An Introduction to Xamarin.Forms

Getting Started with Cross Platform User Interfaces

Xamarin Inc.

# BriEf

Xamarin.Forms is a cross-platform natively backed UI toolkit abstraction that allows developers to easily create user interfaces that can be shared across Android, iOS, and Windows Phone. The user interfaces are rendered using the native controls of the target platform, allowing Xamarin.Forms applications to retain the appropriate look and feel for each platform. This guide will provide a quick introduction to Xamarin.Forms and how to get started writing applications with it.

**Sample Code:**

[LINK TO HELLOXAMARINFORMSWORLD.ZIP]

[LINK TO XAMARINFORMSSAMPLE.ZIP]

**Related APIs**

[INotifyPropertyChanged](http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifypropertychanged(v=vs.110).aspx)

**Related Articles:**

[Link to Xamarin.Forms controls]

[Cross Platform Application Fundamentals](http://docs.xamarin.com/guides/cross-platform/application_fundamentals/)

[Portable Class Libraries](http://docs.xamarin.com/guides/cross-platform/application_fundamentals/pcl/)

[Xamarin.Forms API docs]

[Data Binding Overview on MSDN](http://msdn.microsoft.com/en-us/library/ms752347(v=vs.110).aspx)

## Overview

Xamarin.Forms is a framework that allows developers to rapidly create cross platform user interfaces. It provides is own abstraction for the user interface that will be rendered using native controls on iOS, Android, or Windows Phone. This means that applications can share a large portion of their user interface code and still retain the native look and feel of the target platform.

Xamarin.Forms are written in C# and allow for rapid prototyping of applications that can evolve over time to complex applications. Because Xamarin.Form applications are native applications, they do not have the limitations of other toolkits such as browser sandboxing, limited APIs, or poor performance. Applications written using Xamarin.Forms are able to utilize any of the API’s or features of the underlying platform, such as (but not limited to) CoreMotion, PassKit, and StoreKit on iOS; NFC and Google Play Services on Android; and Tiles on Windows Phone. This also means it is possible to create applications that will have parts of their user interface create with Xamarin.Forms while other parts are created using the native UI toolkit.

Xamarin.Forms applications are architected in the same way as traditional cross-platform applications. The most common approach is to use [Portable Libraries](http://docs.xamarin.com/guides/cross-platform/application_fundamentals/pcl/) or Shared Projects to house the shared code, and then create platform specific applications that will consume the shared code.

This guide will discuss the fundamentals of the Xamarin.Forms framework. It will cover the following topics:

* Installing Xamarin.Forms
* Setting up a Xamarin.Forms solution in Visual Studio or Xamarin Studio.
* How Xamarin.Forms pages and controls are used.
* How to navigate between pages
* How to set up data binding.

## Requirements

Xamarin.Forms applications can be written for the following mobile operating systems:

* Android 4.0 or higher
* iOS 6.1 or higher
* Windows Phone 8 (in Visual Studio only)

Xamarin.Forms also requires the [Windows Phone Toolkit](http://phone.codeplex.com/) for some of its controls (such as the DatePicker) and animations.

It is assumed that the developer has familiarity with [Portable Class Libraries](http://docs.xamarin.com/guides/cross-platform/application_fundamentals/pcl/introduction_to_portable_class_libraries/) and Shared Projects.

## Getting Started with Xamarin.Forms

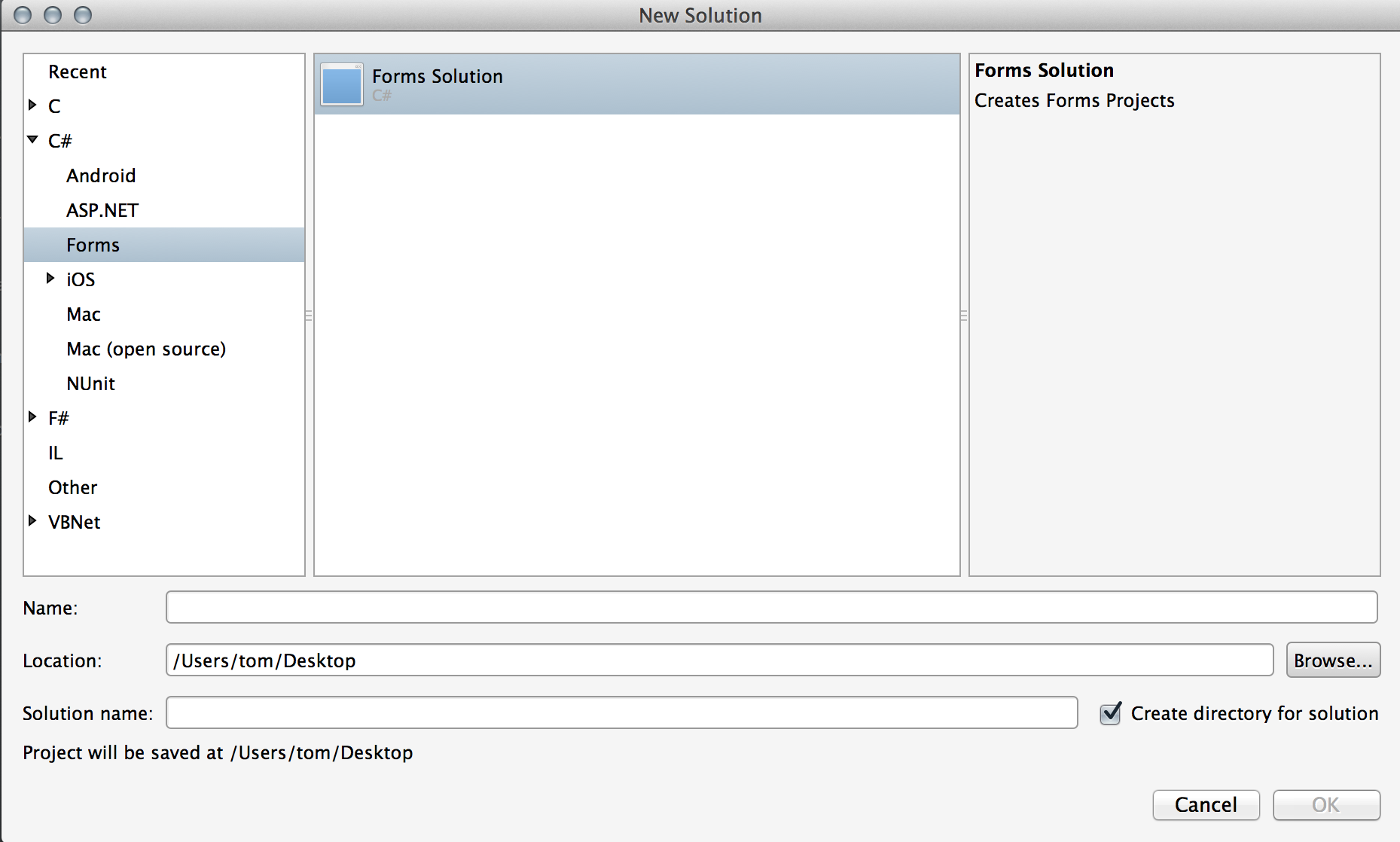
As discussed above, Xamarin.Forms is implemented as a .NET Portable Class Library (PCL), which makes it very easy to share the Xamarin.Forms API’s across a variety of platforms. The first step to getting started is to create a solution for the various projects that will make up the application.

A Xamarin.Forms solution can be created in Xamarin Studio or Visual Studio and will typically contain the following projects:

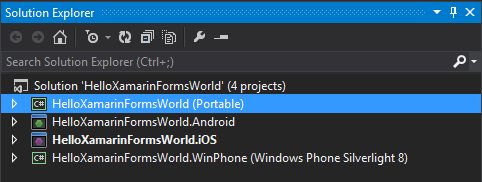
* **Portable Library or Universal Project** – This project is the cross platform application library that holds all of the shared code and share UI.
* **Xamarin.Android Application** – This project holds Android specific code and is the entry point for Android applications.
* **Xamarin**.iOS Application – This project holds iOS specific code and is the entry point for iOS applications.
* **Windows Phone Application** – This project holds the Windows Phone specific code and is the entry point for Windows Phone applications.

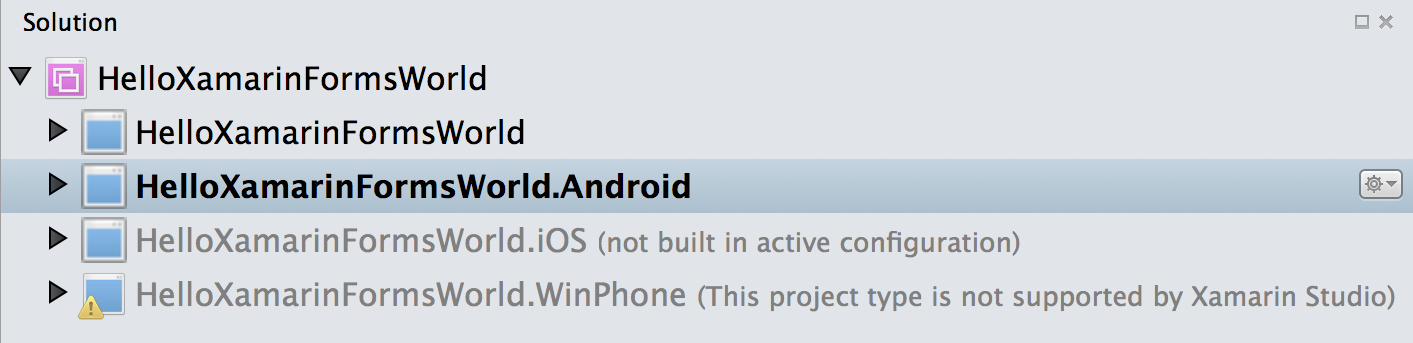
Xamarin 3.0 provides templates that will create a complete solution with all of the necessary projects for a Xamarin.Forms application. In Visual Studio, select File > New > Project. In the New Project Dialog that appears, click on Templates > Visual C# > Forms, and select the Universal Xamarin.Forms App (PCL) project in the centre of the dialog. An example of this can be seen in the following screenshot:





Enter the name of the project and click the OK button. The template will create a new solution with four projects in it. The following screenshots depict the solution loaded in Visual Studio and Xamarin Studio respectively:





Notice that while Xamarin Studio does not support Windows Phone applications it still loaded the Windows Phone project as a part of the solution. This allows you to browse your Windows Phone code while remaining in Xamarin Studio.

### Examining A Xamarin.Forms Application

The default template creates the simplest Xamarin.Forms solution possible. If you run the application, it should appear similar to the following screenshots:



Each screen in the screenshots above corresponds to a Page in Xamarin.Forms. A Xamarin.Forms.Page represents an Activity in Android, a View Controller in iOS, or a Page in Windows Phone. The HelloXamarinFormsWorld in the screenshots above instantiates a Xamarin.Forms.ContentPage object and uses that to display the Label.

To maximize the reuse of the startup code, Xamarin.Forms applications will have a single class named App that is responsible for instantiating the first Page that will be displayed. An example of the App class can be seen in the following code:

public class App

{

public static Page GetMainPage()

{

return new ContentPage

{

Content = new Label

{

Text = "Hello, Forms !",

VerticalOptions = LayoutOptions.CenterAndExpand,

HorizontalOptions = LayoutOptions.CenterAndExpand,

},

};

}

}

This code will instantiate a new ContentPage object that will display a single Label centered both vertically and horizontally on the page.

### Launching the Initial Xamarin.Forms Page on Each Platform

To use this Page inside an application, each platform application must initialize the Xamarin.Forms framework and then provide an instance of the ContentPage as it is starting up. This initialization step varies from platform from platform and will be discussed in the following sections.

#### Android

To launch the initial Xamarin.Forms page in Android, you create an Activity with the MainLauncher attribute just as you would a traditional Android application, except that your activity must inherit from Xamarin.QuickUI.Platform.Android.AndroidActivity, initialize the Xamarin.Forms framework, and then display the initial Page in the OnCreate method. The following code example illustrates this pattern in action:

namespace HelloXamarinFormsWorld.Android

{

[Activity(Label = "HelloXamarinFormsWorld", MainLauncher = true)]

public class MainActivity : **AndroidActivity**

{

protected override void OnCreate(Bundle bundle)

{

base.OnCreate(bundle);

**Xamarin.Forms.Forms.Init(this, bundle);**

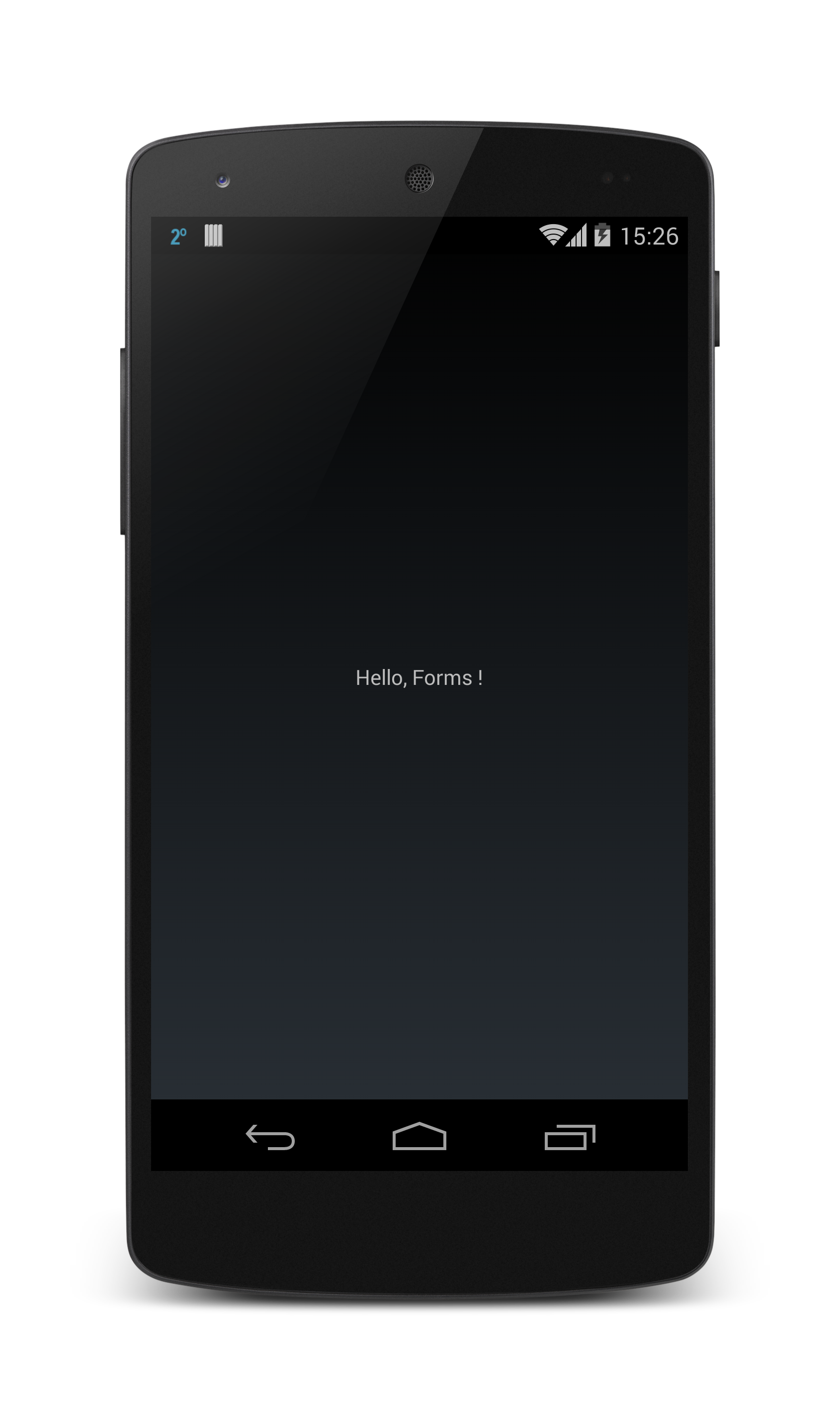
**SetPage(App.GetMainPage());**

}

}

}

The code above will create a Xamarin.Android application that will initialize the Xamarin.Forms framework and then run the shared UI code.



#### iOS

For a Xamarin.iOS application, the AppDelegate class must initialize the Xamarin.Forms framework and then set the RootViewController to the initial Xamarin.Forms Page. This is done inside the FinishedLaunching method, as demonstrated in the following code:

namespace HelloXamarinFormsWorld.iOS

{

[Register("AppDelegate")]

public partial class AppDelegate : UIApplicationDelegate

{

UIWindow window;

public override bool FinishedLaunching(UIApplication app, NSDictionary options)

{

**Forms.Init();**

window = new UIWindow(UIScreen.MainScreen.Bounds);

**window.RootViewController = App.GetMainPage().CreateViewController();**

window.MakeKeyAndVisible();

return true;

}

}

}

As with the Android application, the first step in the FinishedLaunching event is to initialize the Xamarin.Forms framework with a call to Xamarin.Forms.Forms.Init(). This step causes the iOS specific implementation of Xamarin.Forms to be globally loaded in the application.

The next step is set the root view controller of the application. This is done by invoking the CreateViewController() method on an instance of the HelloWordPage class that we created in the cross platform application library.



#### Windows Phone

For a Windows Phone project the start up page will initialize the Xamarin.Forms framework and then set the content of the startup page to that of the Xamarin.Forms Page. An example of how to do this can be seen in the code below:

namespace HelloXamarinFormsWorld.WinPhone

{

public partial class MainPage : PhoneApplicationPage

{

public MainPage()

{

InitializeComponent();

**Forms.Init();**

**Content = HelloXamarinFormsWorld.App.GetMainPage().ConvertPageToUIElement(this);**

}

}

}



Now that we have some familiarity with Xamarin.Forms lets discuss the parts of a Xamarin.Forms application in more detail.

## Parts of a Xamarin.Forms Application

The Hello World application of the previous section demonstrates the simplest Xamarin.Forms application possible. This section will cover the Xamarin.Forms framework in more detail and discuss some of the controls and layouts in Xamarin.Forms. It will then cover how to bind the controls to data and how to navigate from Page to Page. Lets first take a look at the components that make up a screen in Xamarin.Forms.

### Views and Layouts

Xamarin.Forms provides a single API for creating user interfaces with controls and layouts. At runtime, Xamarin.Forms control will be mapped to the appropriate native control and that is what will be rendered. Here is a brief explaination of each of these classes:

* **View** – these are typically referred to as controls or widgets in other platforms. They correspond to UI elements such as labels, buttons, text fields, etc.
* **Page** – a Xamarin.Forms page is a single screen in your application. These are analagous to an Android Activity, a WPF Page, or an iOS UIViewController.
* **Layout** – this is a specialized View subtype. It is meant to act as a container for other Layout or Views. Layout subtypes typically contain logic that is specific to organizing the child views in a certain way.
* **Cell** – This class is a specialized element that is used for items in a list or a table. It describes how each item in a list should be drawn.

The following table lists some of the more common controls:

|  |  |
| --- | --- |
| **Xamarin.Forms Control** | **Description** |
| Label | The label is a read-only text display control. |
| Entry | An Entry is a simple single-line text-input control. |
| Button | Buttons are used to initiate commands. |
| Image | This control is used to display a bitmap. |
| ListView | The ListView presents a scrolling list of items. The items inside a list are known as cells. |

Controls themselves will be hosted inside of a layout. Xamarin.Forms has two different categories of layouts that arrange the controls in very different ways:

* **Managed Layouts** – these are layouts that will take care of positioning and sizing child controls on the screen and follow the CSS box model. Applications should not attempt to directly set the size or position of child controls. One common example of a managed Xamarin.Forms layout is the StackLayout.
* **Unmanaged Layouts** – as opposed to managed layouts, unmanaged layouts will not arrange or position their children on the screen. Typically, the user will specify the size and location of the child control as it is being added to the layout. The AbsoluteLayout is an example of an unmanaged layout control.

Let’s take a look at the StackLayout and AbsoluteLayout in more detail below.

#### StackLayout

The StackLayout is a very common managed layout. The StackLayout greatly simplifies cross-platform application development by automatically arranging controls on the screen regardless of the screen size. Each child element is positioned one after the other, either horizontally or vertically in the order they were added. How much space the StackLayout will use depends on how the HorizontalOptions and the LayoutOptions properties are set, but by default the StackLayout will try to use the entire screen.

The following code is an example of using a StackLayout to arrange three Label controls on the screen:

public class StackLayoutExample: ContentPage

{

public StackLayoutExample()

{

Padding = new Thickness(20);

var red = new Label

{

Text = "Stop",

BackgroundColor = Color.Red,

Font = Font.SystemFontOfSize (20)

};

var yellow = new Label

{

Text = "Slow down",

BackgroundColor = Color.Yellow,

Font = Font.SystemFontOfSize (20)

};

var green = new Label

{

Text = "Go",

BackgroundColor = Color.Green,

Font = Font.SystemFontOfSize (20)

};

Content = new StackLayout

{

Spacing = 10,

Children = { red, yellow, green }

};

}

}

By default the StackLayout assumes a vertical orientation as illustrated by the following screenshots:



It is possible to change the Orientation and Vertical options by using the following code:

public class StackLayoutExample: ContentPage

{

public StackLayoutExample()

{

// Code that creates labels removed for clarity

Content = new StackLayout

{

Spacing = 10,

VerticalOptions = Layout.Options.End,

Orientation = StackOrientation.Horizontal,

HorizontalOptions = LayoutOptions.Start,

Children = { red, yellow, green }

};

}

}

These screenshots show what the screens would look after this code change:



Although it is not possible to explicitly size the child controls in a StackLayout, it is possible to provide hints to the layout engine through the HeightRequest and WidthRequest properties. The following code snippet shows how to request the width for each label:

var red = new Label

{

Text = "Stop",

BackgroundColor = Color.Red,

Font = Font.SystemFontOfSize (20),

WidthRequest = 100

};

var yellow = new Label

{

Text = "Slow down",

BackgroundColor = Color.Yellow,

Font = Font.SystemFontOfSize (20),

WidthRequest = 100

};

var green = new Label

{

Text = "Go",

BackgroundColor = Color.Green,

Font = Font.SystemFontOfSize (20),

WidthRequest = 200

};

Content = new StackLayout

{

Spacing = 10,

VerticalOptions = LayoutOptions.End,

Orientation = StackOrientation.Horizontal,

HorizontalOptions = LayoutOptions.Start,

Children = { red, yellow, green }

};

The following screenshots illustrate how the StackLayout trying to honor these suggestions:



#### AbsoluteLayout

In contrast to the StackLayout, the AbsoluteLayout is an unmanaged layout. Each control must be explicitly positioned within the layout. Conceptually it is very much like how controls are positioned in iOS (without constraints) or the old style Windows Forms. While this allows for very precise positioning of controls, this layout does require extra testing on different screen sizes.

A simple example of an AbsoluteLayout can be seen in the following code snippet:

public class MyAbsoluteLayoutPage : ContentPage

{

public MyAbsoluteLayoutPage()

{

var red = new Label

{

Text = "Stop",

BackgroundColor = Color.Red,

Font = Font.SystemFontOfSize (20),

WidthRequest = 200,

HeightRequest = 30

};

var yellow = new Label

{

Text = "Slow down",

BackgroundColor = Color.Yellow,

Font = Font.SystemFontOfSize (20),

WidthRequest = 160,

HeightRequest = 160

};

var green = new Label

{

Text = "Go",

BackgroundColor = Color.Green,

Font = Font.SystemFontOfSize (20),

WidthRequest = 50,

HeightRequest = 50

};

var absLayout = new AbsoluteLayout();

absLayout.Children.Add(red, new Point(20,20));

absLayout.Children.Add(yello, new Point(40,60));

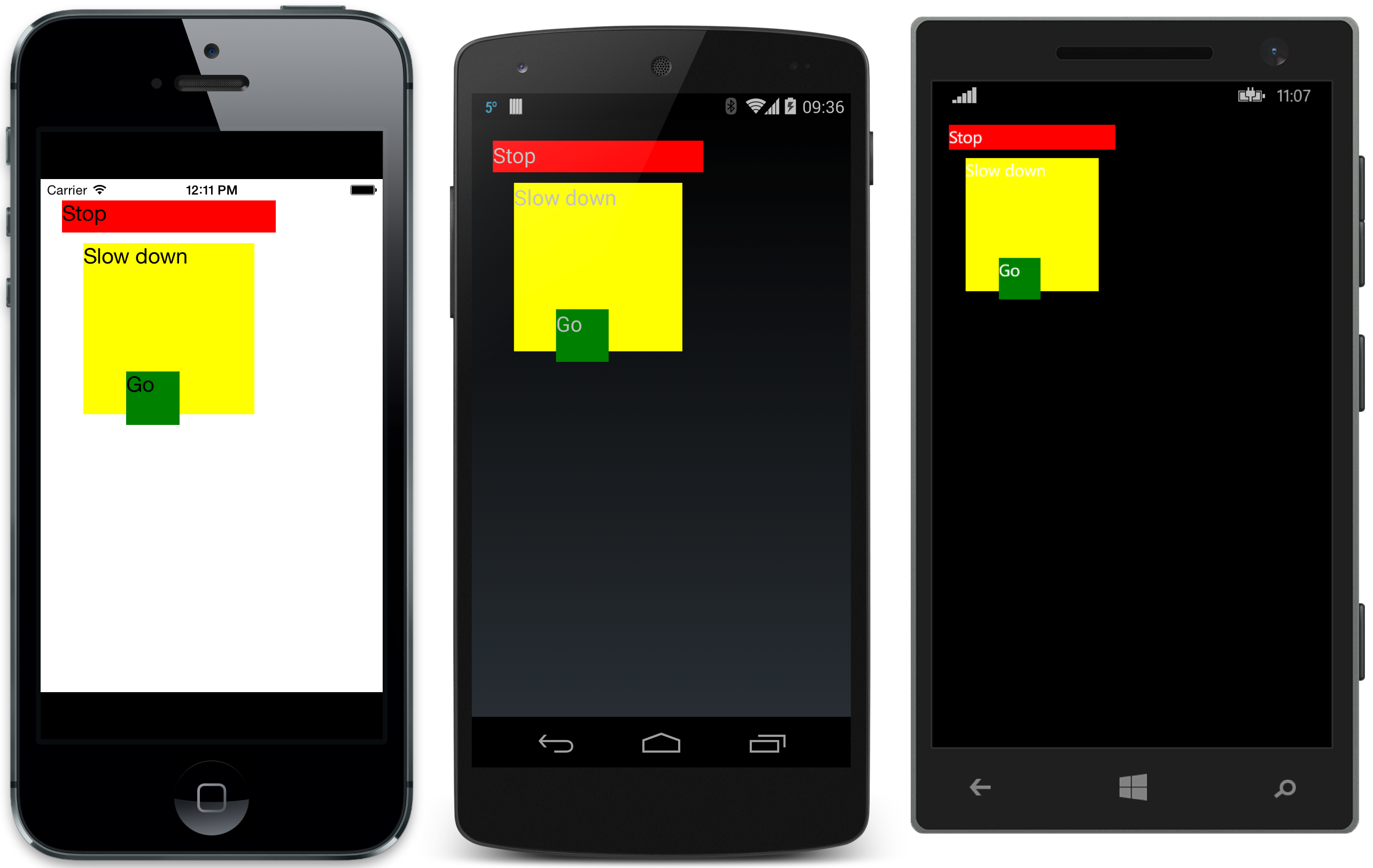
absLayout.Children.Add(red, new Point(80,180));

Content = absLayout;

}

}

When rendered, this page may look something like the following screenshot:



Note that the order the controls are added to the Children collection affects the Z-order of elements on screen – the first control appears at the ‘bottom’ of the Z-order and subsequent controls are added higher, meaning they can overlap (as the green label does in this example). Care must be taken when absolute positioning controls not to hide other controls by completely covering them or to accidentally positioning them off the edge of the screen.

### Lists in Xamarin.Forms

ListViews are a very common control in mobile applications and deserve to be covered in a bit more detail. The ListView is responsible for displaying a collection of items on the screen; each item in the ListView will be contained in a single cell. By default, a ListView will use the built-in TextCellTemplate class and render a single line of text. The code snippet below is a simple example of using the ListView:

var listView = new ListView

{

RowHeight = 40

};

listView.ItemSource = new string []

{

"Buy pears",

"Buy oranges",

"Buy mangos",

"Buy apples",

"Buy bananas"

};

Content = new StackLayout

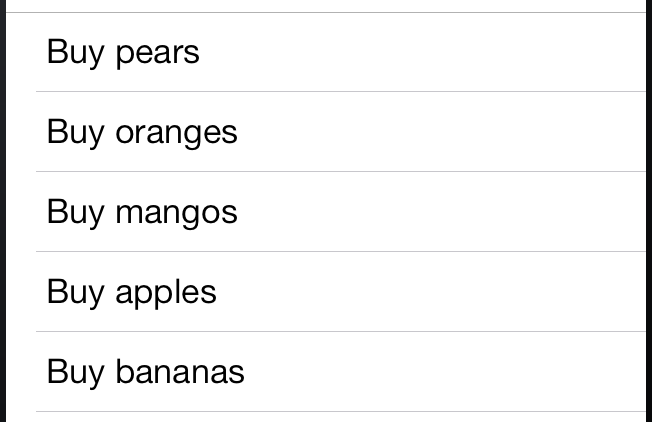
{

VerticalOptions = LayoutOptions.FillAndExpand,

Children = { listView }

};

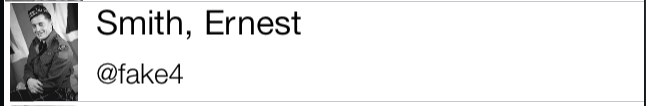
This code will resemble the following screenshot at run time:



#### Customizing the Appearance of a Cell

It is possible to customize the cells by subclassing ViewCell and then setting the type of this class to the ItemTemplate property of the ListView.

Consider the following screenshot of a cell in a ListView:



This cell is composed of one Image control and two Label views. To create this custom layout, we would subclass ViewCell as shown in the sample class below:

class EmployeeCell : ViewCell

{

public EmployeeCell()

{

var image = new Image

{

HorizontalOptions = LayoutOptions.Start

};

image.SetBinding(Image.SourceProperty, new Binding("ImageUri"));

image.WidthRequest = image.HeightRequest = 40;

var nameLayout = CreateNameLayout();

var viewLayout = new StackLayout()

{

Orientation = StackOrientation.Horizontal,

Children = { image, nameLayout }

};

View = viewLayout;

}

static StackLayout CreateNameLayout()

{

var nameLabel = new Label

{

HorizontalOptions= LayoutOptions.FillAndExpand

};

nameLabel.SetBinding(Label.TextProperty, "DisplayName");

var twitterLabel = new Label

{

HorizontalOptions = LayoutOptions.FillAndExpand,

Font = Fonts.Twitter

};

twitterLabel.SetBinding(Label.TextProperty, "Twitter");

var nameLayout = new StackLayout()

{

HorizontalOptions = LayoutOptions.StartAndExpand,

Orientation = StackOrientation.Vertical,

Children = { nameLabel, twitterLabel }

};

return nameLayout;

}

}

The code has a lot happening:

* It adds an Image and binds it to the ImageUri property of the Employee object. Data binding will be covered in more detail in a later section.
* It creates a StackLayout with a vertical orientation to hold the two Labels. The Labels are bound to the DisplayName property and the Twitter property of the Employee object.
* It creates another StackLayout that will host the Image and the StackLayout from the previous two steps. It will arrange its children using a horizontal orientation.

Once the custom cell has been created it can be used with a ListView control by wrapping in a DataTemplate:

List<Employee> myListOfEmployeeObjects = GetAListOfAllEmployees();

var listView = new ListView

{

RowHeight = 40

};

listView.ItemSource = myListOfEmployeeObjects;

listView.ItemTemplate = new DataTemplate(typeof(EmployeeCell));

This code will provide a List<Employee> objects to the ListView. Each cell will be rendered using the EmployeeCell class. The ListView will pass the Employee object to the EmployeeCell as its BindingContext.

### Data Binding

Data binding is used to simplify how a Xamarin.Forms application can display and interact with its data. It establishes a connection between the user interface and the underlying application. When the user edits the value in a text box, the data binding can update a property on an underlying object. The BindableObject class contains much of the infrastructure to support data binding.

Data binding defines the relationship between two objects. The source object will provide data. The target object is another object that will consume/display/interact/manipulate the data from the source object. For example, a Label may display the name from an Employee class. In this case, the Employee object is the source, while the Label is the target.

Setting up data binding on a Xamarin.Forms object (such as a Page or a Control) follows these two steps:

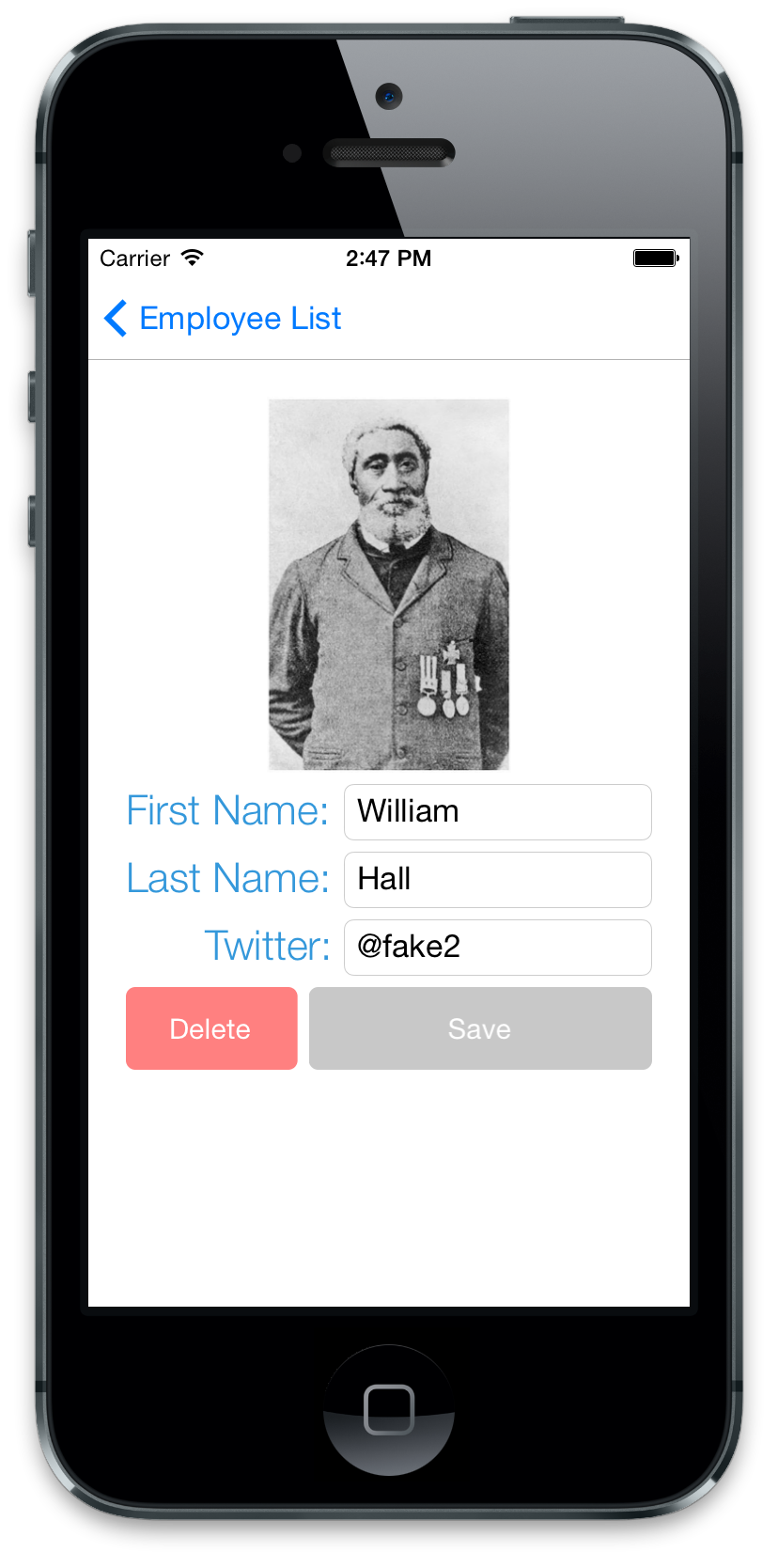
1. Set the BindingContext property to the object that will be bound to. The bound object may be any .NET object that implements the INotifyPropertyChanged interface (discussed below).
2. Invoke the SetBinding method on the Xamarin.Forms object once for each property or method that should be bound.

The SetBinding method takes two parameters. The first parameter specifies information about the type of binding. The second parameter is used to provide information about what to bind to or how to bind. The second parameter is, in most cases, just a string holding the name of property on the BindingContext. If we wanted to bind to the BindingContext directly, then we could use the following syntax:

someLabel.SetBinding(Label.TextProperty, new Binding("."));

The dot syntax tells Xamarin.Forms to use the BindingContext as the data source instead of a property on the BindingContext. This is handy when the BindingContext is a more simple type, such as a string or an integer.

To help understanding how to set up data binding in a Xamarin.Forms page, consider the following the following screenshot:



This Page consists of the following controls:

* Xamarin.Forms.Image
* Xamarin.Forms.Label
* Xamarin.Forms.Entry
* Xamarin.Forms.Button

The Page that makes up this screen would be passed an instance of an Employee object via the constructor. The following code snippet is an example of what this constructor might look like:

public EmployeeDetailPage(Employee employeeToDisplay)

{

this.BindingContext = employeeToDisplay;

var firstName = new Entry()

{

HorizontalOptions = LayoutOptions.FillAndExpand

};

firstName.SetBinding(Entry.TextProperty, "FirstName");

// Rest of the code omitted…

}

The first line of code sets the BindingContext to a .NET object – this tells the underlying data binding API’s what object to bind to. The next line of code instantiates a Xamarin.QuickUI.Entry control. The last line defines the binding between the Xamarin.QuickUI.Entry and employeeToDisplay; the Entry.Text property should be bound to the FirstName property of the object set to the BindingContext. The changes made in the Entry control will automatically be propagated to the employeeToDisplay object. Likewise, if changes are made to employeeToDisplay.FirstName, then Xamarin.Forms will also update the contents of the Entry control. This is known as two-way binding.

In order for two-way binding to work, the model class must implement INotifyPropertyChanged that we will look at next.

#### INotifyPropertyChanged

The INotifyPropertyChanged interface is used to notify a client of an object that a value has changed. The interface is very simple:

public interface INotifyPropertyChanged

{

event PropertyChangedEventHandler PropertyChanged;

}

Objects that implement INotifyPropertyChanged must raise the PropertyChanged event when one of their properties is updated with a new value. An example of one such class can be seem in the following class:

public class MyObject : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

string \_firstName;

public string FirstName

{

get { return \_firstName; }

set

{

if (value.Equals(\_firstName, StringComparison.Ordinal))

{

// Nothing to do - the value hasn't changed;

return;

}

\_firstName = value;

OnPropertyChanged();

}

}

void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

var handler = PropertyChanged;

if (handler != null)

{

handler(this, new PropertyChangedEventArgs(propertyName));

}

}

}

When an instance of MyObject has the FirstName changed, the method OnPropertyChanged is invoked which will raise the PropertyChanged event.

Notice propertyName parameter is adorned with the CallMemberName attribute. If the method OnPropertyChanged is invoke with a null value, the CallMemberName attribute will provide the name of the method that invoked OnPropertyChanged.

### Navigation

Now that we understand how to create pages and arrange controls, lets discuss how to navigate from one page to another. Navigation can be thought of as a last-in, first-out stack of Page objects. To move from one page to another an application will push a new page onto this stack. To return back to the previous page the application will pop the current page from the stack. This navigation in Xamarin.Forms is handled by the INavigation interface which provides the following methods:

public interface INavigation

{

void Push (Page page);

Page Pop ();

void PopToRoot ();

void PushModal (Page page);

Page PopModal ();

}

Xamarin.Forms has a NavigationPage class that implements this interface and will manage the stack of Pages. The NavigationPage class will also add a navigation bar to the top of the screen that displays a title and will also have a platform appropriate Back button that will return to the previous page. The following code shows how to wrap a NavigationPage around the first page in an application:

public static Page GetMainPage()

{

var mainNav = new NavigationPage(new EmployeeListPage());

return mainNav;

}

To display the LoginPage for the current page it is necessary to invoke the INavigation.Push as demonstrated in the following code snippet:

Navigation.Push(new LoginPage());

This causes the new LoginPage object to be pushed on the Navigation stack. To return back to the original page, the LoginPage must invoke:

Navigation.Pop();

Modal navigation is similar. The following snippet will display a new page modally:

Navigation.PushModal(new LoginPage());

To return to the calling page, LoginPage must invoke:

Navigation.PopModal();

## Summary

In this guide we discussed what Xamarin.Forms is and how it can be used to create cross platform applications. We covered how to install Xamarin.Forms and setup a solution. We learned how to create a Xamarin.Forms application with a common user interface that will retain the native look and feel of the underlying platform. We also saw how to set up data binding between the user interface and the underlying data and how to navigate between Pages.