THE EXPERT'S VOICE®

Alpha

Apress®

What's an Alpha Book?

An Alpha Book is a means to give you access to a book's content prior to publication date. They contain only a limited number of chapters in a unedited unfinished pre-release format. Once the final eBook is ready, you'll have full access to that.

How much does an Alpha Book cost?

Alpha Books are priced the same as regular eBooks.

What format is an Alpha Book?

Alpha Books are available as PDF files. Once the finished book publishes, other formats will become available, which you'have access to, just as with a regular eBook.

Where is my Alpha Book's source code?

We don't offer source code with Alpha books. Authors submit any source code once they've finished writing the book, whereupon we offer it for download.

This book was purchased by arosner@rosnertech.com

Apress®

Table of Contents

1. Introduction and Tooling
2. Controls
3. Data Binding
4. Views
5. Local Data
6. Remote data and services
7. Charms and Contracts
8. Notifications
9. Application Model
10. Location & Other Sensors
11. Windows Store

The following chapters are present: 3,4,5,6,7,8

Controls

What's New in Windows 8.1

Windows 8.1 brings serveral new controls as well as updates existing controls. The new controls that we will be discussing in this chapter are the DatePicker, TimePicker, Flyout and MenuFlyout controls.

Updates include the ability to add a Header to the DatePicker, TimePicker, TextBox, PasswordBox, RichEditBox, Combobox, and Slider, and Placeholder Text (watermarks) to the TextBox, PasswordBox, RichEditBox, Combobox,

Getting started with Windows 8Windows 8.1 programming is as easy as opening a new blank project and dragging some controls onto the design surface. You can add controls in any of the following ways:

- Open Blend and drag a control onto the design surface
- Open Blend and create the control by hand in the XAML
- Open Visual Studio and drag a control onto the design surface
- Open Visual Studio and drag a control onto the XAML
- Open Visual Studio and create the control by hand in the XAML

Almost too many choices. For this book, we'll assume that you are doing most of your work in Visual Studio, though we strongly agree that doing the design work (laying out controls, etc.) can often best be done in Blend.

Dragging a control from the toolbox onto either the design surface or the XAML itself is a great way to add controls to your page. If your control needs a special namespace, Visual Studio will add it for you automatically if you drag the control into place; otherwise you'll have to add the namespace by hand.

This chapter will cover some of the more important controls and panels, but does not endeavor to be exhaustive. A few controls, such as GridView won't be covered in detail until later in the book.

The best way to get started looking at the controls is to create a new project. Open Visual Studio, select New Project and select Windows Store application – Blank App (XAML) as shown in figure 1

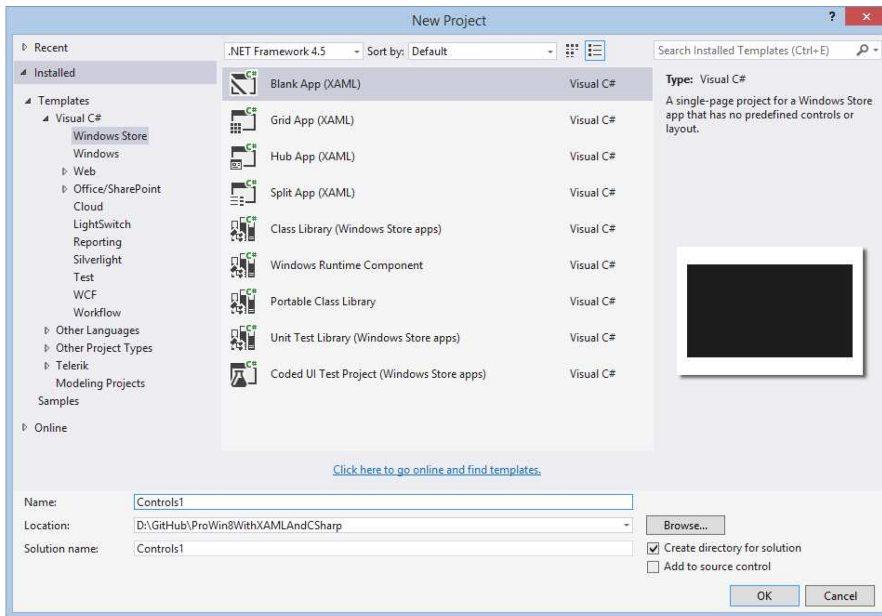


Figure 1 – New Project

Name your project *Controls1*.

Themes

Windows 8 and Windows 8.1 applications use a dark theme by default. Using a dark theme is more user and battery friendly for tablet based applications. However, based on your application, the light theme might create a better user experience. In Windows 8, the Application Theme can be changed in App.xaml by specifying RequestedTheme=”Light || Dark” as in the following code:

```
<Application
    x:Class="Controls1.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Controls1"
    RequestedTheme="Light">
```

In Windows 8.1 applications, setting the RequestedTheme in App.xaml changes the foreground theme but not the background theme for panels.

To see the dark and light themes in action, add a TextBlock to the Grid in MainPage.xaml, and setting the Text to “Hello, World”. The resulting code is shown here:

```
<Grid>
    <TextBlock Text="Hello"/>
</Grid>
```

When you run the program, “Hello, World” is shown in light text on a dark background.

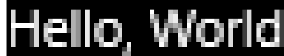


Figure 2 – Text Block with the default Dark theme

In App.xaml, add the RequestedTheme="Light to the <Application /> tag:

```
<Application
  x:Class="Controls1.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Controls1"
  RequestedTheme="Dark">
```

When you run the program again, all that shows is a dark screen. This is because the control is set to use the light theme. In the light theme, controls are expecting a light background, so they are configured to use dark fonts. Panels (covered in the next section) must have their background set to use the ApplicationPageBackgroundThemeBrush for the RequestedTheme to take effect.

Change the Grid it use the ApplicationPageBackgroundThemeBrush as in the following code.

```
 <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
      <TextBlock Text="Hello, World"/>
  </Grid>
```

Run the application again, and you will see dark text on a light background, as in Figure 3.

Figure 3 – Text Block with Light theme and Grid configured to use the Requested Theme

Panels

You can't discuss controls without first discussing panels. Panels are a special kind of control that "hold" other controls, they supply a place to put your controls, and they help with managing the layout of your controls. Panels are not new to Windows 8.1, they have been in XAML since the beginning with WPF and SilverLight. Panels are a special type of UI control that can contain other controls. The type of panel determines the behavior of the contained controls (as described below). Conceptually, they are very similar to ASP.NET Placeholder controls in that they contain other controls and they can both have controls added through code during runtime. However, while ASP.NET Placeholder controls are just another option for creating User Interfaces, the XAML containers are are key components of creating User Interfaces.

There are a number of types of panels available out of the box with Windows 8Windows 8.1, the most important of which are:

- Canvas
- Grid
- StackPane

The Canvas

The Canvas is used primarily in games and other applications where you need precise (to the pixel) control over the placement of every object in your application. We'll show how to use the canvas, but then we won't return to it for the rest of this book, as our focus is not on games.

To start, open MainPage.xaml, and remove the Grid that was added to the page automatically and the TextBlox. by Visual StudioNext, and add a Canvas. Again, to add a canvas (or any other control) you can drag it from the toolbox onto the design surface or onto the XAML or you can enter it by hand by typing the following where the Grid used to be:,

```
<Canvas>

</Canvas>
```

SIDE BAR

Depending on how you add controls to your page, the resulting XAML can be very different. If you add controls by dragging a control from the ToolBox directly into the XAML editor, the resulting XAML will be very clean:

```
<Canvas />
```

If you add controls by dragging a control from the ToolBox onto the design surface in Visual Studio, there will be a lot more attributes set on the control as the design surface interprets where on the design surface the control is dropped. For example, in my test app, dragging onto the design surface resulted in the following XAML:

```
<Canvas HorizontalAlignment="Left" Height="100" Margin="221,399,0,0" Grid.Row="1" VerticalAlignment="Top" Width="100"/>
```

There are advantages and disadvantages to the different methods of adding controls. The best option is to try them all and determine what works best for you.

You'll place controls within the Canvas (that is, between the opening and closing tags). If you drag an Ellipse onto the design surface, Visual Studio will fill in a number of properties for you so that the Ellipse is visible,

```
<Ellipse Fill="#FFF4F4F5"
     Height="100"
     Canvas.Left="205"
     Stroke="Black"
     Canvas.Top="111"
     Width="100" />
```

Setting the Height and Width to the same value (in this case, 100) makes the Ellipse into a circle. The properties Canvas.Left and Canvas.Top set the location of the Ellipse with respect to the left and top boundaries of the Canvas. In this case, the Ellipse is 205 pixels to the right of the left margin and 111 pixels down from the top boundary.

Change the Fill property to "Red" to see the circle more clearly.

You can place as many objects as you like onto the canvas. Try the following XAML,

```
<Canvas>
  <Ellipse Fill="Red"
       Height="100"
       Canvas.Left="205"
       Stroke="Black"
       Canvas.Top="111"
       Width="100"
       Canvas.ZIndex="1"/>
  <Rectangle Fill="Blue"
        Height="188"
```

```
            Canvas.Left="82"
            Stroke="Black"
            Canvas.Top="40"
            Width="118" />
   <Rectangle Fill="Blue"
            Height="137"
            Canvas.Left="278"
            Stroke="Black"
            Canvas.Top="91"
            Width="140"
            Canvas.ZIndex="0"/>
</Canvas>
```

The result is shown in figure 42,

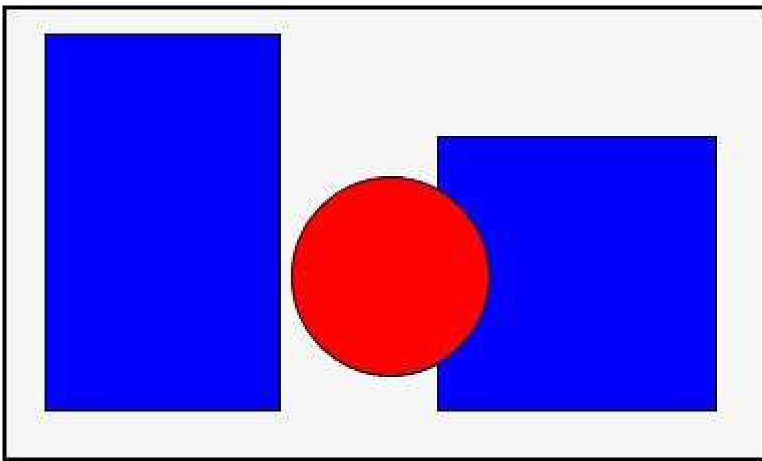


Figure 42 – Two Rectangles and an Ellipse

Normally, this would create two rectangles and a circle, and the second rectangle would partially cover the circle since it is declared after the ellipse. In this case, however, the ellipse occludes the rectangle because the z-Index was set higher for the ellipse (placing the ellipse "on top of" the rectangle). The z-index determines the layering as if the shapes were on a three-dimensional surface as shown in figure 53.

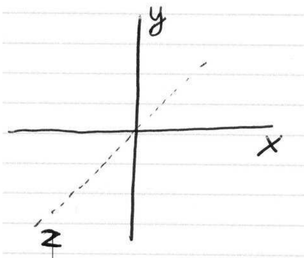


Figure 53 – Z-Index

The canvas really comes into its own when you work with animation, a topic beyond the scope of this book. we cover in chapter ?

The Grid

The Grid is the workhorse panel of Windows 8Windows 8.1. It is so popular and so ubiquitous that new projects are created with a Grid to get you started.

Typically, you will define rows and/or columns for your grid. You can define these using various measurements. The most common are to define the actual size,

```
<RowDefinition Height="50" />
```

Or by defining the relative size of two or more rows,

```
<RowDefinition Height="*" />
<RowDefinition Height="2*"
```

The above indicates that the second row will be twice as big as the first (and will take 2/3 of the available space). An asterisk alone indicates 1*.

Finally, you can declare a size to be “Auto” in which case it will size to the largest object in that row or column.

To see this all at work, create a new project (Controls2) and add the following code within the Grid that is provided,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="100" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Rectangle Fill="Red"
        Height="40"
        Width="20"
        Grid.Row="0"
        Grid.Column="0" />
  <Rectangle Fill="Blue"
        Height="40"
        Width="20"
        Grid.Row="1"
        Grid.Column="0" />
  <Rectangle Fill="Green"
        Height="40"
        Width="20"
        Grid.Row="2"
        Grid.Column="0" />
  <Rectangle Fill="Yellow"
        Height="40"
        Width="20"
        Grid.Row="3"
        Grid.Column="0" />
</Grid>
```

Notice that all the Rectangles are the same size (height=”40”) but the rows are of differening sizes. The result is shown in figure 46,

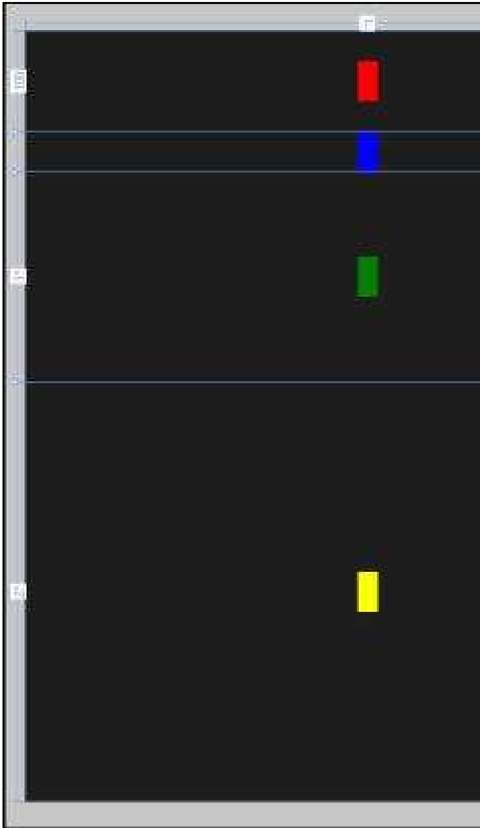


Figure 64 – Width and Height

The things to notice in this figure are that the first (red) rectangle is centered in a row that is 100 pixels high. The second rectangle is fit just to the size of the row (or more accurately, the row has fit to the size of the rectangle. The next two rows divide the remaining space in the proportion 1:2 and the rectangles are centered in each. The image was cropped to make all of this more obvious.

Notice that around the borders of the design surface, the relative sizes are shown (e.g., 100, 1*, 2*, etc.) to make it easier to see (and adjust) the sizes.

In this example, we placed everything in the first column to save space in the figure.

Alignment, Margins and Padding

In setting objects into rows and columns of the grid, you often want finer control over their placement. The first properties you might set are the horizontalAlignment and the VerticalAlignment. These are enumerated constants as shown in figure 75,

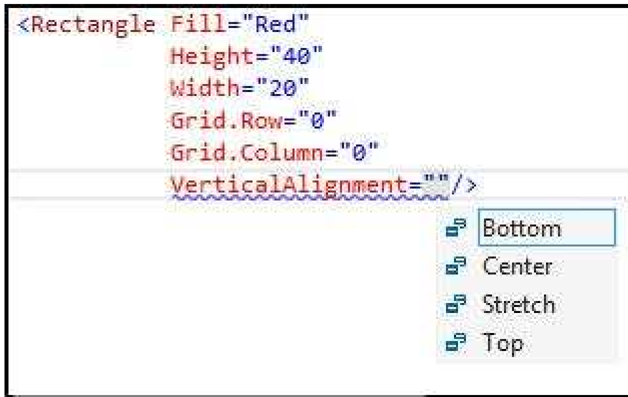


Figure 5 7 – Vertical Alignment

Margins are set to further fine tune the placement of the object within the grid's cell. You can set margins in any of three ways

- A single value
- A pair of values
- Four values

If you set a single value, all four margins (top, left, bottom, right) will be set to the entered value. that margin will be set to the top and bottom, left and right of the object.

If you set a pair of values, the left and right margins will be assigned the value of the first pairnumber, and the top and bottom margins will be divided evenly betweent he left and right margins and the second pair will be set to the second value.will be divided between the top and bottom margins.

Finally, if you set four numbers they will be assigned left, top, right, bottom. This is different from CSS in web development, where the numbers are in top, right, bottom, left order.

Thus, if you set

Margin="5"

You create a margin of five pixels all around the object, but if you set

Margin = "10, 20"

You create a margin of five 10 on the left and right and ten 20 on the top and bottom and, finally, if you set

Margin = "5, 10, 0, 20"

You create a margin of five on the left, ten on the top, 0 on the right and 20 on the bottom.

Padding refers to the space *within* a control between its border and its contents. You can see padding at work when you create a button and set the padding – the contents are spaced further from the edges. In Controls2A you see that we create three buttons, setting the padding to be quite small in the second and quite large in the third,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Margin="50">
        <Button Content="Hello" />
        <Button Content="Hello"
                Padding="5" />
```

```
        <Button Content="Hello"
                Padding="25" />
    </StackPanel>
</Grid>
```

The result is shown in figure 5A7A

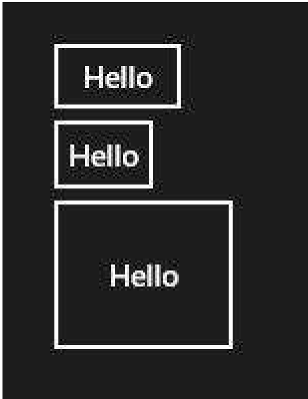


Figure 5A 7A – Padding

You can see in figure 75A that the padding affects the spacing around the content but within the button, not the spacing between buttons (which is controlled by the margin).

The Stack Panel

The StackPanel is both simple and useful. It allows you to “stack” one object on top of another or one object to the right of another (horizontal stacking). No room is created between the objects, but you can easily add space by setting a margin. You can add any control inside a stack, including another stack.

For example, create a new application (Controls3) and add the following code,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel HorizontalAlignment="Left" Margin=”50”>
        <Rectangle Fill="Red"
                   Height="50"
                   Width="50" />
        <Ellipse Fill="Blue"
                 Height="50"
                 Width="50" />
        <StackPanel Orientation="Horizontal">
            <Ellipse Fill="Green"
                     Height="20"
                     Width="20" />
            <Ellipse Fill="Yellow"
                     Height="20"
                     Width="20" />
        </StackPanel>
    </StackPanel>
</Grid>
```

This book was purchased by arosner@rosnertech.com

Here you have two Sstack pPanels, an outer StackPanel and an inner StackPanel. The outer StackPanel consists of a Rectangle and an Ellipse stacked one on top of another. The inner StackPanel has its orientation set to Horizontal (the default is Vertical) and within the inner StackPanel are two somewhat smaller ellipses.

The result is shown in figure 86,

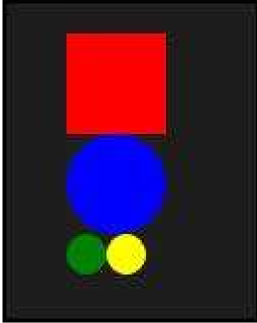


Figure 86 – StackPanel Objects

Border Control

The Border control is a container that can draw a border, background or both around one or more other objects. Technically, the Border control, like many other controls, can only have a single object as its contents; but that object can be, e.g., a StackPanel that can in turn have many objects within it, greatly increasing the utility of the Border. Create another project called Controls4 and add the following XAML:

```
<Border BorderBrush="Red"
      BorderThickness="5"
      Height="150"
      Width="150"
      Background="Wheat">
   <StackPanel VerticalAlignment="Center" >
      <Rectangle Height="50"
               Width="50"
               Fill="Blue"
               Margin="5"/>
      <Rectangle Height="50"
               Width="50"
               Fill="Black"
               Margin="5"/>
   </StackPanel>
</Border>
```

Notice in figure 97 that the two squares have 10 pixels between them. That is because each declared a margin of 5, and the bottom margin of the upper rectangle was added to the top margin of the lower rectangle (5+5).

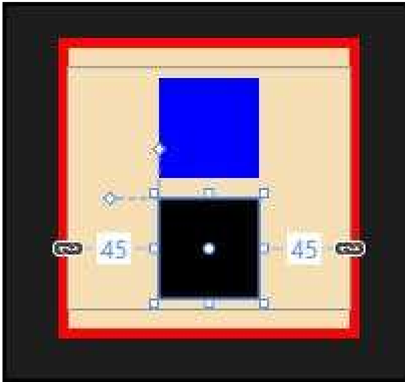


Figure 97 – Border control

Controls

In addition to the various panels, the Windows 8.1 tool box is chock full of additional controls for gathering or displaying information and/or for interacting with the end user.

TextBlock and TextBox

There are a number of simple controls for displaying or retrieving data from the user. For example, text is typically displayed using a TextBlock control, and is retrieved from the user with a TextBox control. In Controls5 we create a very simple data entry form,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
       <RowDefinition Height="50" />
       <RowDefinition Height="50" />
       <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
       <ColumnDefinition />
       <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBlock Text="First Name"
             FontSize="20"
             Margin="5"
             HorizontalAlignment="Right"
             VerticalAlignment="Center" />
    <TextBox x:Name="FirstName"
            Width="200"
            Height="40"
            Grid.Row="0"
            Grid.Column="1"
            HorizontalAlignment="Left"
            Margin="5"
            VerticalAlignment="Center" />
    <TextBlock Text="Last Name"
             Grid.Row="1"
             FontSize="20"
             Margin="5"
```

```
            HorizontalAlignment="Right"
            VerticalAlignment="Center" />
    <TextBox x:Name="LastName"
            Width="200"
            Height="40"
            Grid.Row="1"
            Grid.Column="1"
            HorizontalAlignment="Left"
            Margin="5"
            VerticalAlignment="Center" />

        <TextBlock Text="Job Title"
                   Grid.Row="2"
                   FontSize="20"
                   Margin="5"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Center" />
        <TextBox x:Name="JobTitle"
                 IsSpellCheckEnabled="True"
                 IsTextPredictionEnabled="True"
                 Width="200"
                 Height="40"
                 Grid.Row="2"
                 Grid.Column="1"
                 HorizontalAlignment="Left"
                 Margin="5"
                 VerticalAlignment="Center" />
</Grid>
```

We start by giving the Grid three rows and two columns, and then populating the resulting cells with TextBlocks and TextBoxes. Notice that the default position for a control is in Grid.Row="0" and Grid.Cell="0" and so we only need to specify when we want a different row or cell. While it is best programming practice to specify the default as well, it is so common to assume 0 for cell or row if not specified, that we show it that way here.

The result of this code is shown in figure 108,

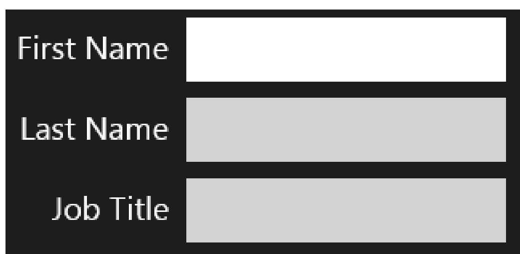


Figure 108 – Simple Form

The TextBoxes have an X that shows up on the right hand side once the user starts typing, allowing the user to delete the text and start over, as shown in figure 119,

Figure 911 – X to Delete

Also notice in the XAML that the two TextBoxes are named.

<TextBox x:Name="FirstName"

This is so that we can refer to them programmatically. That is, we can refer to them in code, such as in an event handler, which we'll see in the next section.

SpellCheck

WinRT Windows 8.1 XAML supports system-driven spell checking and auto-complete. Built in and right out of the box. Figure 119a shows spell checking at work. Notice the squiggly underlining of the misspelled word, the options to correct or add to the dictionary or ignore.

Figure 119a – Spell Check

You get both spell checking and autocompletion by adding setting just two properties on the TextBox.

```
IsSpellCheckEnabled="True"
IsTextPredictionEnabled="True"
```

The complete XAML for the JobTitle TextBox is shown here:

```
<TextBox x:Name="JobTitle"
    IsSpellCheckEnabled="True"
    IsTextPredictionEnabled="True"
    Width="200"
    Height="40"
    Grid.Row="2"
    Grid.Column="1"
    HorizontalAlignment="Left"
    Margin="5"
    VerticalAlignment="Center" />
```

Headers and Watermarks

In the examples above, we used a two column Grid and created TextBlocks next to the TextBoxes in order to create a data entry form. New in Windows 8.1 is the ability to create Headers in place with many input controls, including the TextBox, the PasswordBox, and the ComboBox (both shown later in this chapter).

Create a project Controls5a and add the following row definitions into the Grid. The main difference between this Grid and the one we created in Controls5 is setting the height to “Auto” and not including column definitions.

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

To create headers for the controls, add the Header attribute into the markup as in the sample code shown here.

```
<TextBox x:Name="FirstName"
       Header="First Name"
       Width="200"
       Grid.Row="0"
       HorizontalAlignment="Left"
       Margin="5"
       VerticalAlignment="Center" />
```

To add a watermark, use the PlaceHolderTextAttribute as shown in the sample code below.

```
<TextBox x:Name="FirstName"
       Header="First Name"
       PlaceholderText="[Enter First Name]"
       Width="200"
       Grid.Row="0"
       HorizontalAlignment="Left"
       Margin="5"
       VerticalAlignment="Center" />
```

The full code for the page is list here, and when you run the project you get the result shown in Figure 11a.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <TextBox x:Name="FirstName"
        Header="First Name"
        PlaceholderText="[Enter First Name]"
        Width="200"
        Grid.Row="0"
        HorizontalAlignment="Left"
        Margin="5"
        VerticalAlignment="Center" />
  <TextBox x:Name="LastName"
        Header="Last Name"
        PlaceholderText="[Enter Last Name]"
        Width="200"
        Grid.Row="1"
        HorizontalAlignment="Left"
        Margin="5"
        VerticalAlignment="Center" />
  <TextBox x:Name="JobTitle"
        Header="Job Title"
        PlaceholderText="[Enter Job Title]"
        IsSpellCheckEnabled="True"
        IsTextPredictionEnabled="True"
        Width="200"
        Grid.Row="2"
        HorizontalAlignment="Left"
        Margin="5"
        VerticalAlignment="Center" />
</Grid>
```

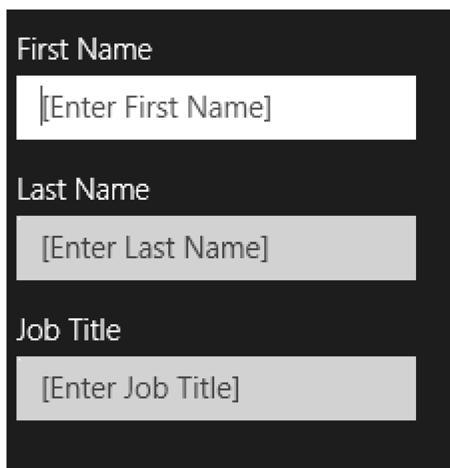


Figure 11a Password Box

Whether you use the Layout Panel or the Header approach is dependent on your particular application requirements. Either way, the goal is to provide the clearest user experience possible. To that end, the use of watermarks is highly recommended for data entry operations, as it greatly increases the clarity of what the user needs to accomplish on the page.

Password Box

A variation on the TextBlock is the PasswordBox, which allows you to collect information from the user that is masked by a “password character” – by default a bullet (●). In Listing Controls56ab we substitute a question mark as the password character and we turn on the reveal button which allows the user to temporarily see the password in clear text,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

        <PasswordBox Margin="5"
                Width="200"
                Header="Password"
                PlaceholderText="Please Enter Your Password"
                IsPasswordRevealButtonEnabled="True"
                PasswordChar="?"
                VerticalAlignment="Top"/>

<StackPanel Orientation="Horizontal" Margin="50">
      <TextBlock Text="Password: "
            Margin="5"
            VerticalAlignment="Top"
            FontSize="40"/>
      <PasswordBox Margin="5"
            Height="50"
            Width="200"
            IsPasswordRevealButtonEnabled="True"
            PasswordChar="?"
            VerticalAlignment="Top"/>
   </StackPanel>
</Grid>
```

The before and after typing an entry results is are shown in figure 9a11b

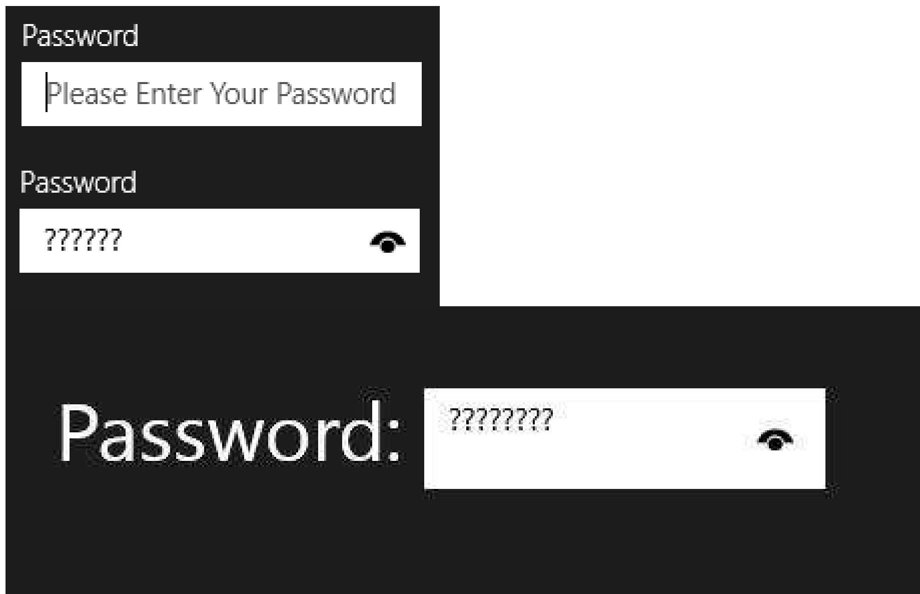


Figure 9a 11b Password Boxes. Waiting for input (top) and with input (bottom)

The “eye” on the right side of the password box is the reveal button. This allows the user to show the actual characters being typed. This is not a toggle, as the characters are only revealed while the reveal button is being pressed.

Buttons and Event Handlers

Controls6 is a form with Let’s two TextBoxes,add a button to the form, and an event handler that will respond to the button being clicked. In that event handler we can get the values in the two textboxes. The result will be displayed in a We’ll also need another TextBlock to display the user name we obtain from the TextBoxes. Controls6 is just like Controls5 except that we add a row, and we add a TextBlock and a Button to the new row.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
   <Grid.RowDefinitions>
       <RowDefinition Height="50" />
       <RowDefinition Height="50" />
       <RowDefinition Height="50" />
       <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
       <ColumnDefinition />
       <ColumnDefinition />
   </Grid.ColumnDefinitions>
   <TextBlock Text="First Name"
              FontSize="20"
              Margin="5"
              HorizontalAlignment="Right"
              VerticalAlignment="Center" />
   <TextBox x:Name="FirstName"
            Width="200"
            Height="40"
            Grid.Row="0"
            Grid.Column="1"
            HorizontalAlignment="Left"
            Margin="5"
            VerticalAlignment="Center" />
   <TextBlock Text="Last Name"
```

```
                   Grid.Row="1"
                   FontSize="20"
                   Margin="5"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Center" />
        <TextBox x:Name="LastName"
                 Width="200"
                 Height="40"
                 Grid.Row="1"
                 Grid.Column="1"
                 HorizontalAlignment="Left"
                 Margin="5"
                 VerticalAlignment="Center" />
        <TextBlock x:Name="Output"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Center"
                   Text=""
                   Margin="5"
                   FontSize="20"
                   Grid.Column="0"
                   Grid.Row="2" />
        <Button Name="ShowName"
                Content="Show Name"
                HorizontalAlignment="Left"
                VerticalAlignment="Center"
                Margin="5"
                Grid.Row="2"
                Grid.Column="1" />
    </Grid>
```

There are numerous ways to add an event handler for the click event. The simplest is to type Click= and when you hit space Visual Studio will offer to create (and name) an event handler for you as shown in figure 120.

Figure 120 – Event Handler

The name created for you by Visual Studio is <is <objectName>_click_1, so if your button is named *ShowName* then the event handler will be *ShowName_Click_1*, though you are free to override this name with anything you like.

Not only does Visual Studio set up the name for your event handler, but it stubs out the event handler in the CodeBehind (e.g., MainPage.xaml.cs).

```
private void ShowName_Click_1( object sender, RoutedEventArgs e )
{
}
```

All you need do is fill in the logic. In this case, we'll obtain the string values from the FirstName and LastName controls Text property, concatenate them and place them into the Text property of the TextBlock *Output*.

```
private void ShowName_Click_1( object sender, RoutedEventArgs e )
{
   string fn = FirstName.Text;
   string ln = LastName.Text;
   Output.Text = fn + " " + ln;
}
```

You can of course shorten this by leaving out the intermediate string variables:

```
private void ShowName_Click_1( object sender, RoutedEventArgs e )
{
   Output.Text = FirstName.Text + " " + LastName.Text;
}
```

We will typically show the longer version as it makes it easier to see what is happening, and it makes it easier to debug if anything goes wrong.

HyperLinkButton

The HyperLinkButton looks like a HyperLink but acts like a button. That is, you can assign a click-event handler to the HyperLinkButton that will be handled before the user is navigated to the hyperlink. HyperlinkButtons are often used inline with text blocks.

Controls6ab shows a simple HyperLinkButton at work

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
   <StackPanel Margin="50">
      <TextBlock Name="xMessage" />
      <HyperlinkButton Content="Jesse Liberty"
                  NavigateUri="http://JesseLiberty.com"
                  Click="HyperlinkButton_Click_1" />
   </StackPanel>
</Grid>
```

Notice that the HyperLinkButton has an event handler. This method is implemented in the code-behind, like any event handler.

```
private void HyperlinkButton_Click_1( object sender, RoutedEventArgs e )
{
   xMessage.Text = "Hello Hyperlink!";
}
```

The code will be called before the hyperlink is called, and so in this case the message "Hello Hyperlink!" will flash just before we navigate to the URL. You can, of course, write code that includes user interaction to take place before the URL is called. More common is to use the HyperLinkButton just to navigate to the URL, giving you the appearance of a normal (HTML) Hyperlink within a XAML application.

CheckBoxes, ToggleSwitches aAnd RadioButtons

There are a number of controls to help the user to make selections. If the user is selecting from a number of options, two of the most popular are CheckBoxes (for multi-select) and RadioButtons (for mutually exclusive selections). ToggleSwitches are the Windows 8Windows 8.1 control of choice for when the user is selecting one of two mutually exclusive choices. Controls7 shows a small form with all three of these controls added.

This book was purchased by arosner@rosnertech.com

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <StackPanel Name="OuterPanel" Margin="50">
    <StackPanel Orientation="Horizontal"
            Name="RadioButtonsPanel">
      <RadioButton Name="Soft"
              Content="Soft"
              GroupName="Loudness"
              Margin="5" />
      <RadioButton Name="Medium"
              Content="Medium"
              GroupName="Loudness"
              Margin="5" />
      <RadioButton Name="Loud"
              Content="Loud"
              IsChecked="True"
              GroupName="Loudness"
              Margin="5" />
    </StackPanel>
    <StackPanel Orientation="Horizontal"
            Name="CheckBoxPanel">
      <CheckBox Name="ClassicRock"
            Content="Classic Rock"
            Margin="5" />
      <CheckBox Name="ProgRock"
            Content="Progressive Rock"
            Margin="5" />
      <CheckBox Name="IndieRock"
            Content="Indie Rock"
            Margin="5" />
    </StackPanel>
    <ToggleSwitch Header="Power"
            OnContent="On"
            OffContent="Off" />
    <ToggleButton Content="Toggle me!" Checked="ToggleButton_Checked_1" Unchecked="ToggleButton_Unchecked_1"/>
    <TextBlock Name="Message"
           Text="Ready..."
           FontSize="40" />
  </StackPanel>
</Grid>
```

The first thing to notice about this form is that rather than creating the layout with cells in a grid, we laid out the controls in stack panels. Either way is valid.

The first StackPanel consists of three RadioButtons. RadioButtons are grouped using a GroupName (every RadioButton in a group is mutually exclusive). We set the IsChecked property to true for the third button – the default is false, and in a RadioButton group only one button will have the IsChecked value set to true at a time.

The second StackPanel has three CheckBoxes and they are not mutually exclusive; the user is free to pick any or all of the music choices.

Next, we have a ToggleSwitch. You can see that it has a header and text for when it is in the on and in the off position.

Finally, we have a ToggleButton, which looks like a button but toggles between checked and unchecked state and has events that correspond to being checked or unchecked.

Run the program and click on the buttons and CheckBoxes to see how they behave, and toggle the swtich to see the text and the switch change, as shown in figure 131,

Figure 113 – Radio Buttons

The only code for all of the above are the event handlers for clicking the toggle button,

```
private void ToggleButton_Checked_1( object sender, RoutedEventArgs e )
{
  Message.Text = "Button was toggled on!";
}

private void ToggleButton_Unchecked_1( object sender, RoutedEventArgs e )
{
   Message.Text = "Button was toggled off!";
}
```

ListBox, ListView & ComboBox

Windows 8Windows 8.1 continues to support the ListBox control though it has, in many ways been replaced by the ListView. They work very similarly, and they both provide a scrolling list of data, though the ListView is more powerful and more suited to Windows 8Windows 8.1 Store applications. In their simplest form, their use and appearance is identical, as shown in Controls8,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Horizontal" Margin="50">
        <ListBox Name="myListBox"
                  Background="White"
                  Foreground="Black"
                  Width="150"
                  Height="250"
                  Margin="5">
            <x:String>ListBox Item 1</x:String>
            <x:String>ListBox Item 2</x:String>
            <x:String>ListBox Item 3</x:String>
            <x:String>ListBox Item 4</x:String>
        </ListBox>
        <ListView Name="myListView"
                  Background="White"
                  Foreground="Black"
                  Width="150"
                  Height="250"
```

```
                              Margin="5">
                        <x:String>ListView Item 1</x:String>
                        <x:String>ListView Item 2</x:String>
                        <x:String>ListView Item 3</x:String>
                        <x:String>ListView Item 4</x:String>
                    </ListView>
                </StackPanel>
            </Grid>
```

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Horizontal" Margin="50">
        <ListView Name="xListBox"
                  SelectionChanged="xListBox_SelectionChanged_1"
                  Background="White"
                  Foreground="Black"
                  Width="150"
                  Height="250"
                  Margin="5">
            <x:String>Item 1</x:String>
            <x:String>Item 2</x:String>
            <x:String>Item 3</x:String>
            <x:String>Item 4</x:String>
        </ListView>
        <ListView Name="xListView"
                  SelectionChanged="xListView_SelectionChanged_1"
                  Background="White"
                  Foreground="Black"
                  Width="150"
                  Height="250"
                  Margin="5">
            <x:String>Item 1</x:String>
            <x:String>Item 2</x:String>
            <x:String>Item 3</x:String>
            <x:String>Item 4</x:String>
        </ListView>
    </StackPanel>
</Grid>
```

The output is nearly identical in this case, as shown in figure 124,

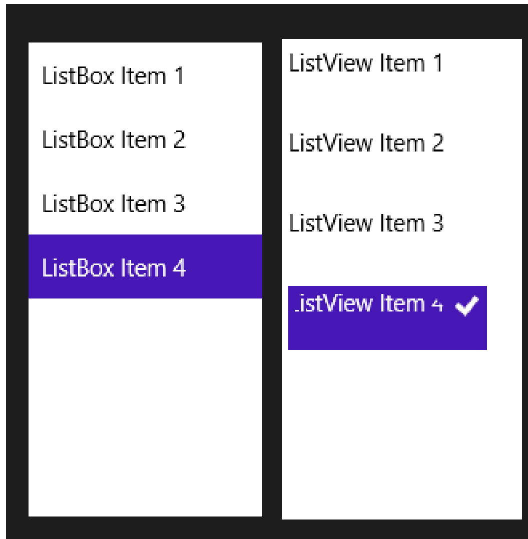


Figure 124 – ListBox and ListView

Later in this book we'll return to the ListView in more detail to look at its full capabilities. At that time, we'll also look at the GridView and how the two work together.

The ComboBox is very similar to the ListBox. It except that it provides a drop down list, but you can also provide watermark text and a header (through the PlaceHolderText and Header attributes respectively), as shown added to in Controls8a,

```
<ComboBox Name="myComboBox"
        Background="White"
        Foreground="Black"
        Width="200"
        Height="40"
        Margin="5"

   PlaceHolderText="Please Select and Item"

        Header="ComboBox Example">
        <x:String>ComboxBox Item 1</x:String>
        <x:String>ComboxBox Item 2</x:String>
        <x:String>ComboxBox Item 3</x:String>
        <x:String>ComboxBox Item 4</x:String>
</ComboBox>
```

```
<ComboBox Name="xComboBox"
        SelectionChanged="xComboBox_SelectionChanged_1"
        Background="White"
        Foreground="Black"
        Width="150"
        Height="250"
        Margin="5">
   <x:String>Item 1</x:String>
   <x:String>Item 2</x:String>
   <x:String>Item 3</x:String>
   <x:String>Item 4</x:String>
</ComboBox>
```

The result, with all three controls, is shown in Figure 14a3,

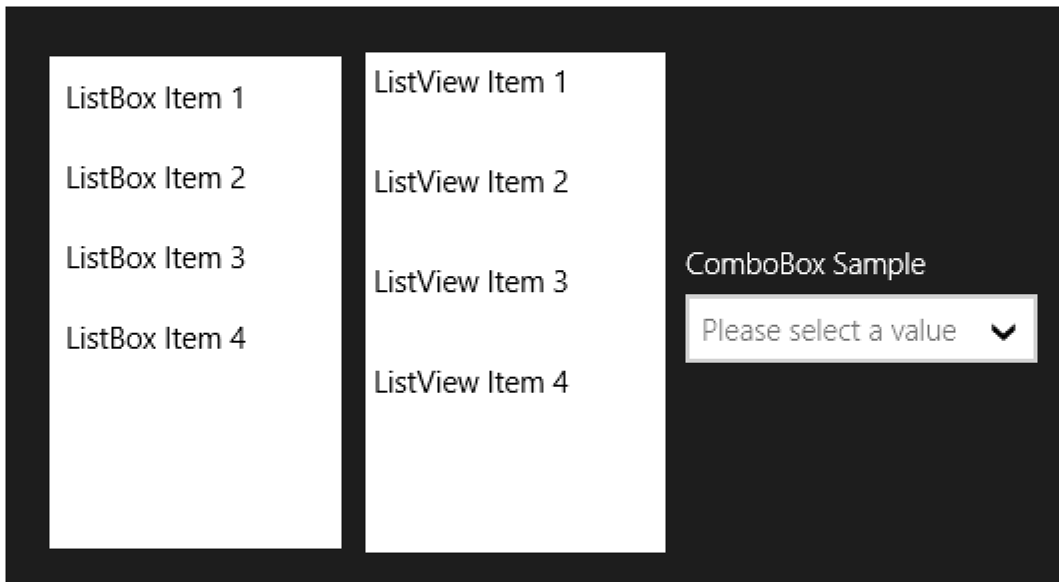


Figure 14a3 With ComboBox

Typically, you'll use the comboBox when space is tight and when users select just one choice at a time. You'll use the ListBox (if you use it at all) when you want to display all the choices at once and when you want to allow multiple selections.

Image Control

The image is used to display (surprise!) an image… typically a jpg or png file. Its most important property is *source* which is the URI of the image itself. Controls9 shows how to add two images to the page,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <StackPanel Margin="50">
    <Image Height="240"
        Width="360"
        Source="Assets/Sheep.jpg"
        Margin="5" />
    <Image Height="240"
        Width="360"
        Source="Assets/PaintedDesert.jpg"
        Margin="5" />
  </StackPanel>
</Grid>
```

The result is shown in figure 154,

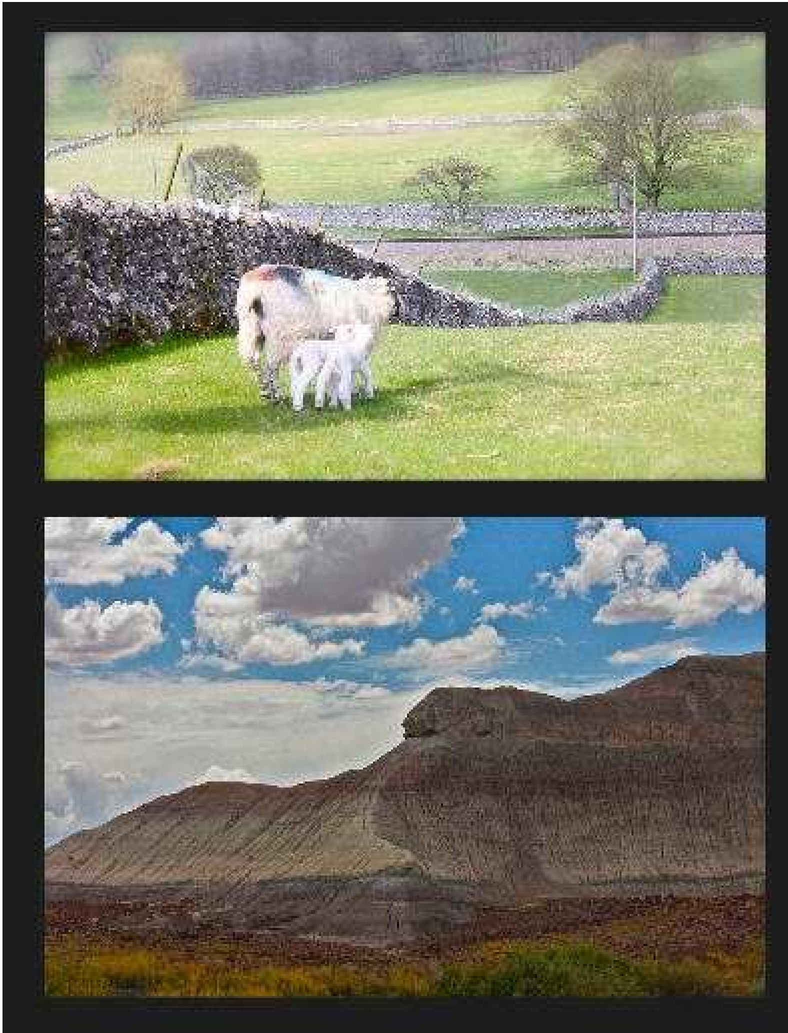


Figure 154 – Images

To make this work copy two images into your Assets folder, click on *Add Existing Item* to add them to the project and then update the name in the Source property of the Image. You can use any images that you have on your local system, or use the images from the sample code provided along with the book.

If your image does not fit the rectangle described for it by the Image control, you can set the Stretch property, which is an enumerated constant, as shown in Figure 15a5,

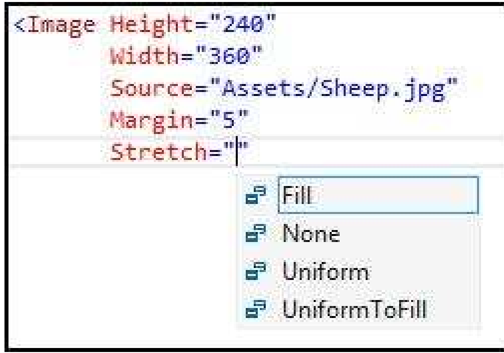


Figure 15a – Stretch Property

The effect of each of these constants can be seen in figure 16

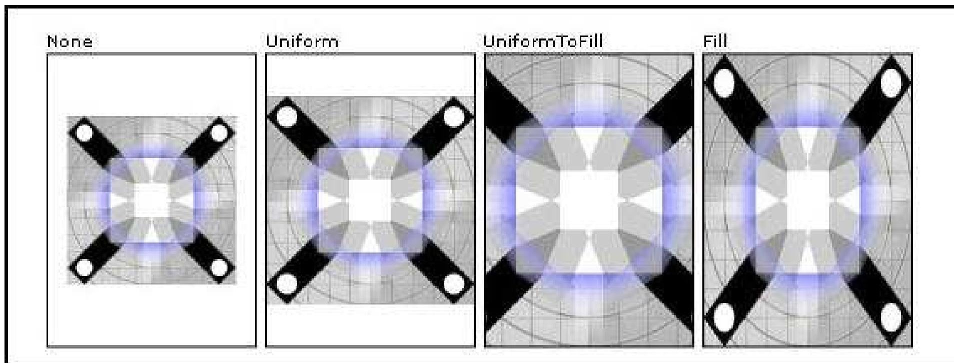


Figure 16 – Stretch – Please Redraw

Slider Control

The Slider allows the user to select a value from a range of values by sliding a thumb control along a track. While the thumb is being moved the actual value is automatically displayed above the slider, as shown in figure 17

Figure 17 – Slider

The code for creating the Slider is shown in Controls 10,

```
<StackPanel Margin="50">
   <Slider
         Header=”Select a Value”Name="xSlider"
         Minimum="0"
         Maximum="100"
```

```
          Value="50"
          Width="100"/>
</StackPanel>
```

Progress Bar

The progress bar is used to indicate (surprise!) progress… typically while waiting for an asynchronous operation to complete. If you know the percentage of progress achieved at any moment, you can use a standard progress bar, setting its value as you progress towards completion. Otherwise you will want to use the indeterminate progress bar.

A ProgressRing works just like the indeterminate Progress Bar but is typically used for system waits rather than for program progress.

The code for creating ProgressBars is shown in Controls11,

```
<StackPanel Margin="50">
<ProgressBar Name="xDeterminateProgress"
           Value="50"
           Width="150"
           Margin="20"/>
<ProgressBar Name="xIndeterminateProgress"
           IsIndeterminate="True"
              Width="150"
              Margin="20" />
   <ProgressRing Name="xProgressRing"
               IsActive="True"
               Margin="20" />
</StackPanel>
```

The result is shown in figure 18

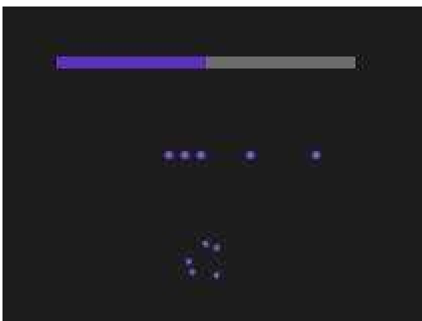


Figure 18 – Progress Bars

ToolTip

Tooltips pop up and display information associated with an element. A tooltip will appear if the user hovers over the element with a mouse or if the user taps and holds the element with touch. Controls12 shows the code for adding a ToolTip to a button,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
   <StackPanel Margin="50">
   <Button Content="xButton"
         Click="Button_Click_1"
         ToolTipService.ToolTip="Click to perform action" />
   </StackPanel>
</Grid>
```

The tooltip is shown in Figure 19

Figure 19 – ToolTip

Date and Time Pickers

Windows 8.1 adds Date and Time pickers for XAML based applications. Create a new project Controls13 and add the following XAML to demonstrate the core functionality of the controls, which are shown in Figure 20.

```
<Grid.ColumnDefinitions>
   <ColumnDefinition Width="120"/>
   <ColumnDefinition Width="Auto"/>
   <ColumnDefinition Width="40"/>
   <ColumnDefinition Width="Auto"/>
   <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<TimePicker Grid.Column="1" Header="Select a Time" />

    <DatePicker Grid.Column="3" Header="Select a Date"/>
```

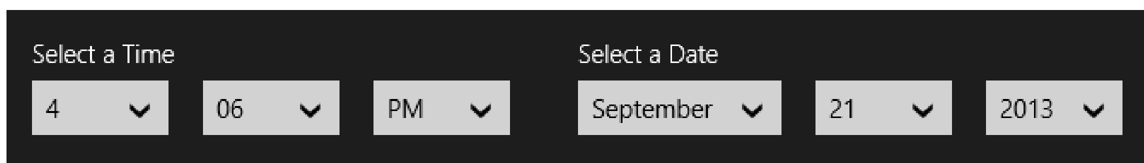


Figure 20 – Time Picker and Date Picker

Flyouts

Flyouts have been available for HTML/JavaScript developers since Windows 8. Windows 8.1 adds Flyouts for XAML developers. A Flyout is an UI element that overlays the current UI to provide additional information to the user or provide a mechanism to get confirmation of an action.

Flyouts are attached to other controls and are shown in response to an action. For example, a Flyout attached to a Button will show when the Button is clicked. Flyouts are container controls themselves, and can hold and number of additional controls by using a Panel.

To show the different types of Flyouts, create another project called Controls14. Add the following Column definitions in preparation for the different examples.

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="120"/>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="Auto"/>

 </Grid.ColumnDefinitions>
```

Basic Flyouts

To show how to add a Flyout, create a new project called Controls14. Add the following XAML to the Grid, run the program, and you will see the result in Figure 21 (after clicking the button).

```
<Button Grid.Column="0" Content="This is a button to launch a flyout">
  <Button.Flyout>
    <Flyout>
      <StackPanel>
        <TextBlock>This is a flyout message</TextBlock>
        <Button>This is a flyout button</Button>
      </StackPanel>
    </Flyout>
  </Button.Flyout>
</Button>
```

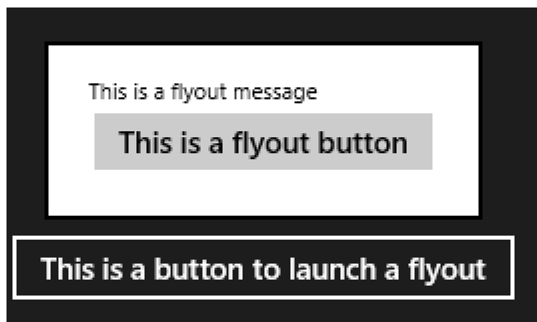


Figure 21 – Flyout attached to a Button

Menu Flyouts

Menu Flyouts allow for building menus that remain hidden until needed. Add the following XAML into Controls14 and run the program to get the result in Figure 21.

```
<Button Grid.Column="1" Content="This is a button to launch Menu Flyout">
  <Button.Flyout>
    <MenuFlyout>
      <MenuFlyoutItem Text="First Item"/>
      <MenuFlyoutSeparator/>
      <ToggleMenuFlyoutItem Text="Toggle Item"/>
    </MenuFlyout>
  </Button.Flyout>
</Button>
```


This book was purchased by arosner@rosnertech.com

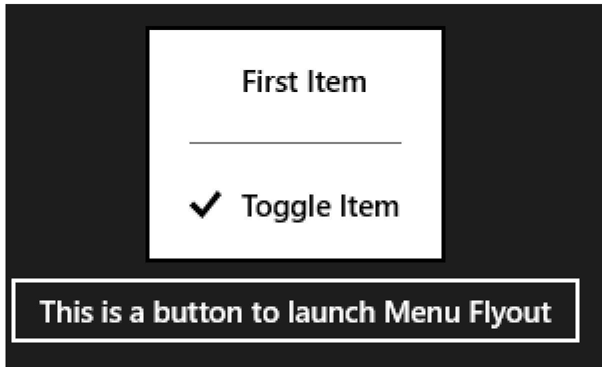


Figure 21 – The Menu Flyout with a regular menu itme and a toggle menu item

Dependency Properties

Dependency properties are like the sewers, electric wires and pipes running under the streets of a big city. You can ignore them, you can even be oblivious to their existence, but they make all the magic happen, and when something breaks you suddenly discover that you need to understand how they work.

Before we begin exploring Dependency Properties, let's review Properties themselves. To do that, we go back into the dark early days of C++ when what we had available in a class were methods and member member variables (fields).

When I was a boy, there were no properties, VCRs or digital watches…

Data Hiding

Back in these dark days, we wanted to store values in fields, but we didn't want clients (methods that use our values) to have direct access to those fields (if nothing else, we might later change how we store the data). Thus we used accessor methods.

The accessor methods had method semantics (you used parentheses) and so data hiding was explicit and ugly. You ended up with code that looked like this:

```
int theAge = myObject.GetAge();
```

which just seems all wrong , what you wanted was

```
int theAge = myObject.age;
```

Field Syntax with Method Semantics

Enter properties. Properties have the wonderful characteristic of looking like a field to the consumer, but looking like a method to the implementer. Now you can change the way the Property is "backed" without breaking the consumer, but the consumer gets the right semantics. Nice.

```
private int _age;
public int Age { get { return _age; } set { _age = value; } }
```

Age is the property, _age is the backing variable. The consumer can now write

```
int theAge = myObject.Age;
```

Age is a property and so you can put anything you like in the getter, including accessing a database or computing the age.

Note, by the way that this idiom of just having a backing variable and getting and setting its value is so common that automatic properties were invented to save typing. Thus, the private field field and public property above could have been written with this single line:

```
public int Age { get; set; }
```

The compiler turns this single statement into a declaration of a private backing variable, and a public property whose get returns that backing variable, and whose set, sets that variable.

Dependency Propoerties

Dependency properties are an extension to the CLR and to C#, and were created because normal properties just don't provide what we need: **declarative syntax that is fast enough to supports animation and that will support data binding as well as storyboards and animations**.

Most of the properties exposed by Windows 8Windows 8.1 elements are dependency properties, and you've been using Dependency Properties without realizing it. That is possible because Dependency Properties are designed to look and feel like traditional C# properties.

In short, what was needed was a system that could establish the value of a property, at run time, based on input from a number of sources. That is, the value of a dependency property is computed based on inputs such as

- User preferences
- Databinding
- Animation and Storyboards
- Templates and styles
- Inheritance

The Dependency Property acts as a backing store for a Windows Runtime property, rather than using a private field. Dependency Properties provide what is known as a "property store;" that is, a set of property identifiers and values that exist for the particular object.

A key value of the Dependency Properties system is the ability to build properties that automatically notify any registered interested party each time the value of the property changes. This free, painless and automatic implementation of the observer pattern[1] is tremendously powerful and greatly reduces the burden on the client programmer (in fact, the data-binding system depends on it!).

In exchange for these capabilities you pay the price of the added complexity of a second type of properties overlaid on top of the inherent properties of the language. In addition, some of your properties are really just wrappers to the Dependency Property. What that means is that the backing value for some of your properties is not a member variable or a value in a database or a computation, but a Dependency Property. That takes a little mind-share

Note that you You normally will not have to create Dependency Properties unless you are creating a custom control (a topic beyond the scope of this book).

The first thing to know is that in order to support a dependency Dependency property Property the object that defines the property (your custom control) must inherit from, DependencyObject. Virtually all of the types you use for a Windows Store app with XAML and C# will be a DependencyObjectg subclass.

You might then declare your property like this:

```
public bool Valuable
{
  get { return (bool) GetValue( ValuableProperty ); }
```

[1] http://msdn.microsoft.com/en-us/library/Ee817669(pandp.10).aspxhttp://msdn.microsoft.com/en-us/library/ms954621.aspx

```
   set { SetValue( ValuableProperty, value ); }
}
```

A pretty standard get and set, except that you access your backing variable using GetValue and SetValue to get and set the value of a Dependency property Property named ValuableProperty (and that is the idiom, the CLR property name + the word Property = the name of the DP, thus Valuable + Property = ValuableProperty.)

The declaration of the Dependency Property itself is a bit weirder,

```
public static readonly DependencyProperty ValuableProperty =
    DependencyProperty.Register(
      "Valuable",
      typeof( bool ),
      typeof( MyCustomControl ),
      new PropertyMetadata( new PropertyChangedCallback(
          MyCustomControl.OnValuablePropertyChanged ) ) );
```

Let's break this down. The first line declares my object (which is really a reference to a DependencyProperty) as public; it must be static and readonly, and its type is DependencyProperty and its name (identifier) is ValuableProperty.

We set that reference to what we'll get back by calling the static Register method on the DependencyProperty class. Register takes four arguments:

- The name of the dependency property wrapper
- The type of the DP being registered
- The type of the object registering it
- The Callback

The Callback is of type *PropertyMetaData*. You can imagine a world in which there are various pieces of MetaData for the DependencyProperty. At the moment, however, in Windows 8Windows 8.1, there is only one: the callback.

The constructor for the PropertyMetaData takes an object of type *PropertyChangedCallback* which will be called any time the effective property value of the DP property changes. We pass it a reference to the method to call (which equates to a callback).

The net of all of this is that we present to the world a CLR property (Valuable) which is in fact backed by a DependencyProperty which will call back to the method OnValuablePropertyChanged any time the effective value of the property changes.

The callback method will take two arguments:

A DependencyObject (the control)

An object of type DependencyPropertyChangedEventArgs

Typically you'll cast the first argument to be the type of the control that contains the property, and you'll cast the NewValue property of the DependencyPropertyChangedEventArgs object to the DependencyProperty that changed. You can then take whatever action you need to based on the change in the DP's value

```
public class MyCustomControl : Control
{
   public static  readonly DependencyProperty
      ValuableProperty = DependencyProperty.Register(
      "Valuable",
      typeof( bool ),
      typeof( MyCustomControl ),
```

```
    new PropertyMetadata( new PropertyChangedCallback(
        MyCustomControl.OnValuablePropertyChanged ) ) );

  public bool Valuable
  {
    get { return (bool) GetValue( ValuableProperty );}

    set { setValue( ValuableProperty, value );}
  }



  private static void OnValuablePropertyChanged(
      DependencyObject d,
      DependencyPropertyChangedEventArgs e )
  {
    MyCustomControl control = d as MyCustomControl;

    bool b = (bool) e.NewValue;
  }
}
```

The key thing to know about Dependency Properties is that when you bind to a property (either through data binding or element binding), animate a property it must be a Dependency Property. Similarly, if you are going to use animations (for example to change the opacity of a control or they style based on values, when you data bind to a property (as we'll discuss in chapter 9) you must bind to a the animation story board only works on Dependency Propertiesy. The reason is that only a Dependency Property can get its value quickly enough from these rapidly changing sources. This is normally not a problem as the properties you will be inclined to animate or bind to (almost always) are dependency properties anyway. You can see that quickly by looking at the documentation for the properties for any of the UI elements. These will all be UIElements (or FrameworkElement which derives from UIElement) and UIElement derives from DependencyObject, which provides the support for Dependency Properties.

Summary

There are a wide variety of controls available for use in your Windows 8.1 apps that ship with Visual Studio, plus many more available through the rich third party ecosystem. Every page starts with a panel control (the most common being the Grid), and then additional controls are added to make the desired user interface.

Using controls and handling events for those controls is very similar to what you find in other XAML environments, such as WPF and Silverlight. This chapter did not cover every type of control available, but provides experience the core controls, and the remaining controls are all conceptual extensions of the ones described.

The next step in your journey into using Windows 8.1 controls is data binding, which is covered in the next chapter.

CHAPTER 4

n n n

Binding

What's New in Windows 8.1

There are several new features for Data Binding in Windows 8.1. The FrameworkElement.DataContextChanged event notifies you when the data context for a control is changed.

Two new binding properties (Binding.FallbackValue and Binding.TargetNullValue) provide mechanisms to handle data binding failures and null values.

The options for the UpdateSourceTrigger property now include Explicit, designed for programmatic control in two-way binding scenarios.

The essential glue in any meaningful Windows 8.1 application is showing data in your user interface. This is referred to as binding. At its simplest, binding allows you to connect the value of one object to a property on another object. This connection is refered to as a "binding". The most common use is binding the properties of a data object (a .NET Class) to the controls in the user interface. For example, a Customer's information might be bound to a series of TextBoxes. This use case is so prevelant that binding is commonly referred to as "Data Binding".

The *target* of the Binding must be a dependency property on a UI element; the source can be any property of any object. That is a very powerful statement; it means that you don't have create anything special in the source. A simple class such as a POCO (Plain Old CLR Object) will do. The classes that make of the data are typically referred to as "models."

It is important to note that *any* dependency property on the control can be bound to and accessible property on the data source. We'll see an example of this later in this chapter.

NOTE: Please see Chapter 3 for more information on dependency properties.

Data Context

As stated earlier, Data Binding needs a model to supply the data and a dependency property to receive the value of one of the model's properties. If the Binding statement specifies the information directly, the binding engine sends the data value(s) from the model to the control. However, when multiple controls are bound to the model, this requires a *lot* of typing as each binding statement must provide the model's information. In addition to the additional work up front to bind the user interface, if anything changes about the model (or if the model must be exchanged for another one), you must update each control with the new information.

If the Binding statement does not contain information that specifies how to find the model, the Binding Engine will use the Data Context. If the control does not have a Data Context specified, the binding engine will walk up the control tree until it finds a Data Context. Once the binding engine finds a control in the UIElement tree that has a data context specified, the binding engine will use that definition in the binding statements for all of the controls contained by that element.

Using a Data Context instead of explicitly referencing a model in each binding statements leads to a much more supportalble user interface. It also greatly enhances the ability to change the data source during runtime with a single statement.

Creating a Simple Binding

The best way to move from theory to practice is to write some simple code. Create a new Windows 8.1 project by select the Blank App (XAML) Windows Store template and name it Binding1.

Add a new class to the project named "Person". This class will become the model to hold the data will be bound to the user interface. The Person class is shown below:

```
public class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
}
```

Open up MainPage.xaml.cs and add the following line of code in the class to create an instance of the Person class that will serve as the data model.

```
public Person person = new Person { FirstName = "Jesse", LastName = "Liberty" };
```

The output we want to display is a prompt and the value of the first and of the last name, as shown in Figure 1.

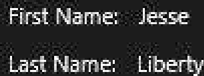


Figure 1 First and Last Name

We certainly could create this by adding four TextBlocks like this,

```
<StackPanel>
    <StackPanel Orientation="Horizontal">
       <TextBlock Text="First Name: " Margin="5"/>
       <TextBlock Name="txtFirstName" Margin="5"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal">
       <TextBlock Text="Last Name: "  Margin="5"/>
       <TextBlock Name="txtLastName" Margin="5"/>
    </StackPanel>
 </StackPanel>
```

We would then populate the TextBlocks by writing the value from the person instance into the Text property of the TextBlock,

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
   txtFirstName.Text = person.FirstName;
   txtLastName.Text = person.LastName;
}
```

The OnNavigated event is raised when a XAML pages loads into view. It will be covered later in the book when we talk about navigation. It is used to setup state for each page.

While this works, and it is perfectly valid code, it is inflexible and incredibly tedious when working with collections. Data Binding is much more powerful, flexible, easier to maintain, and generally more desirable.

To change the controls to Data Bind to the model, start by changing the XAML as follows:

```
<StackPanel>
   <StackPanel Orientation="Horizontal">
      <TextBlock Text="First Name: " Margin="5"/>
      <TextBlock Text="{Binding FirstName}" Margin="5"/>
   </StackPanel>
   <StackPanel Orientation="Horizontal">
      <TextBlock Text="Last Name: "  Margin="5"/>
      <TextBlock Text="{Binding LastName}" Margin="5"/>
   </StackPanel>
</StackPanel>
```

The Data Binding statements are contained as strings enclosed with curly braces and start with the word "Binding". In these examples, the Text properties of TextBlocks are bound to the FirstName and LastName properties, respectively. This is much better; if nothing else we've moved from procedural to declarative code, which is more easily manipulated in tools such as Blend or Visual Studio. We'll look deeper into the binding syntax in the next section, but for now, this is enough detail.

Only the properties on the model are specified, and there isn't any model specified. This causes the data binding engine to next check the TextBlock for a DataContext. Not finding one, it would then check the two StackPanels (starting with the inner StackPanel), finally moving on to the Window itself. Not finding any Data Context,

The Data Context can be set programmatically (in code) or declaratively (in XAML). In this example, I'll set the DataContext at the page level, and I'll do so programmatically by setting the DataContext of the class (which is the page itself). The page is the highest level container, so all of the elements on the page will inherit the person class as the Data Context.To do this, replace the assignments to the TextBlocks in the OnNavigatedTo event to the following code.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
   DataContext = person;
}
```

When the application runs, the DataContext is set to the Person instance, and the values are filled into the TextBlocks. Very gratifying.

Data Binding Statements

As mentioned earlier, a declarative binding statement in its simplest form is as follows:

```
<dependencyobject dependencyproperty="{Binding bindingArgs}" />
```

The bindingArgs are are set of name value pairs that further refine the binding. There are a lot more options to explore, but here are some of the most commonly used binding arguments.

Path

The path defines the property from the model that will be databound to the dependencyproperty. It can be a simple property or a complex property chain in "dot" notation, walking down the properties of the model, such as "Address.StreetName".

If there is only argument, and that argument is not a binding property, the binding engine will use that as the path. For example, these two statements are equivalent.

{Binding Path=Address.Streetname}
{Binding Address.Streetname}

Source

The Source property specifies the model that the binding statement will use to get the data. This is unnecessary when the DataContext is set.

FallbackValue (New in 8.1)

The Binding.FallbackValue provides to display in the control when the binding fails.

TargetNullValue (New in 8.1)

Provides a value to display in the control when the bound value is null.

ElementName

XAML allows a control to one of its properties to the property of another control. This is commonly called Element Binding.

Mode

The binding mode specifies if the binding is "OneWay", "TwoWay", or "OneTime". If Mode is not specified it will be set to OneWay.

UpdateSourceTrigger

Determines when an update will be triggered in two way binding situations. Possible values are Default, Explicit, and PropertyChanged.

Converter

Value Converters enable binding of disparate types, such as changing the color of an element based on the numeric value of a property on the model.

Binding Errors

If you haven't worked in XAML before now, there is a very important fact about data binding. When a binding statement fails, nothing happens. That's right, the user doesn't get any notification, there aren't any error dialogs, just a silent failure.

One one hand, this is very good. If you remember the days of Windows Forms, when there was a binding error, the user would get a modal dialog box. This dialog came from the framework, and could be very discrutpive. Especially if the binding problem was in a grid, the usre would get a steady stream of modal dialog boxes.

Databinding errors in the XAML world are very different. Nothing is presented to the user (unless you purposely add that interaction), and the default behavior is that the error will *only* be displayed in one place – and that's in the output window of Visual Studio. To illustrate this, update the binding statement for the LastName property to this:

```
<TextBlock Text="{Binding LastName1}" Margin="5"/>
```

When you run the app, look in the Output Window in Visual Studio. There will be a statement similar to this (it is actually a lot longer, and might differ slightly on your machine):

```
Error: BindingExpression path error: 'LastName1' property not found on 'Binding1.Person, Binding1,…
```

NOTE: If you don't have the Output window open, you can find it on the Debug menu under Windows->Output.

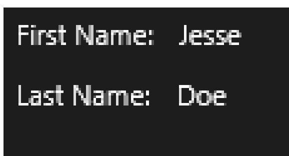
Windows 8.1 provides two new features in data binding specifically for problem situations. The FallbackValue provides a safety valve if the binding is in error, and the TargetNullValue provides a value to display if the source is null.

FallbackValue

New in Windows 8.1 is the ability to specify a value to be shown if the binding fails. To specify a fallback value, update the binding statement to the following:

```
<TextBlock Text="{Binding LastName1, FallbackValue='Doe'}" Margin="5"/>
```

Now when you run the app, the result will be like Figure 2.



This book was purchased by arosner@rosnertech.com

Figure 2 – Window using FallbackValue

It is important to note that the control is not databound. It is essentially showing a watermark.

TargetNullValue

Another new binding option is to specify what to show when the bound property is null. To illustrate this, correct the binding statement to once again bind to LastName and add the TargetNullValue:

```
"{Binding LastName, FallbackValue='Doe', TargetNullValue='Unknown'}"
```

The next change is to change the creation of the Person class to not provide a last name.

```
public Person person = new Person { FirstName = "Jesse"};
```

When you run the project, you get the following result as shown in Figure 3.

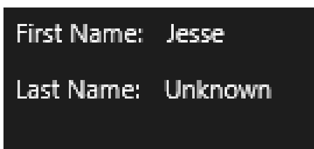


Figure 3 – TargetNullValue

Just like the FallbackValue, the value shown in the UI is window dressing. It does not affect the value of the model.

Binding to Elements

In addition to binding to objects such as the Person object, you are also free to bind to the value of other controls. For example, you might add a slider to your page and a TextBlock and bind the Text property of the TextBlock to the Value property of the slider.

To see this, create a new project and name it Binding2. Add the following XAML,

```
<StackPanel Orientation="Horizontal" Margin="100">
  <Slider Name="MySlider"
      Minimum="0"
      Maximum="100"
      Value="50"
      Width="300"
      Margin="10" />
```

```
   <TextBlock Margin="10"
           Text="{Binding ElementName=MySlider, Path=Value}"
           FontSize="42" />
</StackPanel>
```

That’s all you need; no code required. You’ve bound the TextBlock to the slider, as the slider changes value, the value in the TextBlock will be updated, as shown in figure 4,

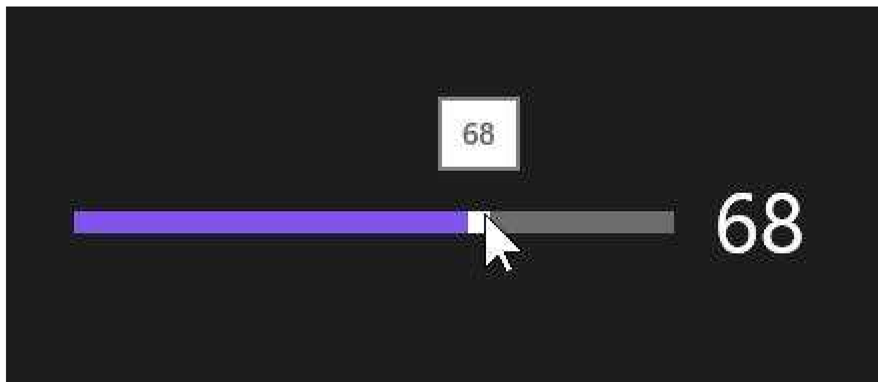


Figure 4 – Element Binding

You might be thinking at this point “That’s interesting. Why whould I use that?” Element Binding is a very powerful technique that can make your applications user interface much more interactive.

A common example is enabling or disabling controls based on the state of other controls. Figure 5 is an example where a user must check a box that they accept the conditions (perhaps an End User License Agreement) before they continue. Checking the checkbox should enable the button so the user can continue.

Figure 5 – Disabled continue Button

This can certainly be done with code, but we can do it entirely in mark up by leveraging Element Binding. Simply binding ths IsEnabled property of the

button to the IsChecked property of the checkbox links them together. The binding statement is writtent like this:

```
IsEnabled="{Binding ElementName=MyCheckBox,Path=IsChecked}"
```

When the check box is checked, the button becomes enabled as shown in Figure 6.

Figure 6 – Enabled button with checked check box

The entire XAML of this example is shown below.

```
<TextBlock Text="I accept the conditions:" Style="{StaticResource BodyTextStyle}"/>
<CheckBox Name="MyCheckBox" HorizontalAlignment="Left"/>
<Button Content="Continue" IsEnabled="{Binding ElementName=MyCheckBox,Path=IsChecked}"/>
```

Binding Modes

The Binding Mode sets the direction(s) that binding takes place. Binding comes in three "modes,"

- OneWay
- TwoWay
- OneTime

Specifying the Mode in a Binding statement is optional. If it's not specified, Mode gets set to the default value of OneWay.

The OneTime mode sets the value of the target when the window first loads and then severes the binding. Any changes to the source will not update the target. To see this, update the binding statement on the TextBlock by adding Mode=OneTime like this.

```
Text="{Binding ElementName=MySlider, Path=Value, Mode=OneTime}"
```

Run the project and the value in the text block will be set to 50 (the initial value of the slider). Changing the slider will not change the value in the text block.

OneWay binding mode sets the initial value just like OneTime, but then keeps the connection alive so that changes to the source can be reflected in the target. Changing the Mode to OneWay (or removing the Mode property completely) returns the example back to one way, and changing the slider will change the value in the text block.

It is important to note that this example works because the target is binding to a dependency property on the Slider. If you were to bind to the Person class we built for the first binding example and updated the source, the target value would not change. This is due to the class (as we have written it) missing some needed infrastructure. Fortunately, the fix is to leverage INotifyPropertyChanged. This is very simple and is covered in the next section.

Two way binding allows for binding back to the source so that user input can update the data source. This is what users expect when your application is binding to application data. To see two way binding at work, change the TextBlock to a TextBox and set the binding mode to TwoWay.

```
<StackPanel Orientation="Horizontal"
        Margin="100">
    <Slider Name="xSlider"
        Minimum="0"
        Maximum="100"
        Value="50"
        Width="300"
        Margin="10" />
    <TextBox Margin="10"
         Text="{Binding ElementName=xSlider, Path=Value, Mode=TwoWay}"
         FontSize="42"
         Height="75"
         VerticalAlignment="Top" />
 </StackPanel>
```

Run the application and move the slider; the value in the TextBox updates. Now type a new value between 0 and 100 into the TextBox and hit tab. The value of the slider changes to reflect the value you entered. That is two way binding at work.

UpdateSourceTrigger

In the previous example, the slider value didn’t change until the text box lost focus. But the text box updates immediately when the slider’s value changes. Why the different behavior? It is actually very clever how the framework determines when to send the update. By default, the update gets sent when the value changes on a non text based control, and when a text based control looses focus. If the screen jumps around at every key stroke (or sends a lot of error messages while the user is still typing), you will not keep the users around for long.

There are three options for the UpdateSourceTrigger binding property,

1. Default,
2. Explicit,
3. PropertyChanged

Default leaves the behavior the same as explained above. PropertyChanged will fire every time a property changes (not on lost focus). Explicit prevents the binding framework from updating the target, making it required to programmatically update values with the call to UpdateSource method.

Change the binding statement from the previous example by adding "UpdateSourceTrigger=PropertyChanged" as follows:

```
Text="{Binding ElementName=MySlider, Path=Value, Mode=TwoWay,  UpdateSourceTrigger=PropertyChanged}"
```

Run the program, and start typing into the text box. As you type, the slider will update.

INotifyPropertyChanged

It is possible for the value of a property in your data to change after it is displayed to the user. You would like for that value to be updated in the View. For example, it might be that you are retrieving the data from a Database and the data was changed by another instance of the program. It can be imperative that the user interface keep up with these changes.

Imagine that our application is showing books and how many are in stock for our store. A customer calls asking for "History of the English Speaking Peoples" by Winston Churchill. I look and see that there is one copy left. While I'm negotiating the price with the potential buyer you are on another computer and you sell the last copy. If my screen does not update to show that we are now sold out, I'm in real danger of selling a book I don't have.

To prevent this, we typically have the classes that will serve as Models in the view implement the INotifyPropertyChanged interface. This interface consists of exactly one event: PropertyChanged. You raise this event each time your property changes, passing in an EventArg that contains the name of your property. The XAML framework listens for this event and will then update any control bound to the property on the model that matches the name in the event arg. If the event arg is blank, all properties will be updated.

Create a copy of Bindings1 and name it Bindings3. Add a helper method to the Person class called NotifyPropertyChanged that wraps the event to cut down on repetitive code.

```
public event PropertyChangedEventHandler PropertyChanged;
private void NotifyPropertyChanged(string caller = "" )
{
```

```
   if ( PropertyChanged != null )
   {
      PropertyChanged( this, new PropertyChangedEventArgs( caller ) );
   }
}
```

To fire this event, change the getters and setters for the Person class so that they call NotifyPropertyChanged method from each Setter, passing in the name of the property that is changed. That way, each time we set the value of the property for any reason, the UI will be notified.

```
private string firstName;
 public string FirstName
 {
    get { return firstName; }
    set
    {
       firstName = value;
       NotifyPropertyChanged("FirstName");
    }
 }
 private string lastName;
 public string LastName
 {
    get { return lastName; }
    set
    {
       lastName = value;
       NotifyPropertyChanged("LastName");
    }
}
```

One problem with this implementation is the prevelance of magic strings. Every time a property is added, you have to remember to add the string name. And if the property name changes and you forget to update the string passed into the NotifyPropertyChanged method, you won't get a compiler error. The databound control just won't get updated.

To solve this problem, we can use the CallerMemberName attribute. This will set the value of the caller parameter to the name of the property that called the method, greatly simplifying the process.

```
public event PropertyChangedEventHandler PropertyChanged;
private void NotifyPropertyChanged( [CallerMemberName] string caller = "" )
{
   if ( PropertyChanged != null )
   {
      PropertyChanged( this, new PropertyChangedEventArgs( caller ) );
   }
}
```

The updated properties are listed here:

```
private string firstName;
 public string FirstName
 {
    get { return firstName; }
    set
    {
       firstName = value;
       NotifyPropertyChanged("FirstName");
    }
 }
 private string lastName;
 public string LastName
 {
    get { return lastName; }
    set
    {
       lastName = value;
       NotifyPropertyChanged("LastName");
    }
}
```

To test this, add a button to MainPage.xaml, below the TextBlocks,

```
<Button Name="cmdChange"
        Content="Change"
        Click="cmdChange_Click" />
```

When the button is pressed, we'll modify the FirstName of the person object,

```
private void cmdChange_Click( object sender, RoutedEventArgs e )
{
   person.FirstName = "Stacey";
}
```

Because of INotifyPropertyChanged, the UI will be updated when the value is changed, as shown in Figure 7,

Figure 7 - Person Changed

Binding to Collections

One of the key places for Data Binding is in showing lists of items. The two main controls to show lists are the ListView and the GridView. To show binding to collections, we will create an app that takes a list of People and displays it in two ways. First a ListView (shown in Figure 8) and then a GridView (shown in Figure 9).

Figure 8 – A Databound ListView

Figure 9 – A Databound GridView

Creating the Collection

Before we see this at work, we need to create a list. Start by creating a new application named Binding4. Instead of manually creating a list of Person objects, we going to create a helper function that will generate names for each of the Person objects. In the Person Class, create arrays to hold some First and Last names as well as an array of cities,

private static readonly string[] firstNames = { "Adam", "Bob", "Carl", "David", "Edgar", "Frank", "George", "Harry", "Isaac", "Jesse", "Ken", "Larry" };

```
private static readonly string[] lastNames = { "Aaronson", "Bobson", "Carlson", "Davidson", "Enstwhile", "Ferguson", "Harrison", "Isaacson", "Jackson", "Kennelworth", "Levine" };
private static readonly string[] cities = { "Boston", "New York", "LA", "San Francisco", "Phoenix", "San Jose", "Cincinnati", "Bellevue" };
```

We can then "assemble" new Person objects by randomly selecting from each of the arrays, creating as many people as we need. To keep things simple, we'll do this in a static method so that we can call it from our MainPage.xaml.cs without instantiating an object. ListViews and GridViews bind very well it IEnumerables, so we'll have it return an IEnumerable<Person>,

```
public static IEnumerable<Person> CreatePeople( int count )
{
    var people = new List<Person>();

    var r = new Random();

    for ( int i = 0; i < count; i++ )
    {
        var p = new Person()
        {
            FirstName = firstNames[r.Next( firstNames.Length )],
            LastName = lastNames[r.Next( lastNames.Length )],
            City = cities[r.Next( cities.Length )]
        };
        people.Add( p );
    }
    return people;
}
```

The entire Person class now looks like this:

```
public class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string City { get; set; }

   private static readonly string[] firstNames = { "Adam", "Bob", "Carl", "David", "Edgar", "Frank", "George", "Harry", "Isaac", "Jesse", "Ken", "Larry" };
   private static readonly string[] lastNames = { "Aaronson", "Bobson", "Carlson", "Davidson", "Enstwhile", "Ferguson", "Harrison", "Isaacson", "Jackson", "Kennelworth", "Levine" };
   private static readonly string[] cities = { "Boston", "New York", "LA", "San Francisco", "Phoenix", "San Jose", "Cincinnati", "Bellevue" };


   public static IEnumerable<Person> CreatePeople( int count )
   {
      var people = new List<Person>();
```

```
        var r = new Random();

        for ( int i = 0; i < count; i++ )
        {
            var p = new Person()
            {
                FirstName = firstNames[r.Next( firstNames.Length )],
                LastName = lastNames[r.Next( lastNames.Length )],
                City = cities[r.Next( cities.Length )]
            };
            people.Add( p );
        }
        return people;
    }
}
```

Creating a Data Bound ListView

A ListView lays out the data vertically in a single column. ListViews are most often used when an application is in Portrait or Snapped Mode due to the reduced width of the view screen when apps are in those two layouts. ListViews scroll vertically, which is the preferred scrolling direction for Portrait and Snapped views.

Open up MainPage.XAML and either drag a ListView from the Tool Box onto the page and name it MyListView or enter the following XAML:

```
<ListView Name="MyListView">
</ListView>
```

We need to set the ItemsSource property of the ListView to the result of calling the static CreatePeople function we added to the Person class. Add the following line of code in MainPage. We'll do this in MainPage.xaml.cs, in the constructor,

```
public MainPage()
 {
    this.InitializeComponent();
    MyListView.ItemsSource = Person.CreatePeople(25);
 }
```

When you run this, you won't get quite what you were hoping for, as shown in the cropped Figure 10,

This book was purchased by arosner@rosnertech.com

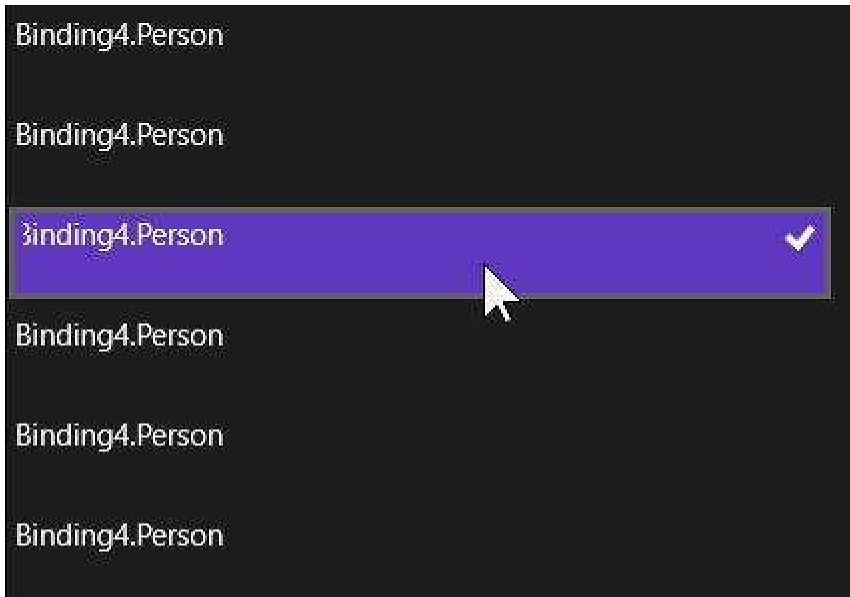


Figure 10 – Binding without data template

The binding is working perfectly; there is one entry for each of the 25 Person objects. The problem is that the ListView doesn't know how to display a Person object and so it falls back to just showing the name of the type. Databound List Controls such as the ListView and the GridView require and ItemTemplate to define how each item in the list is displayed. The ItemTemplate needs to have a DataTemplate that contains the defining markup to display each item. Update the ListView XAML to match the following:

```
<ListView Name="MyListView">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Border Width="300" Height="Auto" BorderBrush="Beige" BorderThickness="1">
      <StackPanel>
        <TextBlock>
          <Run Text="{Binding FirstName}" />
          <Run Text=" " />
          <Run Text="{Binding LastName}" />
          <Run Text=" " />
          <Run Text="(" />
          <Run Text="{Binding City}" />
          <Run Text=")" />
        </TextBlock>
      </StackPanel>
      </Border>
    </DataTemplate>
  </ListView.ItemTemplate>
```

Let’s break down the ItemTemplate. The main container is the DataTemplate whose Data Context is the current Item in the collection. After that, it’s just a matter of formatting the data to your liking. Inside the TextBlock in the sample is a XAML trick that allows for more compext binding into a single TextBlock. By using the <Run/> markup, you can add multiple binding statements to build a single TextBlock. In our carse, we are combining the FirstName, LastName, and City data elements with some punctuation. This pattern will be repeated for all 25 entries in the ListView as shown in Figure 11,

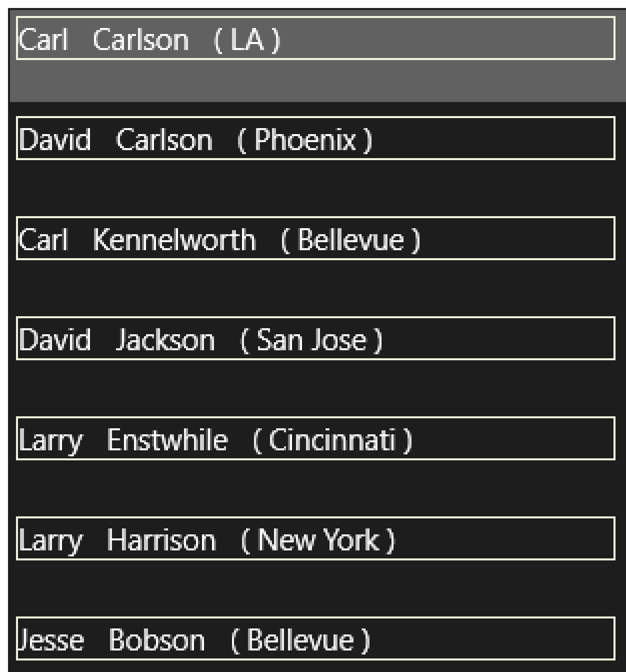


Figure 11 List View

Creating a Data Bound GridView

While the ListView scrolls vertically, the GridView layout fills in vertically until the container filled, then starts in the next column, and so on. If the entire screen is filled, the GridView will scroll horizontally. This layout container is used for apps while they are in Portrait or Filled mode, and usually shows a bit more information than a ListView.

To show a GridView in action, comment out the ListView we just created in MainPage.XAML and add the following XAML into the page.

```
<GridView Name="MyGridView">
  <GridView.ItemTemplate>
    <DataTemplate>
      <Border Width="300" Height="100" BorderBrush="Beige" BorderThickness="1">
```

```
            <StackPanel Orientation="Vertical">
              <TextBlock Text="{Binding FirstName}" Style="{StaticResource ItemTextStyle}"/>
              <TextBlock Text="{Binding LastName}" Style="{StaticResource ItemTextStyle}"/>
              <TextBlock Text="{Binding City}" Style="{StaticResource ItemTextStyle}"/>
            </StackPanel>
          </Border>
        </DataTemplate>
      </GridView.ItemTemplate>
    </GridView>
```

You'll see that the techniques are the same, just with a little different styling. The resulting GridView is shown in Listing 12.

Bob
Bobson
Cincinnati

Harry
Enstwhile
LA

Ken
Harrison
Bellevue

Adam
Carlson
Cincinnati

Carl
Bobson
Phoenix

Edgar
Harrison
San Francisco

Figure 12 – The GridView

INotifyCollectionChanged

Just as you want the application to be updated when the value of a property changes, you want to be notified if the contents of a collection changes (e.g. an item is added, deleted, etc.). Similar to INotifyPropertyChanged, there is an interface made just for this scenario, INotifyCollectionChanged. This interface has just one event,

```
public event NotifyCollectionChangedEventHandler CollectionChanged;
```

This event needs to be called each time an item is added or removed from the collection. We could inherit from a specialized collection and code this ourselves (much like we did in the Person Class for INotifyPropertyChanged), but Microsoft has already done the work for us and provided a custom class

called ObservableCollection<T>. You can use the ObservableCollection<T> class in the same way you use other collection classes.

In fact, you can modify Person.cs to use an ObservableCollection<Person> and the program will continue to work just like it did before with the added feature that the ListView or GridView will be updated if an item is added or removed from the collection. All we need to do is simply change the return type of the CreatePeople method to return ObservableCollection<Person> and change the internal variable from List<Person> to ObservableCollection<Person>,

```
public static ObservableCollection<Person> CreatePeople( int count )
 {
    var people = new ObservableCollection<Person>();
```

Note that the ObservableCollection class only updates the client if an item is added or deleted, but not if the item is changed (e.g., a property on an item in the collection is changed). If you want notification on the item being changed, you need to implement INotifyPropertyChanged on the items.

Data Converters

There are times that you will want to bind an object to a control but the types don't match up correctly. For example, you might want to bind a check box to a value, setting the check mark if the value is anything other than zero. To do this, you need to convert the integer value to a Boolean, and for that you need a DataConverter. A DataConverter is any class that implements IValueConverter which takes two methods: Convert and ConvertBack.

To see this at work, create a new project and name it Binding5. In MainPage.xaml we'll have a CheckBox which will be set if the Num property of the Person object is not zero, a TextBlock which will display the Num property and a Button which will generate a new Num property at random.

```
<StackPanel Margin="100"
         Orientation="Horizontal"
         Height="100">
    <CheckBox Name="xCheckBox"
           Content="Is Not Zero"
           Margin="10"
           IsChecked="{Binding Num, Converter={StaticResource numToBool}}"/>
    <TextBlock Name="xTextBlock"
           Margin="10"
           FontSize="42"
           VerticalAlignment="Center"
           Text="{Binding Num}"/>
    <Button Name="xButton"
```

```
        Content="Generate Number"
        Click="xButton_Click"
        Margin="10" />
</StackPanel>
```

Notice that the IsChecked binding for the CheckBox calls on the DataConverter which was identified in the Resources section,

```
<Page.Resources>
  <local:IntegerToBooleanConverter x:Key="numToBool" />
</Page.Resources>
```

We’ll use the Person class we used in previous iterations, adding a Num property to the Person class so that we can bind to it (e.g., number of children, etc.)

```
  public class Person : INotifyPropertyChanged
  {
    private string firstName;
    public string FirstName
    {
      get { return firstName; }
      set
      {
        firstName = value;
        NotifyPropertyChanged();
      }
    }
    private string lastName;
    public string LastName
    {
      get { return lastName; }
      set
      {
        lastName = value;
        NotifyPropertyChanged();
      }
    }

    private int num;
    public int Num
    {
      get { return num; }
      set
      {
        num = value;
        NotifyPropertyChanged();
      }
    }


    public event PropertyChangedEventHandler PropertyChanged;
```

```
    private void NotifyPropertyChanged( [CallerMemberName] string caller = "" )
    {
        if ( PropertyChanged != null )
        {
            PropertyChanged( this, new PropertyChangedEventArgs( caller ) );
        }
    }

}
```

Key to making this work, of course, is the converter. Create a new class IntegerToBooleanConverter, and have it implement IValueConverter,

```
public class IntegerToBooleanConverter : IValueConverter
{
    public object Convert( object value, Type targetType, object parameter, string language )
    {
        int num = int.Parse( value.ToString() );
        if ( num != 0 )
            return true;
        else
            return false;

    }

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

The binding mechanism will take care of passing in the value (in this case, Num) the targetType (in this case Boolean), a parameter if there is one and the language. Our job is to convert the value to a Boolean, which we do in the body of the Convert method. The net effect is to convert any non-zero value to the Boolean value true, which sets the check box as shown in Figure 13,

Figure 13 - Boolean Converter

ConvertBack is used in TwoWay binding scenarios. For example, if you needed to format a number from the model to resemble the user’s local currency, you can do that in the ConvertMethod. In order to save that number back into the model, it would need to be changed back into a number from the string representation that includes the currency symbol and the commans.

Summary

Binding is a key element of programming for Windows 8.1. With Data Binding you can connect a data source, which can be virtually any object, to a property on a visual element (or you can bind two visual elements together). Along the way you can convert data from one type to another to facilitate the binding. The INotifyPropertyChanged intrerface allows your view to be updated when your data changes. Two way binding allows your data object to be changed based on user input.

CHAPTER 5

n n n

Views

In this chapter we will examine the four visual states available for a Windows 8 application:

- Full
- Landscape
- Fill
- Snapped

In addition, we will examine the two principal controls used to display data in these states: GridView for the first three and ListView for the fourth, and we will take a look at the Grid App and Split App templates provided by Visual Studio.

GridView and ListView

GridView and ListView are nearly identical controls, both inheriting all their methods, properties and events from ListViewBase. The key difference is that GridView is designed to scroll horizontally and ListView vertically.

You can get a GridView out of the box using the Grid App template, as will be covered later in this chapter, but things are a lot clearer and simpler if you start with a blank application and then implement your own GridView.

GridView is one of the most powerful out-of-the-box controls in Windows 8, but fully understanding how to use it is not necessarily trivial. The complexity is a result of the fact that GridViews are often used to hold groups of collections, rather than simple collections. The GridView template assumes that it will be displaying groups of collections of items, and this can add complexity to working with the GridView.

A basic GridView is shown in figure 1, with people sorted into groups by city.

Figure 1 – Gridview: People sorted into groups by City

Getting Started

Creating a program to illustrate GridView is nearly trivial. Create a new Blank Application and name it GridView1. We'll divide our page's Grid (not to be confused with GridView) into rows and columns. In the left column we'll add two buttons: Add and Insert. In the right hand column we'll add a very simple, GridView.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
   <Grid.ColumnDefinitions>
      <ColumnDefinition Width="1*" />
      <ColumnDefinition Width="3*" />
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*" />
```

```
    </Grid.RowDefinitions>
    <Button Name="Add"
            Content="Add"
            Grid.Row="0"
            Grid.Column="0"
            Click="Add_Click"/>
    <Button Name="Insert"
            Content="Insert"
            Grid.Row="1"
            Grid.Column="0"
            Click="Insert_Click" />
    <GridView Name="xGridView"
              Grid.RowSpan="4"
              Grid.Column="1"
              Grid.Row="0" />

</Grid>
```

The event handler for both the add button and the insert button are nearly intuitive,

```
private int counter = 1;

private void Add_Click( object sender, RoutedEventArgs e )
 {
    xGridView.Items.Add( "Item " + counter++ );
 }

 private void Insert_Click( object sender, RoutedEventArgs e )
 {
    xGridView.Items.Insert( 0, "Item " + counter++ );
 }
```

Clicking Add and/or Insert a number of times will fill the first column of the grid, and then the second column will begin to fill. Unfortunately, the grid will size all the items based on the size of the first item, and things get crowded quickly, as shown in figure 2.

This book was purchased by arosner@rosnertech.com

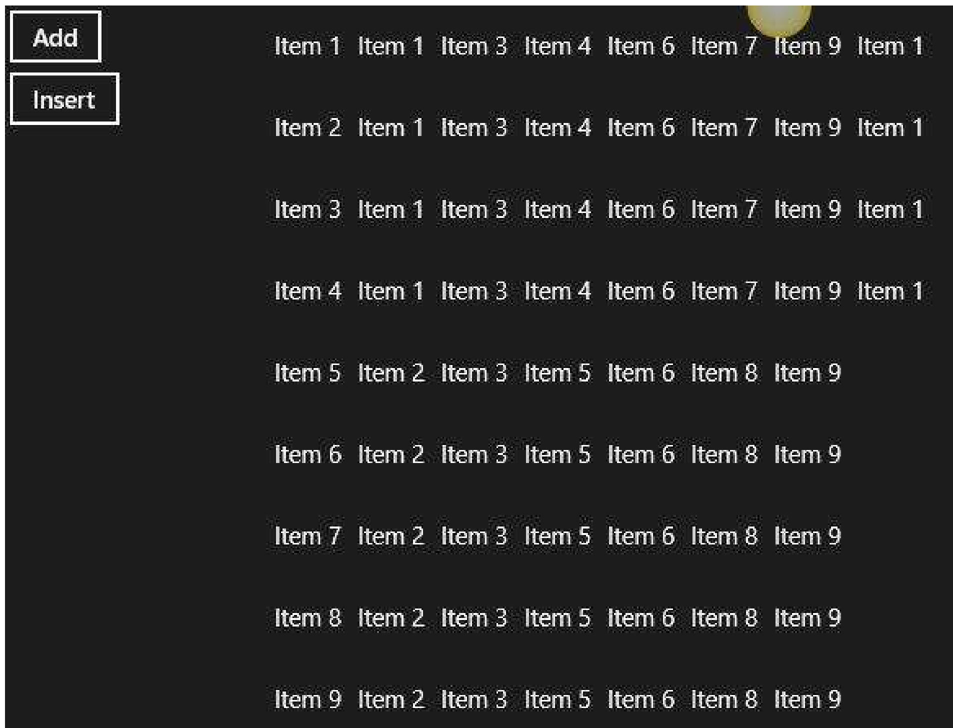


Figure 2 – Crowded Grid

We can fix this by adding a style,

```
<GridView Name="xGridView"
        Grid.RowSpan="4"
        Grid.Column="1"
        Grid.Row="0" >
   <GridView.ItemContainerStyle>
      <Style TargetType="GridViewItem">
         <Setter Property="Width"
               Value="150" />
      </Style>
   </GridView.ItemContainerStyle>
```

```
</GridView>
```

Now when the grid is populated there is a bit more breathing room between the columns and nothing is truncated, as shown in Figure 3.

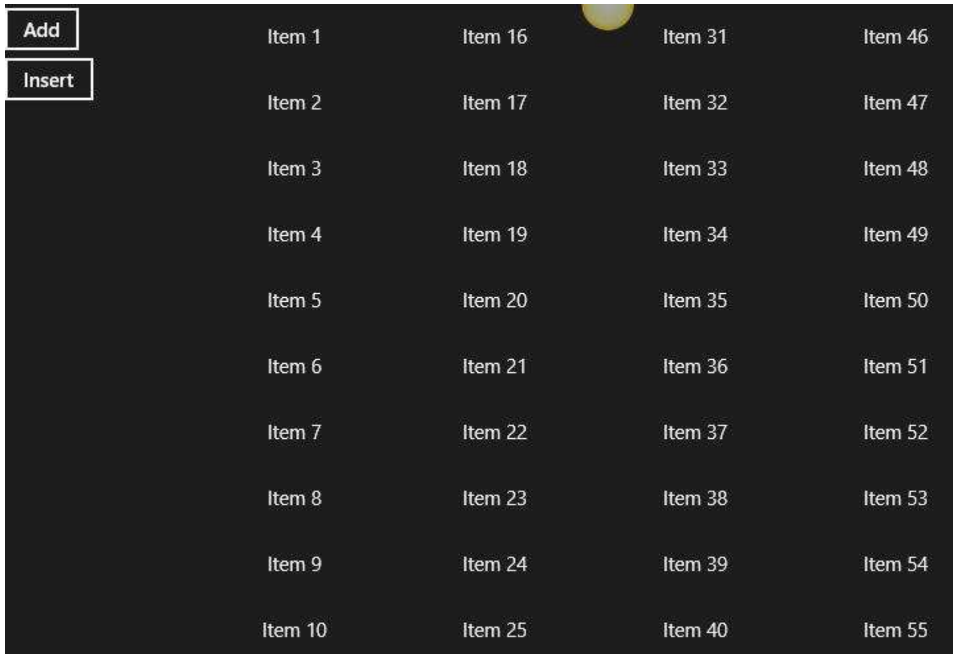


Figure 3 – Grid With Style

GridView with Collections of Collections

GridViews are typically used to display more complex data: specifically groups of collections, and implementing that is not trivial.

Let's assume that our data consists of people, with each person having a first name, a last name and a city.

```
public class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string City { get; set; }
```

We'd like to display these people in a GridView, but organized (grouped) by city. The trick is to create a Group class that will allow you to have a key on which you'll group the people (in this case city) and a list of people who match that key,

```
public class Group<TKey, TItem>
{
    public TKey Key { get; set; }
    public IList<TItem> Items { get; set; }
}
```

To generate your initial (ungrouped) list of people you might go out to a service, or read in XML, or in our case, you might generate names and cities at random. To do so, we'll have three arrays,

```
private static readonly string[] firstNames = { "Adam", "Bob", "Carl", "David", "Edgar", "Frank", "George", "Harry", "Isaac", "Jesse", "Ken", "Larry" };
private static readonly string[] lastNames = { "Aaronson", "Bobson", "Carlson", "Davidson", "Enstwhile", "Ferguson", "Harrison", "Isaacson", "Jackson", "Kennelworth", "Levine" };
private static readonly string[] cities = { "Boston", "New York", "LA", "San Francisco", "Phoenix", "San Jose", "Cincinnati", "Bellevue" };
```

And you'll need a method to generate a given number of people randomly mixing names and cities,

```
public static IEnumerable<Person> CreatePeople(int count)
{
    var people = new List<Person>();

    var r = new Random();

    for (int i = 0; i < count; i++)
    {
        var p = new Person()
        {
            FirstName = firstNames[r.Next(firstNames.Length)],
            LastName = lastNames[r.Next(lastNames.Length)],
            City = cities[r.Next(cities.Length)]
        };
        people.Add(p);
    }
    return people;
}
```

Creating the GridView

Let's turn now to the view. In MainPage.xaml you'll add a GridView control which will bind its itemsSource to a CollectionViewSource.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
<GridView x:Name="myGridView"
      ItemsSource="{Binding Source={StaticResource cvs}}">
```

The CollectionViewSource manages the collection of lists, and is created in the Resources Section. Be sure to set the IsSourceGrouped property to True and set the ItemsPath to the list of objects (in our case, the Items property in the Group class.)

```
<Page.Resources>
  <CollectionViewSource x:Name="cvs"
              IsSourceGrouped="True"
              ItemsPath="Items" />
</Page.Resources>
```

Within the GridView itself we want to determine how the groups will be organized, and for that we use the ItemsControl.ItemsPanel,

```
<ItemsControl.ItemsPanel>
  <ItemsPanelTemplate>
    <StackPanel Orientation="Horizontal" />
  </ItemsPanelTemplate>
</ItemsControl.ItemsPanel>
```

To ensure that there is sufficient room between each entry, we'll want to add some padding around each item. We do that with the ItemsControl.ItemContainerStyle,

```
<ItemsControl.ItemContainerStyle>
  <Style TargetType="GridViewItem">
    <Setter Property="HorizontalContentAlignment"
        Value="Left" />
    <Setter Property="Padding"
        Value="5" />
```

```
   </Style>
</ItemsControl.ItemContainerStyle>
```

Next we want to determine how each group will be laid out. This consists of two templates: one for the header and one for the items themselves. The Headertemplate will, in our case, display the city which is, you'll remember the Key property in the Group class.

```
<ItemsControl.GroupStyle>
   <GroupStyle>
      <GroupStyle.HeaderTemplate>
         <DataTemplate>
            <TextBlock Text="{Binding Key}"
                  FontSize="26.67" />
         </DataTemplate>
      </GroupStyle.HeaderTemplate>
```

We then add an ItemsPanelTemplate to ensure that the members of each group are listed vertically and have an appropriate width,

```
<GroupStyle.Panel>
   <ItemsPanelTemplate>
      <VariableSizedWrapGrid Orientation="Vertical"
                  ItemWidth="220" />
   </ItemsPanelTemplate>
</GroupStyle.Panel>
```

The Supporting Code

In the code behind I generate the 200 people into a list and then use a LINQ statement to obtain a list of Group objects, which I set as the source for the CollectionViewSource. To begin, open MainPage.xaml.cs and create two lists,

```
private IList<Person> people;
private IList<Group<object, Person>> groupedPeople;
```

Next, add the code to generate 200 people. Add this to the constructor,

```
people = Person.CreatePeople( 200 ).ToList();
groupedPeople = ( from person in people
            group person by person.City into g
```

```
        orderby g.Key
        select new Group<object, Person>
        {
            Key = g.Key.ToString(),
            Items = g.ToList()
        } ).ToList();
cvs.Source = groupedPeople;
```

Notice that I don’t need an item template as I’ve overridden the ToString method of Person,

```
public override string ToString()
{
    return string.Format("{0} {1} ({2})", FirstName, LastName, City);
}
```

The result is a fully populated grid with people rather than random data, as shown in figure 4

Bellevue		Boston
Frank Enstwhile (Bellevue)	Larry Enstwhile (Bellevue)	Carl Levine (Boston)
Ken Isaacson (Bellevue)	Frank Carlson (Bellevue)	George Davidson (Boston)
Bob Kennelworth (Bellevue)	Bob Bobson (Bellevue)	Bob Ferguson (Boston)
Ken Davidson (Bellevue)	George Bobson (Bellevue)	Carl Davidson (Boston)
Larry Enstwhile (Bellevue)	Ken Kennelworth (Bellevue)	Adam Aaronson (Boston)
Jesse Carlson (Bellevue)	George Jackson (Bellevue)	Bob Aaronson (Boston)
George Levine (Bellevue)	Ken Kennelworth (Bellevue)	Ken Carlson (Boston)
Carl Enstwhile (Bellevue)		David Ferguson (Boston)
Larry Enstwhile (Bellevue)		Carl Bobson (Boston)

Figure 4 - Gridview with people

SnapView

Every Windows 8 application must support SnapView, in which your application is allocated 320x768 pixels – that is, your application is squeezed into a relatively thin sliver on the left or right of the screen.

GridView leaves much to be desired when seen in SnapView as seen in Figure 5.

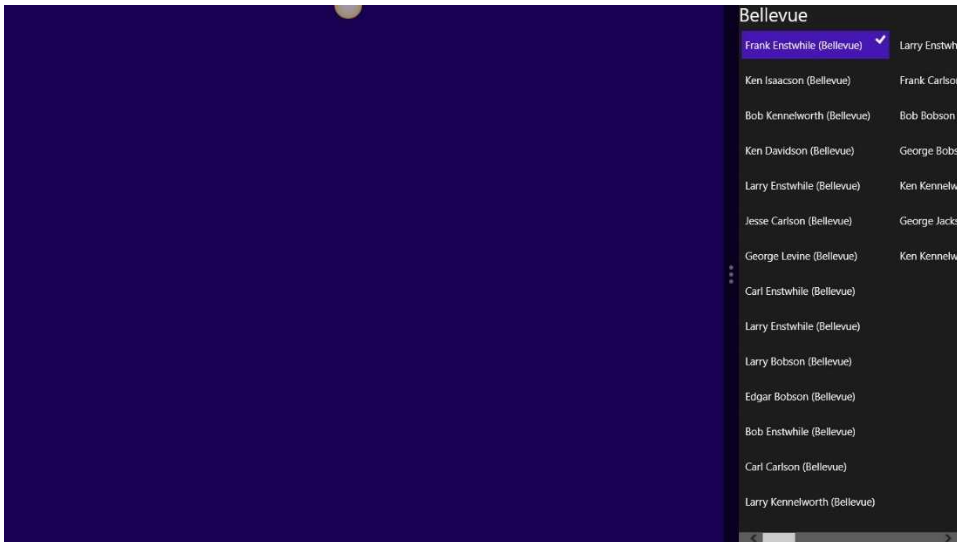


Figure 5 GridView in SnapView

The common solution to how poor a GridView looks in SnapView is to hide your GridView and to display the same data in a ListView, which works well in those dimensions as it scrolls vertically, as shown in figure 6

Figure 6 – In Snap View with List View

Don’t be confused: GridView and ListView are controls, SnapView is a state of the application

---

There are a few steps to implementing an application that supports SnapView well with these controls. What you want is:

- In Full or Landscape View show the GridView
- In FillView show the GridView
- In SnapView show the ListView

First, you need to know which view you are in, and you need to respond to a change in views. Here are the steps:

- Handle the Window SizeChanged event to notice that the user may have put you into or out of snapped view
- Obtain the current visual state (e.g., Snapped)
- Tell the VisualStateManager to go to the appropriate corresponding state
- Implement the Storyboard that handles the ViewState change
  - Collapse the GridView
  - Show the ListView
  - Make any other appropriate changes (e.g., change the title font size)

Creating the Sample Application

We’ll build on the application we have already created. Run it again and put it into snap view, either by dragging another application in until your running application snaps, or by pressing Windows-dot. Notice that what you get is a thin slice of a GridView. The GridView itself is unchanged.

Add a Title

We want to add a title to the page, so update the outermost grid with two rows and place the title TextBlock in the top row,

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <TextBlock x:Name="Title"
         Text="Cities"
         FontSize="42"
         Grid.Row="0" />
```

Run the application and notice that your GridView scrolls underneath the title, which is just what we want.

Adding the ListView

When we are in SnappedView we want to display this same data in a ListView. Since ListView and GridView share all the same methods, events and properties, you can start by copying and pasting the GridView so that you have two, one above the other. Change the second GridView to a ListView and fix up any place in that ListView that says GridView (e.g., change the ItemContainerStyle TargetType from GridViewItem to ListViewItem).

Name the GridView *FullView* and name the ListView *SnappedView* (this is arbitrary and you can name them anything you want as long as you match up the names with the storyboard targets).

Here's the complete ListView,

```
<ListView x:Name="SnappedView"
      Grid.Row="1"
      ItemsSource="{Binding Source={StaticResource cvs}}"
      Visibility="Collapsed">
  <ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel Orientation="Vertical" />
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>
  <ItemsControl.ItemContainerStyle>
    <Style TargetType="ListViewItem">
      <Setter Property="HorizontalContentAlignment"
          Value="Left" />
      <Setter Property="Padding"
```


This book was purchased by arosner@rosnertech.com

```
            Value="5" />
        </Style>
    </ItemsControl.ItemContainerStyle>
    <ItemsControl.GroupStyle>
        <GroupStyle>
            <GroupStyle.HeaderTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding Key}"
                               FontSize="26.67" />
                </DataTemplate>
            </GroupStyle.HeaderTemplate>
            <GroupStyle.Panel>
                <ItemsPanelTemplate>
                    <VariableSizedWrapGrid Orientation="Vertical"
                                           ItemWidth="220" />
                </ItemsPanelTemplate>
            </GroupStyle.Panel>
        </GroupStyle>
    </ItemsControl.GroupStyle>
</ListView>
```

Notice that it is nearly identical to the GridView and that, most important, its ItemsSource is bound to the same CollectionViewSource as the GridView. This is very important as it will enable both controls to point to the same data. In fact, by using the CollectionViewSource, if we select an item in the GridView and then switch to SnapView, the same item will be selected.

You now need to add the ViewStateManager markup to move into and out of the SnapView. **Put this code *below* the GridView and ListView but within the outermost Grid.**

Handling Application State

You begin managing Application State by declaring the Visual State Group and within that the ApplicationViewStatesGroup. Within the ApplicationViewStatesGroup you'll declare four Visual States:

- FulllScreenLandscape
- Filled
- FullScreenPortrait
- Snapped

They must have these exact names. The first three will be empty but the fourth, Snapped, will have a Storyboard to implement the changes you want to have occur when you move into Snapped view.

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ApplicationViewStates">
    <VisualState x:Name="FullScreenLandscape" />
    <VisualState x:Name="Filled" />
    <VisualState x:Name="FullScreenPortrait" />
    <VisualState x:Name="Snapped">
      <Storyboard>
        <!--storyboard code goes here -->
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Inside the Storyboard we’ll add markup to change the size of the Title from its initial size of 42 to 20,

```
<ObjectAnimationUsingKeyFrames Storyboard.TargetName="Title"
                    Storyboard.TargetProperty="FontSize">
  <DiscreteObjectKeyFrame KeyTime="0"
                  Value="20" />
</ObjectAnimationUsingKeyFrames>
```

Notice that, while we are using animation syntax, there really is no animation; the change happens instantly (KeyTime=”0”). We’ll add two other AnimationKeyFrames, one to collapse the GridView and one to show the ListView,

```
<ObjectAnimationUsingKeyFrames Storyboard.TargetName="FullView"
                    Storyboard.TargetProperty="Visibility">
  <DiscreteObjectKeyFrame KeyTime="0"
                  Value="Collapsed" />
</ObjectAnimationUsingKeyFrames>
<ObjectAnimationUsingKeyFrames Storyboard.TargetName="SnappedView"
                    Storyboard.TargetProperty="Visibility">
  <DiscreteObjectKeyFrame KeyTime="0"
                  Value="Visible" />
</ObjectAnimationUsingKeyFrames>
```

All of these changes happen, essentially, instantaneously.

We’re all set, when the view state changes to snapped we’ll diminish the size of the title and we’ll hide the GridView and show the ListView. When we move out of SnappedView we’ll return to the default settings (since there is no storyboard for those states).

But how do we get the view state to change?

LayoutAwarePage

Many of the templates used to create new applications create a class, LayoutAwarePage, in the Common folder. This page is somewhat complex, and we’ll create our own simpler version. Create a new class named LayoutAwarePage in the Common folder. The new class will inherit from Page and will implement two event handlers: one for when the window changes size and one for when the page is loaded,

```
public class LayoutAwarePage : Page
{
   public LayoutAwarePage()
   {
      Window.Current.SizeChanged += WindowSizeChanged;
      Loaded += LayoutAwarePage_Loaded;
   }
```

The code for when the page is loaded and for when the window changes size is identical; both call a helper method: SetVisualState,

```
private void LayoutAwarePage_Loaded( object sender, RoutedEventArgs e )
{
   SetVisualState();
}

private void WindowSizeChanged( object sender, WindowSizeChangedEventArgs e )
{
   SetVisualState();
}
```

The job of SetVisualState is to obtain the current visual state and to ask the VisualStateManager to go to the state with that name. This forces the change in ViewState that is picked up by the page.

```
public void SetVisualState()
{
   string viewValue= ApplicationView.Value.ToString();
   VisualStateManager.GoToState( this, viewValue, false );
}
```

We want our MainPage to derive from LayoutAwarePage. To accomplish this, open MainPage.xaml.cs and add a using statement,

```
using GridView.Common;
```

and then set the MainPage class to derive from LayoutAwarePage,

```
public sealed partial class MainPage : LayoutAwarePage
```

In addition, add a namespace to MainPage.xaml,

```
xmlns:common="using:GridView.Common"
```

and change the type of the page from Page to LayoutAwarePage

```
<common:LayoutAwarePage x:Class="GridView.MainPage"
```

Remember to do the same in the resources section,

```
<common:LayoutAwarePage.Resources>
   <CollectionViewSource x:Name="cvs"
                  IsSourceGrouped="True"
                  ItemsPath="Items" />
</common:LayoutAwarePage.Resources>
```

You can now run the application and switch between snapped and unsnapped views and you should see the application “do the right thing.”

The Grid App Template

As mentioned earlier, the GridView control is perhaps the most complex control you'll work with in Windows Store XAML applications since it works with collections of collections. The Grid App template was designed to help you get started with a GridView-based application, but it's (necessarily) a bit complex too, for the same reason.

Compounding things a bit, there's no owner's manual or even an obvious "Start here!" to tell you what's going on. It's worth figuring out, though: once you understand the Grid App template, it can really help by providing a useful architecture for managing multi-dimensional data.

What's included in the Grid App Template

Create a new Grid App project and take a look at the generated files:

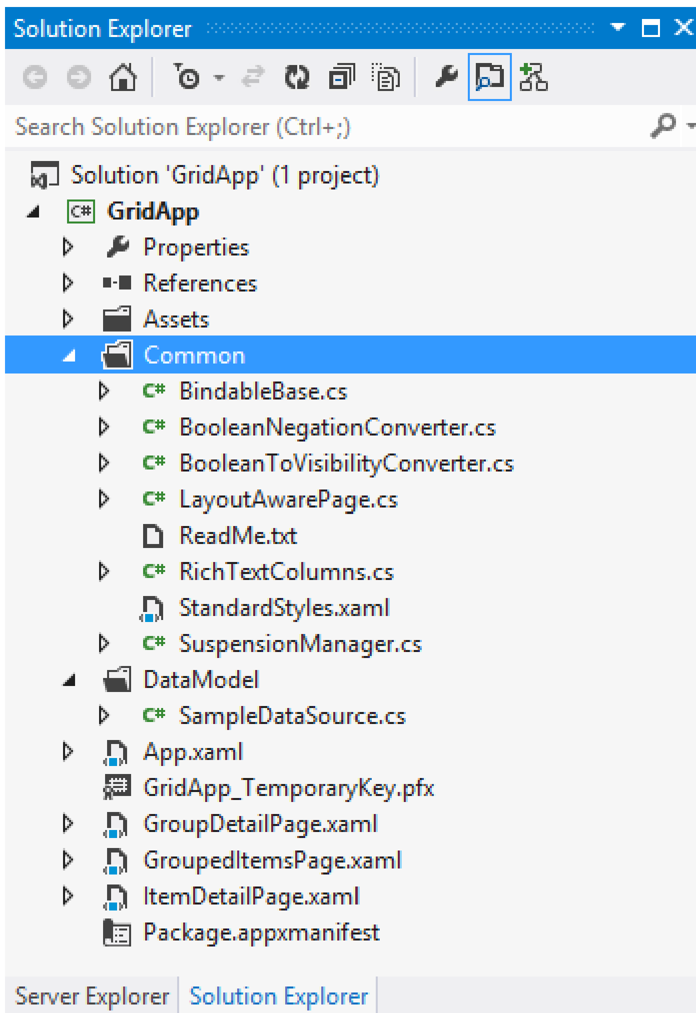


Figure 7

The Pages

There are three Pages: GroupedItemsPage, GroupDetailPage and ItemDetailPage. **GroupedItemsPage** is the home page. It displays grouped collections of items.

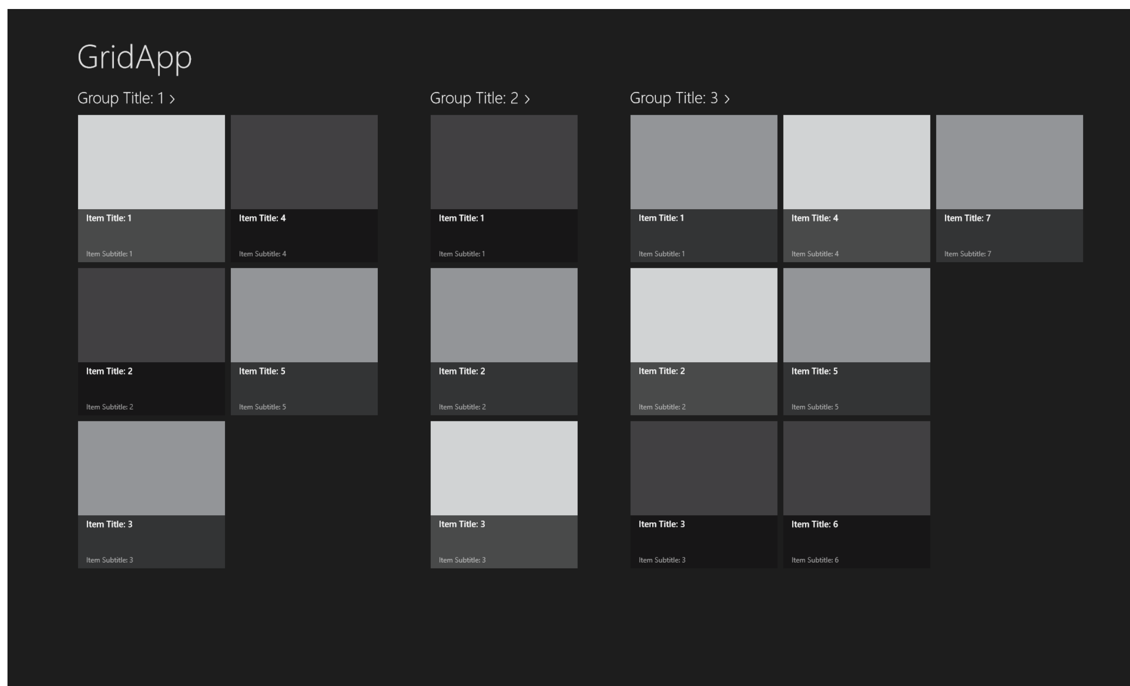


Figure 8

Clicking on a group title shows the **GroupDetailPage,** which displays all items in that group:

Figure 9

Finally, clicking on an individual item displays the **ItemDetailPage**, which (as you probably guessed) shows details for that one item.

Figure 10

The Common folder

The Common folder contains several files, introduced via a ReadMe.txt file which basically says "Hands off the XAML, be very careful with the class files." Of primary interest in this section is the BindableBase, which implements INotifyPropertyChanged and raises PropertyChangedEventHandler in a setter. There are only a few lines of code here, but this base simplifies several classes in the SampleDataSource.

The SampleDataSource

SampleDataSource.cs (found in /DataModel) actually contains four classes:

SampleDataCommon – This is a base class for both SampleDataItem and SampleDataGroup. Since both Items and Groups have similar properties (Title, Subtitle, Description, Image) they can leverage a common base class. This class inherits from BindableBase and calls SetProperty in each property setter, so any changes you make to properties in either an item or a group will raise a property changed event and update the UI.

SampleDataItem – This class represents a single item. In addition to the common properties in SampleDataCommon, an Item adds two properties. It has a Group property, which references the group it belongs to, and a Content property, which includes extended textual content.

SampleDataGroup – A Group adds a few properties to the common properties: Items (an ObservableCollection of all items in the group) and TopItems (an ObservableColleciton of the first 12 items). The TopItems collection exists to allow for rapid interaction and binding, and it's automatically maintained – whenever an item in a group is added, edited, or removed, the TopItems is automatically updated if necessary.

SampleDataSource – This class populates the items and groups on startup and maintains a catalog in a static collection. It's also got a few static utility methods to allow retrieving a group or item by unique ID.

Options for integrating your data into a Grid App project

This book was purchased by arosner@rosnertech.com

The Grid App template works fine out of the box, but it's just displaying hardcoded data. The SampleDataConstructor initializer is really simple – it just populates groups and items with simple strings.

There are a few ways to integrate with the Grid App template:

1. You can just modify the way that SampleDataSource loads data. This is ideal if the properties in SampleDataCommon, SampleDataItem and SampleDataGroup fit your data model. That's not all that far-fetched – these classes have very generic names which would work for quite a few cases.

2. You can throw out the SampleDataSource class completely and load the Group and Item collections using your own custom logic. You can see exactly where you'd make these changes by searching the application for "// TODO:" since the LoadState() method of all pages (GroupedItemsPage.xaml.cs GroupDetailPage.xaml.cs, and ItemDetailPage.xaml.cs) all begin with the comment " // TODO: Create an appropriate data model for your problem domain to replace the sample data"

3. An important third option to keep in mind is that the Grid App may not be a good fit for your specific application. If your Item properties and display need to be substantially different, you're going to end up doing more work than if you just built custom views in a Blank application, as shown earlier in this chapter.

If you've settled on option 3, our work here is done!

If you're choosing whether to go with option 1 or 2, it's a little bit harder. While it may be tempting to think that your application is a unique snowflake, keep in mind that the SampleDataSource classes do handle a lot of helpful things for you, like raising property change notifications through BindableBase and keeping the TopItems collection for each group up to date. Unless you're fairly certain that you're in that narrow range that will profit from the Grid App template but can't work with the SampleDataSource, I'd recommend starting with option 1 until you're certain you've outgrown it.

For this example, we'll look at integrating with the SampleDataSource.

Integrating your own data into SampleDataSource

While it may be tempting to delete all the sample data loaded in the SampleDataSource constructor, remember that this data is helpful when you're editing the application pages in design mode. A better solution is to include a check for design mode and, if so, load the existing sample data.

You can do this by enclosing the existing code in the SampleDataSource constructor in an if-block that checks if you're in design mode; otherwise we'll call out to a service that loads the data.

```
if (Windows.ApplicationModel.DesignMode.DesignModeEnabled)
{
var group1 = new SampleDataGroup("Group-1",
        "Group Title: 1",
        "Group Subtitle: 1",
        "Assets/DarkGray.png",
        "Group Description: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus tempor scelerisque lorem in vehicula. Aliquam tincidunt, lacus ut sagittis tristique, turpis massa volutpat augue, eu rutrum ligula ante a ante");
group1.Items.Add(new SampleDataItem("Group-1-Item-1",
        "Item Title: 1",
        "Item Subtitle: 1",
        "Assets/LightGray.png",
        "Item Description: 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.",
        ITEM_CONTENT,
        group1));
group1.Items.Add(new SampleDataItem("Group-1-Item-2",
        "Item Title: 2",
        "Item Subtitle: 2",
        "Assets/DarkGray.png",
        "Item Description: 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.",
        ITEM_CONTENT,
        group1));
```

```
group1.Items.Add(new SampleDataItem("Group-1-Item-3",
        "Item Title: 3",
        "Item Subtitle: 3",
        "Assets/MediumGray.png",
        "Item Description: 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.",
        ITEM_CONTENT,
        group1));
group1.Items.Add(new SampleDataItem("Group-1-Item-4",
        "Item Title: 4",
        "Item Subtitle: 4",
        "Assets/DarkGray.png",
        "Item Description: 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.",
        ITEM_CONTENT,
        group1));
group1.Items.Add(new SampleDataItem("Group-1-Item-5",
        "Item Title: 5",
        "Item Subtitle: 5",
        "Assets/MediumGray.png",
        "Item Description: 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.",
        ITEM_CONTENT,
        group1));
this.AllGroups.Add(group1);
  //Rest of original code for this method goes here
}
else
{
  //TODO: Load our non-design data here
}
```

Add the People class

Our first task is to create the classes and services needed to generate the “real” data for the application – that is the data that is shown at run time (rather than at de4sign time). We’ll use a modified version of the Person class defined earlier in this chapter. Notice that we will create the names randomly, but we will assign the city from a parameter passed in to the CreatePeople method.

Add the following People class to the DataModels folder:

```
using System;
```

```
using System.Collections.Generic;

namespace GridApp
{
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }

        private static readonly string[] firstNames = { "Adam", "Bob", "Carl", "David", "Edgar", "Frank", "George", 
"Harry", "Isaac", "Jesse", "Ken", "Larry" };
        private static readonly string[] lastNames = { "Aaronson", "Bobson", "Carlson", "Davidson", "Enstwhile", 
"Ferguson", "Harrison", "Isaacson", "Jackson", "Kennelworth", "Levine" };

        public override string ToString()
        {
            return string.Format( "{0} {1} ({2})", FirstName, LastName, City );
        }

        public static IEnumerable<Person> CreatePeople( int count, string city )
        {
            var people = new List<Person>();

            var r = new Random();

            for ( int i = 0; i < count; i++ )
            {
                var p = new Person()
                {
                    FirstName = firstNames[r.Next( firstNames.Length )],
                    LastName = lastNames[r.Next( lastNames.Length )],
                    City = city
                };
                people.Add( p );
            }
            return people;
        }
    }
}
```

Creating the PeopleService

Normally, we will go out to a database or a web service to retrieve the collection of people to display. We will mimic that by creating local class with a "GetTimes" method that will create all the people we need for the application.

Add the following PeopleService class to the DataModels folder:

```
using GridApp.Data;
using System;

namespace GridApp.DataModel
{
    public class PeopleService
    {
        public string[] GetGroups()
        {
            string[] cities = { "Boston", "New York", "LA", "San Francisco", "Phoenix", "San Jose", "Cincinnati",
"Bellevue" };
            return cities;
        }

        internal Data.SampleDataGroup GetItems(string group)
        {
            var r = new Random();
            int i = 0;
            var people = Person.CreatePeople(r.Next(50), group);

            SampleDataGroup dataGroup = new SampleDataGroup("Group-" + group,
                    group,
                    string.Empty,
                    "Assets/DarkGray.png",
                    "Group Description: Some description for " + group + " goes here.");

            foreach (var person in people)
            {
                dataGroup.Items.Add(new SampleDataItem("Group-" + group + "-Item-" + ++i,
                    person.FirstName + " " + person.LastName,
                    "(" + person.City + ")",
                    "Assets/LightGray.png",
                    "Person Description: (none)",
                    "Here's where the extended content for each person goes",
                    dataGroup));
            }
            return dataGroup;
        }
    }
```

```
}
```

Finally, go back to the SampleDataSource constructor and replace the "//TODO: Load sample data here" comment with the following code:

```
this.AllGroups.Clear();
var peopleService = new PeopleService();

string[] groups = peopleService.GetGroups();
foreach(string group in groups)
{
   this.AllGroups.Add(peopleService.GetItems(group));
}
```

This method will now do the following:

- Clear the SampleDataSource groups
- Retrieve the list of groups
- Populate each group

Now, run the app to see that information's shown for each group and item. Add images and you have a very professional looking application.

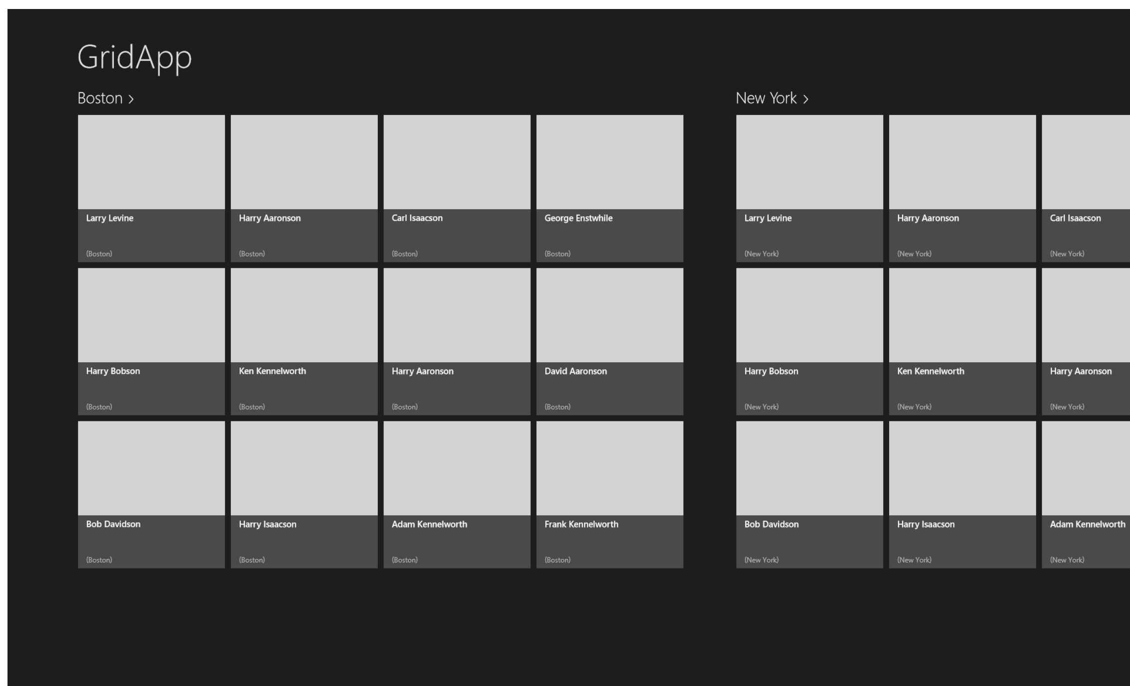


Figure 11

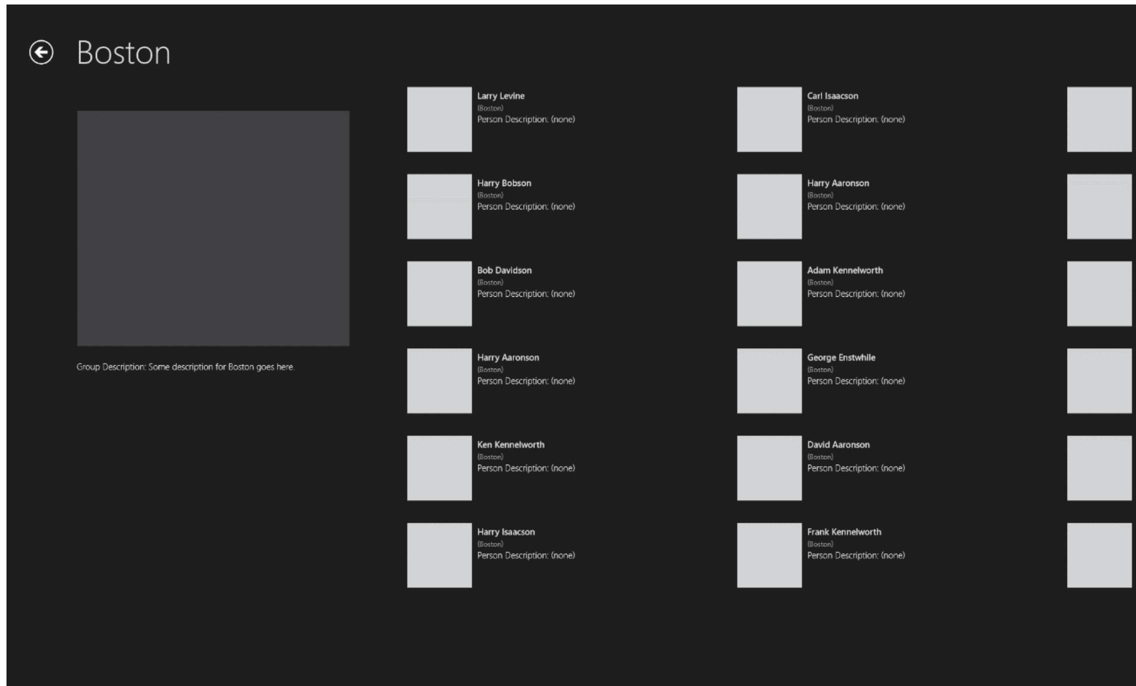


Figure 12

Split App

In addition to the Grid App template, Visual Studio offers the Split App template as shown in figure 13

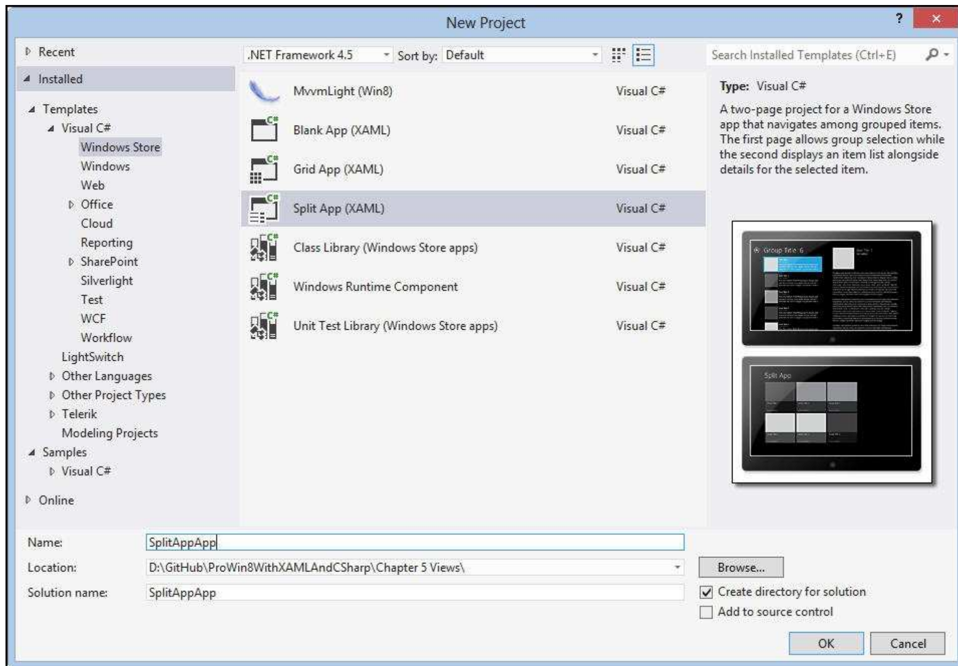


Figure 13 Split App

The Split App is in essence a splified version of the grid application and you can use it in exactly the same way. In fact, to see this, create a new project of type Split App and then follow the identical steps from above for the Grid App. The net effect is that your data is now shown in the SplitApp main page, as shown in figure 14,

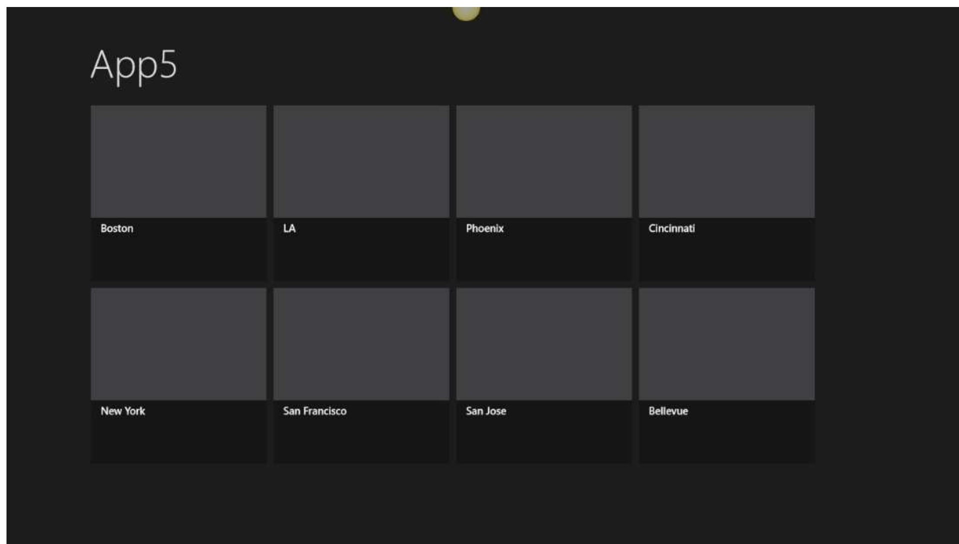


Figure 14 – Split App Groups

Notice that figure 14 shows the Groups. Clicking on a group brings you to the split page, with the list of members of the group on the left and the details about the selected item on the right, as shown in figure 15,

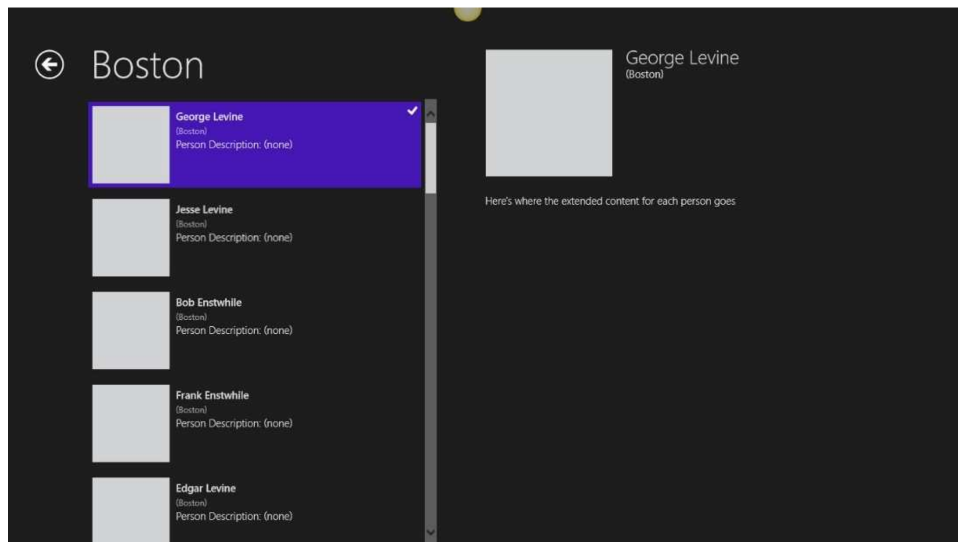


Figure 15 – Split App Details

This book was purchased by arosner@rosnertech.com

CHAPTER 6

n n n

Local Data

What's New in Windows 8.1

Windows 8.1 has removed the Documents Library capability from the app manifest. This is due to security concerns with giving apps access to the user's entire document library. Instead, you must use File Pickers to access files on the user's device. Since File Pickers require the user to specifically open or save files, the app can't access anything that the user hasn't granted access.

SQLite has been updated for Windows 8.1. The version compatible with Windows 8.0 will not work with apps being targeted for Windows 8.1. This means if you have existing apps in the wild that are using SQLite, there isn't an upgrade path for SQLite. You will have to determine the best way to move installed apps from one version to the next.

All significant Windows 8 Store applications manage and store data of some sort. This data can be arbitrarily divided into application data and user data. The former, application data, refers to the state of the application at any given time, including

- What page the user is on
- What form fields have been filled out
- Selections and other choices

User data, on the other hand is data entered by the user specifically to be processed and stored by the application. User data can be stored in a number of formats, and it can be stored in a number of places on your disk or in the cloud.

This chapter will start off by showing how to store application data, and then will explore some of the most popular ways to store user data.

Application Data

While your Windows 8 Store application is running, you want to be saving data all the time. After all, the user can switch to another application at any time, and you only have five seconds to store out your data at that point.

While that is true for major data points, there are certain "status" values that you want to store only at the last minute. This might include which information is visible, what is selected, currently filled in fields, and so forth.

A great way to store that information is in an ApplicationDataContainer. This can be stored with your local settings or your roaming settings, but it requires no special permission from the operating system; these files are considered safe.

Settings Containers

There are three settings containers that can be used for local storage: Local, Roaming, and Temporary. Local is stored on the device and persists across app launches. Roaming is also stored locally but will sync across devices based on the currently logged in user. Temporary is, well, temporary. It can be deleted by the system at anytime.

Saving, Reading, and Deleting Locak Data

To see this, lLet's create an application that stores, readsrestores and deletes application data. Our UI is very simple, a text box to put information into, a textBlock to display recalled information, and three buttons as shown in Figure 1:

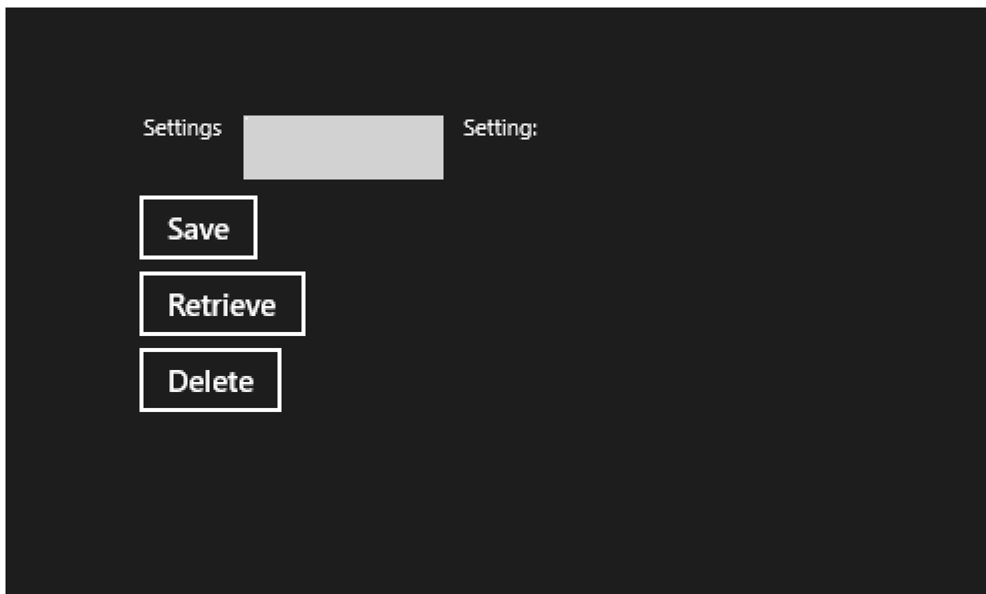


Figure 1 – App for saving, retrieive, and deleting data

Start with a new project based on the BlankApp template and name your app AppSettings. Open MainPage.XAML.cs and add the following declaration to the top of the class:

```
private ApplicationDataContainer settings = ApplicationData.Current.LocalSettings;
```

This creates a local variable that references the Local data storage.

Next, open MainPage.XAML and raplace the default Grid with the following XAML:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
   <Grid.ColumnDefinitions>
      <ColumnDefinition Width="120"/>
      <ColumnDefinition Width="*"/>
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
      <RowDefinition Height="140"/>
      <RowDefinition Height="*"/>
   </Grid.RowDefinitions>
   <StackPanel Grid.Column="1" Grid.Row="1">
      <StackPanel Orientation="Horizontal">
         <TextBlock Text="Settings"
                  Margin="5" />
         <TextBox Width="100"
                  Height="30"
                  Name="Settings"
                  Margin="5"/>
         <TextBlock Text="Setting: " Margin="5"/>
         <TextBlock Name="SettingOutput"
                  Text=""
                  Margin="5"/>
      </StackPanel>
      <Button Name="SaveSettings"
            Content="Save"
            Click="SaveSettings_Click_1" />
      <Button Name="RetrieveSettings"
            Content="Retrieve"
            Click="RetrieveSettings_Click_1" />
      <Button Name="DeleteSettings"
            Content="Delete"
            Click="DeleteSettings_Click_1" />
   </StackPanel>
</Grid>
```

Our UI is very simple, a text box to put information into, a textBlock to display recalled information, and three buttons:

```
<StackPanel Background="{StaticResource ApplicationPageBackgroundThemeBrush}" Margin="100">

 <StackPanel Orientation="Horizontal">
```

```
            <TextBlock Text="Settings"
                       Margin="5" />
            <TextBox Width="100"
                     Height="30"
                     Name="xSettings"
                     Margin="5"/>
            <TextBlock Text="Setting: " Margin="5"/>
            <TextBlock Name="xSettingOutput"
                       Text=""
                       Margin="5"/>
        </StackPanel>
        <Button Name="SaveSettings"
                Content="Save"
                Click="SaveSettings_Click_1" />
        <Button Name="RetrieveSettings"
                Content="Retrieve"
                Click="RetrieveSettings_Click_1" />
        <Button Name="DeleteSettings"
                Content="Delete"
                Click="DeleteSettings_Click_1" />

    </StackPanel>
```

The event handlers are fairly simple. When you click Save we obtain the content from the TextBox and write it into the settings collection, and then we blank the text box:

```
string userValue = txtSettings.Text;
settings.Values["UserSetting"] = userValue;
txtSettings.Text = String.Empty;
```

```
private void SaveSettings_Click_1( object sender, RoutedEventArgs e )
{
    string userValue = Settings.Text;
    settings.Values["UserSetting"] = userValue;
    Settings.Text = String.Empty;

}
```

If “UserSetting” already exists in the dictionary, the current value will be overwritten. If it doesn’t exist, it will be created.

Retrieving the text is equally simple, we pull the text out of the settings collection as an object, and if it is not null we turn it into a string. If the setting does not exist, it will come back as null.,

```
private void RetrieveSettings_Click_1( object sender, RoutedEventArgs e )
{
    object val = settings.Values["UserSetting"];
    if ( val != null )
    {
        SettingOutput.Text = val.ToString();
    }

}
```

```
object val = settings.Values["UserSetting"];
if (val != null)
{
    txtSettingOutput.Text = val.ToString();
}
```

Finally, deleting a setting just requires a call to the Remove method. We also clear out the output text blox.

```
private void DeleteSettings_Click_1( object sender, RoutedEventArgs e )
```

```
{
   settings.Values.Remove( "UserSetting" );
   SettingOutput.Text = String.Empty;
}
```

```
settings.Values.Remove("UserSetting");
txtSettingOutput.Text = String.Empty;
```

That's all it takes to manage your application state. Piece of cake. Easy as pie. Except when you spell the setting name wrong in one of the methods, and nothing works as expected. A much better practice is to define a class level variable that will ensure consistency across event handlers:

```
string settingName = "UserSetting";
```

The complete code is listed here:

```
private void SaveSettings_Click_1(object sender, RoutedEventArgs e)
{
   string userValue = txtSettings.Text;
   settings.Values[settingName] = userValue;
   txtSettings.Text = String.Empty;
}

private void RetrieveSettings_Click_1(object sender, RoutedEventArgs e)
{
   object val = settings.Values[settingName];
   if (val != null)
   {
      txtSettingOutput.Text = val.ToString();
   }
}

private void DeleteSettings_Click_1(object sender, RoutedEventArgs e)
{
   settings.Values.Remove(settingName);
   txtSettingOutput.Text = String.Empty;

}
```

Creating the DataModel, ViewModel, and Repository Before we move into additional methods for dealing with local data, I am going to create an object model and repository that I will use in subsequent examples.

To begin, create a new Windows 8 Store Application using the blank application template, and call it LocalFolderSample.

Create the Model

Add a folder named DataModel and add a Customer class,

```
public class Customer
{
   public int Id { get; set; }
   public string Email { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Title { get; set; }
}
```

Notice that this is a POCO (Plain Old CLR Object) class, nothing special about it.

Create the Repository

Next, we want to build a file repository, but to do that we'll start by defining a DataRepository interface. Create a new file, IDataRepository,

```
public interface IDataRepository
{
   Task Add(Customer customer);
   Task<ObservableCollection<Customer>> Load();
   Task Remove(Customer customer);
   Task Update(Customer customer);
}
```

There's a lot going on in that Interface, and it merits some discussion before we move on. The ObservableCollection class has been around since WPF, and it is a class that implements INotifyCollectionChanged and INotifyPropertyChanged. This notifies the UI when items are added or deleted to the collection, or if any other changes are made to the collection class.

All of the methods have been defined of type Task to enable asynchronous operations. More on this later.

Creating the ViewModel

The final file in the DataModel folder is ViewModel.cs. This will act as the data context for the view.

It begins by declaring a member variable of type IDataRepository,

```
IDataRepository _data;
```

The constructor takes an IDataRepository and initializes the local variable,

```
public ViewModel(IDataRepository data)
{
    _data = data;
}
```

Since this is a view model we want to make sure we implement INotifyPropertyChanged. The listing below shows where the class is currently with the repository being injected into the class via Construct Injection and the implementation of RaisePropertyChanged:

```
public class ViewModel:INotifyPropertyChanged
{
    IDataRepository _data;

    public ViewModel(IDataRepository data)
    {
        _data = data;
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void RaisePropertyChanged([CallerMemberName] string caller = "")
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(caller));
        }
    }


}
```

There are two public properties in the ViewModel: SelectedItem and Customers

```
private Customer _selectedItem;
public Customer SelectedItem
{
   get
   {
      return this._selectedItem;
   }
   set
   {
      if (value != _selectedItem)
      {
         _selectedItem = value;
         RaisePropertyChanged();
      }
   }
}

private ObservableCollection<Customer> _customers;
public ObservableCollection<Customer> Customers
{
   get
   {
      return _customers;
   }
   set
   {
      _customers = value;
      RaisePropertyChanged();
   }
}
```

In the view model initialize, we tell the repository to load its data,

```
public async void Initialize()
{
   Customers = await _data.Load();
}
```

With these, we are ready to implement the CRUD operations, delegating the work to the repository,

```
internal void AddCustomer(Customer cust)
{
   _data.Add(cust);
   RaisePropertyChanged();
```

```
}

internal void DeleteCustomer(Customer cust)
{
   _data.Remove(cust);
   RaisePropertyChanged();
}
```

That's it for the ViewModel. By using a repository, we keep the VM simple, clean and reusable with different storage approaches.

Local Data

Local Data files include any data in your app, including data that the user entered into your program that and should be you want to persisted. One option is to persist this data in the local or roaming settings as discussed in the previous section. This can be less than ideal, especially with large data.

Another option is to store the data in a file in the AppData folder under the signed-in user's name. This is not the same as using LocalSettings. The LocalSettings is a dictionary object that is stored alondside the app's data on disk. Local Data refers to data of any shape, size, and/or format that is stored locally on disk.

Local Data Containers

Just like the settings containers, there are three that can be used for local storage: Local, Roaming, and Temporary. Local is stored on the device and persists across app launches. Roaming is also stored locally but will sync across devices based on the currently logged in user. Temporary is, well, temporary. It can be deleted by the system at anytime.

The actual storage of the file is pretty simple, but we're going to build out a fully reusable Repository model so that we can reapply it to other storage approaches later. We begin with the data file itself. To keep things simple, I'll stay with the idea of storing and retrieving customer data.

Creating the FileRepositoryApplication

To begin, create a new Windows 8 Store Application using the blank application template, and call it LocalFolderSample. Add a folder named DataModel and in DataModel add a Customer class,

```
public class Customer
{
   public int Id { get; set; }
   public string Email { get; set; }
```


This book was purchased by arosner@rosnertech.com

```
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Title { get; set; }
}
```

The actual storage of the file is pretty simple, but we're going to build out a fully reusable Repository model so that we can reapply it to other storage approaches later. We begin with the data file itself. To keep things simple, I'll stay with the idea of storing and retrieving customer data.

The file repository implements the IDataRepository that we created above. Add a new class to the DataModel folder named FileRepository that implements IDataRepository. We need add three member variables:

- A StorageFolder representing a local folder under application data
- A string for the file name of our specific storage file
- An observable collection of Customer
- After expanding the required members of the interface and adding the member variables, your code should look like the following:

```
public class FileRepository:IDataRepository
{
    public Task Add(Customer customer)
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }

    public Task<ObservableCollection<Customer>> Load()
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }

    public Task Remove(Customer customer)
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }

    public Task Update(Customer customer)
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }
```

```
}
```

Notice that this is a POCO (Plain Old CLR Object) class, nothing special about it. Next, we want to build a file repository, but to do that we'll start by defining a DataRepository interface. Create a new file, IDataRepository,

```
public interface IDataRepository
{
   Task Add(Customer customer);
   Task<ObservableCollection<Customer>> Load();
   Task Remove(Customer customer);
   Task Update(Customer customer);
}
```

This interface has four methods. The only unusual one is Load, which returns a Task of ObservableCollection of Customer. This is so that Load can be run asynchronously, as we'll see later in this chapter.

With this interface, we can build our implementation, in a file named FileRepository.cs,

```
public class FileRepository : IDataRepository
{
   StorageFolder folder = ApplicationData.Current.LocalFolder;
   string fileName = "customers.json";
   ObservableCollection<Customer> customers;
```

Our FileRepository class implements IDataRepository, and has three member variables:

- A StorageFolder representing a local folder under application data
- A string for the file name of our specific storage file
- An observable collection of Customer

The constructor calls the Initialize method, which in this case does nothing. The initialize method will be more important when we cover SQLite, which we will later in this chapter.

```
public FileRepository()
{
   Initialize();
}

private void Initialize()
{
}
```

Using JSON to Format Data

Before getting into the meat of reading and writing the datafiles, we need to consider the different storage formats available. Local Data files are text files, and as such, all of the data stored will be

simple strings. There was a time that XML was the predominant format for string data. However, XML is heavy with all of the tags, and can be difficult to work with.
The current leader in textual data is JSON – JavaScript Object Notation. Initially championed by the web for it's light weight format and it's ease of conversion between text and an object graph, JSON is winning the format war, and most Windows 8 developers are using JSON for local storage of text.

The leading package for C# developers to work with JSON data is JSON.NET. One way to add JSON.NET into your project is to download the package from http://json.codeplex.com/, install (or unzip based on the download), and reference the correct assemblies. An easier way is to use NuGet to quickly and easily add JSON into your project. See the sidebar for more information.

USING NUGET TO INSTALL JSON.NET

To install Json.net through NuGet, click on Tools –> Library Package Manager –> Package Manager Console. This will open the console window at the bottom of your screen, providing you the Package Manager prompt (PM>). Enter the command to load Json.NET:

```
PM> Install-Package Newtonsoft.Json
```

Remember to put in the hyphen between Install and Package. A few seconds later the package will have been installed.

```
public FileRepository()
 {
    Initialize();
 }

 private void Initialize()
 {
 }
```

The interface demands that we implement four methods. Before we implement the repository methods that we stubbed out earlier, there are two helper methods that we need to create.

```
private Task WriteToFile()
{
   return Task.Run(async () =>
   {
      string JSON = JsonConvert.SerializeObject(customers);
      var file = await OpenFileAsync();
      await FileIO.WriteTextAsync(file, JSON);
   });
}

private async Task<StorageFile> OpenFileAsync()
{
   return await folder.CreateFileAsync(fileName, CreationCollisionOption.OpenIfExists);
```

```
}
```

Notice that WriteToFile converts the Customers collection to JSON, then asynchronously opens the file to write to, and then finally writes to the file, again asynchronously.

To open the file, we add the helper method OpenFileAsync. Notice that when opening the file we handle CreationCollisions by saying that we want to open the file if it already exists.

The Add method adds a customer (passed in as a parameter) to the customers collection and then calls WriteToFile,

```
public Task Add(Customer customer)
 {
    customers.Add(customer);
    return WriteToFile();
 }
```

The Remove method removes a customer (passed in as a parameter) to the customers collection and then calls WriteToFile,

```
public Task Remove(Customer customer)
 {
    customers.Remove(customer);
    return WriteToFile();
 }
```


```
public Task Add(Customer customer)
 {
    customers.Add(customer);
    return WriteToFile();
 }
```


```
//Write to file is a helper method,
private Task WriteToFile()
{
   return Task.Run(async () =>
   {
      string JSON = JsonConvert.SerializeObject(customers);
      var file = await OpenFileAsync();
      await FileIO.WriteTextAsync(file, JSON);
   });
}
```

Notice that WriteToFile converts the Ccustomers collection to JSON and then asynchronously opens the file to write to and then writes to that file, again asynchronously. To open the file, we add the helper method OpenFileAsync,

```
private async Task<StorageFile> OpenFileAsync()
 {
    return await folder.CreateFileAsync(fileName, CreationCollisionOption.OpenIfExists);
```

```
}
```

Notice that when opening the file we handle CreationCollisions by saying that we want to open the file if it already exists.

The second of the four methods we must implement is Remove, which is pretty much the inverse of Add,

```
public Task Remove(Customer customer)
 {
    customers.Remove(customer);
    return WriteToFile();
 }
```

The third interface method is Update. Here we have slightly more work to do: we must find the record we want to update and if it is not null then we remove the old version and save the new,

```
public Task Update(Customer customer)
{
   var oldCustomer = customers.FirstOrDefault(c => c.Id == customer.Id);
   if (oldCustomer == null)
   {
      throw new System.ArgumentException("Customer not found.");
   }
   customers.Remove(oldCustomer);
   customers.Add(customer);
   return WriteToFile();
}
```


```
public Task Update(Customer customer)
{
   var oldCustomer = customers.FirstOrDefault(c => c.Id ==
customer.Id);
   if (oldCustomer == null)
   {
      throw new System.ArgumentException("Customer not found.");
   }
   customers.Remove(oldCustomer);
   customers.Add(customer);
   return WriteToFile();
}
```


Reading and Writing The Entire File

So why remove the old record and add the new record? While JSON is a very efficient storage and transport mechanism for text, it is not a relational database. It is much more efficient to simple remove the old record and

replace it with the new one than to loop through all of the properties and update the record. Since there isn't any concept of an autoincrement id (or any other relational database concept for that matter), we don't loose anything, and gain only speed.

A similar question can be asked as to why save the entire file each time there is a change instead of just updating the individual record. The answer is pretty much the same. The time and effort it would take to replace/add/change individual records compared to writing the entire file on each change makes writing the file each time a much faster and less fragile option.

What if you have a really large text file? Then I would suggest looking at a more database centric solution, such as SQLLite, discussed later in this chapter.

Finally, we come to Load. Here we create our file asynchronously and if it is not null, we read the contents of the file into a string.

```
public async Task<ObservableCollection<Customer>> Load()
{
    var file = await folder.CreateFileAsync(fileName,
CreationCollisionOption.OpenIfExists);

    string fileContents = string.Empty;
    if (file != null)
    {
        fileContents = await FileIO.ReadTextAsync(file);
    }

    IList<Customer> customersFromJSON =
        JsonConvert.DeserializeObject<List<Customer>>(fileContents)
            ?? new List<Customer>();

    customers = new ObservableCollection<Customer>(customersFromJSON);

    return customers;
}
```


```
public async Task<ObservableCollection<Customer>> Load()
 {
    var file = await folder.CreateFileAsync(fileName, CreationCollisionOption.OpenIfExists);

    string fileContents = string.Empty;
    if (file != null)
    {
       fileContents = await FileIO.ReadTextAsync(file);
    }
```

We then dDeserialize the customer from that string of JSON into a IList of customer, and create an ObservableCollection of customer from that IList,

Creating the View

```
    IList<Customer> customersFromJSON =
        JsonConvert.DeserializeObject<List<Customer>>(fileContents)
            ?? new List<Customer>();

    customers = new ObservableCollection<Customer>(customersFromJSON);

    return customers;
}
```

Note that this JSON manipulation requires that we add the JSON.NET library which you can obtain through NuGet or CodePlex. The easiest route is through NuGet as explained in the sidebar.

USING NUGET TO INSTALL JSON.NET

To install Json.net through NuGet, click on Tools –> Library Package Manager –> Package Manager Console. This will open the console window at the bottom of your screen, providing you the Package Manager prompt (PM>). Enter the command to load Json.NET:

```
PM> Install-Package Newtonsoft.Json
```

Remember to put in the hyphen between Install and Package. A few seconds later the package will have been installed.

Creating the ViewModel

The final file in the DataModel folder is ViewModel.cs. This will act as the data context for the view.

It begins by declaring a member variable of type IDataRepository,

```
IDataRepository _data;
```

The constructor takes an IDataRepository and initializes the local variable,

```
public ViewModel(IDataRepository data)
{
   _data = data;
}
```

In the view model initialize, we tell the repository to load its data,

```
async public void Initialize()
{
    Customers = await _data.Load();
}
```

There are two public properties in the VM: SelectedItem and Customers

```
private Customer selectedItem;
public Customer SelectedItem
{
    get { return this.selectedItem; }
    set
    {
        if (value != selectedItem)
        {
            selectedItem = value;
            RaisePropertyChanged();
        }
    }
}

private ObservableCollection<Customer> customers;
public ObservableCollection<Customer> Customers
{
    get { return customers; }
    set
    {
        customers = value;
        RaisePropertyChanged();
    }
}
```

With these, we are ready to implement the CRUD operations, delegating the work to the repository,

```
internal void AddCustomer(Customer cust)
{
    _data.Add(cust);
    RaisePropertyChanged();
}

internal void DeleteCustomer(Customer cust)
{
    _data.Remove(cust);
    RaisePropertyChanged();
}
```

Don’t forget to have the VM implement INotifyPropertyChanged,

```
public event PropertyChangedEventHandler PropertyChanged;
private void RaisePropertyChanged(
   [CallerMemberName] string caller = "")
{
   if (PropertyChanged != null)
   {
      PropertyChanged(this, new PropertyChangedEventArgs(caller));
   }
}
```

That's it for the ViewModel. By using a repository, we keep the VM simple, clean and reusable with different storage approaches.

The View begins with a couple styles,

```
<Page.Resources>
   <Style TargetType="TextBlock">
      <Setter Property="FontSize"
            Value="20" />
      <Setter Property="Margin"
            Value="5" />
      <Setter Property="HorizontalAlignment"
            Value="Right" />
      <Setter Property="Grid.Column"
            Value="0" />
      <Setter Property="Width"
            Value="100" />
      <Setter Property="VerticalAlignment"
            Value="Center" />
   </Style>
   <Style TargetType="TextBox">
      <Setter Property="Margin"
            Value="5" />
      <Setter Property="HorizontalAlignment"
            Value="Left" />
      <Setter Property="Grid.Column"
            Value="1" />
   </Style>
</Page.Resources>
```

It then adds an AppBar for saving data. Note that the way CommandBar and AppBars are created in Windows 8.1 is completely different and (thankfully) much simpler.

```
   <Page.BottomAppBar>
      <CommandBar>
         <CommandBar.SecondaryCommands>
            <AppBarButton x:Name="cmdSave" Label="Save" Click="cmdSave_Click"
Icon="Save"/>
```

```
            <AppBarButton x:Name="cmdDelete" Label="Delete" Click="cmdDelete_Click"
Icon="Delete"/>
        </CommandBar.SecondaryCommands>
    </CommandBar>
</Page.BottomAppBar>
```

```
<Page.BottomAppBar>
  <AppBar x:Name="BottomAppBar1"
      Padding="10,0,10,0"
      AutomationProperties.Name="Bottom App Bar">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="50*" />
        <ColumnDefinition Width="50*" />
      </Grid.ColumnDefinitions>
      <StackPanel x:Name="LeftPanel"
            Orientation="Horizontal"
            Grid.Column="0"
            HorizontalAlignment="Left">
        <Button x:Name="Save"
            Style="{StaticResource SaveAppBarButtonStyle}"
            Tag="Save"
            Click="Save_Click" />
        <Button x:Name="Delete"
            Style="{StaticResource DeleteAppBarButtonStyle}"
            Tag="Delete"
            Click="Delete_Click" />
      </StackPanel>
    </Grid>
  </AppBar>
</Page.BottomAppBar>
```

Next we create a set of stack panels to gather the data and a listview to display the data.,

```
    <StackPanel Margin="150">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Email"
              Margin="5" />
            <TextBox Width="200"
             Height="40"
             Name="Email"
             Margin="5" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="First Name"
              Margin="5" />
            <TextBox Width="200"
             Height="40"
             Name="FirstName"
```


This book was purchased by arosner@rosnertech.com

```
            Margin="5" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="Last Name"
            Margin="5" />
          <TextBox Width="200"
           Height="40"
           Name="LastName"
           Margin="5" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="Title"
            Margin="5" />
          <TextBox Width="200"
           Height="40"
           Name="Title"
           Margin="5" />
        </StackPanel>
      <ScrollViewer>
        <ListView Name="xCustomers"
              ItemsSource="{Binding Customers}"
              SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
              Height="400">
          <ListView.ItemTemplate>
            <DataTemplate>
              <StackPanel>
                <TextBlock Text="{Binding FirstName}" />
                <TextBlock Text="{Binding LastName}" />
                <TextBlock Text="{Binding Title}" />
              </StackPanel>
            </DataTemplate>
          </ListView.ItemTemplate>
        </ListView>
      </ScrollViewer>
 </StackPanel>
```

```
<StackPanel Margin="150">
  <StackPanel Orientation="Horizontal">
    <TextBlock Text="Email"
           Margin="5" />
    <TextBox Width="200"
          Height="40"
          Name="Email"
          Margin="5" />
  </StackPanel>
  <StackPanel Orientation="Horizontal">
    <TextBlock Text="First Name"
```

```
                Margin="5" />
        <TextBox Width="200"
                Height="40"
                Name="FirstName"
                Margin="5" />
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="Last Name"
                Margin="5" />
        <TextBox Width="200"
                Height="40"
                Name="LastName"
                Margin="5" />
    </StackPanel>
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="Title"
                Margin="5" />
        <TextBox Width="200"
                Height="40"
                Name="Title"
                Margin="5" />
    </StackPanel>
```

Finally, we add a ListView to display the customers we had on disk, and now have in memory

```
<ScrollViewer>
    <ListView Name="Customers"
            ItemsSource="{Binding Customers}"
            SelectedItem="{Binding SelectedItem, Mode=TwoWay}"
            Height="400">
        <ListView.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding FirstName}" />
                    <TextBlock Text="{Binding LastName}" />
                    <TextBlock Text="{Binding Title}" />
                </StackPanel>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ScrollViewer>
</StackPanel>
```

Notice the binding both for the ItemsSource and the SelectedItem.

The code behind is straightforward. The first thing we do is instantiate an IDataRepository and declare the viewmodel,

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

```
private IDataRepository data = new FileRepository();
private ViewModel _vm;
```

In the constructor, we create the ViewModel passing in the repository, then call initialize on the VM and finally set the VM as the DataContext for the page,

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

    _vm = new ViewModel(data);
    _vm.Initialize();
    DataContext = _vm;
 }
```

All that is left is to implement the two event handlers

```
private void Save_Click(object sender, RoutedEventArgs e)
 {
    Customer cust = new Customer
    {
       Email = Email.Text,
       FirstName = FirstName.Text,
       LastName = LastName.Text,
       Title = Title.Text
    };
    _vm.AddCustomer(cust);
 }

 private void Delete_Click(object sender, RoutedEventArgs e)
 {
    if (_vm.SelectedItem != null)
    {
       _vm.DeleteCustomer(_vm.SelectedItem);
    }
 }
```

When you run the application, you are presented with the form. Fill in an entry and save it, and it immediately appears in the list box. *Note that the save button is in the app bar*. To access the app bar, either swipe up from the bottom or down from the top of your screen (on a touch device) or right click the form with the mouse.

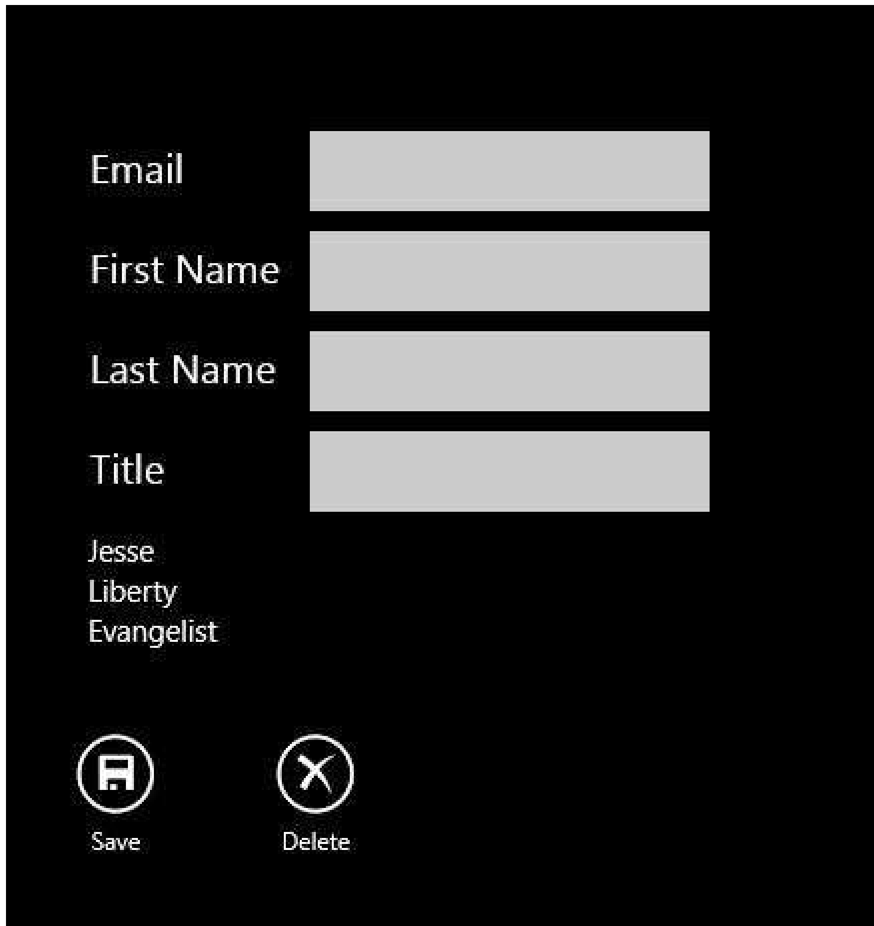


Figure 2 – The User Interface

Once you have saved at least one record, More important, your file, Customers.json, has been saved in application data. You can find it by searching for it under Application Data on your C drive,

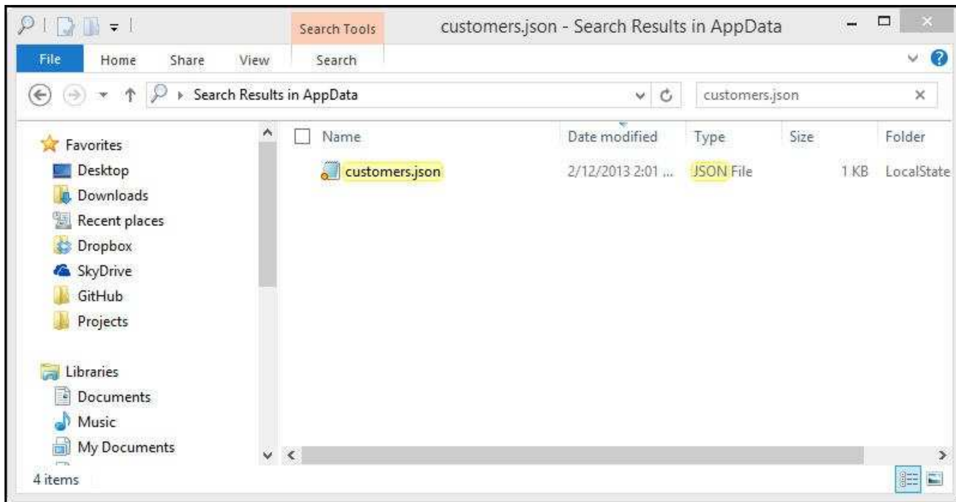


Figure 3 – The data file in AppData

Double click on that file and see the JSON you've saved:

[{"Id":0,"Email":"jesse.liberty@telerik.com","FirstName":"Jesse","LastName":"Liberty","Title":"Evangelist"}]

Roaming

To change from storing your data in local storage to roaming storage, you must change *one line* of code. Back in FileRepository.cs change the LocalFolder to a RoamingFolder,

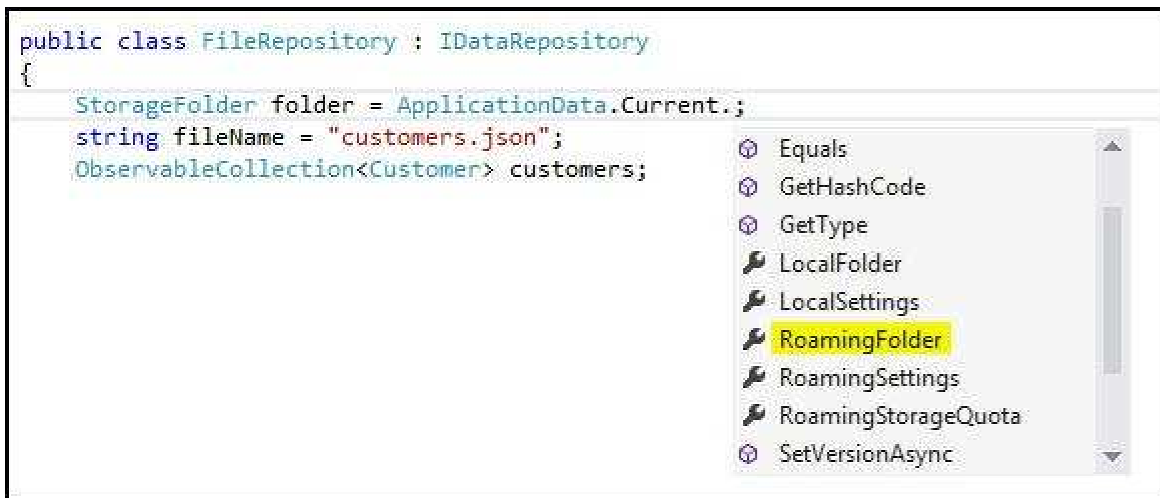


Figure 4 – Available Settings Options

Hey! Presto! Without any further work, your application data is now available on any Windows 8 computer that has the application installed computer and you sign into.

User Specified Known Locations

Local and roaming files are a valid the solution for storing most application data; but sometimes you want to write your data to a more well known public location. You can do so in Winddnows 8.1 eitheehr by having the user explicitly pick a location at run time, or by placing your file in what is called a "known" location such as MyDocuments.

In Windows 8.0, there was a capability in the app manifest that allowed access to the Documents Library. This has been removed in favor of the stronger security model of requiring the user to specifically open and/or save files.

To do this, we have to make some a couple modifications to the program we wrote above. The Interface remains the same, but the implementation of the DataRepository changes a bit, and we have to add another class to wrap of the FilePickers.

Add a new class to the DataModel folder named FileOperations and add three static class lever variables:

```
static ApplicationDataContainer _settings = ApplicationData.Current.LocalSettings;
private static string _mruToken;

    private static string _tokenKey = "mruToken";
```

We need a method to Create the file if it doesn't exist.

```
private static async Task<StorageFile> CreateFile(string fileName)
{
   FileSavePicker savePicker = new FileSavePicker();
   savePicker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
   savePicker.SuggestedFileName = fileName;
   savePicker.FileTypeChoices.Clear();
   savePicker.FileTypeChoices.Add("JSON", new List<string>() { ".json" });
   var file = await savePicker.PickSaveFileAsync();
   return SaveMRU(file);
}
```

Once the user has selected a file, we want to save the file into the MostRecentlyUsedList. This list remembers that the user gave permission to open the file, and therefore the files that are stored in the list are accessible on successive loads.

```
private static StorageFile SaveMRU(StorageFile file)
{
   if (file != null)
   {
      _mruToken = StorageApplicationPermissions.MostRecentlyUsedList.Add(file);
      _settings.Values["mruToken"] = _mruToken;
      return file;
   }
   else
   {
      return null;
   }
}
```

We also use a FileOpenPicker to enable loading a file off the computer in case the MRUToken isn't available:

```
private static async Task<StorageFile> GetFile()
{
   FileOpenPicker openPicker = new FileOpenPicker();
   openPicker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
   openPicker.ViewMode = PickerViewMode.List;

   // Filter to include a sample subset of file types.
   openPicker.FileTypeFilter.Clear();
   openPicker.FileTypeFilter.Add(".json");
   // Open the file picker.
   var file = await openPicker.PickSingleFileAsync();
   return SaveMRU(file);
}
```

If the MRUToken does exist, we can greatly simplify getting the file:

```
private static async Task<StorageFile> GetFileFromMRU()
{
   return
   await StorageApplicationPermissions.MostRecentlyUsedList.GetFileAsync(_mruToken);
}
```

Finally, create the entry method into the FileOperations.

```
public static async Task<StorageFile> OpenFile(string fileName)
{
   _mruToken = (_settings.Values[_tokenKey] != null) ?
      _settings.Values[_tokenKey].ToString() : null;
   if (_mruToken != null)
   {
      return await GetFileFromMRU();
   }
   var file = await GetFile();
   if (file != null)
   {
      return file;
   }
   else
   {
      return await CreateFile(fileName);
   }
}
```

With the FileOperations class built, there are just a few changes to make to the FileRepository class. At the top of the FileRepository class we need to change delete the instantiation of the StorageFolder: from

StorageFolder folder = ApplicationData.Current.RoamingFolder;
Update the OpenFileAsync to the following:

```
private async Task<StorageFile> OpenFileAsync()
{
   return await FileOperations.OpenFile(fileName);
}
```

Finally, update the Load method by deleting the first line:

```
   var file = await folder.CreateFileAsync(fileName,
CreationCollisionOption.OpenIfExists);
```

and replacing is this this line:

```
var file = await FileOperations.OpenFile(fileName);
```

Adding the File Association for JSON files

Open the Package.appxmanifest file and To using the KnownFolders Document library,

```
StorageFolder rootFolder = Windows.Storage.KnownFolders.DocumentsLibrary;
```

Notice that is the rootFolder. We're also going to declare a StorageFolder and a StorageFolder name,

```
StorageFolder folder;
string folderName = "KnownFoldersSample";
```

At the top of the Load method we need to create our new folder,

```
folder = await rootFolder.CreateFolderAsync(folderName, CreationCollisionOption.OpenIfExists);
```

That's it for code changes, but we also have to inform the operating system that we'll be writing to the Documents library and what we'll be writing. Thus, we need to double click on Package.appxmanifest. Once that is open, click on the Capabilities tab,

Check the Documents Library checkbox as shown in figure 1,

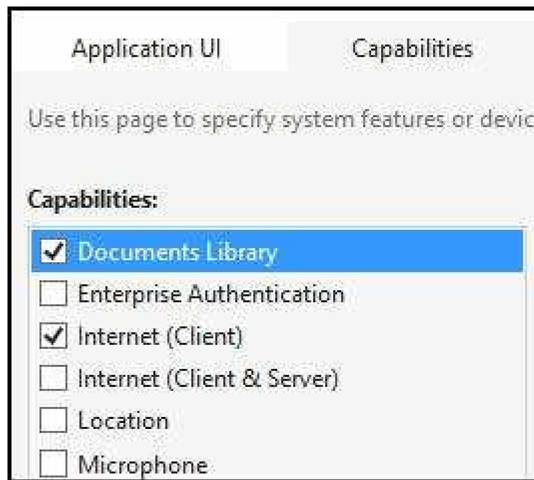


Figure 1 - Capabilities

Next, click on the Declarations tab. Drop down the Available Declarations, and select File Type Associations, and click Add as shown in figure 52,

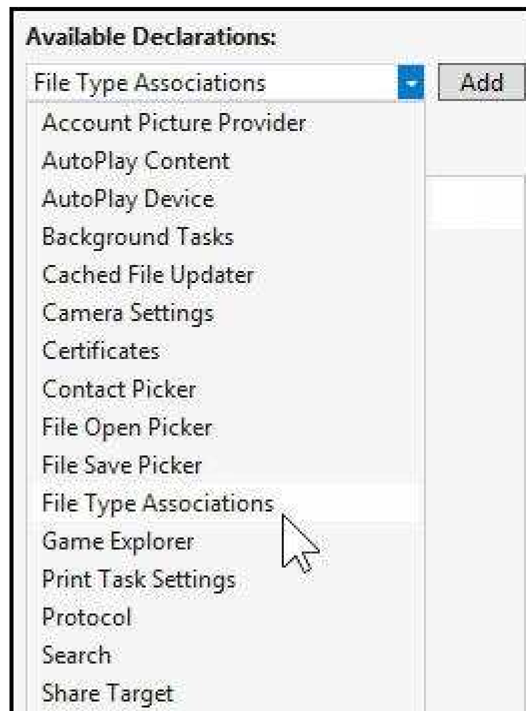


Figure 52 – Declaring File Type Associations

You must also fill in a number of fields in the Declarations tab:

Display name: json files

Name: json

Content type: text/plain

File type: .json

And be sure to check *Open is safe* as shown in figure 36

This book was purchased by arosner@rosnertech.com

Declarations Packaging

properties.

Description:

Registers file type associations, such as .jpeg, on behalf of the app.

Multiple instances of this declaration are allowed in each app.

More information

Properties:

Display name: json files

Logo:

Info tip:

Name: json

Edit flags

☑ Open is safe

☐ Always unsafe

Supported file types

At least one file type must be supported. Enter at least one file type; for example, ".jpg".

Supported file type Remove

Content type: text/plain

File type: .json

Figure 36 – Declarations

After adding in the File Association declaration, your app can load *.json files from File Exporer, as shown in Figure 7.

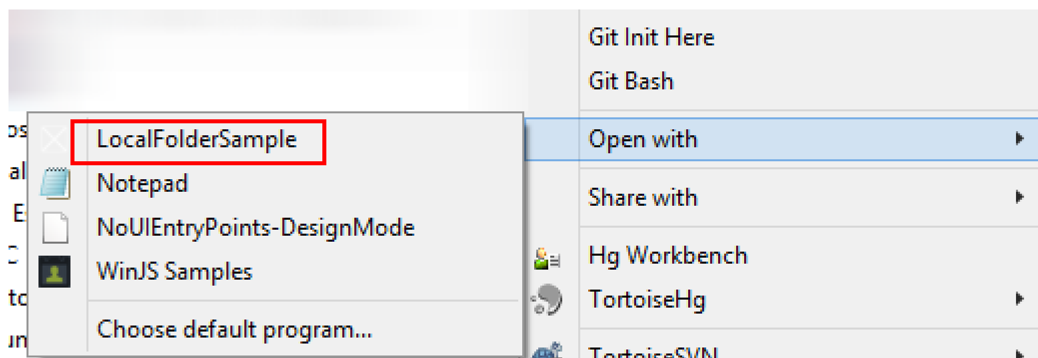


Figure 7 – Open .json files with the app

With this accomplished you can now write .json files to MyDocuments or one of its subdirectories.

SQLite

Interestingly, we can swap in SQLite for our repository and the only file that has to change significantly is the implementation of IDataRepository. , that is SQLiteRepository.cs. The other files (IDataRepository.cs and ViewModel.cs) remain unchanged, and Customer.cs and MainPage.XAML.cs both receives a tiny tweak. To see this in action, create a new project based on the BlankApp template named SQLiteSample.

Before we can start, however, wWe must install SQLite . To do so, you must first install and the SQLite Visual Studio Extension. To do so, click on Tools -> Extensions and Updates. Then click on Online and type *Sqlite for windows runtime* into the search box. When it comes up, as shown in figure 84, click Download,

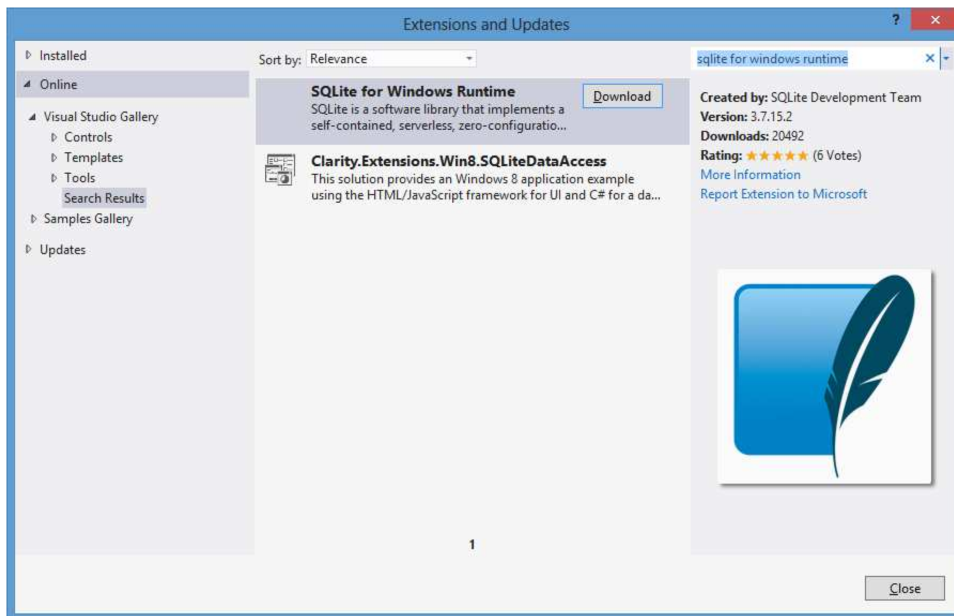


Figure 84 the SQLite extension

As of this writing, the SQLite for Windows Runtime installs the version of SQLite compatible with Windows 8.0. By the time you are reading this, the extension should be updated to Windows 8.1. However, if the extension is still not current, in order to use the extension with Windows 8.1, you must manually install the SQLite VSIX package for Windows 8.1 and VS2013 from the SQLite home page, located here: http://www.sqlite.org/download.html. Once that is installed, remove the references SQLite for Windows Runtime and the Microsoft Visual C++ file, and add references for SQLite for Windows Runtime (Windows 8.1) and Microsoft Visual C++ 2013 Runtime Package for Windows, as in Figure 9.

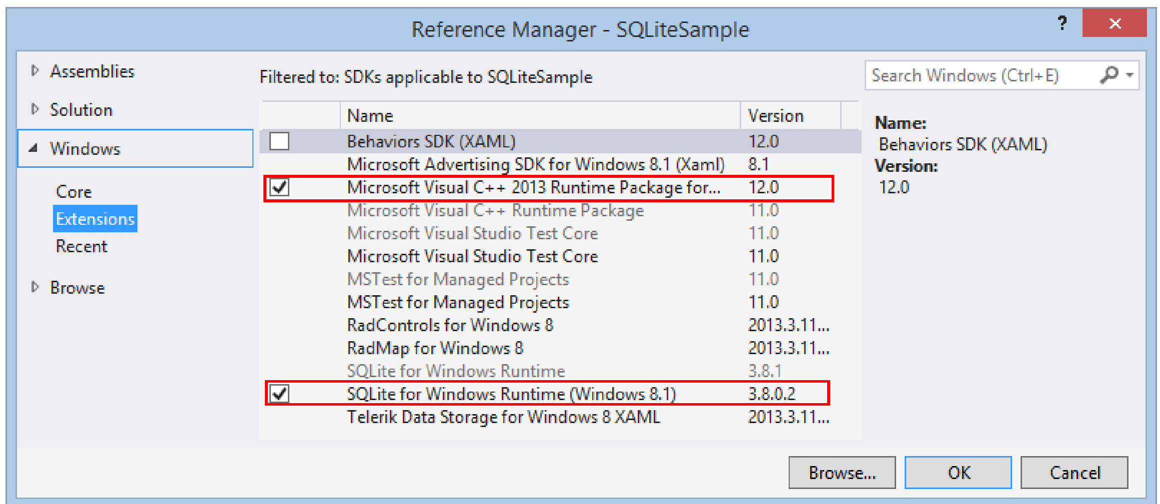


Figure 9 SQLite References for Windows 8.1

Create a new application from the BlankApp template called SQLiteSample and

In your new application (which you create with the Blank template) add the SQLite references, as shown in figure 5,9

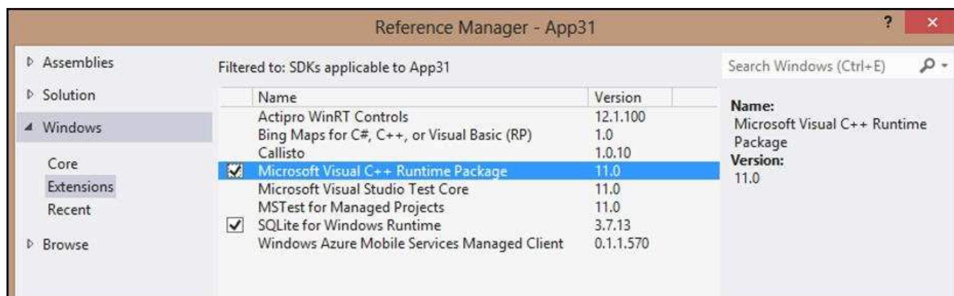


Figure 95 – Reference Manager

Note that your project configuration cannot be set to *AnyCpu*. You must go into the configuration manager by clicking on Build->Configuration Manager and changinge it to x86 or x64 or ARM (whichever you are using). This is due to the different C++ operating system runtime libraries required by SQLite, one for each platform. At the time of this writing, there isn't a mechanism for creating an any CPU install.

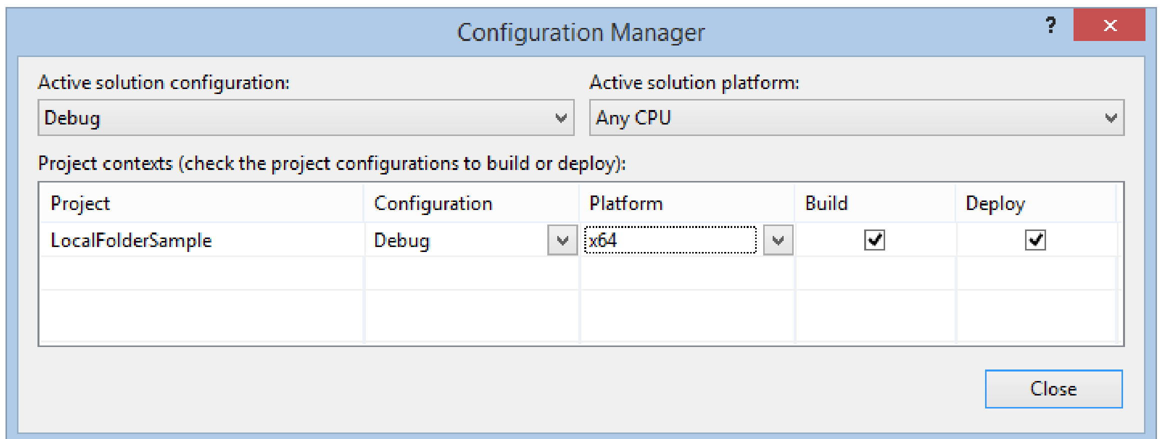


Figure 10 – Changin the target Platform

Finally, you'll want to install the sqlite-net NuGet package. This time, rather than using the console, click on Tools -> Library Package Manager -> Manage NuGet Packages for Solution which brings up the wizard, and from which you can search for and install sqllite, as shown in figure 116,

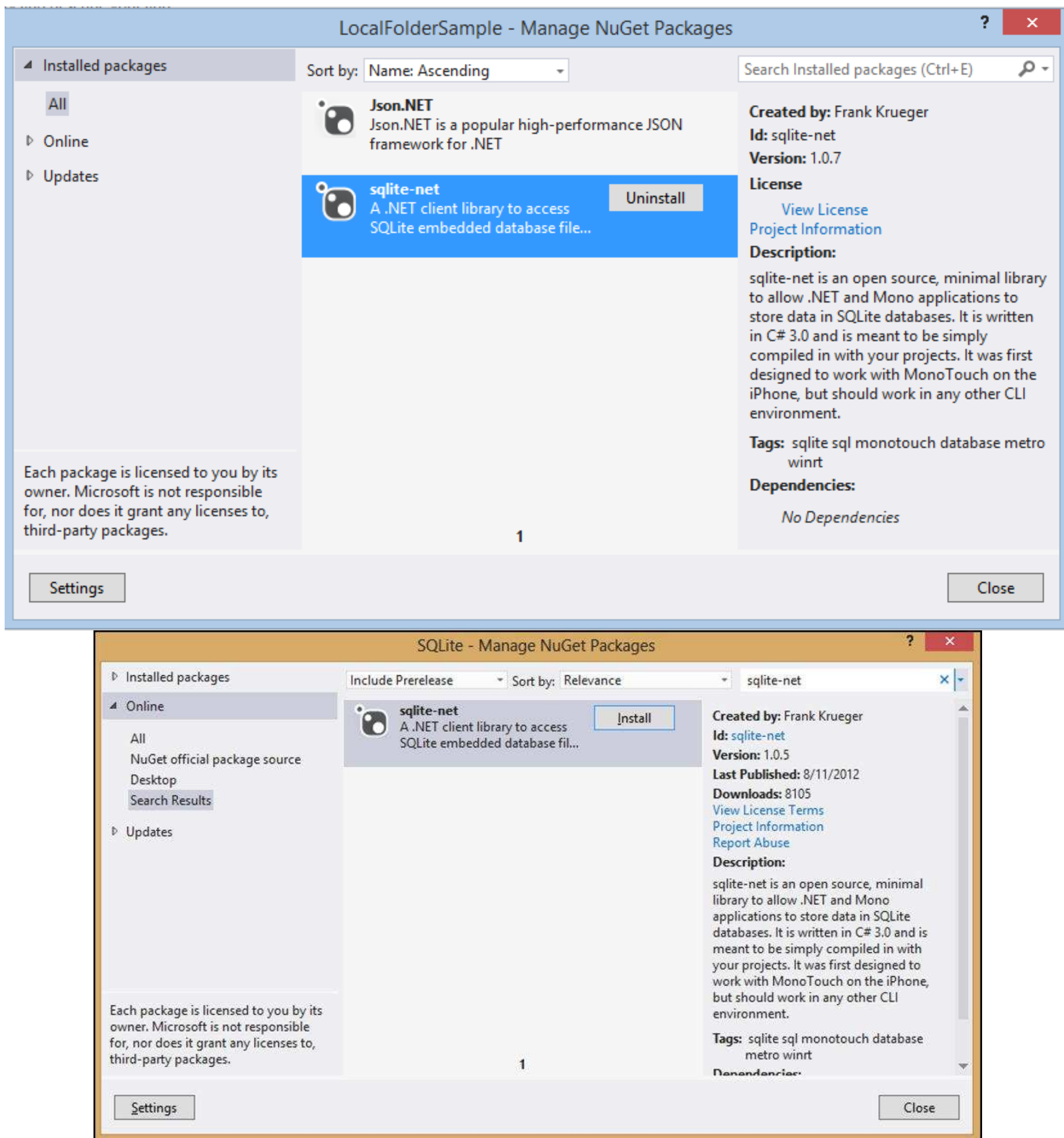


Figure 6 11 -– SQLite-net NuGet Package

Justlike our earlier example, create a folder named DataModel, and add the IDataRepository and ViewModel classes. Also, add the Customer class and Continue with your blank application, but make the one line change to Customer.add the following code to that class. You'll see that this is the same as what we did in the prior example, but there are two new attributes (*PrimaryKey, AutoIncrement*) on the Id fieldcs,

```
public class Customer
```

```
{
   [PrimaryKey, AutoIncrement]
   public int Id { get; set; }
   public string Email { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Title { get; set; }
}
```

Create a new class in the DataModel directory named SQLiteRepository, and implement the public class Customer

```
{
   [PrimaryKey, AutoIncrement]
   public int Id { get; set; }

   public string Email { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Title { get; set; }
}
```

Notice the addition of the attributes *PrimaryKey, AutoIncrement* to the Id field.

As you would expect, SQLiteRepository implments IDataRepository interface. We will be using all of the same methods, so there is minimal change to the rest of our code, but our implementation of the repository , but does so using SQLite and so is somewhat different.

We begin by declaring where we want to store the database,

```
public class SQLiteRepository : IDataRepository
 {
    private static readonly string _dbPath =
      Path.Combine(Windows.Storage.ApplicationData.Current.LocalFolder.Path, "app.SQLite");
```

We also declare an ObservableCollection<Customer>,

```
ObservableCollection<Customer> _customers;
```

Once again the constructor calls Initialize, but this time initialize has real work to do,

```
public SQLiteRepository()
{
   Initialize();
}
```

```
public void Initialize()
{
   using (var db = new SQLite.SQLiteConnection(_dbPath))
   {
      db.CreateTable<Customer>();

      //Note: This is a simplistic initialization scenario
      if (db.ExecuteScalar<int>("select count(1) from Customer") == 0)
      {
         db.RunInTransaction(() =>
         {
         db.Insert(new Customer() { FirstName = "Phil", LastName = "Japikse" });

            db.Insert(new Customer() { FirstName = "Jon", LastName = "Galloway" });
            db.Insert(new Customer() { FirstName = "Jesse", LastName = "Liberty" });
         });
      }
      else
      {
         Load();
      }
   }
}
public SQLiteRepository()
{
   Initialize();
}

public void Initialize()
{
   using (var db = new SQLite.SQLiteConnection(_dbPath))
   {
      db.CreateTable<Customer>();

      //Note: This is a simplistic initialization scenario
      if (db.ExecuteScalar<int>("select count(1) from Customer") == 0)
      {

         db.RunInTransaction(() =>
         {
            db.Insert(new Customer() { FirstName = "Jon", LastName = "Galloway" });
            db.Insert(new Customer() { FirstName = "Jesse", LastName = "Liberty" });
         });
      }
      else
      {
```

```
            Load();
        }
    }
}
```

The first step in Initialize is to “use” the SqlLiteConnection. The using statement, of course, ensures that this resource will be relesased the moment we’re done with it. SQLite is built using COM and other unmanaged resources, so it’s extremely important to make sure all of the resources are disposed of.

We then call dbCreateTable<Customer> which, using Sqlite.net creates our Customer table based on the Customer class.

If we don’t have any customers in the table we insert two three “seed” cstomers to start, otherwise we call Load.

Load’s job is to make a connection to the database and to extract all the customers,

```
public async Task<ObservableCollection<Customer>> Load()
{
    var list = new ObservableCollection<Customer>();
    var connection = new SQLiteAsyncConnection(_dbPath);

    _customers = new ObservableCollection<Customer>(await
connection.QueryAsync<Customer>("select * from Customer"));
    return _customers;
}public async Task<ObservableCollection<Customer>> Load()
{
  var list = new ObservableCollection<Customer>();
  var connection = new SQLiteAsyncConnection(_dbPath);

  customers = new ObservableCollection<Customer>(await   connection.QueryAsync<Customer>("select * from
Customer"));
  return customers;
}
```

Similarly, the Add and Remove methods make a connection to the database and Insert or Delete (respectively) the Customer, while also updating the in-memory customers collection,

```
public Task Add(Customer customer)
{
    _customers.Add(customer);
    var connection = new SQLiteAsyncConnection(_dbPath);
    return connection.InsertAsync(customer);
}
```


q

```
public Task Remove(Customer customer)
{
    _customers.Remove(customer);
    var connection = new SQLiteAsyncConnection(_dbPath);
    return connection.DeleteAsync(customer);
```

```
}public Task Add(Customer customer)
{
   customers.Add(customer);

   var connection = new SQLiteAsyncConnection(_dbPath);
   return connection.InsertAsync(customer);
}

public Task Remove(Customer customer)
{
   customers.Remove(customer);

   var connection = new SQLiteAsyncConnection(_dbPath);
   return connection.DeleteAsync(customer);
}
```

Finally, the Update method, much as it did in the previous cases, finds the original value for the customer and then removes the old value and replaces it with the new value. It then does an update on the customer in the database.

```
public Task Update(Customer customer)
{
    var oldCustomer = _customers.FirstOrDefault(c => c.Id == customer.Id);
    if (oldCustomer == null)
    {
        throw new System.ArgumentException("Customer not found.");
    }
    _customers.Remove(oldCustomer);
    _customers.Add(customer);
    var connection = new SQLiteAsyncConnection(_dbPath);
    return connection.UpdateAsync(customer);
}
```

```
public Task Update(Customer customer)
 {
    var oldCustomer = customers.FirstOrDefault(c => c.Id == customer.Id);

    if (oldCustomer == null)
    {
       throw new System.ArgumentException("Customer not found.");
    }
```

```
    customers.Remove(oldCustomer);
    customers.Add(customer);

    var connection = new SQLiteAsyncConnection(_dbPath);
    return connection.UpdateAsync(customer);
}
```

Copy the MainPage.XAML and MainPage.XAML.cs from the previous example and overwrite the file in this project. We just need to update one line of code where we instantiate the IDataRepository. The update line should read like this:

```
private IDataRepository data = new SQLiteRepository();
```

You can see that we were able to carry the repository pattern that we used in the first example, all the way forward to the SQLite example, successfully and saving us a lot of re-thinking and re-design. You have a number of choices for where and how you store user data, but none of them is terribly difficult to implement.

Summary

There are a variety of options to choose from when your app needs to store data locally on the device. Simple data can be stored in application settings, or in files on the device. More complex data can be stored in SQLite. One noticeably absent item from the list is SQL Server. SQL Server doesn't run on ARM devices or under WinRT.

If your app needs to access SQL Server, it must be done through a service. In the next chapter, we will look into accessing remote data.

This book was purchased by arosner@rosnertech.com

CHAPTER 7

n n n

Remote Data and Services

Who cares?

This is a book on Windows 8, we are building client-side applications. Why do we care about remote data and services? Experience says that any non-trivial application will deal with data, and that data is most useful when it can be shared outside of the local application.

Retrieving data from or saving data to a remote service fulfills one of the key aspects of Microsoft's recommended data goals for Windows 8 applications: *Be Connected*. There is good reason for this directive, a connected application is far more valuable to the end user; the application has now broken the boundaries of the box it is running on and is connected to the terrabytes of information available world wide.

In addition, once you can exchange data with a server you can share data between devices all running the same application. The server acts as a central switch, taking in and dispensing data to various instances of your application.

The ability to share data among instances of your application can be extended to the ability to share data among users, in a controlled and secure manner. This includes everything from games (Words With Friends) to banking (transfer money to my daughter's account).

With remote data you also have the opportunity to back up local data to the cloud and cloud storage can be far more secure and reliable than local storage.

Finally, with remote data you can access standard remote services or data repositories such as Twitter, FaceBook and other applications.

ASP.NET Web API

ASP.NET Web API is a framework for building HTTP services which reach a variety of clients, from browsers to mobile and desktop applications. ASP.NET Web API has been designed to work well with RESTful (where REST is an acronym for REpresentational State Transfer) design principles.)

What this means is that ASP.NET Web API has been designed to make it very easy to write services that work well with the HTTP protocol and closely adhere to current industry standards and best practices.

If you've used Windows Communication Foundation (WCF) in the past, you can think of ASP.NET Web API as a streamlined version of WCF that's easier to use (both as a service provider and consumer) because it's optimized for the most common scenarios – communicating using XML or JSON over HTTP. That doesn't mean it's dumbed down, though. It's been very carefully designed to give you easy access to extend and customize it if you need to; you just shouldn't need to very often.

REST

REST, short for REpresentational State Transfer, is an architectural style which was first described in Roy Fielding's doctoral dissertation. Roy was one of the principle authors of the HTTP specification.

Roy's dissertation covers quite a bit, and there are numerous books about REST written by REST fanatics who fall asleep every night thinking deep RESTful thoughts. REST doesn't easily fit in a nutshell.

In the context of web services, though, REST focuses on a few key values:

- Service calls map to resources (nouns) like users, products and orders.
- HTTP verbs (e.g. GET, POST, PUT, DELETE) describe the actions you're taking on these resources.
- Resources can be represented in different formats (XML, JSON, CSV, text, image, etc.). Rather than have different service endpoints for each format type, the same resource is requested with an HTTP accept header which states the requested response type.
- Interactions are stateless – no state is maintained between successive calls. Each interaction is self-descriptive.

- URL's are important. The URL for a request should look like a directory (e.g. /products/electronic/5) rather than a server-side technology mouthing off (products.aspx?catid=204&productid=5).

- RESTful services should be navigable as hypermedia. For instance, a category would have links to products, which would have links to services that can be performed against the products.

These are some big concepts to really master. Fortunately, you've got two things on your side:

First, ASP.NET Web API guides you into creating services that follow a RESTful style. When you create a new API Controller (which will define the service URLs and how they'll map to your application code), it will be pre-populated with methods that follow good patterns – method names match HTTP verbs, clean URLs are generated, etc.

Secondly, you should view REST as one of many respected and time proven patterns in software development. It's recommended because it's been found to work pretty well, and if you follow it other software developers will have an easier time understanding your code. But like any software development pattern, it exists to serve you. The only real test your code needs to meet is in the user experience they provide to your users.

Building a service with ASP.NET Web API

One of the best and most effective ways to see how to interact with a web service is to build one and to build a Windows 8 client to interact with it. We will do so in this part of the chapter, using ASP.NET MVC 4/ Web API.

To get started, create a new application and in the templates select ASP.NET MVC4 from the templates, as shown in figure 1,

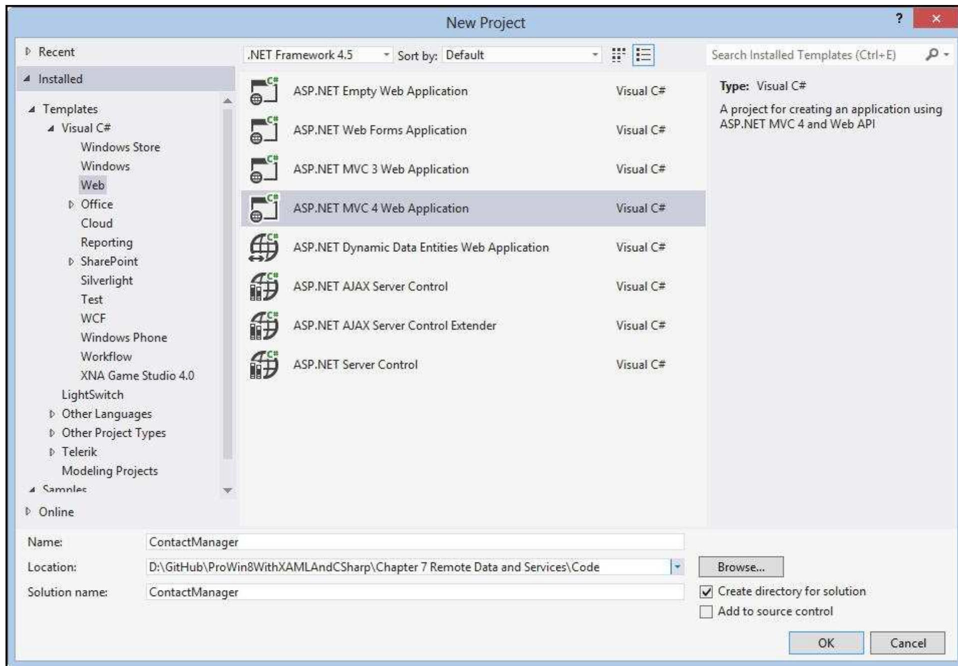


Figure 1 – New MVC Application

After clicking OK, select the Web API template, as shown in Figure 2.

Note: ASP.NET MVC and ASP.NET Web API are closely related, so you can use ASP.NET Web API with any of the ASP.NET MVC project templates. We'll use the Web API template because it adds some useful things like API help generation, but there's little difference between the MVC templates when it comes to ASP.NET Web API.

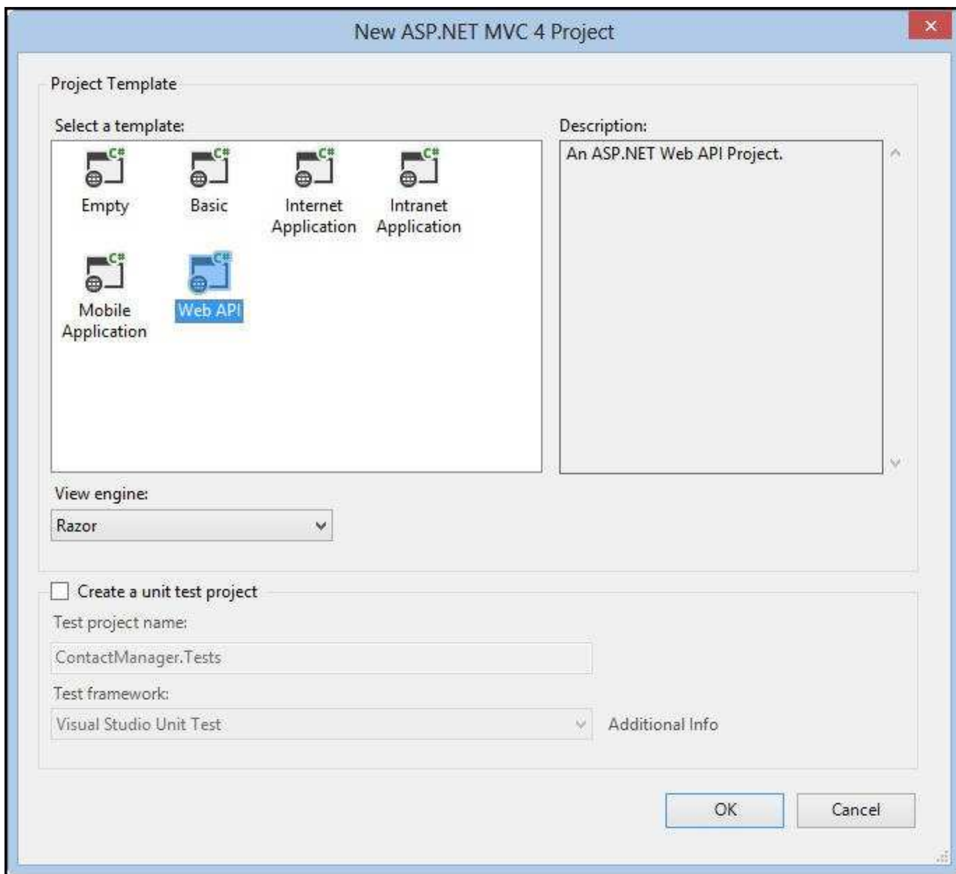


Figure 2 – Web API Template

Convention over Configuration

- Both MVC and EF CF make use of convention over config
- Putting files in the provided folders
- Naming files, properties and classes appropriately

Creating The Model

The first step is to create our Contact class. We're building a simple contact manager, and the Contact class will be the fundamental data that we'll use on both the server and the client.

Right click on the Models folder that was created for you and add a Contact.cs class as shown in Example 1,

---

Note: This example is loosely based on the Microsoft Contact Manager sample for ASP.NET Web API, which shows web, Windows Phone and a more complex Windows 8 client sample. It's available here: http://librt.me/ContactManagerSample

---

Example 1 – Contact Class

```
public class Contact
{
    [ScaffoldColumn(false)]
    public int ContactId { get; set; }
    [Required]
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    public string Email { get; set; }
    public string Twitter { get; set; }
}
```

You will see a red squiggly line under ScaffoldColumn and Required. As usual, pressing control-dot will bring up Intellisense and you can thus add the following using statement:

```
using System.ComponentModel.DataAnnotations;
```

There are two data annotation attributes used in this class. The first, *ScaffoldColumn*, indicates that the ContactId is controlled by the application and should not be set externally. The *Required* attribute indicates that, at a minimum, a Contact must have a name.

We are ready to create a Controller, but to do so we must first Build the application so that the Contact Model class will be available.

Creating the Controller

Right-click on the Controllers folder and select Add / Controller. Change the name of the controller to ContactsController. Drop down the Template list and select

API controller with read/write actions, using Entity Framework.

Drop down the Model class list. If it is empty, you neglected to build the application, so cancel this dialog and build. From the drop down, select *Contact (ContactManager.Models)*.

Drop down the Data context class list and select *New Data Context*. Name the new context

ContactManagerContext, as shown in figure 3,

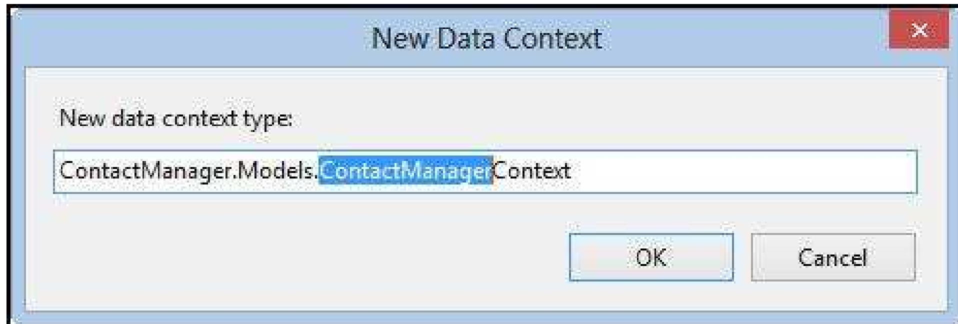


Figure 3 – New Data Context

Your Add Controller dialog should now be filled in as shown in figure 4. Click the Add button.

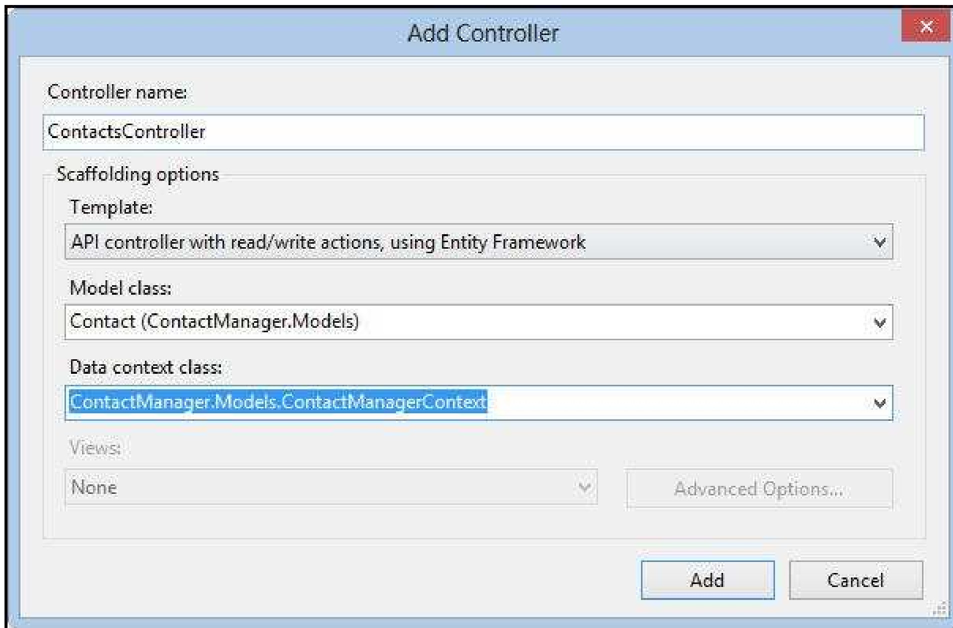


Figure 4 - Add Controller

A new ContactsController class is created. It inherits from ApiController and includes some working default code with the following methods:

GetContacts

This will return all contacts

GetContact(int id)

This will find the contact with the matching ID and return it. If none is found, it will return an HTTP 404 (not found) response – the same HTTP response your browser would encounter if you browsed to a non-existant web URL.

PutContact(int id, Contact contact)

This allows you to replace an existing contact by ID.

PostContact(Contact contact)

This allows you to create a new contact.

DeleteContact(int id)

This deletes a contact by id.

Conventions in ASP.NET Web API

ASP.NET Web API (like ASP.NET MVC) uses conventions to simplify code while conforming to HTTP standards. As you can see, all of the controller action methods are named using HTTP verbs (GET, PUT, POST, DELETE), and ASP.NET Web API will automatically route HTTP requests to the appropriate method based on name without any additional configuration, web.config, annotations, etc. on your part. Thus an HTTP GET request to /api/contacts will automatically route to the ContactsController's GetContacts method, simply by following naming conventions.

Entity Framework and Sample Data

At this point we have a functioning web service which uses Entity Framework to handle data persistence! We're using Entity Framework Code First, which lets us model our database using standard .NET classes (our Contact class above). It will automatically create our database – complete with tables based on our entity classes – when the service is first accessed.

EFCF uses a lot of conventions, for more detail than is provided here, see http://librt.me/EFCFGetStarted

Entity Framework also provides a method for inserting sample or initial data into our database when it's created, known as a *database initializer*. We'll make use of that here.

Creating the Database Initializer

Right click on the Models folder and create a new class named *ContactManagerDatabaseInitializer*.

This class has one simple purpose – provide data when the database is initialized. Here's the code for that class:

```
public class ContactManagerDatabaseInitializer :
   DropCreateDatabaseIfModelChanges<ContactManagerContext>
{
   protected override void Seed( ContactManagerContext context )
   {
      base.Seed( context );

      context.Contacts.Add(
         new Contact
         {
            Name = "Jon Galloway",
            Email = "jongalloway@gmail.com",
            Twitter = "jongalloway",
            City = "San Diego",
            State = "CA"
         } );

      context.Contacts.Add(
         new Contact
         {
            Name = "Jesse Liberty",
            Email = "jesseliberty@gmail.com",
            Twitter = "jesseliberty",
            City = "Acton",
            State = "MA"
         } );
   }
}
```


This book was purchased by arosner@rosnertech.com

Let’s look at this in a bit of detail. The first thing to notice is the base class, DropCreateDatabaseIfModelChanges<ContactManagerContext>. This is one of two standard base classes, the other is DropCreateDatabaseAlways.

The ContactManagerContext was created for you and is used when overriding the Seed method (as we do above). When you register the initializer (see below) the initializer will be called by Entity Framework when it creates the database, and it will call your overridden seed method.

The context Contacts collection that you are adding to above is of type DbSet<Contact> which was created for you when you selected to create a new Data Context, as shown above in figure 4. You can see that collection in ContactManagerContext.cs (within the Models folder),

```
public DbSet<Contact> Contacts { get; set; }
```

Registering the Database Initializer

We need to register this database initializer in Global.asax.cs. We do so in the Application_Start method.

```
Database.SetInitializer(new ContactManagerDatabaseInitializer());
```

You'll need to add using statements to reference the System.Entity and your ContactManagerDatabaseInitializer's namespaces, as shown in figure 5,

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register( GlobalConfiguration.Configuration );
    FilterConfig.RegisterGlobalFilters( GlobalFilters.Filters );
    RouteConfig.RegisterRoutes( RouteTable.Routes );
    BundleConfig.RegisterBundles( BundleTable.Bundles );
    Database.SetInitializer( new ContactManagerDatabaseInitializer() );
}
                      using ContactManager.Models;
                      ContactManager.Models.ContactManagerDatabaseInitializer
                      Generate class for 'ContactManagerDatabaseInitializer'
                      Generate new type...
```

Figure 5 - Application Start

With that, our server is ready to provide data to our Windows 8 client. We won't deploy the server (that is another chapter in itself) but we'll run it locally as you'll see shortly.

Accessing the ASP.NET Web API service from a Windows 8 Client

We are ready now to create a Windows 8 project that will use the data supplied by the Web API project we've just completed. Right click on the solution and choose Add New Project. Select *Windows Store* in the left column, and *Grid App* in the right column. Name your application ContactManager.WindowsStore as shown in figure 6,

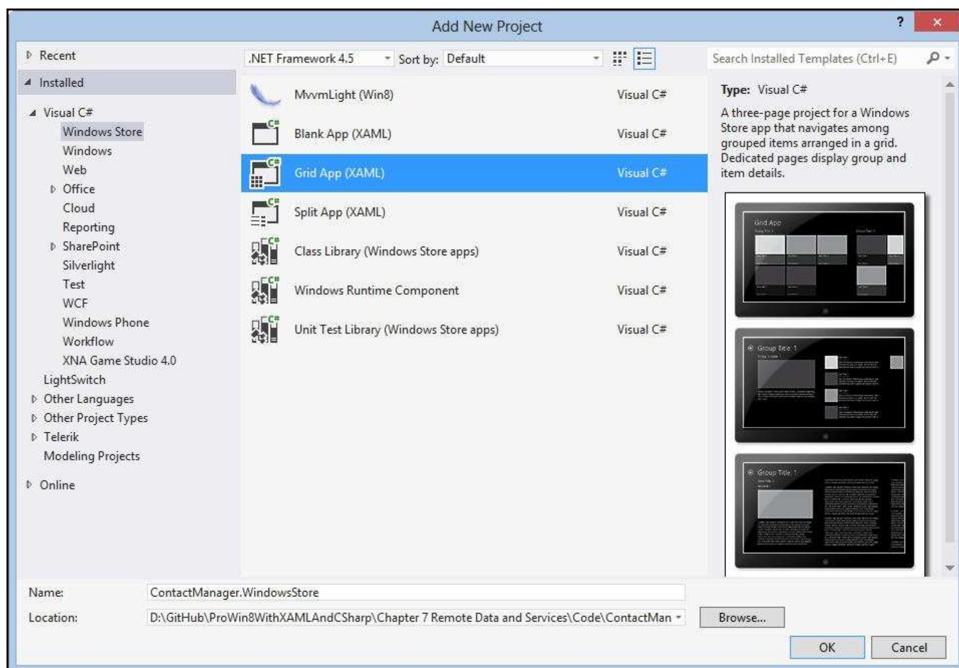


Figure 6 – Grid Project

Adding the WebApi Client package from NuGet

We'll need the WebApi client package to work with the WebAPI application we've built. Fortunately, this can be installed through NuGet. To do this, select View->Other Windows ->Package Manager Console. This window will open at the bottom of your screen. At the Package Manager prompt (PM>) enter the following command:

Install-Package Microsoft.AspNet.WebApi.Client -ProjectName ContactManager.WindowsStore –IncludePrerelease

As shown in figure 7,

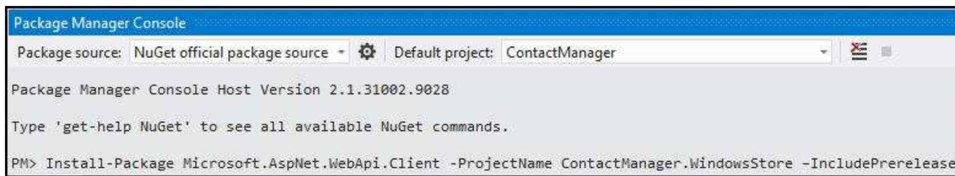


Figure 7 – PM Command

Hit Enter and the command will be executed and the package will be installed into your project. You should see success messages, as shown highlighted in figure 8,

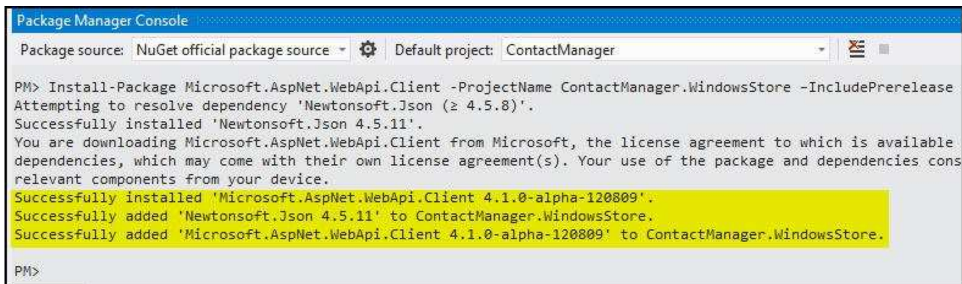


Figure 8 – PM Success

The NuGet installation can be performed via menus and dialogs, but for a walk-through like this it is far simpler to type in the single command.

Adding the Contact Class

We need to share the Contact class in the Server class as well as in the client. At the risk of making experienced developers feel light headed, we're going to copy and paste the class into the client project.

Right click on the DataModel folder and create a new class named Contact. Paste in the Contact class from ContactManager, eliminating the attributes,

```
public class Contact
{
   public int ContactId { get; set; }
   public string Name { get; set; }
   public string Address { get; set; }
   public string City { get; set; }
   public string State { get; set; }
   public string Zip { get; set; }
   public string Email { get; set; }
   public string Twitter { get; set; }
}
```

Editing The SampleDataSource class

Within the ContactManager.WindowsStore project, open the SampleDataSource.cs file in the DataModel folder. Add the following funtions to the end of the SampleDataSource class:

```
public static async Task<SampleDataGroup> LoadDataAsync( bool clear = true )
    {
      // Load this from configuration
      const string serviceUrl = "http://localhost:16854/";

      if ( clear )
      {
         _sampleDataSource.AllGroups.Clear();
      }

      HttpClient client = new HttpClient();

      client.BaseAddress = new Uri( serviceUrl );
      HttpResponseMessage response = await client.GetAsync( "api/contacts" );
      Contact[] contacts = await response.Content.ReadAsAsync<Contact[]>();

      SampleDataGroup contactsGroup = new SampleDataGroup( "ContactsGroup", "My Contacts", null, null, null );

      foreach ( Contact contact in contacts )
      {
         contactsGroup.Items.Add( new SampleDataItem(
            contact.ContactId.ToString(),
            contact.Name,
            GetContactInfo( contact ),
            //new Uri(client.BaseAddress, contact.Self + ".png").AbsoluteUri,
```

```
                new Uri( "http://avatars.io/auto/" + contact.Twitter + "?size=large", UriKind.Absolute ).AbsoluteUri,
                null,
                null,
                contactsGroup ) );
        }
        _sampleDataSource.AllGroups.Add( contactsGroup );

        return contactsGroup;
    }

static string GetContactInfo( Contact contact )
    {
        return string.Format(
            "{0}, {1}\n{2}\n@{3}",
            contact.City ?? "City?",
            contact.State ?? "State?",
            contact.Email ?? "Email?",
            contact.Twitter ?? "Twitter?" );
    }
```

You will need to fix a number of references by adding the required using statements.

The second function, GetContactInfo, just formats some Contact properties into a string for display.

Let's walk through the first function, LoadDataAsync:

```
// Load this from configuration
const string serviceUrl = "http://localhost:16854/";
```

This function will be calling into our service. In the current example application, this will need to match the port number used by the ASP.NET Web API application. To find this, scroll up to the ContactManager project and double click on the Properties as shown in figure 9,

Figure 9 - Clicking on Properties

When the Properties display opens, click on Web (in the left column) to open the Web tab. About half way down the page you'll find the Servers section. Within that, find the Project Url for the Local IIS Web Server and note the Port Number, as shown in figure 10,

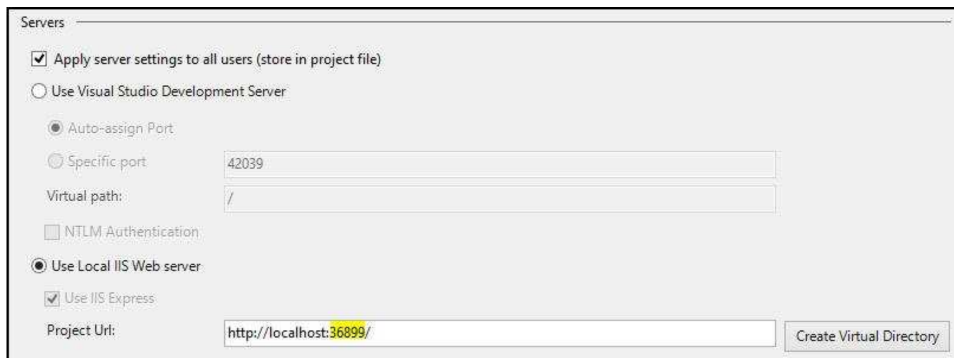


Figure 10 Finding the Port

Change the port number in your code accordingly,

```
public static async Task<SampleDataGroup> LoadDataAsync( bool clear = true )
    {
      // Load this from configuration
      const string serviceUrl = "http://localhost:36899/";
```

We've included a boolean parameter (clear) which is defaulted to true. If true, it will clear the data source before fetching data from the web service. In this case, we do want to have sample data available for design-time support, but we don't want to have the sample data flash on the screen when run by end-users, so we'll leave the default to true.

```
if (clear)
{
    _sampleDataSource.AllGroups.Clear();
}
```

We then instantiate an HttpClient object and set its BaseAddress. We use that to get the contacts asynchronously. They are returned as JSON data that we can then read into a collection of Contact objects.

```
HttpClient client = new HttpClient();

client.BaseAddress = new Uri(serviceUrl);
HttpResponseMessage response = await client.GetAsync("api/contacts");
Contact[] contacts = await response.Content.ReadAsAsync<Contact[]>();
```

Surprisingly, this is all you need to do to call a webservice which returns JSON data. The HttpClient object will make the request, wait asynchronously for the response, and deserialize the result to an array of Contact objects.

```
    SampleDataGroup contactsGroup = new SampleDataGroup("ContactsGroup", "My
Contacts", null, null, null);
```

```
foreach (Contact contact in contacts)
{
    contactsGroup.Items.Add(new SampleDataItem(
        contact.ContactId.ToString(),
        contact.Name,
        GetContactInfo(contact),
        //new Uri(client.BaseAddress, contact.Self + ".png").AbsoluteUri,
        new Uri("http://avatars.io/auto/" + contact.Twitter + "?size=large",
UriKind.Absolute).AbsoluteUri,
        null,
        null,
        contactsGroup));
}
```

```
        _sampleDataSource.AllGroups.Add(contactsGroup);

        return contactsGroup;
```

As discussed in a previous chapter, the SampleDataSource can hold separate groups of data. We're returning all contacts in an AllGroups grouping, but we could split these up if we had a meaningful way to do that – by geographical region, for instance. We'd add those groups to the _sampleDataSource.

We're using another web service for our avatars, avatars.io. This site takes a handle and tries to find the best avatar image from propular social networking sites like Twitter and Facebook.

```
new Uri( "http://avatars.io/auto/" + contact.Twitter + "?size=large", UriKind.Absolute ).AbsoluteUri
```

Calling LoadDataAsync

Finally, we'll need to call the LoadDataAsync method from App.xaml.cs. Add the line

```
SampleDataSource.LoadDataAsync();
```

above the call to Window.Current.Content.RootFrame,

```
protected override async void OnLaunched(LaunchActivatedEventArgs args)
 {
   Frame rootFrame = Window.Current.Content as Frame;

   // Do not repeat app initialization when the Window already has content,
   // just ensure that the window is active

   if (rootFrame == null)
   {
      //…

     SampleDataSource.LoadDataAsync();
     // Place the frame in the current Window
     Window.Current.Content = rootFrame;
   }
```

Running the Client

Right click on the ContactManager.WindowsStore project and choose *Set As Startup Project*. Run the application. You should see the two contacts we created in the sample data displayed in the GridView as shown in figure 11,

Figure 11 Running Application

Deploying a site to Windows Azure Web Services

Of course, your service is of little use sitting on your hard drive. It needs to be deployed to a public web server. As you'd expect, ASP.NET Web API runs great on IIS.

Windows Azure Web Sites is a good hosting option for your web services. You can run ten Windows Azure Web Sites for free, and in most cases the only reasons you'd need to upgrade to a paid account is if you need to support

really high volume or need a custom domain (other than mysitename.azurewebsites.net). Since your users will never see your domain name or URL when used just for backing services, there's a very good chance that you'll never need to upgrade from the free offering.[1]

There's a great end-to-end tutorial on the official Windows Azure site, so rather than repeat it, we'll recommend you follow it here: http://librt.me/RestWebAPI

Common API Formats

Just as web servers and browsers communicate via HTML, APIs use some common media format formats (like XML, JSON) to communicate.

The previous sample didn't even mention a format, because if you're using ASP.NET Web API on the server and the ASP.NET Web API Client (HttpClient) in your Windows Store Application, they'll automatically negotiate the format for you.

XML

XML (eXtensible Markup Language) has been used by services for a while.

```
<album>
  <artist>The Beatles</artist>
  <title>Meet the Beatles!</title>
  <year>1964</year>
</album>
```

It has some benefits:

- As the name implies, it's extensible. New tags can be created to define properties as needed.

- It's human readable and usually pretty self-descriptive.

There's optional support for document validation. An XML document can include a Data Type Definition (DTD) or an XML Schema Definition (XSD) which defines the elements and the types the document is expected to contain.

Because there are common formats for defining documents, there are some common industry standard document types. This is expecially useful in fields like finance, business and healthcare in which businesses interoperate via APIs

[1] Caveat: One of the authors is a Technical Evangelist for Microsoft Azure. We have been careful to recommend Microsoft services in general, and Azure in particular only when we believe it is the very best alternative, and especially when it is free.

This book was purchased by arosner@rosnertech.com

and both sides need to be able to ensure a document is valid. Additionally, common syndication formats like RSS and ATOM have been built on top of XML.

Since XML is pretty tightly structured, there are specifications that allow programs to query, parse, format and otherwise manipulate XML.

There are two main downsides to XML:

- It's verbose. The start and end tags frequently take as many characters as the actual data they contain. That's partly mitigated by the fact that since the tags are very repetitive, XML documents compress pretty well.

- The rigid structure of XML (especially with a schema or definition) is useful, as described above. However, over time some programmers have become tired of all the structure and question the actual real-world value.

JSON

JavaScript Object Notation (JSON) has recently become very popular as an alternative to XML.

```
{
 "artist" : "The Beatles",
 "title" : "Meet the Beatles!",
 "year" : "1964"
}
```

Benefits:

- It's very terse and "lightweight"

- While JSON has outgrown its JavaScript roots, JavaScript Object Notation began life as a lightweight format for serializing simple JavaScript objects. Because of that, it can be easily deserialized into JavaScript objects

- JSON has a simple grammar that isn't concerned with document validity. Those who are tired of XML's structure usually prefer JSON.

Cons:

- Since there are no surrounding tags, JSON isn't as self descriptive as XML

- Since JSON isn't as structured and doesn't address document validity, JSON requires that client code know more about the API structure. This means that API documentation is a manual process, JSON messages can't be automatically validated, etc.

OData

Web API will use JSON or XML. However, if you wish to communicate with a public OData source, you'll need to write your code somewhat differently.

OData (Open Data Protocol) is a data-focused web protocol that layers data-centric activities like querying and updates on top of existing formats like Atom (an XML syndication format mentioned above) and JSON. It was originally developed by Microsoft but has been submitted as an open standard.

OData support wasn't included in the first release of ASP.NET Web API. Since then, a NuGet package has been made available which includes partial support for OData. As OData server support is still evolving, we'll focus on accessing OData API's in a Windows Store Application.

Accessing a Public OData Service using the OData Client Tools

To see OData at work, create a new Windows 8 Blank application and name it OData. Be sure to install *OData Client Tools for Windows Store Apps* from http://www.microsoft.com/en-us/download/details.aspx?id=30714

Add a service reference, as shown in figure 12

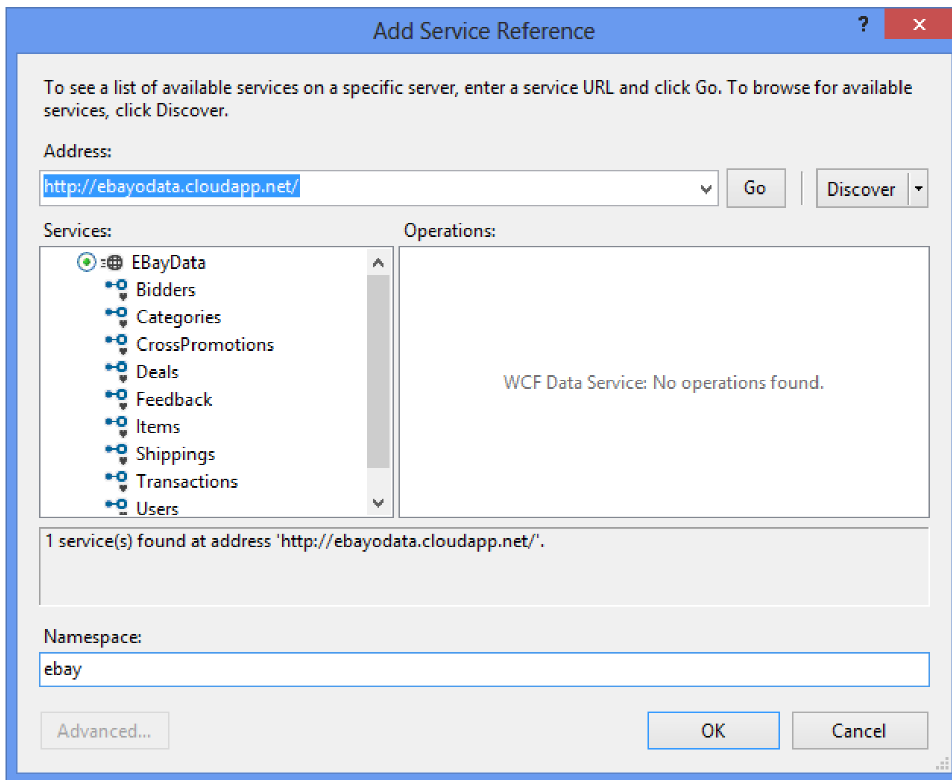


Figure 12, Adding a service reference

In this case we'll be querying against the EBay OData service. Place a GridView onto your MainPage.xaml and bind it to a CollectionViewSource. In tetheridView's ItemTemplate add a StackPanel with an image and set the image's source to bind to GalleryUrl. Also add a TextBlock and bind that to Title,

```
<Page.Resources>
    <CollectionViewSource x:Name="cvs"
                    IsSourceGrouped="False" />
 </Page.Resources>
 <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    <GridView x:Name="myGridView"
          ItemsSource="{Binding Source={StaticResource cvs}}">
       <GridView.ItemTemplate>
          <DataTemplate>
             <StackPanel>
                <Image Stretch="Uniform" Width="150" Height="150">
                   <Image.Source>
```

```
                <BitmapImage UriSource="{Binding GalleryUrl}"/>
              </Image.Source>
            </Image>
            <TextBlock Text="{Binding Title}"/>
          </StackPanel>
        </DataTemplate>
      </GridView.ItemTemplate>
    </GridView>
 </Grid>
```

At the top of the code-behind declare a DataServiceCollection of ebay Items,

```
private DataServiceCollection<OData.ebay.Item> data;
```

In your constructor, initialize a Uri for the ebay Service and use that to create an instance of EBayData (defined by the EBay OData service and generated for you by the tool).

```
public MainPage()
 {
   this.InitializeComponent();
   Uri ebayService = new Uri("http://ebayodata.cloudapp.net");
   EBayData ebayData = new EBayData(ebayService);
```

Initialize the DataServiceCollection with the EBayData,

```
data = new DataServiceCollection<Item>(ebayData);
```

Create an IQueryable of Item, to search for the first 50 furby items,

```
IQueryable<Item> serviceQuery =
   ebayData.Items.AddQueryOption("search", "furby")
   .AddQueryOption("$top", 50);
```

Just to see what can be done, narrow your query to items currently priced greater than $20 and for whom there are at least 6 bids,

```
serviceQuery = from item in serviceQuery
               where item.CurrentPrice > 20.0
               where item.BidCount > 5
               select item;
```

Make the query and assign the resulting data to the CollectionViewSource

```
data.LoadAsync(serviceQuery);
 cvs.Source = data;
```

The result is a very gratifying gridview of Furbys as shown in Figure 13.

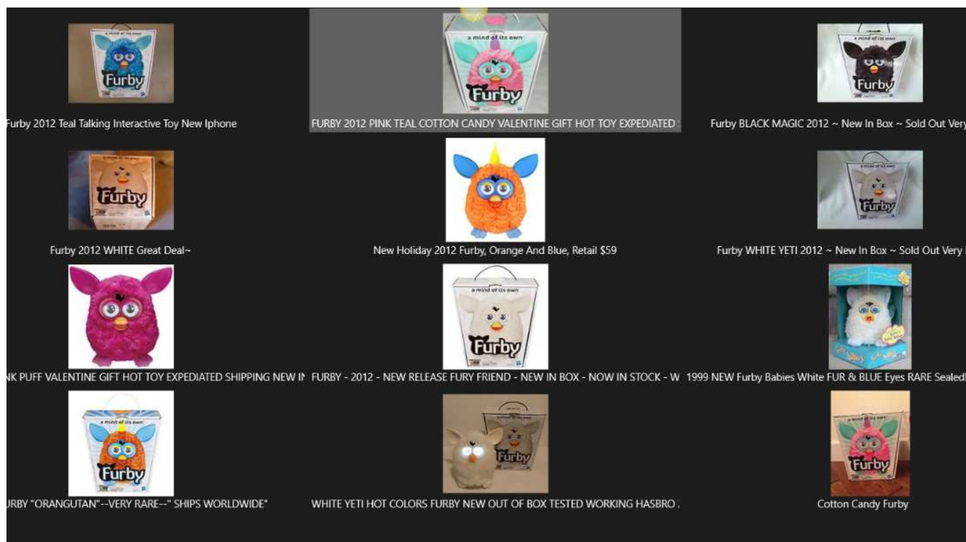


Figure 13, Furbys

8

Charms and Contracts

For the first time in the history of Windows, there is a built-in way to Search (and Share and set your options) across all applications. This is presented to the user as Charms, which are brought forward by swiping in from the right hand side, as shown in figure 1

Clicking, e.g., on the search charm (or pressing Windows-Q) brings up the Search menu, as shown in figure 2. If you are in an application that supports searching, it will be selected in the Search menu and the search will be restricted to that application (though you are free to choose a different application). If you are at the Start menu, the Search will be against applications.

All of the applications that support searching are listed. If you choose a different application, that application becomes the current application and the search is limited to that application.

Support for all of the charms is implemented with contracts. A contract establishes the relationship between your application and the charm service (e.g., searching).

01) Searching

To see this, let's create an application that will simply retrieve the search term from the search box. To begin, create a new Blank Application and call it Searching1. Double-click on the Package.appmanifest and click on the Declarations tab. Drop down the available declarations and select Search, as shown in figure 3.

Let's add a TextBlock to MainPage.xaml that will serve to display the search term,

```
<TextBlock Name="StatusBlock"
           FontSize="40"
           Margin="50"
           Text="Ready..." />
```

In the code behind, create a static variable that will point to the MainPage

```
public static MainPage Current;
```

In the constructor, assign the current MainPage to Current,

```
public MainPage()
{
    this.InitializeComponent();
  Current = this;
}
```

Finally, in the MainPage.xaml.cs file, add a method ShowQueryText that will display whatever is passed in within the TextBlock we just created,

```
public void ShowQueryText( string queryText )
{
    StatusBlock.Text = "Query submitted: " + queryText;
}
```

In App.xaml.cs you can now override the virtual method OnSearchActivated which will be called when Search is activated for this application. This method takes one argument, of type SearchActivatedEventArgs.

This method will do two things. First, it will call a helper method to ensure that the MainPage is ready, and then it will check the QueryText it is provided, and if it is not empty it will display that query text by calling ShowQueryText on the MainPage.

```
async protected override void OnSearchActivated(
SearchActivatedEventArgs args )
{
    await EnsureMainPageActivatedAsync( args );
    if ( args.QueryText == "" )
    {
        // navigate to landing page.
    }
    else
    {
        // display search results.
        MainPage.Current.ShowQueryText( args.QueryText );
    }
}
```

Notice the use of the keyword async at the top of the method and the await keyword used when calling EnsureMainPageActivatedAsync. These are part of the new asynchronous support in C# 5, which eliminates the need for callback methods. Execution will wait for the

EnsureMainPageActivatedAsync method to complete, *without blocking the UI thread*, and will resume on the next line on completion.

EnsureMainPageActivatedAsync is an asynchronous method that makes sure that your main page is ready to receive the call to ShowQueryText,

```
async private Task EnsureMainPageActivatedAsync(
IActivatedEventArgs args )
{
    if ( args.PreviousExecutionState ==
ApplicationExecutionState.Terminated )
    {
        // Do an asynchronous restore
    }

    if ( Window.Current.Content == null )
    {
        var rootFrame = new Frame();
        rootFrame.Navigate( typeof( MainPage ) );
        Window.Current.Content = rootFrame;
    }

    Window.Current.Activate();
}
```

Note that you will need to add *using Threading.Tasks* to App.xaml.cs.

Run the application and bring up the Search Charm. Type in some text and notice, as shown in figure 4, that your application is the selected application and that the text you type into the search box is successfully submitted to the application and displayed in the TextBlock.

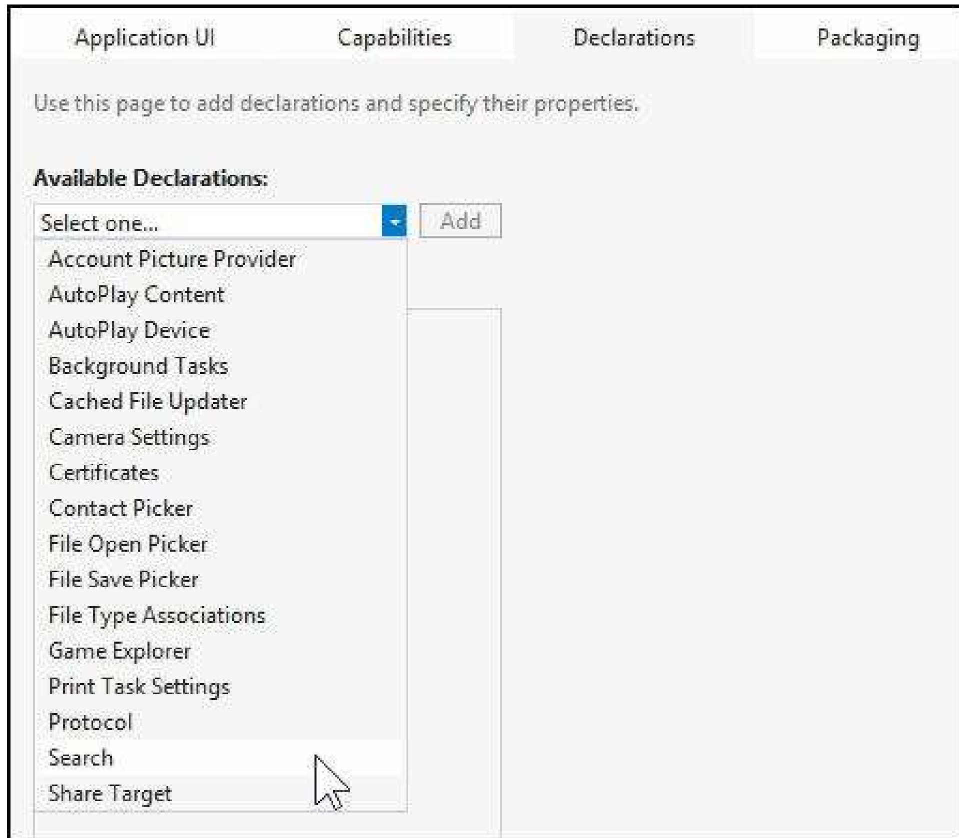
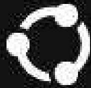
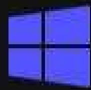
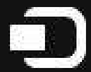


Figure 3 – Declaring Search

Search


Share


Start


Devices

Settings

This book was purchased by arosner@rosnertech.com

Figure 1 – Charm Bar

Search

Apps

Apps

Settings

Files

Store

Amazon

AppMock by Telerik

Bing

Feed Reader

Games

Google Search

Internet Explorer

Kindle

Mail

Figure 2 – Search Bar

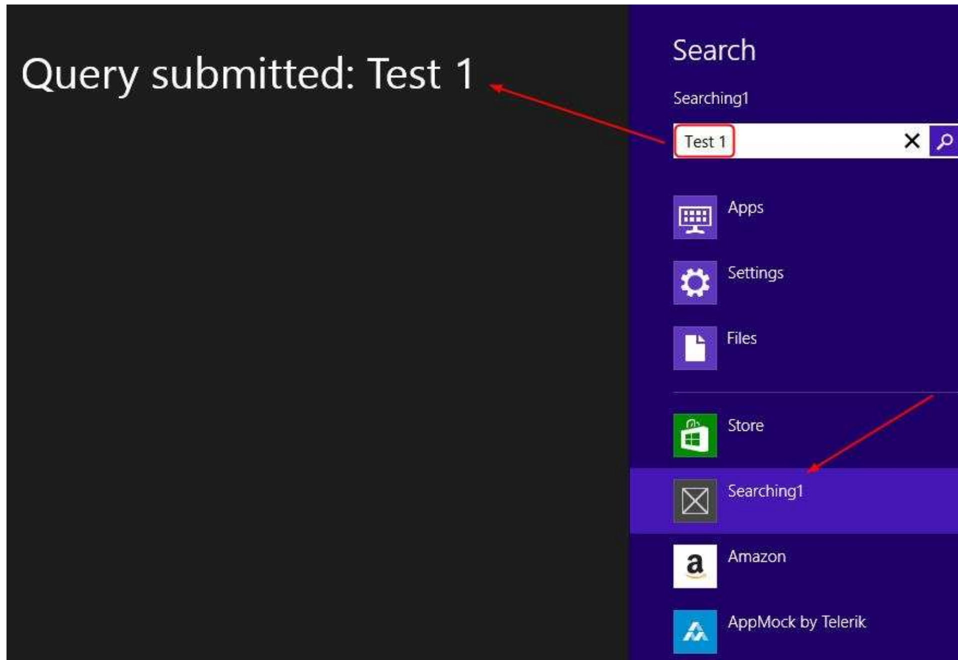


Figure 4 – Query Submitted

02) Matching Search Terms

Retrieving the search term is only half the battle, the other half is returning a match that is appropriate for your application.

Create a new application, Searching2 by copying over Searching 1. Confirm that it still works. Rename the project and the solution and then open Package.appxmanifest and in the ApplicationUI tab rename the Display name, the entry point and the description as shown in figure 5,

Application UI   Capabilities   Declarations   Packaging

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

Display name: Searching2

Entry point: Searching2.App

Default language: en-US   More information

Description: Searching2

Figure 5 – Display Name

The search contract specifies that you must be able to make search suggestions (on partial matches) and search recommendations on full matches. The search recommendation should have the application name, a one line description and an image to set it apart. Search suggestions are illustrated in figure 6 and search recommendations in figure 7

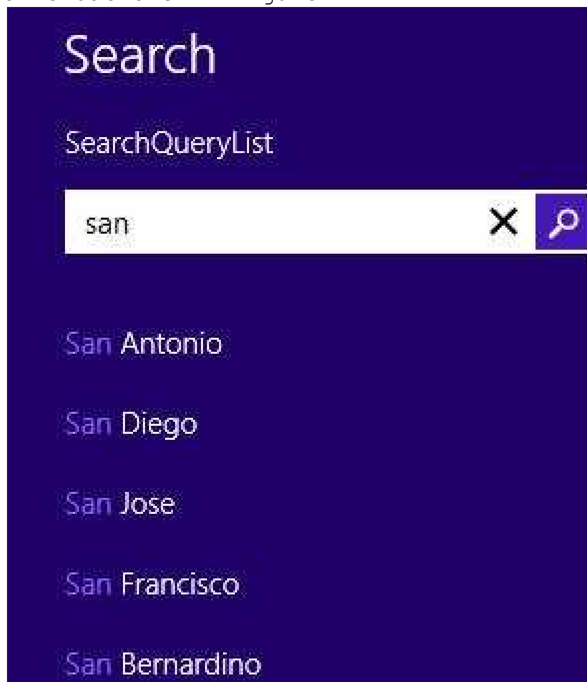


Figure 6 – search Suggestion

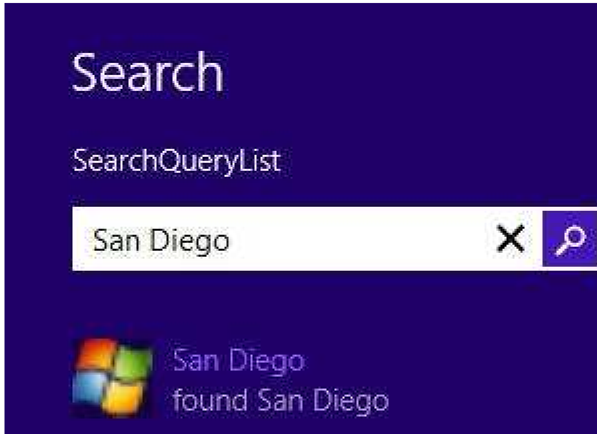


Figure 7 – Search recommendation

All of the work to accomplish this can be done in MainPage.xaml.cs (or can be moved out into a helper class). We start by declaring the static MainPage Current again, and then add a private instance of SearchPane and a constant value for how many suggestions we'll show for Search suggestions,

Adding SearchPane requires adding a using statement. The easiest way to do this is to place the cursor on the term SearchPane and enter control-dot. This brings up Intellisense and you can just hit enter to add the required using statement, as shown in figure 8,

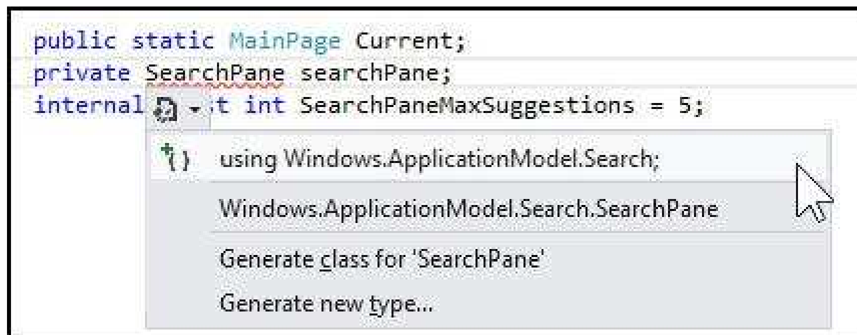


Figure 8 – Adding Using Statement

We need data to search when the user enters search criteria. Normally we'd retrieve this data form a database or a web service. For now, we'll hardcode this data into a static array of strings,

```
private static readonly string[] suggestionList =
     {
         "Shanghai", "Istanbul", "Karachi", "Delhi", "Mumbai",
"Moscow", "São Paulo", "Seoul", "Beijing", "Jakarta",
         "Tokyo", "Mexico City", "Kinshasa", "New York City",
"Lagos", "London", "Lima", "Bogota", "Tehran", "Ho Chi Minh City",
         "Hong Kong", "Bangkok", "Dhaka", "Cairo", "Hanoi", "Rio
de Janeiro", "Lahore", "Chonquing", "Bangalore", "Tianjin",
         "Baghdad", "Riyadh", "Singapore", "Santiago", "Saint
Petersburg", "Surat", "Chennai", "Kolkata", "Yangon", "Guangzhou",
         "Alexandria", "Shenyang", "Hyderabad", "Ahmedabad",
"Ankara", "Johannesburg", "Wuhan", "Los Angeles", "Yokohama",
         "Abidjan", "Busan", "Cape Town", "Durban", "Pune",
"Jeddah", "Berlin", "Pyongyang", "Kanpur", "Madrid", "Jaipur",
         "Nairobi", "Chicago", "Houston", "Philadelphia",
"Phoenix", "San Antonio", "San Diego", "Dallas", "San Jose",
         "Jacksonville", "Indianapolis", "San Francisco",
"Austin", "Columbus", "Fort Worth", "Charlotte", "Detroit",
         "El Paso", "Memphis", "Baltimore", "Boston", "Seattle
Washington", "Nashville", "Denver", "Louisville", "Milwaukee",
         "Portland", "Las Vegas", "Oklahoma City", "Albuquerque",
"Tucson", "Fresno", "Sacramento", "Long Beach", "Kansas City",
         "Mesa", "Virginia Beach", "Atlanta", "Colorado Springs",
"Omaha", "Raleigh", "Miami", "Cleveland", "Tulsa", "Oakland",
         "Minneapolis", "Wichita", "Arlington", " Bakersfield",
"New Orleans", "Honolulu", "Anaheim", "Tampa", "Aurora",
         "Santa Ana", "St. Louis", "Pittsburgh", "Corpus Christi",
"Riverside", "Cincinnati", "Lexington", "Anchorage",
         "Stockton", "Toledo", "St. Paul", "Newark", "Greensboro",
"Buffalo", "Plano", "Lincoln", "Henderson", "Fort Wayne",
         "Jersey City", "St. Petersburg", "Chula Vista",
"Norfolk", "Orlando", "Chandler", "Laredo", "Madison", "Winston-
Salem",
         "Lubbock", "Baton Rouge", "Durham", "Garland",
"Glendale", "Reno", "Hialeah", "Chesapeake", "Scottsdale",
         "North Las Vegas", "Irving", "Fremont", "Irvine",
"Birmingham", "Rochester", "San Bernardino", "Spokane",
         "Toronto", "Montreal", "Vancouver", "Ottawa-Gatineau",
"Calgary", "Edmonton", "Quebec City", "Winnipeg", "Hamilton"
     };
```

In the cunstroctor, after you set the Current static member, set the searchPane to be the result of calling the static method GetForCurrentView

```
public MainPage()
```

```
 {
     this.InitializeComponent();
     Current = this;
     searchPane = SearchPane.GetForCurrentView();
 }
```

In the OnNavigatedTo method we'll register for three events:

- The Query Was Submitted
- Suggestions have been Requested
- A Result Suggestion has been chosen

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    SearchPane.GetForCurrentView().QuerySubmitted +=
        new TypedEventHandler<SearchPane,
SearchPaneQuerySubmittedEventArgs>
            (OnQuerySubmitted);

    searchPane.SuggestionsRequested +=
        new TypedEventHandler<SearchPane,
SearchPaneSuggestionsRequestedEventArgs>
            (OnSearchPaneSuggestionsRequested);

    searchPane.ResultSuggestionChosen +=
OnSearchPaneResultSuggestionChosen;
}
```

The event handler for Query Submitted (OnQuerySubmitted) simply passes the query text to a helper method that calls NotifyUser with the text which in turn displays the text as we did last time,

```
internal void OnQuerySubmitted(object sender,
SearchPaneQuerySubmittedEventArgs args
{
    ProcessQueryText(args.QueryText);
}
internal void ProcessQueryText(string queryText)
{
    NotifyUser("Query submitted: " + queryText);
}

public void NotifyUser(string strMessage)
{
    StatusBlock.Text = strMessage;
}
```

The more interesting work is done in the other event handlers. The simpler of these is OnSearchResultsSuggestionChosen which simply notifies the user of which recommendation was selected,

```
void OnSearchPaneResultSuggestionChosen(
    SearchPane sender, SearchPaneResultSuggestionChosenEventArgs
args)
{
   NotifyUser("Recommendation picked: " + args.Tag);
}
```

Finally, we turn to OnSearchPaneSuggestionsRequested. Here we have a good bit of work to do, determining how much of a match, if any, we have and displaying the results accordingly. The first test is whether the search string is empty or null, if so we notify the user,

```
private void OnSearchPaneSuggestionsRequested(
     SearchPane sender, SearchPaneSuggestionsRequestedEventArgs e)
 {
     var queryText = e.QueryText;
     if (string.IsNullOrEmpty(queryText))
     {
        NotifyUser("Use the search pane to submit a query");
     }
```

Assuming it is a valid query string, we initialize a Boolean, isRecommendationFound to false and iterate through our contents looking for a match,

```
var request = e.Request;
bool isRecommendationFound = false;
foreach (string suggestion in suggestionList)
{
```

If we get an exact match we retrieve the image that we've put in the Assets folder (in this case a copy of the Windows logo, but you can use any image you like) and we obtain the SearchSuggestionCollection from the request object and call AppendResultSuggestion, passing in the text to display, the details, the tag (uusable when called by by ResultSuggestionChosen event) an image (if any) and alternate text for the image. We also set our Boolean to true,

```
if (suggestion.CompareTo(queryText) == 0)
{
    RandomAccessStreamReference image =
        RandomAccessStreamReference.CreateFromUri(new Uri("ms-
appx:///Assets/windows-sdk.png"));

    request.SearchSuggestionCollection.AppendResultSuggestion(
        suggestion,              // text to display
        "found " + suggestion,   // details
        "tag " + suggestion,     // tags usable when called back
by ResultSuggestionChosen event
```

```
        image,                   // image if any
        "image of " + suggestion // image alternate text
        );
    isRecommendationFound = true;
}
```

If we do not get an exact match, we look for a partial match and add it to the SearchSuggestionCollection,

```
else  // otherwise, this is a suggestion
    if (suggestion.StartsWith(queryText,
StringComparison.CurrentCultureIgnoreCase))
    {
        // Add suggestion to Search Pane

request.SearchSuggestionCollection.AppendQuerySuggestion(suggestio
n);
    }
```

After each iteration of the foreach loop we check to make sure that we haven't exceeded the maximum number of suggestions,

```
if (request.SearchSuggestionCollection.Size >=
MainPage.SearchPaneMaxSuggestions)
{
    break;
}
```

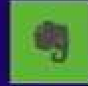
Finaly, we'll indicate a recommendation with an asterisk in the Message text and call notify user with our recommendation or our sugestions,

```
if (request.SearchSuggestionCollection.Size > 0)
{
    // Demo: show if it is a recommendation or just a partial
suggestion
    string prefix = isRecommendationFound ? "* " : "";
    NotifyUser(prefix + "Suggestions provided for query: " +
queryText);
}
else
{
    NotifyUser("No suggestions provided for query: " + queryText);
}
```

I) Sharing

Everybody loves to share. (At least after they are beaten into doing so in pre-school.)

This book was purchased by arosner@rosnertech.com



The ability to share from one application to another can be anything from useful to critical. Historically, we've seen a number of ways of doing this, the most enduring of which is the clipboard. Windows 8, however, introduces a system-wide sharing mechanism that is like the clipboard on steroids, and more important, which is easy to program and easy to use.

Architecture

The process of sharing *sounds* complicated, but the programmer's role is pretty straightforward, and made even easier by the sharing target contract that we'll talk about in the next posting.

Here's the architecture:

One application is the source. This is the provider of information. The source registers with the DataTransferManager and when the user wants to share, the source receives an event which it responds to by filling a DataPackage.

The (invisible, behind-the-scenes) ShareBroker filters the list of Target Apps that can handle the types of data you are sharing. The user selects the target app of choice and that application is activated. The targret processes the data package contents and reports complete.

Implementation

Creating a Sharing source is incredibly easy. Create a new application and call it ShareSource. Give the view a simple TextBlock that says “hello.”

```
<Grid Background="{StaticResource
ApplicationPageBackgroundThemeBrush}">
   <TextBlock FontSize="20"
              Text="Welcome to Sharing!" />
</Grid>
```

Turn to OnNavigatedTo. Here we create an instance of the DataTransferManager by calling GetForCurrentView, a static method on the DataTransferManager class. Once we have that, we can register for the DataRequested event handler.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
   DataTransferManager dtm =
           DataTransferManager.GetForCurrentView();
   dtm.DataRequested += dtm_DataRequested;
}
```

In the event handler we set the Title and the description through the args parameter,

```
void dtm_DataRequested(
     DataTransferManager sender, DataRequestedEventArgs args )
{
   args.Request.Data.Properties.Title =
         "My data from my application";
   args.Request.Data.Properties.Description =
         "Description of my data from my application";
```

We then set the data, using any number of data types. For example, to set text, we would write,

```
args.Request.Data.SetText( "This is text from my source" );
```

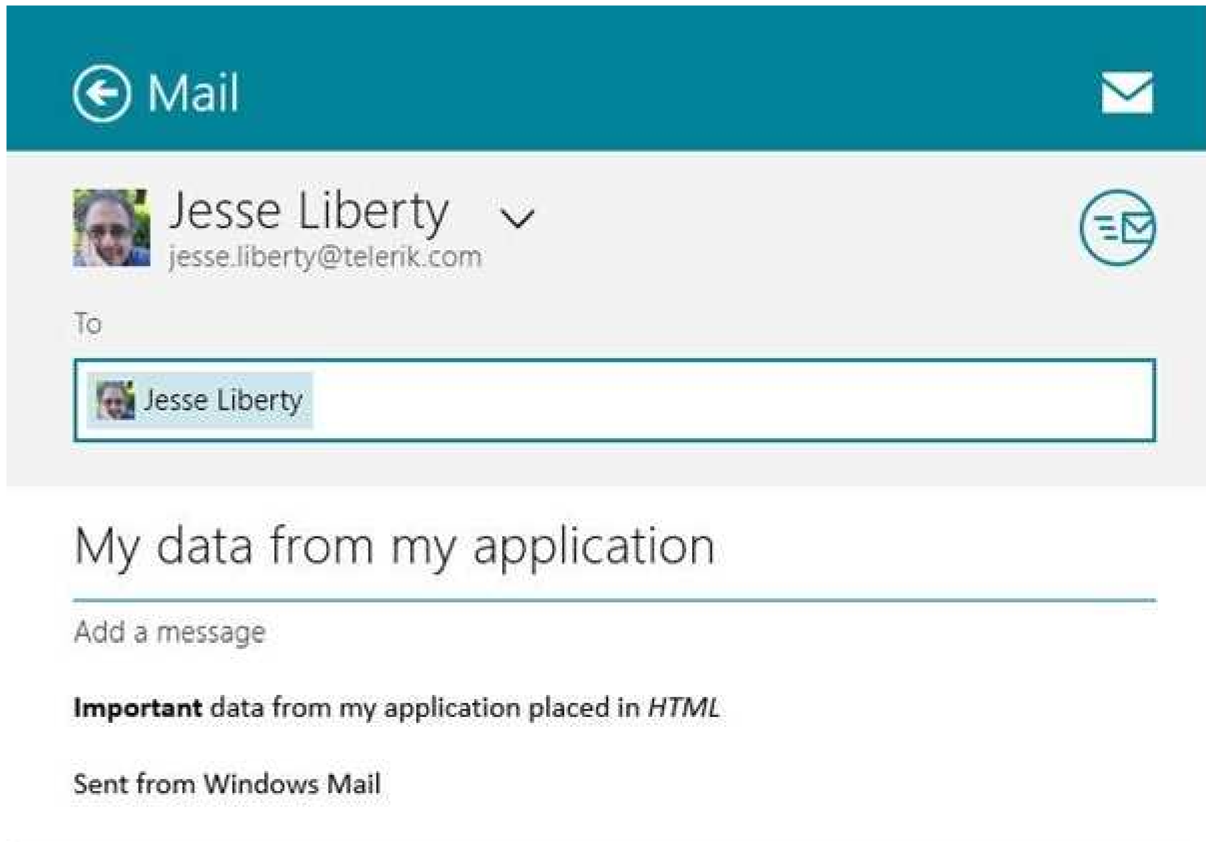
Similarly you can set HTML, but HTML is slightly tricky as a special format is expected. You simplify this problem by using the static CreateHtmlFormat method of the HtmlFormatHelper class,

```
args.Request.Data.SetHtmlFormat(
     HtmlFormatHelper.CreateHtmlFormat(
         "<b>Important</b> data from  my application placed
              in <i>HTML</i>" ) );
```

That’s it! You can now run your application and when it starts, swipe to bring in the charms. Click on Search and you will be presented with a list of all the applications registered on your computer that can handle the data types you are presenting

(text and html), as shown in the cropped image at the top of this posting.

Touch on Mail and notice that the Title you entered is used as the subject, and the HTML you entered is used as the body of the message.



If the target you chose only supported plain text, then the text message would have been used instead of the HTML, as we'll see in the next section.

II) Creating A Sharing Target

Being a target *would* be more complex than being a sharing source, except that Visual Studio provides a template that greatly simplifies the process.

By using the template, we avoid having to set the package appxmanifest and we avoid having to muck about in App.xaml.cs. All that work is done for us. To see this at work, create a new application and name it SharingTarget.

Right click on the project in Solution Explorer, and choose Add -> New Item. From the list, select *Share Target Contract* and use the default name.

You will be prompted that there are required files that Visual Studio can add for you, as shown in figure 9. Click Yes.

Figure 9 – Additional files

Wait a few seconds and you'll see that a number of files have been added to the Common Directory and that ShareTargetPage1.xaml has been added to your solution.

That's it! You now have a target. The sample page provides all the needed information, including the ability to add a comment.

To see this at work, run the program once (to register it) and then stop the new target program. Remember that targets don't have to be running to be in the target list; if the user selects your target application it will be started for you.

Return to the source application that we built yesterday and start that up. Swipe to bring in the charms and touch the sharing charm. Notice the list of target applications now includes your new application as shown in figure 10,

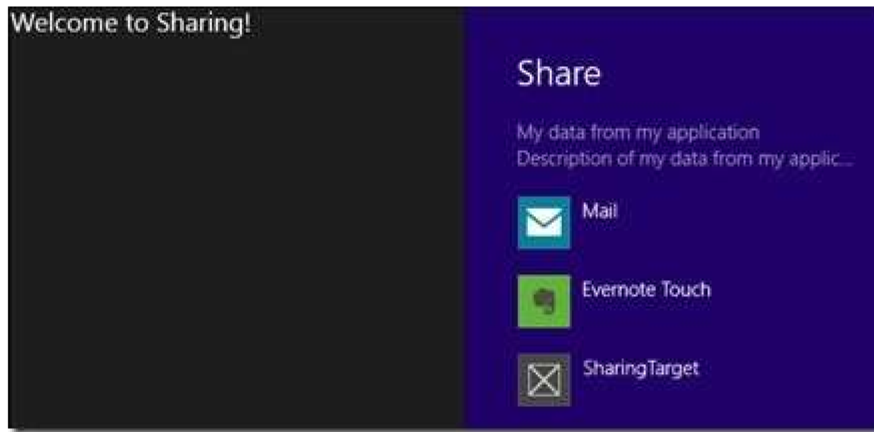


Figure 10 – Source Target

If you click on SharingTarget (the application you just created) the *text* version of the data is picked up as our target currently only supports text (if you try Mail you'll still see the HTML version).

Note that this new application doesn't do anything with the data you've shared; you'll have to write additional code to integrate that data into your new application.