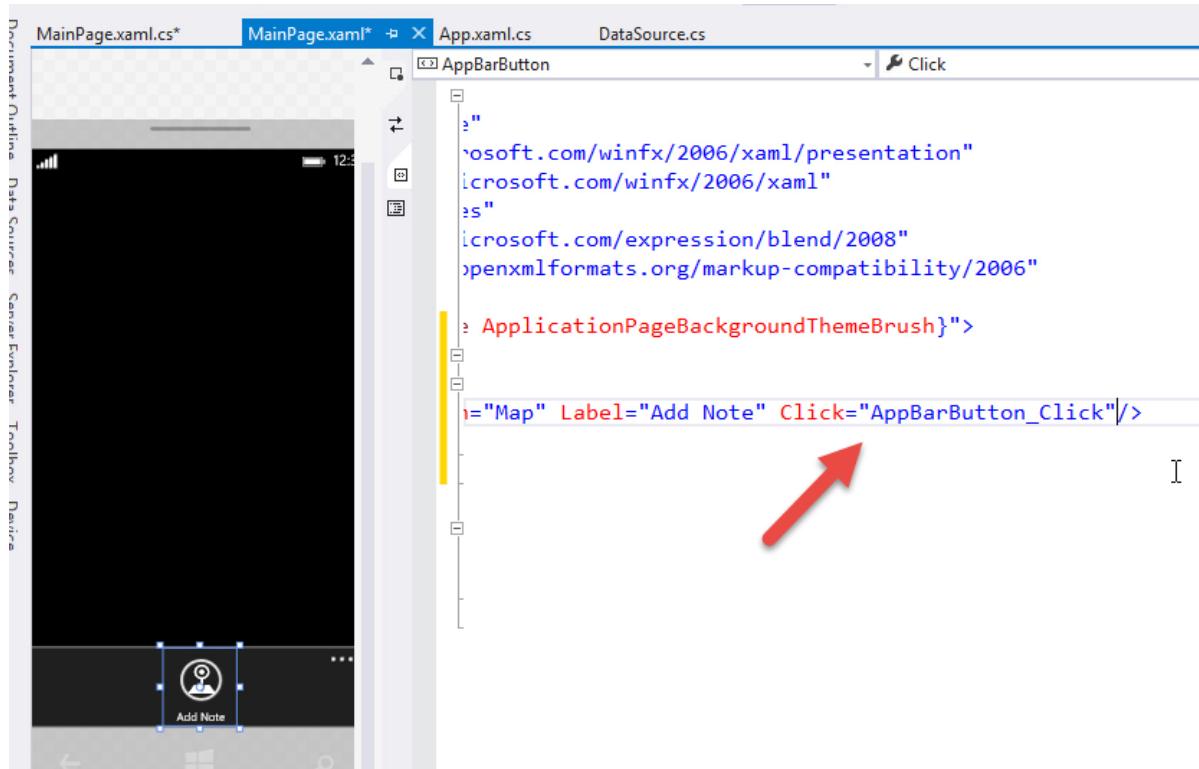
I'll (1) click the Add button to add a new AppBarButton, (2) select the newly added button in the Items list box on the left, then (3) make sure to change the Icon to type SymbolIcon, and (4) select the Map symbol from the dropdown list:



(1) I'll scroll down to the Label property and set that to "Add Note", then (2) click the OK button:



Finally, in the XAML editor, I want to add a click event handler for the new button, so I'll add a Click="AppBarButton_Click" event handler method.



I'll use the F12 technique to create the method stub in the MainPage.xaml.cs file. Here, I'll want to set the DataContext for the page to the collection of MapNotes in the OnNavigatedTo() method (like we've demonstrated before). I'll also Frame.Navigate() to AddMapNote.xaml when the user clicks the AppBarButton:

```

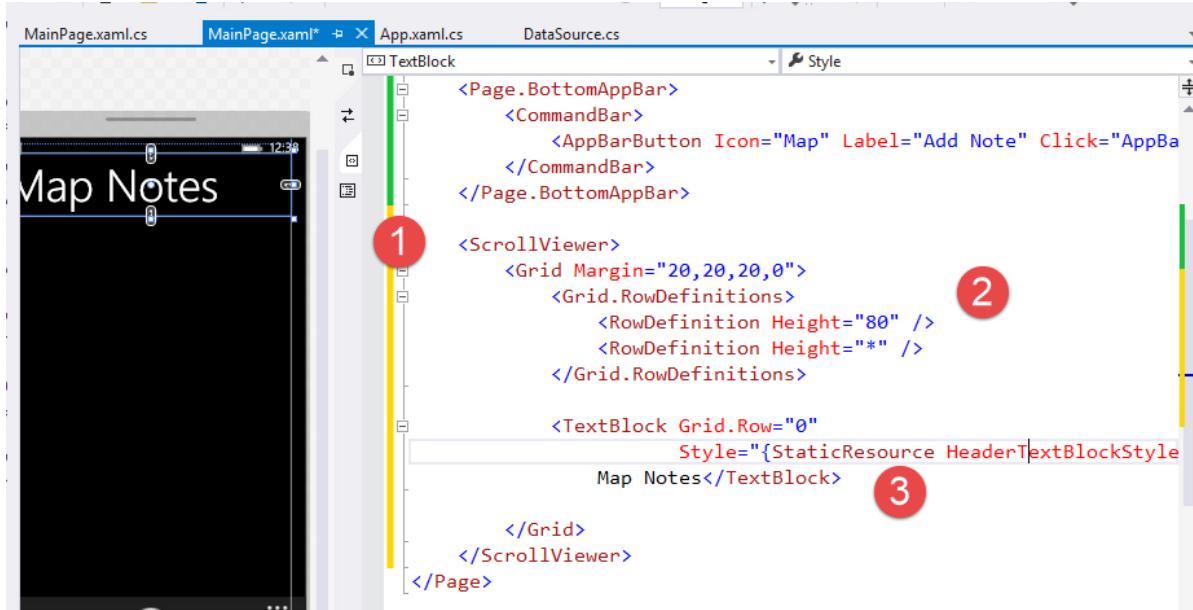
/// <summary>
/// Invoked when this page is about to be displayed in a Frame.
/// </summary>
/// <param name="e">Event data that describes how this page was reached.
/// This parameter is typically used to configure the page.</param>
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    var mapNotes = await App.DataModel.GetMapNotes();
    this.DataContext = mapNotes;
}

private void AppBarButton_Click(object sender, RoutedEventArgs e)
{
    Frame.Navigate(typeof(AddMapNote));
}

```

Back on the MainPage.xaml, I'll lay out the main area of the page by (1) surrounding the Grid with a ScrollViewer so that we can view a long list of items, (2) I'll add two RowDefinition

objects to create an area for the app title and for the list of items, (3) I'll add a TextBlock with the title of the app into that first RowDefinition with the style set to the built-in HeaderTextBlockStyle:

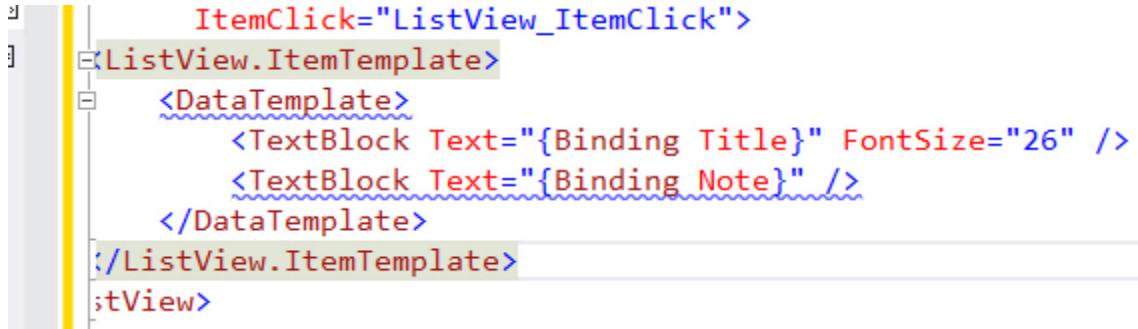


I'll add a ListView control into the second RowDefinition, binding its ItemsSource to the Page's DataContext, and setting properties to ensure that a given list item can be tapped, not selected. This involves setting the SelectionMode to none, the IsItemClickEnabled to true, and creating an event handler for the ItemClick event. Furthermore, I'll add the ItemTemplate and DataTemplate so that we can begin fleshing those out in the next step:

```
<ListView ItemsSource="{Binding}"
    Grid.Row="1"
    SelectionMode="None"
    IsItemClickEnabled="True"
    ItemClick="ListView_ItemClick">
    <ListView.ItemTemplate>
        <DataTemplate>

            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
```

Each list view item will be comprised of two TextBlocks ... one bound to the Title of the note, the other to the Note property of the MapNote:



```
        ItemClick="ListView_ItemClick">
<ListView.ItemTemplate>
    <DataTemplate>
        <TextBlock Text="{Binding Title}" FontSize="26" />
        <TextBlock Text="{Binding Note}" />
    </DataTemplate>
</ListView.ItemTemplate>
;tView
```

Now, we'll move on to the AddMapNote.xaml page. Here, I'll add four row definitions and set the margins on the Grid:

```
<Grid Margin="10,0,10,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="40" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

</Grid>
```

I'll leave the top row empty in order to give the app some vertical spacing. I suppose I could add my app's title or other branding elements there as well. In the second row, I'll add a StackPanel containing the Title and Note TextBoxes:

```
<StackPanel Grid.Row="1">
    <TextBlock Text="Title:" />
    <TextBox x:Name="titleTextBox" TextWrapping="Wrap" />
    <TextBlock Text="Note:"/>
    <TextBox x:Name="noteTextBox" TextWrapping="Wrap" Height="125" />
</StackPanel>
```

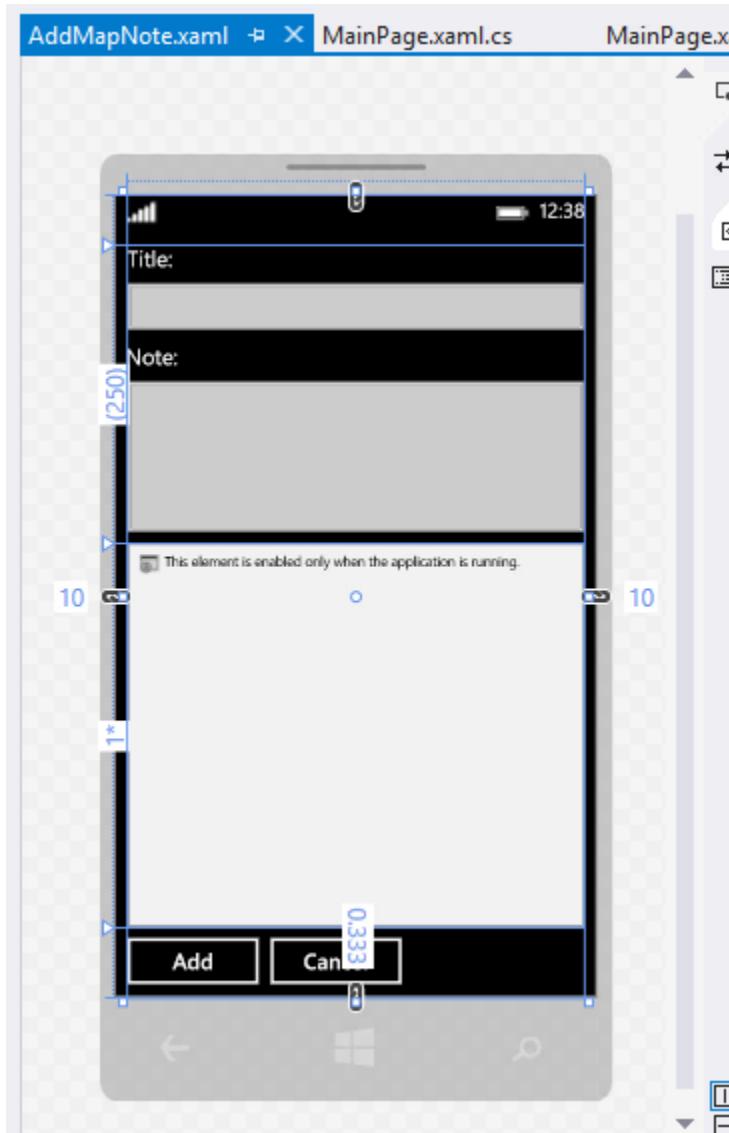
Next, I'll drag-and-drop a Map Control to the designer which adds the MapControl and its namespace to the Page. I'll clean out the extraneous properties and set it to the third row:

```
<Maps:MapControl x:Name="MyMap" Grid.Row="2" />
```

In the fourth row, I'll add two buttons in a StackPanel along with event handler methods for their Click events:

```
<StackPanel Orientation="Horizontal" Grid.Row="3">
    <Button x:Name="addButton" Content="Add" Click="addButton_Click"
Margin="0,0,10,0" />
    <Button x:Name="cancelButton" Content="Cancel" Click="cancelButton_Click" />
</StackPanel>
```

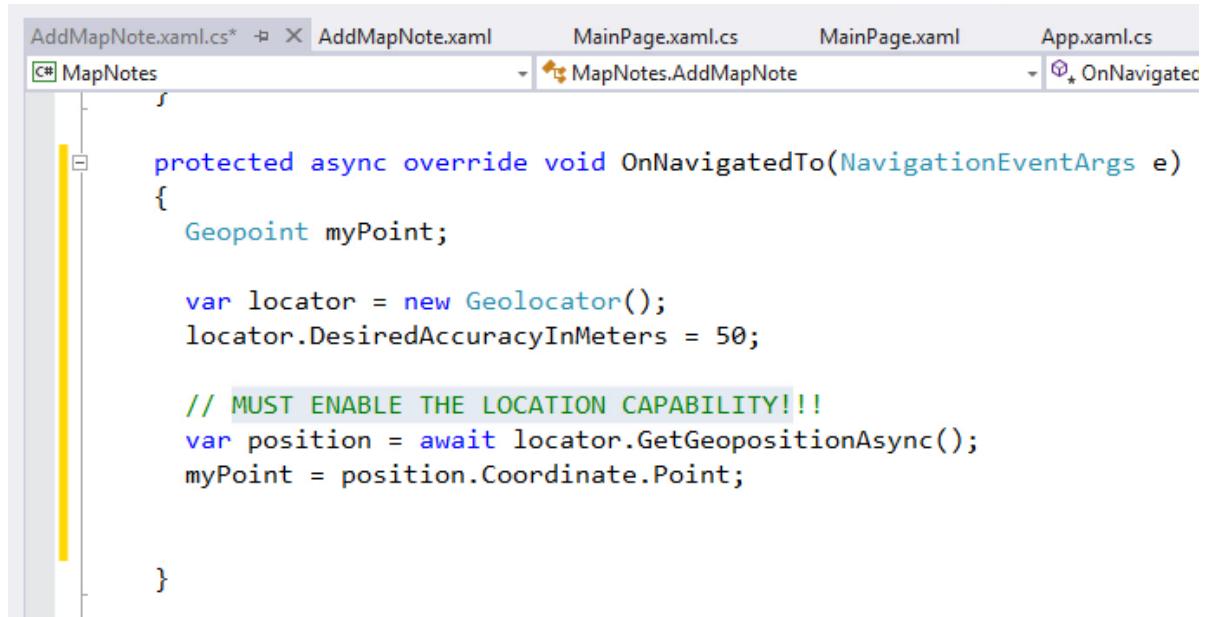
When I'm finished, the preview looks like this:



While the name of the page is `AddMapNote.xaml`, I intend to use it for two scenarios: adding a MapNote and viewing / deleting a MapNote. To accomplish this I'll need to often determine the current scenario I'm in ... am I adding or viewing / deleting?

We see how that affects the `OnNavigatedTo()` event handler ... if I'm adding a new MapNote, then I'll want to determine the current location of the Phone. However, if I'm viewing / deleting a MapNote, I'll want to retrieve the location from the existing MapNote and set the Map Control to display it on the Page.

I'll start with the "Add" scenario. I'll use the Geolocator object to determine the current position:



```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    Geopoint myPoint;

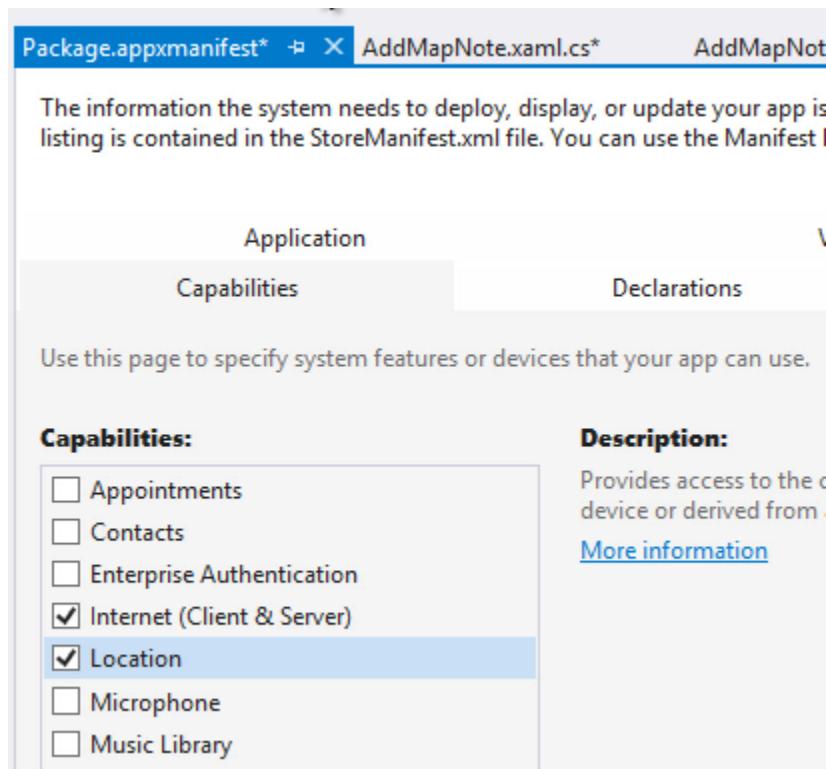
    var locator = new Geolocator();
    locator.DesiredAccuracyInMeters = 50;

    // MUST ENABLE THE LOCATION CAPABILITY!!!
    var position = await locator.GetGeopositionAsync();
    myPoint = position.Coordinate.Point;

}
```

I still have a lot of work to do here, such as set the Map Control to the current Geopoint, however this is a good start.

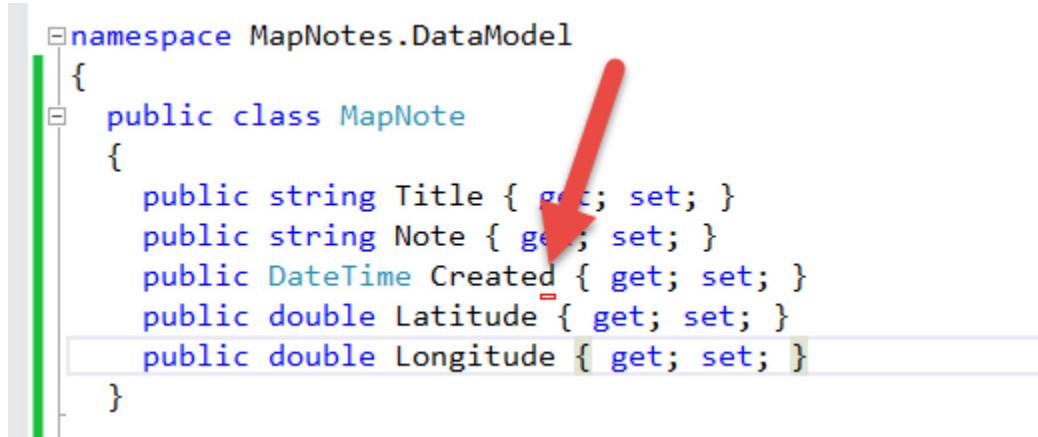
The note reminds me I need to add a Capability for Location. I'll open the Package.appxmanifest file, go to the Capabilities page, and click the check box next to Location:



Back in the AddMapNote.xaml, I'll handle the "Add" scenario for the addButton_Click() event handler method. I'll need to resolve some namespaces, and I'll need to fix a problem with the names of one of my properties which I misspelled (will do that in the next step). Here, we'll create a new instance of the MapNote class, set its properties including the Latitude and Longitude, then call the App.DataModel.AddMapNote() passing in the new instance of MapNote. Finally, I'll navigate back to the MainPage.xaml:

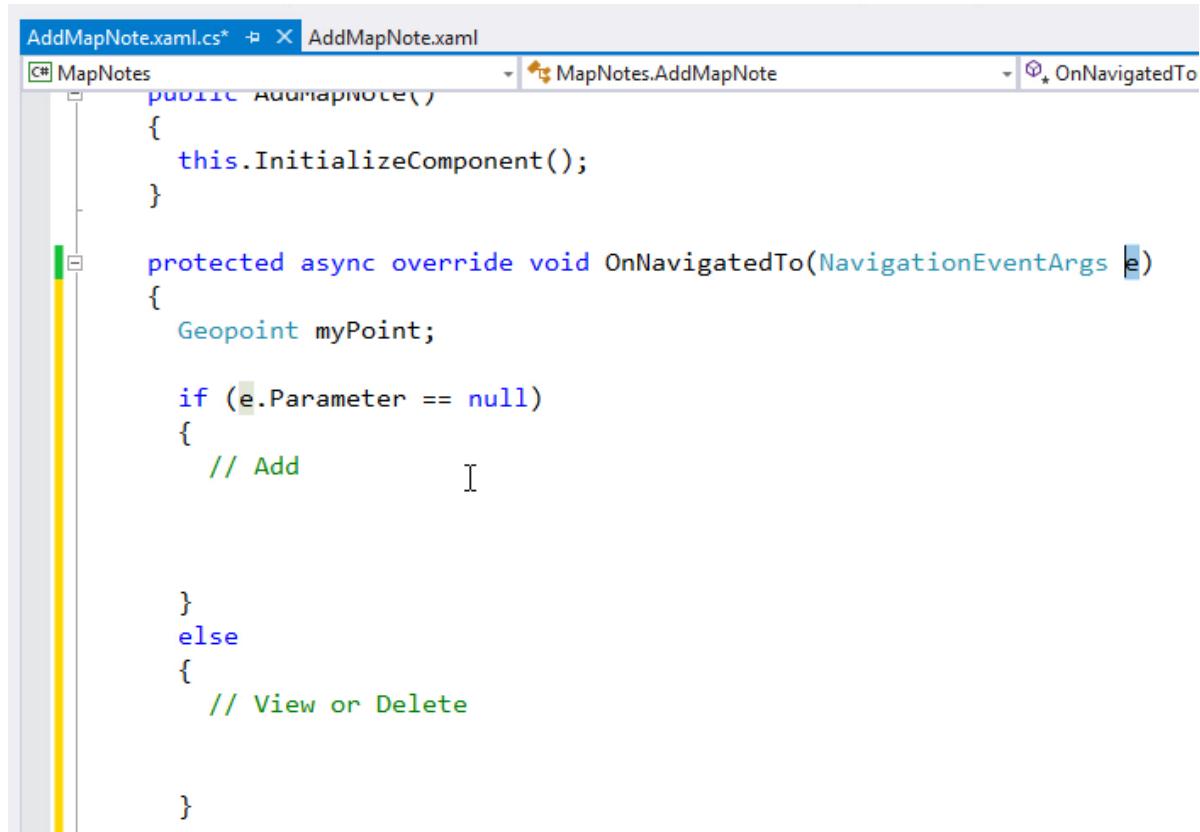
```
private void addButton_Click(object sender, RoutedEventArgs e)
{
    // Add
    MapNote newMapNote = new MapNote();
    newMapNote.Title = titleTextBox.Text;
    newMapNote.Note = noteTextBox.Text;
    newMapNote.Created = DateTime.Now;
    newMapNote.Latitude = MyMap.Center.Position.Latitude;
    newMapNote.Longitude = MyMap.Center.Position.Longitude;
    App.DataModel.AddMapNote(newMapNote);
    Frame.Navigate(typeof(MainPage));
}
```

As I said a moment ago, I realize I forgot the letter 'd' on the property named Created. I'll fix that back in the MapNote class:



Back in the AppNoteNote.xaml.cs, I want to think about the OnNavigatedTo() event handler again. This time, I want to think about how I'll handle both the "add" and "view / delete" scenarios. I'll use the NavigationEventArgs input parameter that contains the input

parameter from the previous page. If the parameter is empty (null), that will signal the “add” scenario. Otherwise, it will signal the “view / delete” scenario. The objective of either case is to determine the Geopoint so that I can set the map to it:



The screenshot shows the Visual Studio IDE with the code editor open. The title bar indicates the file is AddMapNote.xaml.cs. The code itself is as follows:

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

protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    Geopoint myPoint;

    if (e.Parameter == null)
    {
        // Add
    }

    else
    {
        // View or Delete
    }
}
```

I'll copy and paste the code I previously had added to OnNavigatedTo() into the “add” case:

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    Geopoint myPoint;

    if (e.Parameter == null)
    {
        // Add
        var locator = new Geolocator();
        locator.DesiredAccuracyInMeters = 50;

        // MUST ENABLE THE LOCATION CAPABILITY!!!
        var position = await locator.GetGeopositionAsync();
        myPoint = position.Coordinate.Point;
    }
    else
    {
        // View or Delete
    }
}
```

Then in the “view / delete” case I’ll cast the NavigationEventArgs’s Parameter property to MapNote and populate the textboxes. I’ll also create a Geopoint based on the latitude and longitude of the MapNote. Finally, outside of the if statement, I attempt to call MyMap.TrySetViewAsync() passing in the Geopoint that was created in either case, as well as a default zoom level:

```
,  
else  
{  
    // View or Delete  
  
    var mapNote = (MapNote)e.Parameter;  
    titleTextBox.Text = mapNote.Title;  
    noteTextBox.Text = mapNote.Note;  
    addButton.Content = "Delete";  
  
    var myPosition = new Windows.Devices.Geolocation.BasicGeoposition();  
    myPosition.Latitude = mapNote.Latitude;  
    myPosition.Longitude = mapNote.Longitude;  
  
    myPoint = new Geopoint(myPosition);  
  
}  
  
await MyMap.TrySetViewAsync(myPoint, 16D);  
}
```

I realize that when I click the Add button, I'll need some way to determine whether we're currently in the "add" or "view / edit" scenario, so I create a bool flag called isViewing:

```
/// </summary>
public sealed partial class AddMapNote : Page
{
    private bool isViewing = false;

    public AddMapNote()
    {
        InitializeComponent();
    }
}
```

In the "add" scenario, I set it to false:

```
Geopoint myPoint;

if (e.Parameter == null)
{
    // Add
    isViewing = false;
}

var locator = new Geolocator();
locator.DesiredAccuracyInMeters = 50;
```

In the "view / delete" scenario, I set it to true. Now I can tell what scenario I'm handling anywhere in the AddMapNote.xaml.cs:

```

else
{
    // View or Delete
    isViewing = true;

    var mapNote = (MapNote)e.Parameter;
    titleTextBox.Text = mapNote.Title;
}
```

I'll use the isViewing flag to branch the logic of what happens in each scenario:

```
private void addButton_Click(object sender, RoutedEventArgs e)
{
    if (isViewing)
    {
        // Delete
    }
    else
    {

        // Add
        MapNote newMapNote = new MapNote();
        newMapNote.Title = titleTextBox.Text;
        newMapNote.Note = noteTextBox.Text;
    }
}
```

I'll clean up a little by removing an extraneous Frame.Navigate():

```
else
{
    // Add
    MapNote newMapNote = new MapNote();
    newMapNote.Title = titleTextBox.Text;
    newMapNote.Note = noteTextBox.Text;
    newMapNote.Created = DateTime.Now;
    newMapNote.Latitude = MyMap.Center.Position.Latitude;
    newMapNote.Longitude = MyMap.Center.Position.Longitude;
    App.DataModel.AddMapNote(newMapNote);
    Frame.Navigate(typeof(MainPage));
}

Frame.Navigate(typeof(MainPage));
```



If we're currently in the “view / delete” scenario, I'll need to first delete the MapNote, then navigate back to MainPage.xaml. However, I have a problem ... I currently have no way of getting to the MapNote that was loaded into the page. This will require I step back a bit and create a private reference to the MapNote...

```
private void addButton_Click(object sender, RoutedEventArgs e)
{
    if (isViewing)
    {
        // Delete
        App.DataModel.DeleteMapNote(mapNote);
        Frame.Navigate(typeof(MainPage));
    }
    else
```

... here I add a private field of type MapNote. Now, when in the “view / delete” scenario, I’ll use that instead of my locally scoped variable of the same name:

```
/// \summary
public sealed partial class AddMapNote : Page
{
    private bool isViewing = false;
    private MapNote mapNote; | 
    public AddMapNote()
```

Therefore, I’ll remove the var keyword in front of the mapNote like so:

```
else
{
    // View or Delete
    isViewing = true;

    mapNote = (MapNote)e.Parameter;
    titleTextBox.Text = mapNote.Title;
```



When someone clicks the delete button, I want a popup dialog to ask the user if they’re sure they want to delete the MapNote. This will prevent an accidental deletion. To accomplish this, I add code to display a MessageDialog object. The MessageDialog will have two Commands (rendered as buttons) ... “Delete” and “Cancel”. Regardless of which one the user clicks, both will trigger the execution of a handler method called “CommandInvokedHandler”. Finally, there are two lines of code that are unnecessary in this instance, but useful when creating a Windows Store app: I set the default command that should be executed when the user clicks the Esc key on their keyboard. Again, not pertinent here, but it won’t hurt and you

can see how to implement that. Finally, once we've properly set up the MessageDialog we call ShowAsync() to display it to the user:

```
    if (isViewing)
    {
        // Delete
        //Delete
        var messageDialog = new Windows.UI.Popups.MessageDialog("Are you sure?");

        // Add commands and set their callbacks; both buttons use the same callback f
        messageDialog.Commands.Add(new UICommand(
            "Delete",
            new UICommandInvokedHandler(this.CommandInvokedHandler)));
        messageDialog.Commands.Add(new UICommand(
            "Cancel",
            new UICommandInvokedHandler(this.CommandInvokedHandler)));

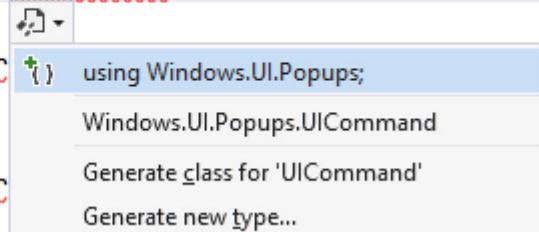
        // Set the command that will be invoked by default
        messageDialog.DefaultCommandIndex = 0;

        // Set the command to be invoked when escape is pressed
        messageDialog.CancelCommandIndex = 1;

        // Show the message dialog
        await messageDialog.ShowAsync();
```

Of course, I'll have to resolve namespaces by adding some using statements:

```
// Add commands and set their callbacks; both buttons use the
messageDialog.Commands.Add(new UICommand(
    "Delete",
    new UICommandInvokedHandler
messageDialog.Commands.Add(new
    "Cancel",
    new UICommandInvokedHandler
```



The screenshot shows an IDE's code editor with a dropdown menu open over the word 'UICommand'. The menu contains the following items: '+> using Windows.UI.Popups;', 'Windows.UI.Popups.UICommand', 'Generate class for 'UICommand'', and 'Generate new type...'. The first item, 'using Windows.UI.Popups;', is highlighted with a blue background.

Next, I'll implement the CommandInvokedHandler(). I'll use the Label property of the command button that was clicked to determine which action to take, whether Cancel or Delete. I'm only interested in the Delete scenario. I'll call DeleteMapNote() then Frame.Navigate():

```
    }
    private void CommandInvokedHandler(IUICommand command)
    {
        if (command.Label == "Delete")
        {
            App.DataModel.DeleteMapNote(mapNote);
            Frame.Navigate(typeof(MainPage));
        }
    }
}
```

I'll need to clean up those two lines of code in the addbutton_Click() event handler method since I don't need them any more. In the screenshot below, I removed them both from the "view / delete" scenario:

```
// Set the command that will be invoked by default
messageDialog.DefaultCommandIndex = 0;

// Set the command to be invoked when escape is pressed
messageDialog.CancelCommandIndex = 1;

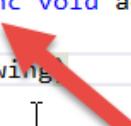
// Show the message dialog
await messageDialog.ShowAsync();
}

```



Also, to resolve all of the compilation errors, I'll need to add the async keyword since I'm calling a method that can be awaited:

```
private async void addButton_Click(object sender, RoutedEventArgs e)
{
    if (isViewing)
    {
        // Delete
    }
}
```



When I'm finished, this should be the result ... the entire code listing for AddMapNote.xaml.cs:

```
using MapNotes.DataModel;
```

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Devices.Geolocation;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Popups;
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.Navigation;

namespace MapNotes
{
    public sealed partial class AddMapNote : Page
    {
        private bool isViewing = false;
        private MapNote mapNote;

        public AddMapNote()
        {
            this.InitializeComponent();
        }

        protected async override void OnNavigatedTo(NavigationEventArgs e)
        {
            Geopoint myPoint;

            if (e.Parameter == null)
            {
                // Add
                isViewing = false;

                var locator = new Geolocator();
                locator.DesiredAccuracyInMeters = 50;

                // MUST ENABLE THE LOCATION CAPABILITY!!!
                var position = await locator.GetGeopositionAsync();
                myPoint = position.Coordinate.Point;
            }
        }
    }
}

```

```

    }
else
{
    // View or Delete
    isViewing = true;

    mapNote = (MapNote)e.Parameter;
    titleTextBox.Text = mapNote.Title;
    noteTextBox.Text = mapNote.Note;
    addButton.Content = "Delete";

    var myPosition = new Windows.Devices.Geolocation.BasicGeoposition();
    myPosition.Latitude = mapNote.Latitude;
    myPosition.Longitude = mapNote.Longitude;

    myPoint = new Geopoint(myPosition);
}
await MyMap.TrySetViewAsync(myPoint, 16D);
}

private async void addButton_Click(object sender, RoutedEventArgs e)
{
    if (isViewing)
    {
        // Delete
        var messageDialog = new Windows.UI.Popups.MessageDialog("Are you sure?");

        // Add commands and set their callbacks; both buttons use the same callback function
        // instead of inline event handlers
        messageDialog.Commands.Add(new UICommand(
            "Delete",
            new UICommandInvokedHandler(this.CommandInvokedHandler)));
        messageDialog.Commands.Add(new UICommand(
            "Cancel",
            new UICommandInvokedHandler(this.CommandInvokedHandler)));

        // Set the command that will be invoked by default
        messageDialog.DefaultCommandIndex = 0;

        // Set the command to be invoked when escape is pressed
        messageDialog.CancelCommandIndex = 1;

        // Show the message dialog
    }
}

```

```

        await messageDialog.ShowAsync();

    }
    else
    {
        // Add
        MapNote newMapNote = new MapNote();
        newMapNote.Title = titleTextBox.Text;
        newMapNote.Note = noteTextBox.Text;
        newMapNote.Created = DateTime.Now;
        newMapNote.Latitude = MyMap.Center.Position.Latitude;
        newMapNote.Longitude = MyMap.Center.Position.Longitude;
        App.DataModel.AddMapNote(newMapNote);
        Frame.Navigate(typeof(MainPage));
    }
}

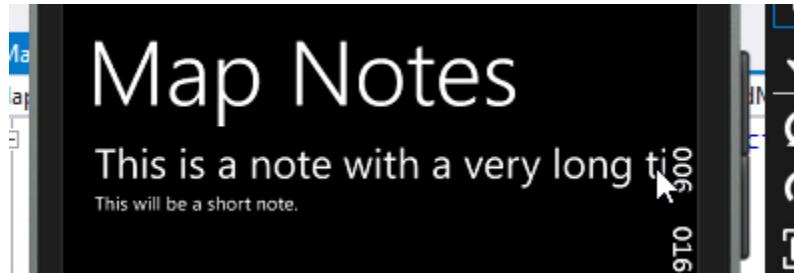
private void cancelButton_Click(object sender, RoutedEventArgs e)
{
    Frame.Navigate(typeof(MainPage));
}

private void CommandInvokedHandler(IUICommand command)
{
    if (command.Label == "Delete")
    {
        App.DataModel.DeleteMapNote(mapNote);
        Frame.Navigate(typeof(MainPage));
    }
}
}

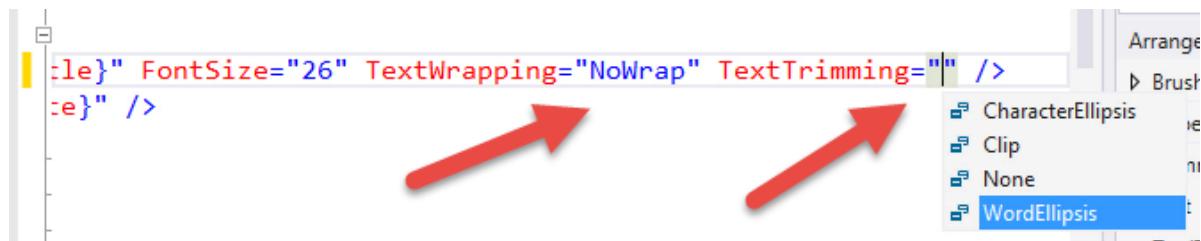
```

The app should work as I test all the scenarios of adding, viewing and deleting a MapNote as I move the Emulator's location around the world.

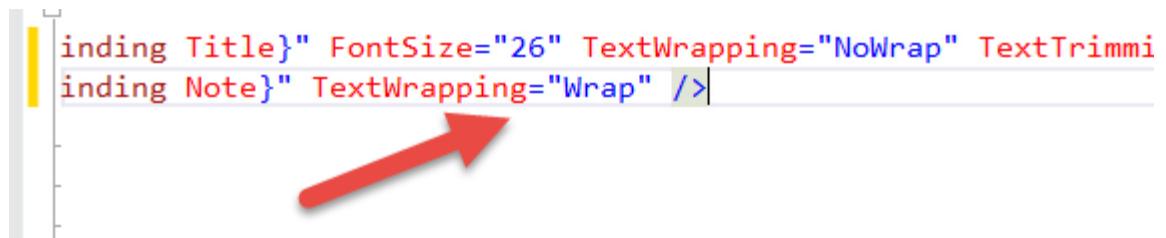
The final change I want to make is to account for the possibility that a given note has a very long title. Currently, the letters will disappear off the right-hand side or possibly wrap to the next line (unless you set the height of the TextBlock):



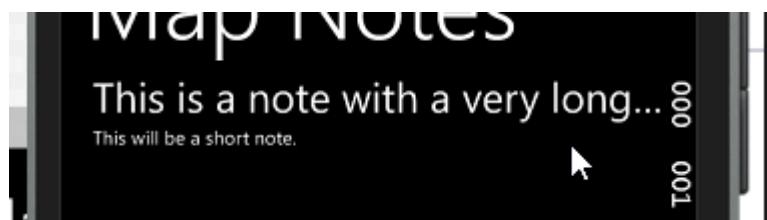
At the very least, I can add a few properties to prevent both wrapping AND trim the final letters that will appear off screen using the TextWrapping and TextTrimming properties, respectively:



I'll also add the TextWrapping property to the Note TextBlock:



Now, my MapNotes will appear correctly on screen:



Another successful Exercise. Hopefully this exercise was helpful in cementing many of the ideas we've already covered. At a minimum, you saw how we were able to reuse a lot of the

DataSource / Data Model code. If your data model is flat (i.e., not a deep hierarchy of relationships between classes), you now have a good recipe / template that you can re-use. It allows you to store your object graph to the Phone's storage and grab it back out. All you have to do is create a new class that can be serialized. Even if you have Commands defined you can add the `IgnoreDataMemberAttribute` to those members that you do not want to be serialized.

We also looked at how to utilize the Map Control, how to change its Geoposition using the Geolocator and how to create an instance of a Geopoint using a latitude, longitude, how to display a MessageDialog and handle the callback function and much more.

Lesson 30: Series Conclusion

Let me congratulate you on making it all the way through this series. Think about how far you've come in such a short amount of time! It takes a high degree of commitment to work your way through 10 hours of content, but you did it, and you definitely have what it takes to see a project through to the end. And so, I'm confident that you can build the next great app. I would encourage you to take your time, aim high, and test, test, test your app to make sure your app is polished and ready for others to use it.

I love to hear from people who have watched these series I've created and who have built an app and submitted it to the app store. That's probably the best feedback that we can get ... that this series helped you in some small way and that you actually built an app and released it into the Windows Store. If that describes you, and if you need a beta tester, by all means please send me a tweet @bobtabor or write me at bob@learnvisualstudio.net.

Before I wrap this up, I want to thank Larry Lieberman and Matthias Shapiro who sponsored this series. If you like this content, please let them know ... there's a feedback link on the footer, or just leave a comment below this video.

Also, I want to thank Channel9 for their ongoing support of my work, including Golnaz Alibeigi, and those I've worked with in the past who have taught me so much, like Clint Rutkas who got a major promotion to the Halo team and of course, Dan Fernandez who I've been working with for over 10 years and who has keep me connected with Microsoft. It is much appreciated.

Allow me to make one last plug for my website, www.LearnVisualStudio.NET. I try to share everything I know with my clients about C#, Visual Studio, Windows and web development, data access, architecture and more.

Finally, I honestly wish you the best in your career and in life. Thank you.