UWP-037 - Utilizing the VisualStateManager to Create Adaptive Triggers

Starting in this lesson we'll focus on the techniques and strategies for creating responsive apps that adapt based on the given device.

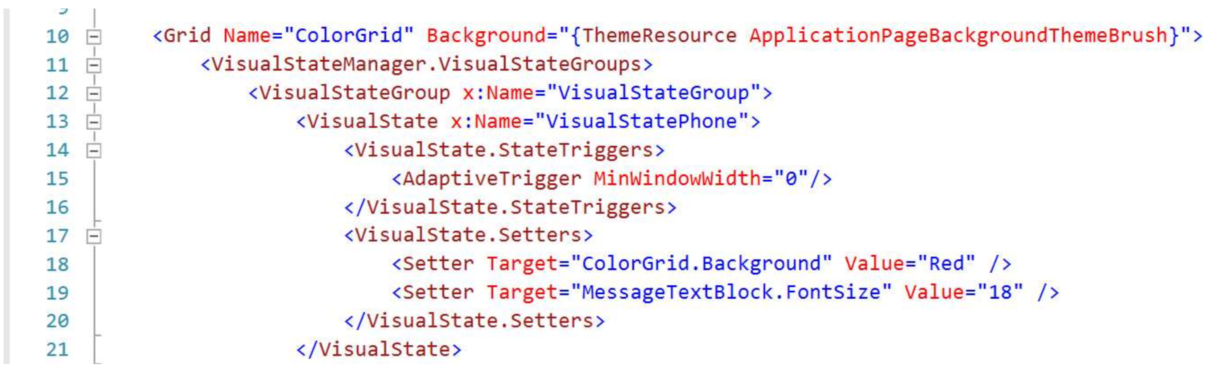
In the resources that accompany this lesson, please open and start debugging SimpleVisualStateTriggerExample. Once running, resize the app's window as large and as small as possible. As you can see, a certain screen sizes the color of the background and the size of the font changes.

This is made possible by a class called the VisualStateManager and it does exactly what it sounds like; it manages the visual state of your application, including the position, sizes, colors, fonts, etc. -- virtually any properties on any objects you want to manipulate based on the current size of the window.

To use the VisualStateManager, you define a series of VisualStates with StateTriggers (what should prompt a change) and Setters (the values of target object properties that should change).

Think about how that applies to the Universal Windows Platform. One of the selling points is that you're able to write one code base and then use it across all these different form factors. This allows you to accommodate different screen resolutions for different form factors with the same code base. The VisualStateManager will be leverages to change the entire layout of your application based on the screen size.

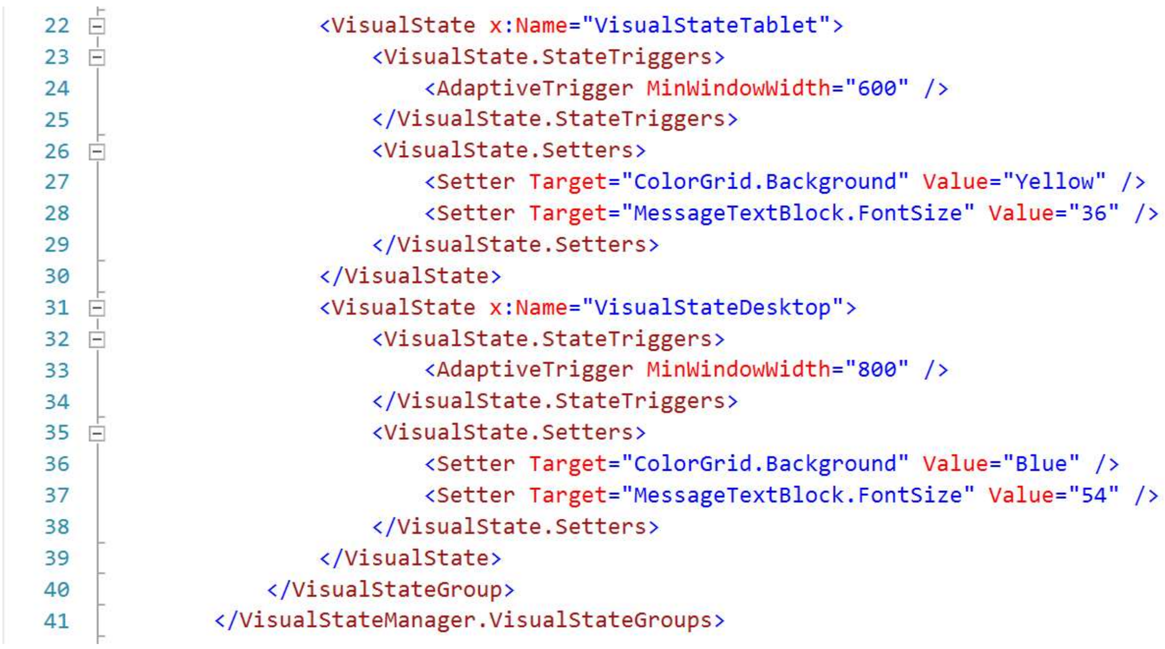
Here's one of the three VisualStates defined for the project.



```
<Grid Name="ColorGrid" Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="VisualStateGroup">
            <VisualState x:Name="VisualStatePhone">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="0"/>
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="ColorGrid.Background" Value="Red" />
                    <Setter Target="MessageTextBlock.FontSize" Value="18" />
                </VisualState.Setters>
            </VisualState>
```

First you define one or more VisualStateGroups. Each VisualStateGroup is comprised of a series of VisualStates. This VisualState is called "VisualStatePhone" inferring that this "state" is valid for the smallest Windows 10 form factor. Each VisualState is comprised of one or more StateTriggers and one or more Setters. The StateTrigger is set by using the AdaptiveTrigger element along with a MinWindowWidth or MinWindowHeight. In this case, since the MinWindowWidth="0" the Setters will apply to any screen size. In this case, there are two Setters: the first changes the ColorGrid's Background to red, and the MessageTextBlock's FontSize to 18.

There are two other VisualStates defined below it.



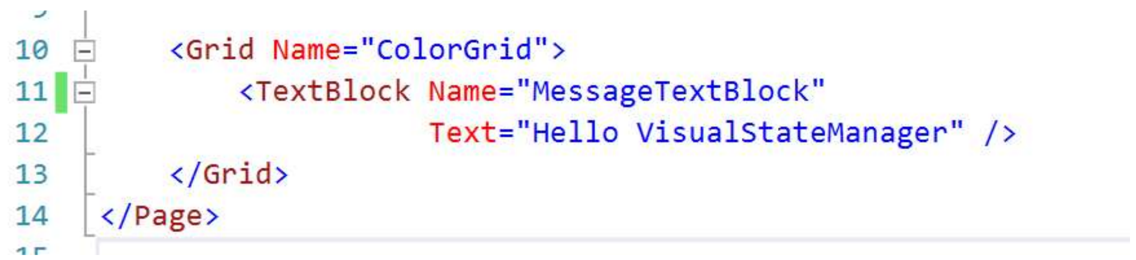
```xml
                <VisualState x:Name="VisualStateTablet">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="600" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Target="ColorGrid.Background" Value="Yellow" />
                        <Setter Target="MessageTextBlock.FontSize" Value="36" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="VisualStateDesktop">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="800" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Target="ColorGrid.Background" Value="Blue" />
                        <Setter Target="MessageTextBlock.FontSize" Value="54" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
```

The VisualStateTablet will modify those same object properties when the window's width is 600 or greater. The VisualStateDesktop will modify those same object properties when the window's width is 800 or greater.

Conceptually it's very easy to understand. This ability to define visual states allows you to get creative with how you want to make changes to your application to conform to a given screen size. We define the Trigger and then once that Trigger is fired off, we apply the Setters.

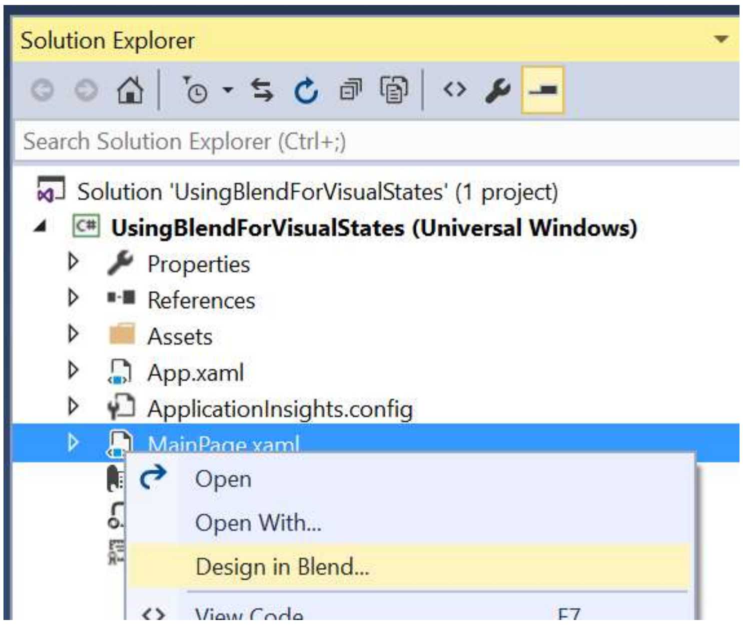
While conceptually it is easy to understand, it may be difficult at first to remember exactly what objects and their relationship to each other is necessary to get this to work. Microsoft Blend will help you build VisualStates if you prefer to use a visual editor.

I've created a new project called UsingBlendForVisualStates, and I have the same color Grid and TextBlock, but I haven't added anything else to it just yet.



```xml
    <Grid Name="ColorGrid">
        <TextBlock Name="MessageTextBlock"
                   Text="Hello VisualStateManager" />
    </Grid>
</Page>
```

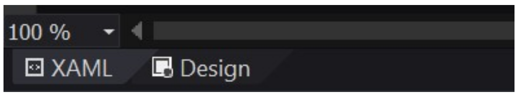
Here in the Solution Explorer right-click on the project and select Design in Blend.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'UsingBlendForVisualStates' (1 project)
UsingBlendForVisualStates (Universal Windows)
Properties
References
Assets
App.xaml
ApplicationInsights.config
MainPage.xaml
Open
Open With...
Design in Blend...
View Code F7

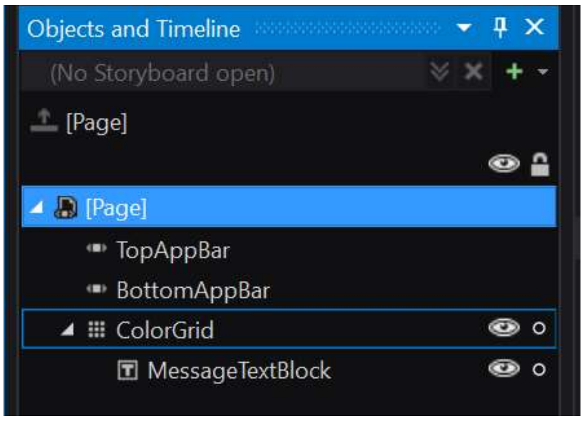
And so Blend is a tool that is usually installed along with Visual Studio (unless you chose custom install and de-selected it for installation). Blend was intended developers who focus on aesthetic design work. It has many of the same features of VisualStudio. By default, on the right-hand side is the Properties window. A Solution Explorer docked by default on the left. While things are in slightly different positions the overall look and feel should be familiar.

There are two things that Blend does for you that you can't easily do in Visual Studio. First, is it gives you tools to creating VisualStates in a visual manner. Second, there are tools that allow you to working with animation, which we're not going to talk about in this series.

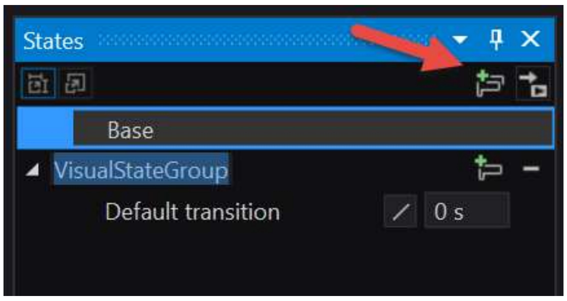
With the MainPage.xaml open in the main area, I'll switch to design view by clicking the tab at the very bottom.



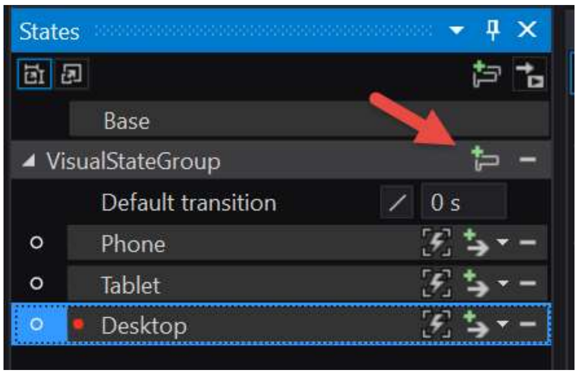
The Objects and Timeline window is typically docked on the left-hand side. I can drill down and see the ColorGrid and whatever is contained inside the ColorGrid, in this case, the MessageTextBlock.



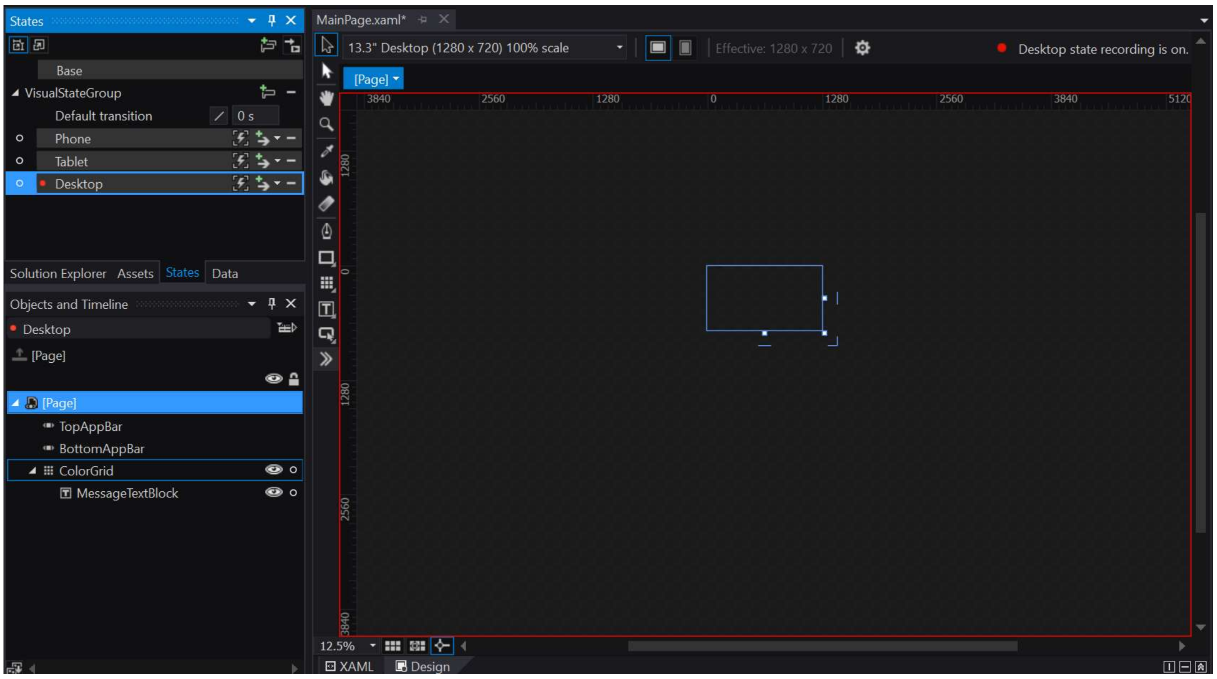
Next, I'll go to the States tab that's usually docked up on the upper left-hand corner. I can add a new "state" by adding a state group, so I'll click the small button in the upper right-hand corner (see red arrow, below). This creates a VisualStateGroup.



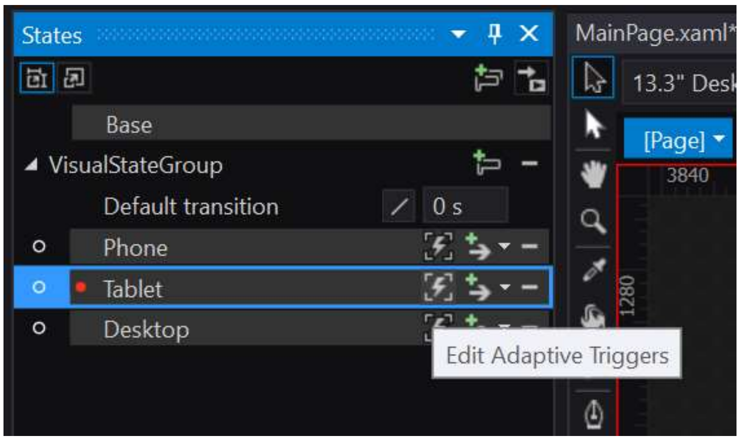
In the default VisualStateGroup, I'll click the small button to its right (see red arrow, below) to create a VisualState. I will call this state "Phone". I'll add another state called "Tablet". And add another state that I call "Desktop".



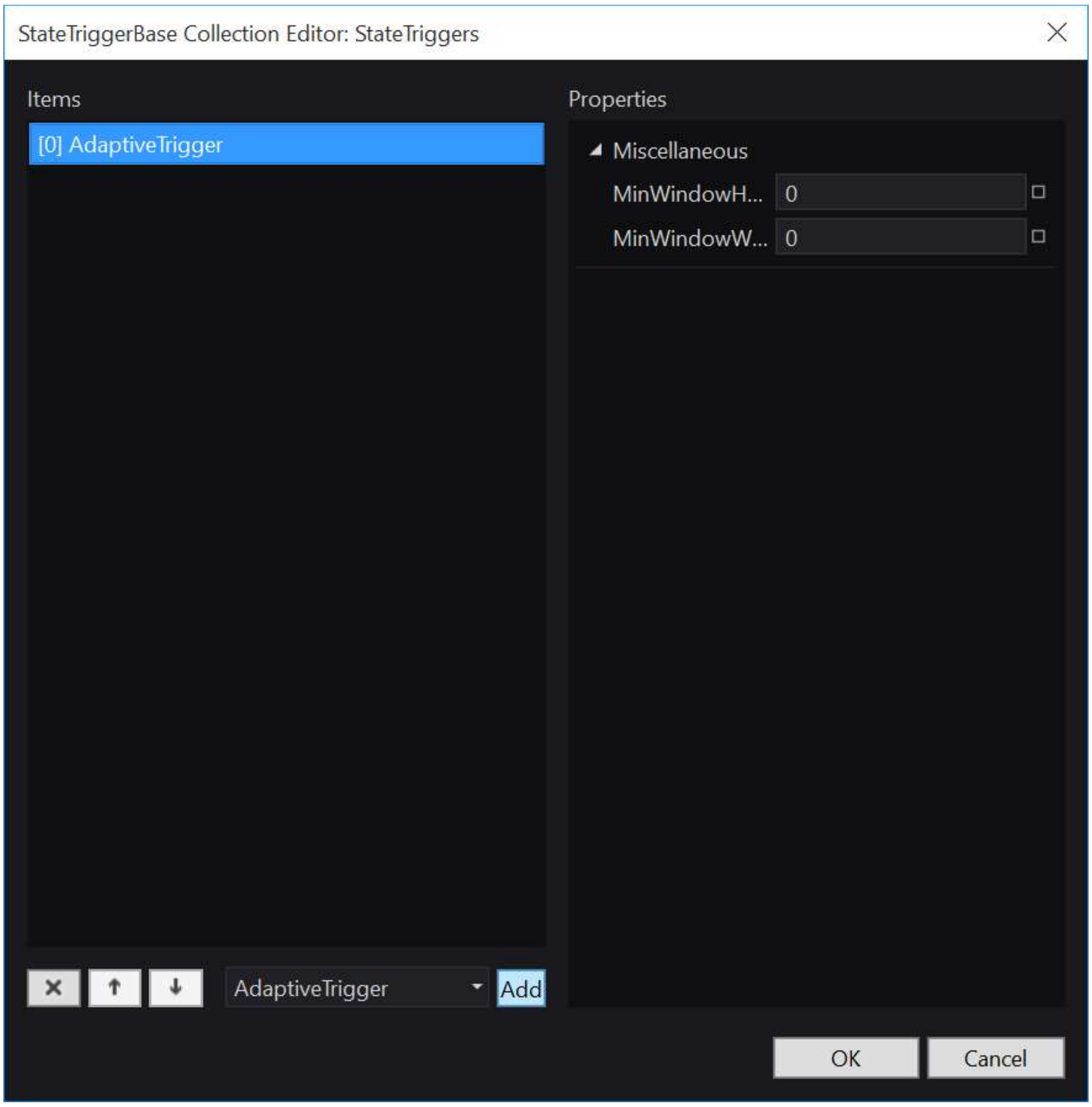
And notice that whenever I select any of these, that a little red light is selected on the left-hand side, and there are a red border all around the designer area. You might also see that Phone state recording is on.



When select the Tablet VisualState, any changes I make in the Design and Properties window will create VisualState Setters. To create the Triggers click the lightning bolt icon to the right:



This will open a dialog that allows you to first add an AdaptiveTrigger (bottom, left), then set its MinWindowHeight and / or MinWindowWidth.



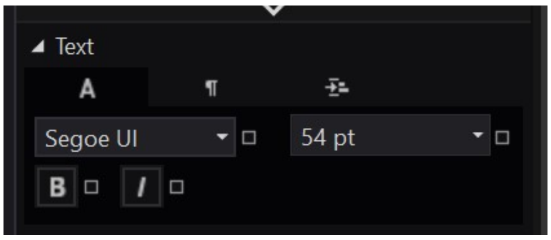
MinWindowH... 0
MinWindowW... 0

Since we're working with the Tablet state, I'll set the MinWindowWidth to 600, then click OK.

Note: The sizes I'm choosing are arbitrary numbers. You would want to test this for your own app.

I'll repeat the steps to create an AdaptiveTrigger for the Desktop VisualState setting the MinWindowWidth to 800.

When in Desktop mode I want the MessageTextBlock's FontSize to be 54. I select the MessageTextBox in the Objects and Timeline window, then in the Properties window I locate the Text section and change the font's size to 54.

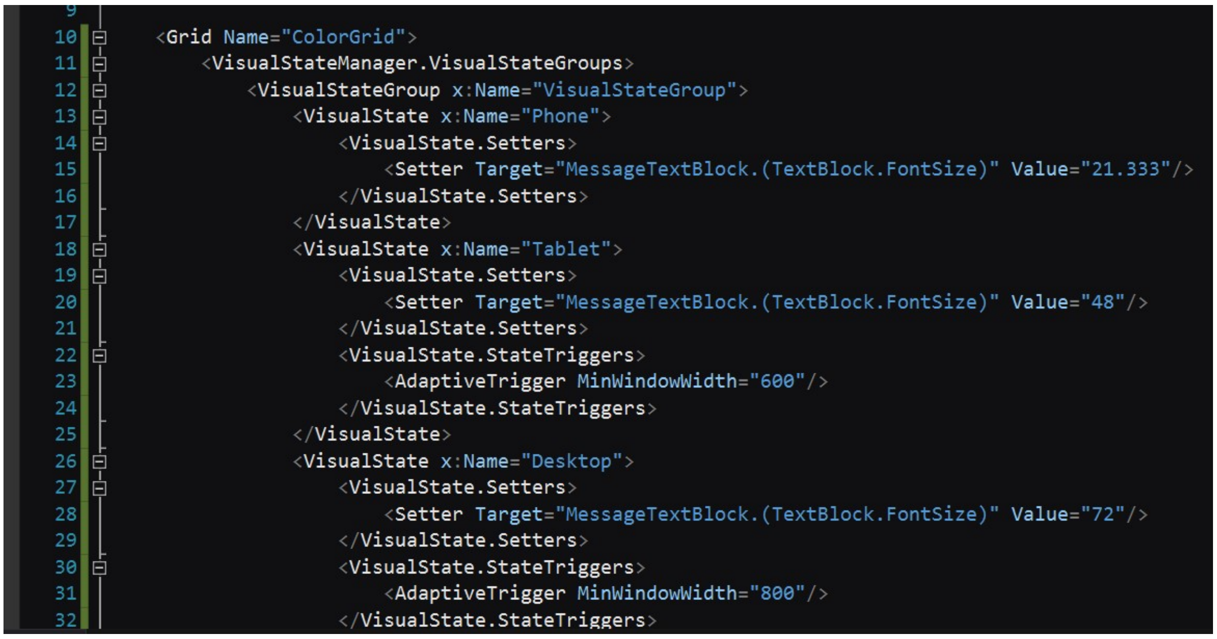


I'll repeat this process for the Tablet VisualState changing the MessageTextBlock's FontSize to 36.

Finally, I'll repeat this process for the Phone VisualState changing the MessageTextBlock's FontSize to 16.

Running the application, you should see the TextBlock's FontSize change as you resize the window.

If you peek at the XAML designer you can see all the XAML that was generated for you by Blend.



```
    <Grid Name="ColorGrid">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="VisualStateGroup">
                <VisualState x:Name="Phone">
                    <VisualState.Setters>
                        <Setter Target="MessageTextBlock.(TextBlock.FontSize)" Value="21.333"/>
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="Tablet">
                    <VisualState.Setters>
                        <Setter Target="MessageTextBlock.(TextBlock.FontSize)" Value="48"/>
                    </VisualState.Setters>
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="600"/>
                    </VisualState.StateTriggers>
                </VisualState>
                <VisualState x:Name="Desktop">
                    <VisualState.Setters>
                        <Setter Target="MessageTextBlock.(TextBlock.FontSize)" Value="72"/>
                    </VisualState.Setters>
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="800"/>
                    </VisualState.StateTriggers>
```

I prefer to just type it in myself. While Blend does create nice clean XAML, but there are some cases like working with colors, and brushes where the generated XAML was overly verbose.

When you close down Blend, you'll return to Visual Studio and you may see that a message asking whether you want to re-load the file because it has changed.

I want to emphasize that this lesson is a foundational concept for building real Universal Windows Platform applications. In the next lesson we'll use the VisualStateManager to create a more realistic app that changes its layout based on the available screen resolution.

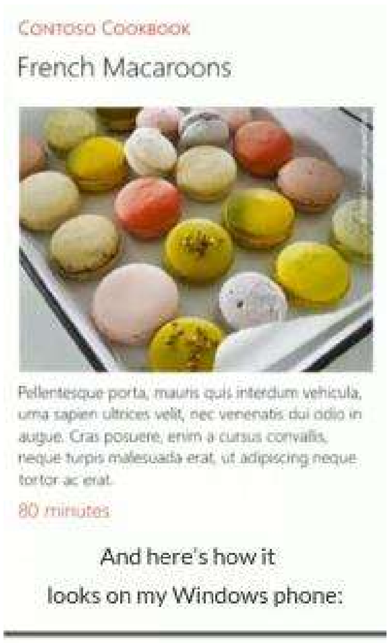
UWP-038 - Working with Adaptive Layout

In this lesson we will cover a bit on adaptive layout. This takes what was covered in the previous lesson, talked about the nuts and bolts of actually using the Visual State Manager and AdapativeTriggers to change attributes of objects in XAML based on screen size, to a higher level.

Adaptive layout is critical to understanding the Universal Windows Platform story, where we build one code base and we can use it across multiple form factors. Here is a creative example that illustrates this, found on “Wintellect's Blog” by Jeff Prosise, who wrote how 'To Build Adaptive UIs in Windows 10'.

http://bit.do/adaptive-ui

This lesson is based on Jeff’s “Contoso Cookbook” example.



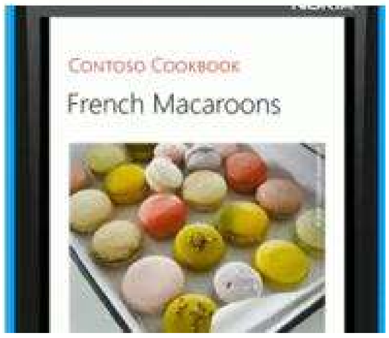
CONTOSO COOKBOOK

French Macaroons

Pellentesque porta, mauris quis interdum vehicula, urna sapien ultrices velit, nec venenatis dui odio in augue. Cras posuere, enim a cursus convallis, neque turpis malesuada erat, ut adipiscing neque tortor ac erat.

80 minutes

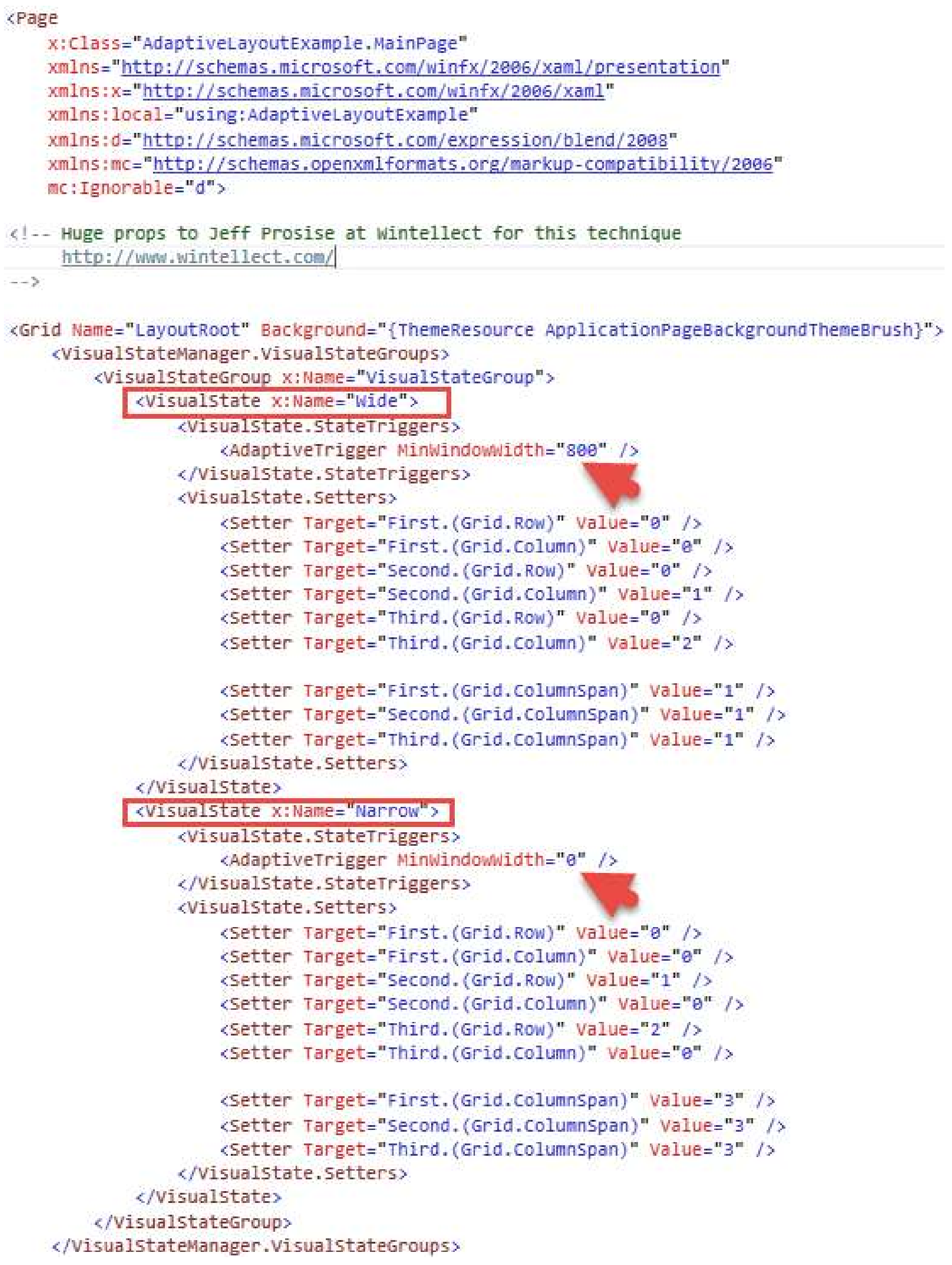
And here's how it looks on my Windows phone:



CONTOSO COOKBOOK

French Macaroons

Here is an example for how you could construct an adaptive layout in your MainPage.xaml

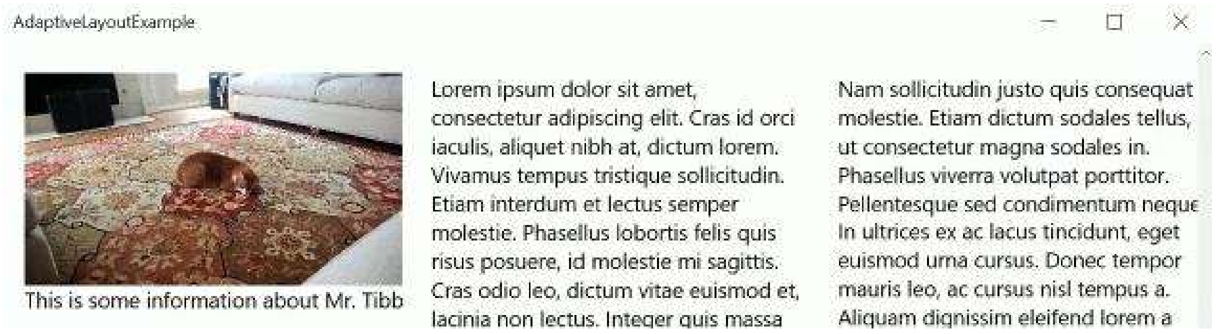


```
<Page
    x:Class="AdaptiveLayoutExample.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AdaptiveLayoutExample"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/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

<!-- Huge props to Jeff Prosise at Wintellect for this technique
     http://www.wintellect.com/
-->

<Grid Name="LayoutRoot" Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="VisualStateGroup">
            <VisualState x:Name="Wide">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="800" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="First.(Grid.Row)" Value="0" />
                    <Setter Target="First.(Grid.Column)" Value="0" />
                    <Setter Target="Second.(Grid.Row)" Value="0" />
                    <Setter Target="Second.(Grid.Column)" Value="1" />
                    <Setter Target="Third.(Grid.Row)" Value="0" />
                    <Setter Target="Third.(Grid.Column)" Value="2" />

                    <Setter Target="First.(Grid.ColumnSpan)" Value="1" />
                    <Setter Target="Second.(Grid.ColumnSpan)" Value="1" />
                    <Setter Target="Third.(Grid.ColumnSpan)" Value="1" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="Narrow">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="0" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="First.(Grid.Row)" Value="0" />
                    <Setter Target="First.(Grid.Column)" Value="0" />
                    <Setter Target="Second.(Grid.Row)" Value="1" />
                    <Setter Target="Second.(Grid.Column)" Value="0" />
                    <Setter Target="Third.(Grid.Row)" Value="2" />
                    <Setter Target="Third.(Grid.Column)" Value="0" />

                    <Setter Target="First.(Grid.ColumnSpan)" Value="3" />
                    <Setter Target="Second.(Grid.ColumnSpan)" Value="3" />
                    <Setter Target="Third.(Grid.ColumnSpan)" Value="3" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

Notice the different VisualState settings for either a Wide or a Narrow layout. And below all of that is a series of StackPanels

```
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <ScrollViewer Grid.Row="1">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*"/>
                    <ColumnDefinition Width="*"/>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>

                <StackPanel Name="First" Margin="20,20,0,0">
                    <Image Source="Assets/Tibbles.jpg" HorizontalAlignment="Left" />
                    <TextBlock>Information on my cat, Mr. Tibbles</TextBlock>
                </StackPanel>
                <StackPanel Name="Second" Grid.Row="1" Margin="20,20,0,0">
                    <TextBlock TextWrapping="Wrap">
                        Lorem ipsum dolor...
                    </TextBlock>
                </StackPanel>
                <StackPanel Name="Third" Grid.Row="2" Margin="20,20,0,0">
                    <TextBlock TextWrapping="Wrap">
                        Nam sollicitudin justo ..
                    </TextBlock>
                </StackPanel>
            </Grid>
        </ScrollViewer>
    </Grid>
</Page>
```

You can see how the image and text is organized within the StackPanels, forming 3 distinct sections (labeled accordingly), and when you run it in a wide viewport it would appear something like this



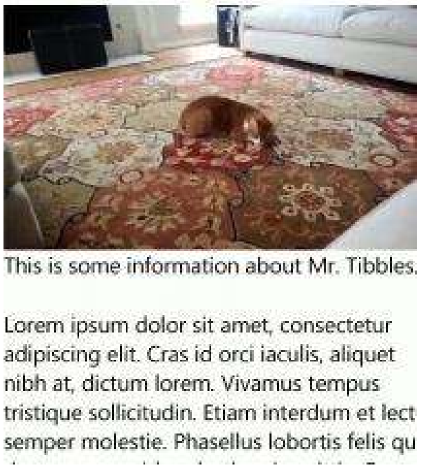
AdaptiveLayoutExample

This is some information about Mr. Tibb

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras id orci iaculis, aliquet nibh at, dictum lorem. Vivamus tempus tristique sollicitudin. Etiam interdum et lectus semper molestie. Phasellus lobortis felis quis risus posuere, id molestie mi sagittis. Cras odio leo, dictum vitae euismod et, lacinia non lectus. Integer quis massa

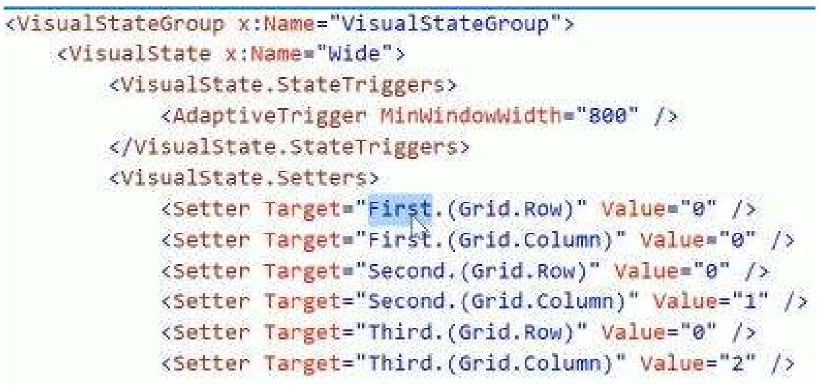
Nam sollicitudin justo quis consequat molestie. Etiam dictum sodales tellus, ut consectetur magna sodales in. Phasellus viverra volutpat porttitor. Pellentesque sed condimentum neque In ultrices ex ac lacus tincidunt, eget euismod urna cursus. Donec tempor mauris leo, ac cursus nisl tempus a. Aliquam dignissim eleifend lorem a

However, when the viewport dramatically narrows (such as on a Phone) it stacks the columns on top of each other, rendering a result that looks something like this

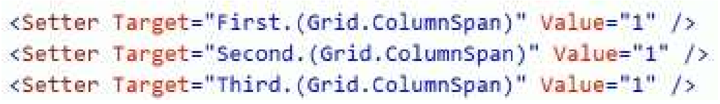

This is some information about Mr. Tibbles.

Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Cras id orci iaculis, aliquet
nibh at, dictum lorem. Vivamus tempus
tristique sollicitudin. Etiam interdum et lect
semper molestie. Phasellus lobortis felis qu

Now, this switching between visual states is done up near the top of the code, within the VisualStateManager

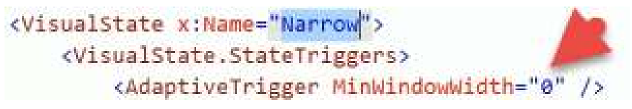

```
<VisualStateGroup x:Name="VisualStateGroup">
    <VisualState x:Name="Wide">
        <VisualState.StateTriggers>
            <AdaptiveTrigger MinWindowWidth="800" />
        </VisualState.StateTriggers>
        <VisualState.Setters>
            <Setter Target="First.(Grid.Row)" Value="0" />
            <Setter Target="First.(Grid.Column)" Value="0" />
            <Setter Target="Second.(Grid.Row)" Value="0" />
            <Setter Target="Second.(Grid.Column)" Value="1" />
            <Setter Target="Third.(Grid.Row)" Value="0" />
            <Setter Target="Third.(Grid.Column)" Value="2" />
```

So, in the wide states, it takes constructs the StackPanel (here labeled, “First,” “Second,” “Third”) with the row/column arrangement you would expect to find in a wide viewport. And, the column span is also set accordingly below that

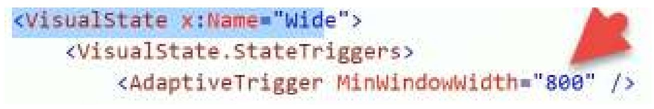

```
<Setter Target="First.(Grid.ColumnSpan)" Value="1" />
<Setter Target="Second.(Grid.ColumnSpan)" Value="1" />
<Setter Target="Third.(Grid.ColumnSpan)" Value="1" />
```

And, of course you can understand the narrow state using much the same kind of reasoning

So, essentially the different visual states activate depending on if the width of the device (or the size of the window on the device) is either less than 800 pixels (the narrow state), or greater than 800 pixels (the wide state)



```
<VisualState x:Name="Narrow">
    <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0" />
```



```
<VisualState x:Name="Wide">
    <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="800" />
```

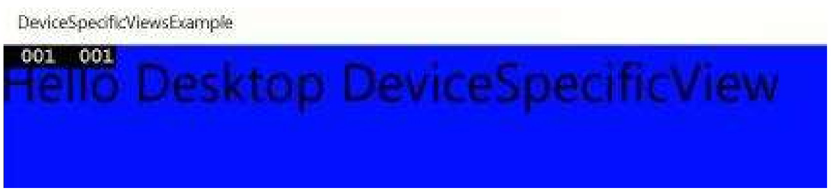
So, what this demonstrates it that you have the tools to make your application change layout based on the break points you set, and you get to decide how those layout changes need to happen. It's all up to you. So, you will want to choose carefully, and test, thinking each step of the way what would make sense on the phone versus sitting on the couch, scrolling through and working with a larger application. Consider how the layout changes, as well as how the user expects to interact with the UI depending on these different contexts.

UWP-039 - Adaptive Layout with Device Specific Views

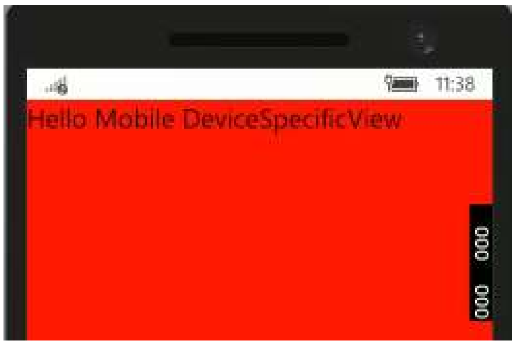
Having now looked at how to use AdaptiveTriggers, and the VisualStateManager, to change the layout of your application based on the current window size, let's now take a look at a second technique that allows you to create a dedicated view for a given device family that your application could potentially run on.

This would work by identifying the device that your app is running on, and then triggering a dedicated view, or what's called the "DeviceSpecificView." It's a very simple technique that Microsoft has made extremely easy to use, however there are a few advanced techniques which you will see in a great article that will be presented to you later. But, first, here is a demo of this technique in action.

Below is an example app running as an ordinary Windows app



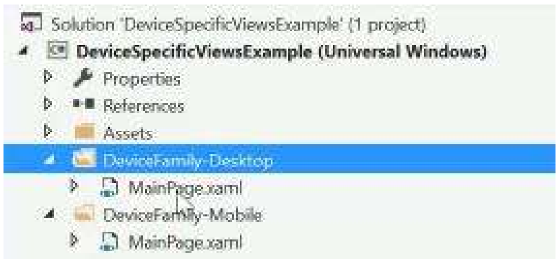
And, then running on Windows Phone (via the Visual Studio emulator)



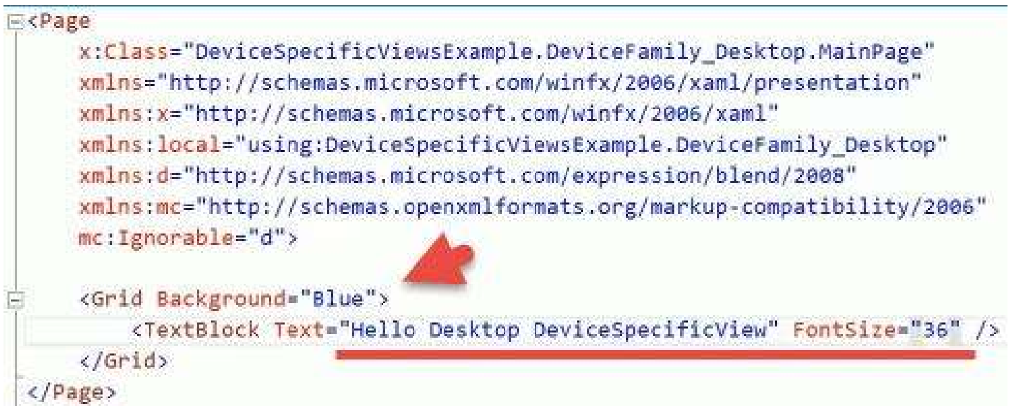
On the phone you can see a dramatically different background and font shows up. Also, notice that the wording is changed to reflect the platform the app is running on.

Let's go through the steps detailing how you could get this kind of result (bear in mind, this is the simplest possible way of doing this in order to just show the technique used. You can take this as far as you want)

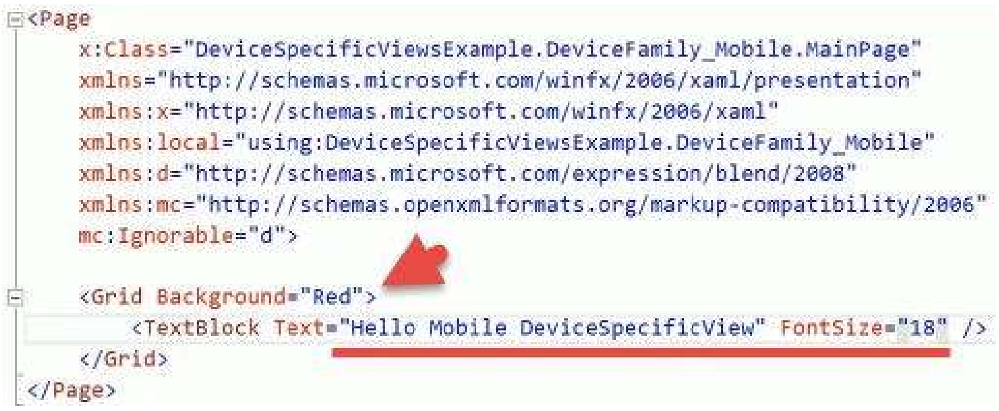
Within your project you would create 2 separate folders for the Desktop/Mobile settings, respectively. Notice how each folder has its own MainPage.xaml


Solution 'DeviceSpecificViewsExample' (1 project)
DeviceSpecificViewsExample (Universal Windows)
Properties
References
Assets
DeviceFamily-Desktop
MainPage.xaml
DeviceFamily-Mobile
MainPage.xaml

And in each folder's MainPage.xaml are the specific settings for each intended device



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

    <Grid Background="Blue">
        <TextBlock Text="Hello Desktop DeviceSpecificView" FontSize="36" />
    </Grid>
</Page>
```



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

    <Grid Background="Red">
        <TextBlock Text="Hello Mobile DeviceSpecificView" FontSize="18" />
    </Grid>
</Page>
```

It may immediately strike you that this is a much cleaner, simpler solution to adaptive layout than using the VisualStateManager in the previous lesson. That is definitely one of the benefits of using this approach. However, you may still want to use a VisualStateManager or AdaptiveTriggers for your application because there are different screen resolutions, even within the context of these different device families.

For more information on these techniques, including advanced uses, refer to this helpful resource

http://bit.do/device-specific-views