Data Binding to controls on Windows Phone

This article explains how bind data objects in code to objects declared in your Windows Phone app's XAML UI.

Introduction

Windows Phone apps typically define their UI declaratively using XAML (in Expression Blend). While some UI component values may be hardcoded in XAML, many will be provided by data sources (objects, databases, XML feeds). Data binding is the technique which provides the connection between the XAML components and their values as defined in code.

There are a number of different binding modes, as defined in the BindingMode enumeration:

- `TwoWay` - changes to the UI or model automatically update the other. This is used for editable forms or other interactive scenarios.
- `OneWay` - updates the UI when the model changes. This is appropriate for read-only controls populated from data.
- `OneTime` - updates the UI when the application starts or when the data context changes. This is used when the source data is static/does not change.
- `OneWayToSource` - changes to the UI update the model
- *Default* - uses the default mode for the particular binding target (UI control). This differs based on the control.

This code example shows how bind `TextBox` controls in XAML to a source object defined in the C# file. It covers `OneTime`, `OneWay` and `TwoWay` modes. For the modes where the source (model) updates the UI (target) it also shows how to use the INotifyPropertyChanged interface class to ensure that the UI is updated whenever the underlying code changes.

The associated example creates three text boxes in XAML, and uses binding to update their values with information from C# objects.

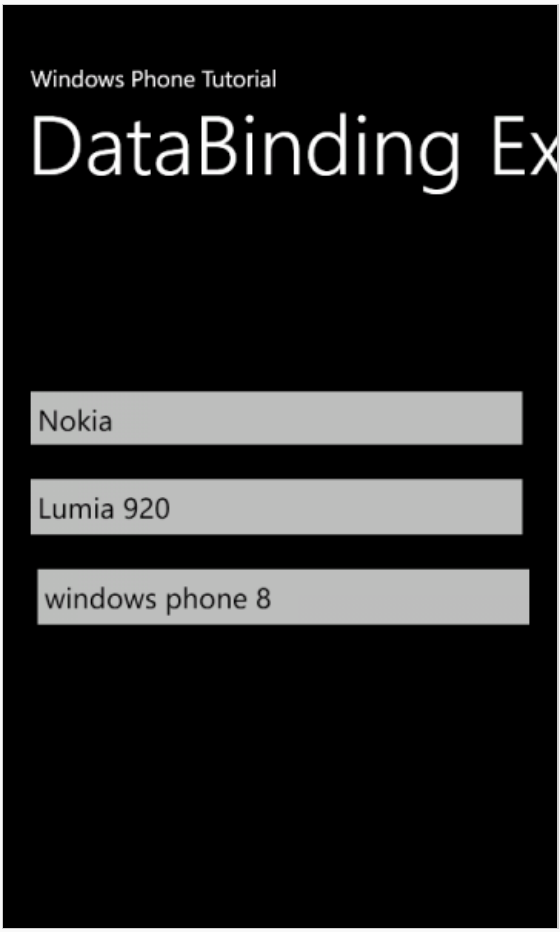

Figure 1: Data Binding in Windows Phone

Creating the XAML UI

First we create a simple Windows Phone page which contains a title panel followed by a content panel (`Grid`) containing three

(C) Copyright Nokia 2014. All rights reserved.

`TextBox` - which will display manufacturer, software and model text. Note that the content `Grid` is named `ContentPanel`, and we will later use this name to bind to the XAML from the C# code (and similarly the `Name`s of each of the `TextBox`es).

```
<!--LayoutRoot is the root grid where all page content is placed-->
<Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <!--TitlePanel contains the name of the application and page title-->
    <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
        <TextBlock x:Name="ApplicationTitle" Text="MY APPLICATION"
Style="{StaticResource PhoneTextNormalStyle}"/>
        <TextBlock x:Name="PageTitle" Text="DataBindingExample" Margin="9,-7,0,0"
Style="{StaticResource PhoneTextTitle1Style}"/>
    </StackPanel>

    <!--ContentPanel - place additional content here-->
    <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <TextBox Height="70" HorizontalAlignment="Left" Margin="0,130,0,0"
Name="manufacturerBox" Text="manufacturer" VerticalAlignment="Top" Width="450" />
        <TextBox Height="72" HorizontalAlignment="Left" Margin="0,206,0,0"
Name="modelBox" Text="model" VerticalAlignment="Top" Width="450" />
        <TextBox Height="72" HorizontalAlignment="Left" Margin="6,284,0,0"
Name="softwareBox" Text="software" VerticalAlignment="Top" Width="450" />
    </Grid>
</Grid>
```

The above XAML code contains hard-coded `Text` property values for the manufacturer, model and software. To bind the controls to data we first need to update the required properties with the `{Binding}` keyword. The keyword allows us to specify the name of the property we want to bind to, and the binding mode (ie whether changes to the UI will propagate to the underlying object and visa versa). Both of these two properties are optional: if you just specify `{Binding}` then the bound object's `ToString()` method will be called to get the property value and the binding will be the default for the target object..

In this example we specify the particular properties of the bound object to use for the Text (ie the `manufacturer` property of the bound object will be used for the text value of the `manufacturerBox` property). We also specify the mode as *OneTime* so that the UI is updated from the model on page load (only).

Tip: It is always best to specify the mode you want to use explicitly as this will often result in better performance.

If we omitted the mode the code would still "work" - the default mode of the `TextBox` is *TwoWay*. But without implementing `INotifyPropertyChanged` the result would be a one time update of the TextBox's content from code. Input into the `TextBox` will still be stored in the bound property though.

The updated XAML code looks like:

```
 <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
      <TextBox Height="70" HorizontalAlignment="Left" Margin="0,130,0,0"
Name="manufacturerBox" Text="{Binding manufacturer,Mode=OneTime}"
VerticalAlignment="Top" Width="450" />
      <TextBox Height="72" HorizontalAlignment="Left" Margin="0,206,0,0" Name="modelBox"
Text="{Binding model,Mode=OneTime}" VerticalAlignment="Top" Width="450" />
       <TextBox Height="72" HorizontalAlignment="Left" Margin="6,284,0,0"
Name="softwareBox" Text="{Binding software,Mode=OneTime}" VerticalAlignment="Top"
Width="450" />
</Grid>
```


(C) Copyright Nokia 2014. All rights reserved.

At this point we still haven't defined the data source (though we know from the XAML it has properties `manufacturer`, `model` and `software`) and bound it to the XAML. This is done in the C# code, which we define in the following sections.

Create and bind the data source

Next we create **PhoneModel.cs**, and define our `PhoneModel` class. Note how the class has C# properties that match the bound properties in our XAML.

```
namespace DataBindingEx
{
    public class PhoneModel
    {
        public string manufacturer { get; set; }
        public string model { get; set; }
        public string software { get; set; }

    }
}
```

The next step is to create an instance of the `PhoneModel` and bind it to the XAML, which we do using the DataContext property. The best place to perform the binding is when page initialisation completes, in the `MainPage_Loaded()` event handler as shown below.

```
namespace DataBindingEx
{
    public partial class MainPage : PhoneApplicationPage
    {
       // declare PhoneModel object
        PhoneModel _phnModel;

        // Constructor
        public MainPage()
        {
            InitializeComponent();

           // add page load event handler
            Loaded += MainPage_Loaded;
        }

        void MainPage_Loaded(object sender,RoutedEventArgs e)
        {
            // initialize _phnModel object
            _phnModel = new PhoneModel
            {
                manufacturer = "Nokia",
                model = "Lumia 920",
                software = "windows phone 8"
            };

            // call utility method to set DataContext
            setDataContext();
        }
```


(C) Copyright Nokia 2014. All rights reserved.

```
        private void setDataContext()
        {
            ContentPanel.DataContext = _phnModel;
        }
    }
}
```

The interesting part of the code is the `setDataContext()` method:

```
private void setDataContext()
    {
    ContentPanel.DataContext = _phnModel;
    }
```

The `ContentPanel` is a reference to the Grid object defined in XAML (`Grid x:Name="ContentPanel"`). We bind to this by assigning our `PhoneModel` instance to the referenced object's `DataContext`. Now when the page is loaded the the properties will be taken from our `PhoneModel` instance when the page is loaded. This results in the screen shown in #Figure1.

At this point the binding is *OneTime* (the properties are updated in the UI only on page load and changes to the UI do not affect the C# object values). The following sections explain how you can make the UI responsive to changes to the model, and how to make the model responsive to changes in the UI after loading (*TwoWay* binding).

Updating the UI when the properties change in code

To update the UI in response to changes in the source (model) we first need to update the XAML to state that this the Mode is *OneWay* or *TwoWay* binding. For a one-way binding all the TextBox definitions would be changed as below:

```
<TextBox Height="70" HorizontalAlignment="Left" Margin="0,130,0,0"
Name="manufacturerBox" Text="{Binding manufacturer,Mode=OneWay}" VerticalAlignment="Top"
Width="450" />
```

In addition, in order to update the UI when a property changes in code, we also need to update the `PhoneModel` class to implement INotifyPropertyChanged. The new **PhoneModel.cs** with this change is shown below:

```
using System.ComponentModel;
using System.Windows.Data;

namespace DataBindingEx
{
    public class PhoneModel:INotifyPropertyChanged
    {
        private string _manufacturer;
        private string _model;
        private string _software;


        public string manufacturer
        {
            get { return _manufacturer; }

            set
            {
                _manufacturer = value;
                NotifyPropertyChanged("manufacturer");
```


(C) Copyright Nokia 2014. All rights reserved.

```
            }
        }
        public string model
        {
            get { return _model; }

            set
            {
                _model = value;
                NotifyPropertyChanged("model");
            }
        }
        public string software
        {
            get { return _software;}
            set
            {
                _software = value;
                NotifyPropertyChanged("software");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void NotifyPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }

    }
}
```

The above code snippet creates a utility method named `NotifyPropertyChanged`, that raises the `PropertyChanged` event with the name of the property which needs to be updated in the UI. All properties need to call this `NotifyPropertyChanged` utility method.

To update the properties of the model and to see whether the UI get's updated or not we will add a button and on click event of the button we will change our properties and will see if the UI get updates or not, so the code for **MainPage.xaml.cs** goes like:

```
namespace DataBindingEx
{
    public partial class MainPage : PhoneApplicationPage
    {
        PhoneModel _phnModel;

        // Constructor
        public MainPage()
        {
            InitializeComponent();
            Loaded += MainPage_Loaded;
        }

        void MainPage_Loaded(object sender,RoutedEventArgs e)
        {
```


(C) Copyright Nokia 2014. All rights reserved.

```
            _phnModel = new PhoneModel
            {
                manufacturer = "Nokia",
                model = "Lumia 920",
                software = "windows phone 8"
            };


            setDataContext();
        }

        private void setDataContext()
        {
            ContentPanel.DataContext = _phnModel;
        }

        // utility method which changes the PhoneModel properties
        private void setPhoneProperties(String manufacturer, String model, String
software)
        {
            _phnModel.manufacturer = manufacturer;
            _phnModel.model = model;
            _phnModel.software = software;

        }

        // called when update button is clicked
        private void updateBtn_Click(object sender, RoutedEventArgs e)
        {
            setPhoneProperties("Nokia", "Lumia 900", "windows Phone 7.8");
        }
    }
}
```

When you click the button the screen should look like Figure 2:

(C) Copyright Nokia 2014. All rights reserved.

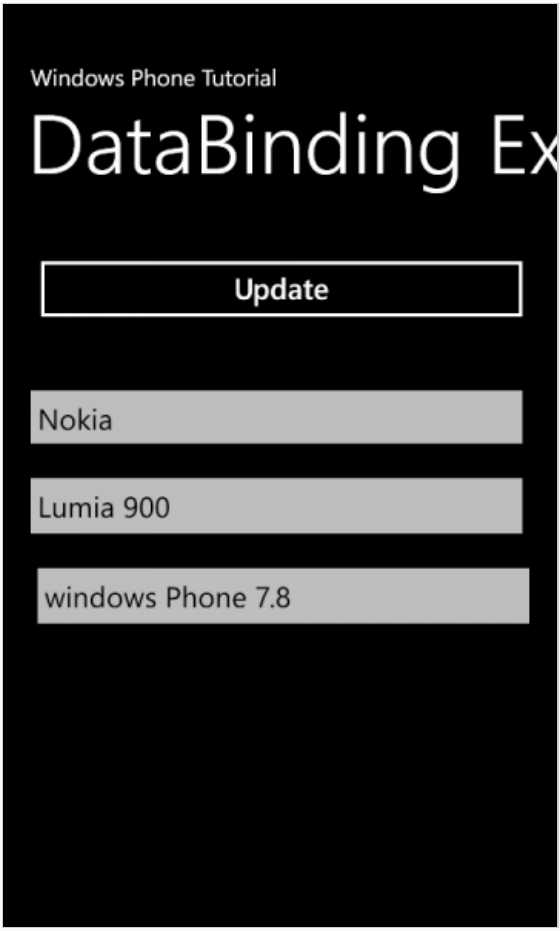

Figure2: Propagating data from the model to the UI

Two-way binding - updating the model from the UI Changes

OneWay binding (as covered in the previous section) is suitable for read-only controls. For editable controls we need *TwoWay* binding so that changes in the UI can update the model.

We don't need to change any other C# code for binding to be bi-directional. All we need to do is change the Mode parameter in XAML to *TwoWay*, as shown:

```
{Binding software,Mode=TwoWay}
```

Actually this is optional - the default mode for a `TextBox` is *TwoWay*. However you are less likely to make a mistake if the binding is explicitly specified.

The final XAML code is therefore:

```
  <TextBox Height="70" HorizontalAlignment="Left" Margin="12,80,0,0"
Name="manufacturerBox" Text="{Binding manufacturer,Mode=TwoWay}" VerticalAlignment="Top"
Width="438" />
  <TextBox Height="72" HorizontalAlignment="Left" Margin="12,156,0,0" Name="modelBox"
Text="{Binding model, Mode=TwoWay}" VerticalAlignment="Top" Width="438" />
  <TextBox Height="72" HorizontalAlignment="Left" Margin="12,234,0,0" Name="softwareBox"
Text="{Binding software,Mode=TwoWay}" VerticalAlignment="Top" Width="438" />
```

To integrate *TwoWay* mode binding in our example application, lets add one `Button` and three `TextBlock`'s to the Grid. We then change the data in the `TextBox`, and will see if updated data is pushed back to the `DataContext` by clicking the **Read Model** Button - this reads the data from the model object and should update the `TextBlock`'s Text property.

The update XAML for the example is:

(C) Copyright Nokia 2014. All rights reserved.

```
  <!--ContentPanel - place additional content here-->
   <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
      <TextBox Height="70" HorizontalAlignment="Left" Margin="12,80,0,0"
Name="manufacturerBox" Text="{Binding manufacturer,Mode=TwoWay}" VerticalAlignment="Top"
Width="438" />
      <TextBox Height="72" HorizontalAlignment="Left" Margin="12,156,0,0"
Name="modelBox" Text="{Binding model, Mode=TwoWay}" VerticalAlignment="Top" Width="438"
/>
      <TextBox Height="72" HorizontalAlignment="Left" Margin="12,234,0,0"
Name="softwareBox" Text="{Binding software,Mode=TwoWay}" VerticalAlignment="Top"
Width="438" />
      <Button Content="Update" Height="72" HorizontalAlignment="Left" Margin="9,18,0,0"
Name="changeButton" VerticalAlignment="Top" Width="441" Click="updateBtn_Click" />
      <Button Content="Read Model" Height="72" HorizontalAlignment="Left"
Margin="12,322,0,0" Name="readBtn" VerticalAlignment="Top" Width="438"
Click="readBtn_Click" />
      <TextBlock Height="30" HorizontalAlignment="Left" Margin="32,400,0,0"
Name="manufacBlock" Text="TextBlock" VerticalAlignment="Top" Width="401" />
      <TextBlock Height="30" HorizontalAlignment="Left" Margin="32,452,0,0"
Name="modelBlock" Text="TextBlock" VerticalAlignment="Top" Width="401" />
      <TextBlock Height="30" HorizontalAlignment="Left" Margin="32,506,0,0"
Name="softwareBlock" Text="TextBlock" VerticalAlignment="Top" Width="401" />
     </Grid>
```

and click event of *Read Button* will look like:

```
        private void readBtn_Click(object sender, RoutedEventArgs e)
        {
           // reading data from model and changing Text of the TextBlocks
            manufacBlock.Text = _phnModel.manufacturer;
            modelBlock.Text = _phnModel.model;
            softwareBlock.Text = _phnModel.software;
        }
```

If everything implemented is correct than it should look like:

(C) Copyright Nokia 2014. All rights reserved.

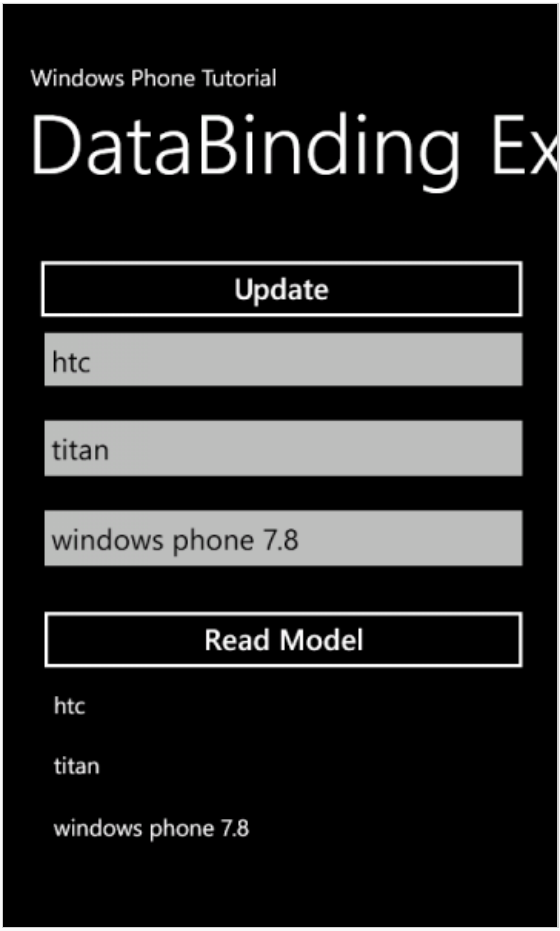

Figure 3: TwoWay data binding in Windows Phone

Summary

Property Binding is one of the best and simplest ways to glue your data with UI attributes.

References

- INotifyPropertyChanged
- Binding Class

(C) Copyright Nokia 2014. All rights reserved.