Introduction to Xamarin.Forms

What is Xamarin.Forms?

Xamarin is offering Xamarin.Forms as an application scaffolding framework. It provides a cross-platform user-interface abstraction that is rendered using native controls on the target platforms: Android, iOS, and Windows Phone.

Xamarin.Forms is simple enough to allow rapid prototyping of user interfaces, complete with complex application logic, written entirely in C# and using the .NET framework. Unlike other tools that provide similar functionality, Xamarin.Forms doesn't have limitations related to browser sandboxing or limited API access – because Xamarin.Forms apps are also genuine native apps it is easy to enhance them with features from the underlying platform SDKs. This means Xamarin.Forms apps have the dual benefit of rapid development startup and shared code, but they can also utilize features including (but not limited to) CoreMotion, PassKit or StoreKit in iOS; NFC, Google Play Services and hardware accessories in Android; and Tiles in Windows Phone.

A Xamarin.Forms prototype may exist solely as shared scaffolding code, 100% re-used across all three platforms. It is expected that as applications are fleshed out and polished they will pull in native features from each platform using Xamarin.Forms's extensibility points, or even be flipped around to work predominantly with native controls (like ViewControllers or Activities) and reserve Xamarin.Forms screens for a subset of the application.

This ability to kickstart app development with a working cross-platform solution, and gradually add platform-specific features gives developers the best of both worlds and really leverages Xamarin's cross-platform C# and .NET framework tools, along with Microsoft's Windows Phone SDK.

Xamarin.Forms apps can be built in both Xamarin Studio and Visual Studio. Once released the capabilities will be available via NuGet, and easily added to any Xamarin.Android, Xamarin.iOS or Windows Phone project. The core framework is available as a Portable Class Library (PCL), meaning the bulk of a Xamarin.Forms application's code can also be written in a PCL.

Table of Contents

Samples

This document discusses the following sample solutions:

- **FormsBasics** – Simple control examples. Uncomment different lines in the App class to see each example.
- **Todo** – SQLite database-driven to-do list.
- **TodoMvvm** – the SQLite database-driven example written using the Mvvm pattern.
- **TodoXaml** – the SQLite database-driven example using Xaml for layouts instead of C# code.
- **Evolve13** – sample conference app using SQLite
- **Roget1911** – a 'navigable' thesaurus that demonstrates PCLStorage to open and parse XML files from within the Xamarin.Forms PCL.

These aren't official Xamarin samples (especially since Xamarin.Forms is still in beta). Final guidance on Xamarin.Forms best practices will be provided when the product ships.

References

The samples use the following external Nugets to provide additional functionality:

- **SQLite.NET-PCL** - http://www.nuget.org/packages/SQLite.Net-PCL/
- **PCLStorage** - http://www.nuget.org/packages/PCLStorage/0.9.4

General Application Structure

A Xamarin.Forms application consists of a single Portable Class Library (PCL) that defines the application logic and then a platform-specific project for each operating system (iOS, Android and Windows Phone).
The bulk of your code will live in the PCL. The platform-specific projects contain bootstrapping code to load Xamarin.Forms and configure any dependencies, such as implementations for data storage and other platform features.

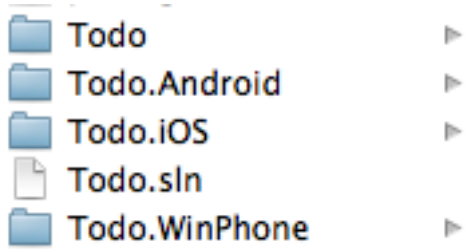
This document aims to help you understand:

- the structure of a Xamarin.Forms solution
- how Xamarin.Forms pages and controls are used
- writing platform-specific code inside your Xamarin.Forms PCL
- how to incorporate a database
- writing platform-dependent code in each target platform project

We'll start with a general overview of how a Xamarin.Forms application is structured on the filesystem and in your IDE (Xamarin Studio or Visual Studio) and then examine the code in a simple cross-platform Xamarin.Forms application.

Files on disk

The suggested approach to structure a Xamarin.Forms application on disk is to keep the projects in a single solution directory, and use a naming convention to help manage the code. A simple "to-do list" sample app might look like the screenshot below:



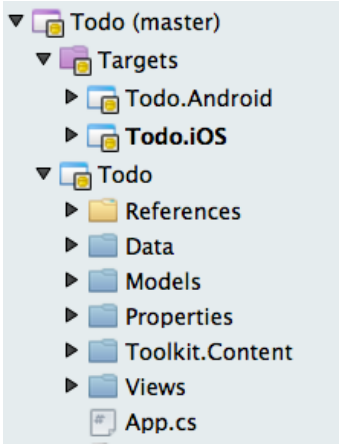
The purpose of these directories is:

- **Todo** – a Portable Class Library that contains all custom code: the Xamarin.Forms user interface, business and application logic.
- **Todo.Android** – Android application project that contains the Android bootstrapping app. This is also where you would configure your application name & icon, Android Manifest permissions, target Android SDK and other Android-specific options.
- **Todo.iOS** – iOS application project that contains the iOS app bootstrapping code. You will also configure you app icon, Bundle ID and other iOS-specific options in this project.
- **Todo.WinPhone** – Windows Phone application project that contains the Windows Phone bootstrapping code. You can also configure Windows Phone specific options for your app. Note that Windows Phone projects cannot be opened in Xamarin Studio.

The bootstrapping code is covered in more detail in the Code Structure section below.

Solution Structure

The preferred approach for the solution structure is to use a Solution Folder to group the platform-specific projects together and leave the PCL Xamarin.Forms library at the root of the solution. This screenshot shows a simple example:

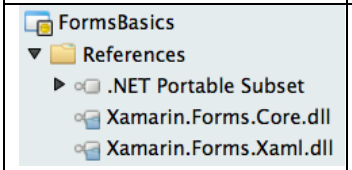
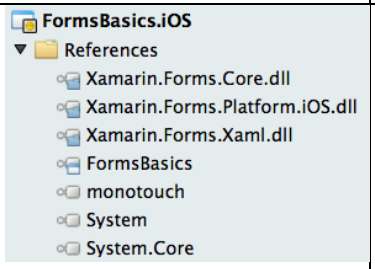
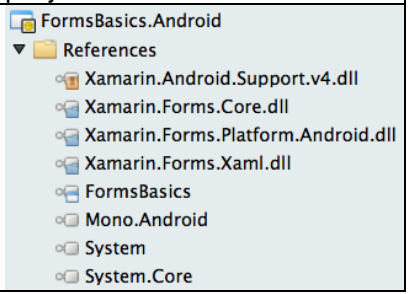


The main reason for this recommendation is the ability to collapse the Targets folder and hide the platform specific projects once they've been configured – this reinforces the idea that you're working on a single codebase. You generally shouldn't need to work with the application projects much at all, except to select them to start debugging.

Note that one of the application projects will always be your **Startup Project**.

References

Each project in the solution contains references to other libraries: they all need Xamarin.Forms assemblies and the platform-specific projects need to reference the PCL.

The PCL project must reference the Xamarin.Forms core and Xaml assemblies.	The iOS project references the Xamarin.Forms core and platform assemblies, plus a reference to the PCL **Todo** project where all the application code lives.	The Android project references Xamarin.Forms core and platform assemblies, plus the Android.Support.v4 component. It must also reference the **Todo** PCL project.
		

The Windows Phone project (not shown here) also requires a reference to WPtoolkit NuGet.

Portable Class Library Profile

The Xamarin.Forms App project will be a Portable Class Library (PCL). The recommended **PCL Profile is 78** that includes Xamarin, Windows Phone and Windows Store apps.

Code Structure

Your application code lives in a Portable Class Library that references Xamarin.Forms. Within this library there should be one instance of an application class – App – that should generally be structured with a single static class to return the main page. *This will in the future be replaced by the Xamarin.Forms.Application class when it is ready.* Generally speaking, this means your application will contain a class that looks like:

```
public static class App
{
    public static Page GetMainPage ()
    {
        return new MyFirstPage ();
    }
}
```

The first page you return in your app should be the top-level navigation page. The remained of your app will consist of other pages and logic defined in the Xamarin.Forms library.
To understand how your Xamarin.Forms app gets started on each platform, the bootstrap code for each platform is shown below. For each platform you must call `Xamarin.Forms.Init()` before anything else, then use a platform-specific method to start your first page.

Xamarin.iOS

In the AppDelegate class set the iOS application's RootViewController to the initial Xamarin.Forms page:

```
public override bool FinishedLaunching (UIApplication app, NSDictionary
options)
{
    Xamarin.Forms.Init ();
    window.RootViewController = App.GetMainPage ().CreateViewController ();
    window.MakeKeyAndVisible ();
}
```

Xamarin.Android

Ensure the first activity loaded implements the Xamarin.Forms AndroidActivity and display the initial Xamarin.Forms page:

```
[Activity (Label = "FormsBasics", MainLauncher = true)]
public class Activity1 : Xamarin.Forms.Platform.Android.AndroidActivity
{
    protected override void OnCreate (Bundle bundle)
    {
```

```
        base.OnCreate (bundle);
        Xamarin.Forms.Xamarin.Forms.Init (this, bundle);
        SetPage (App.GetMainPage ());
    }
}
```

Windows Phone (to come)

…

Once each application project is set up you will be able to compile and debug or deploy them to emulators or devices. Before that we need to learn how to create pages.

Creating Pages

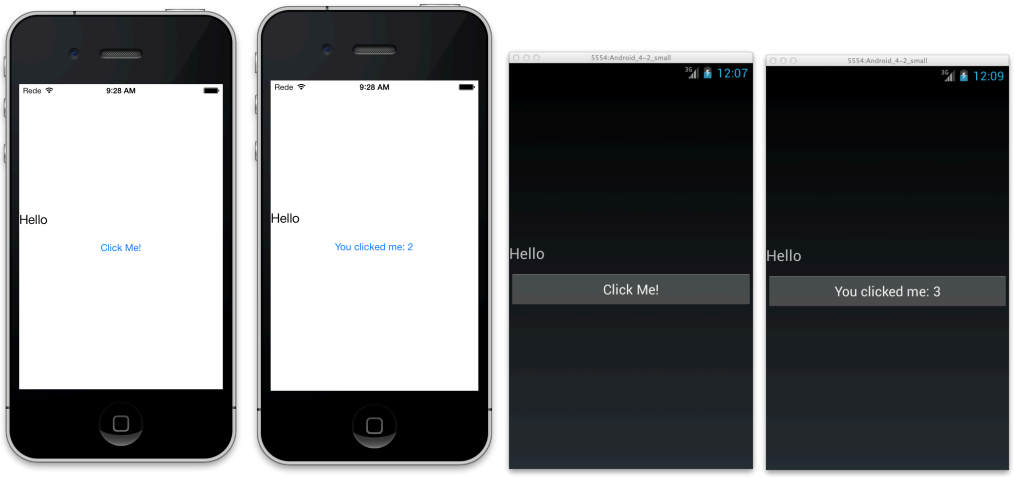
Pages should be created by subclassing the `ContentPage` class. A factory approach for pages can also be used, however subclassing `ContentPage` is the simplest approach. Ideally all pages will retain a default constructor. This makes using them in things like Xaml possible.

Further create widgets as they are needed, not before. This means the elements that go into a layout should be created before the layout itself is. This is not enforced in any way but it improves the readability of the code and can make it much more concise.

For example, the following simple page code creates and then adds a label and button to the screen using Xamarin.Forms controls. Note that the `Content` property of `ContentPage` contains all the visual elements for that screen:

```
using Xamarin.Forms;
public class MyFirstPage : ContentPage
{
    public MyFirstPage ()
    {
        var label = new Label {
                  Text = "Hello",
                  Font = Font.SystemFontOfSize (20),
        };

        var button = new Button {Text = "Click Me!"};

        int i = 1;
        button.Clicked += (s, e) => button.Text = "You clicked me: " + i++;

        Content = new StackLayout {
            Spacing = 10,
            VerticalOptions = LayoutOptions.Center,
            Children = {
                label,
                button
            }
        };
    }
}
```

Running this very simple app produces the following screen (shown before and after clicking). Notice the order of the controls and the affect that the `Spacing` and `VerticalOptions` has on the screen.



This simple example only shows a single page of content. To build more complicated applications we need to be able to navigate between pages.

Creating Navigation

Navigation in Xamarin.Forms is provided by TabbedPages and NavigationPages, as well as to a lesser degree MasterDetailPages and CarouselPages. Unlike ContentPages it is not generally advisable to subclass these pages. The one exception to this would be simply to provide semantic naming for instances where it benefits MVVM users.

We can extend our simple, single page app by adding another `ContentPage` as shown below:

```
using Xamarin.Forms;
public class MySecondPage : ContentPage
{
    public MySecondPage ()
    {
        var label = new Label {
            Text = "This is the second page",
            Font = Font.SystemFontOfSize (36),
        };

        Content = new StackLayout {
            Spacing = 30,
            VerticalOptions = LayoutOptions.Start,
            Children = {
                label,
            }
        };
    }
}
```

However we have no way to navigate back and forth between this new page and our first page. Two changes are required:

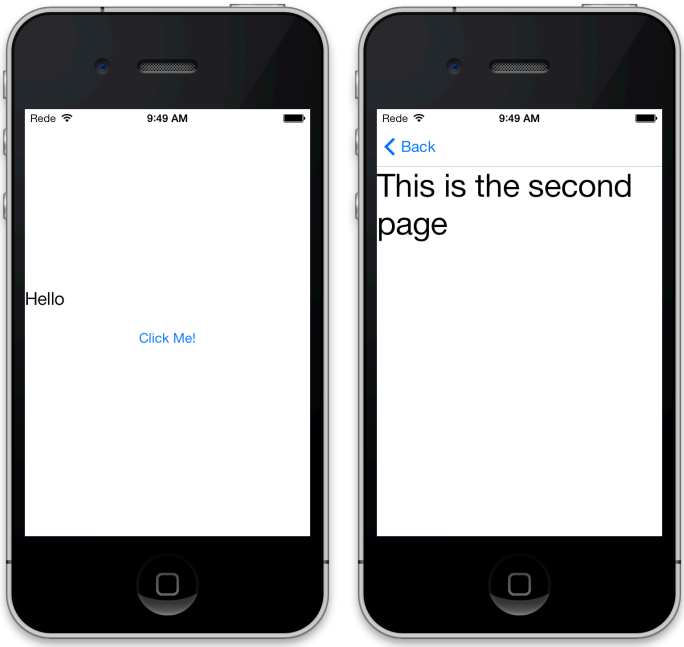
1) In our App class, add a NavigationPage which *contains* the first Page we want to display.

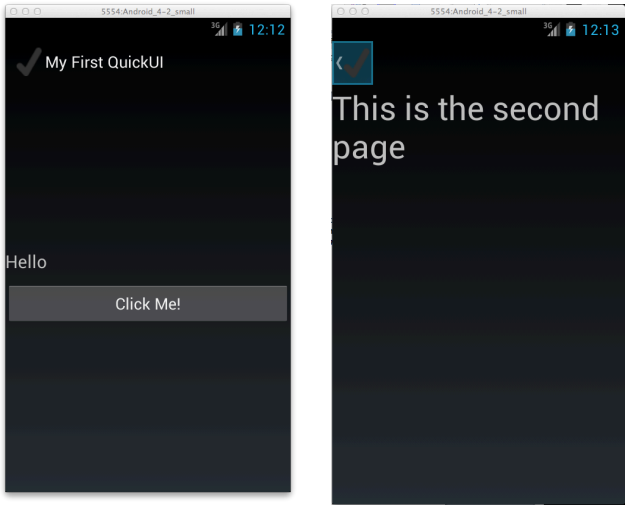
```
public static Page GetMainPage ()
{
    return new NavigationPage (new MyFirstPage ());
}
```

2) In our button Clicked event handler, instead of incrementing a count, let's show the second page. This is done using ContentPage's Navigation property, which will always contain an INavigation implementation *if the page is contained within a NavigationPage*.

```
button.Clicked += (s, e) => Navigation.Push(new MySecondPage());
```

The screenflow for our simple app now looks like this:





It's difficult to see, but the first page now has an empty NavigationBar at the top (we could have set the text using `Title = "My First Xamarin.Forms";`). Clicking on the button now shows the second page, which also has a **Back** button automatically added to return to the first page.

Layouts

One of the biggest pain points in Xamarin.Forms is understanding the difference between managed and unmanaged layouts.

A managed layout has the following aspects:

- It observes the box model
- It will at some point during a Layout cycle call on each of its children child.Bounds = someRectangle;
- It will respect LayoutOptions, WidthRequest/HeightRequest of their children
- It will *ignore* setting the Width and Height properties directly (use the Request properties instead)

An unmanaged layout has the following aspects:

- It is up to the user to respect the box model, though it will behave as a good citizen as the child of a managed layout.
- The user will in some fashion be responsible for directly setting the bounds of the children. (in AbsoluteLayout you just assign the bounds before adding, in RelativeLayout you pass a lambda which calculates it)
- They largely ignore LayoutOptions and WidthRequest/HeightRequest of their children

Managed Layouts:

- StackLayout,
- GridLayout,
- ContentView,
- ScrollView,
- ContentPage (though not technically a layout, it has a content which is a view and it is managed)

Unmanaged Layouts:

- AbsoluteLayout,
- RelativeLayout

StackLayout

StackLayout works in a similar fashion to the StackPanel in WPF or the LinearLayout in Android: each child element is positioned one after the other like steps in a ladder. The orientation can be set to horizontal or vertical.
A simple StackLayout might look like this:

```
public class MyStackLayout : ContentPage
{
    public MyStackLayout ()
    {
        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,
            VerticalOptions = LayoutOptions.Center,
            Children = {
                red, yellow, green
            }
        };
    }
}
```

By default the controls are laid out vertically in the order they are added to the `Children` collection, as shown:

We can change the Orientation (and the VerticalOptions) with the following code:

```
    VerticalOptions = LayoutOptions.End,
    Orientation = StackOrientation.Horizontal,
    HorizontalOptions = LayoutOptions.Start,
```

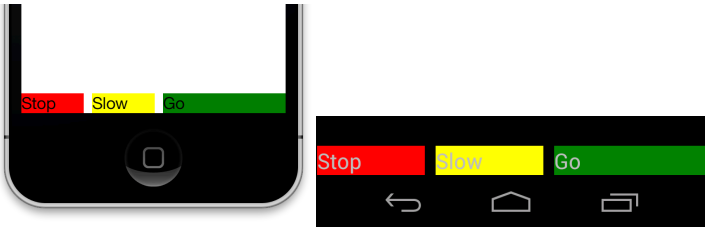
to achieve this effect:

See the controls reference for more information about the effect `LayoutOptions` has on the `StackLayout`'s contents. The important thing to note is that we did not specify any X, Y coordinates or Width, Height values to position the controls – `StackLayout` uses all the screen area it has available and positions the controls in a stack according to the required orientation and layout options specified.

Because `StackLayout` is responsible for laying out all the controls in the layout, you cannot specify an *explicit* size for the controls. You can set `WidthRequest` and `HeightRequest`, which the layout engine will try to honor when positioning the controls. The following code and screenshot shows the effect `WidthRequest` has on the previous sample:

```
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
    }
};
```

Notice how the ratio between the width requests has also been honored (100, 100, 200)



In StackLayout you should never set the Width and Height properties directly – always use WidthRequest and HeightRequest, then let the StackLayout determine the final values.

The flexibility of the StackLayout makes it very useful for cross-platform applications – Xamarin.Forms apps will run on a large variety of screen sizes and a control that can adapt to different sizes dynamically helps you to build screens with confidence that they will work on all those devices. Because the child controls are arranged in a stack, they cannot overlap each other but will always appear adjacent.

AbsoluteLayout

By contrast, the AbsoluteLayout behaves more like positioning controls in iOS (without using constraints) or the old style Windows Forms. Each control requires coordinates and dimensions in order to be displayed on the screen – failure to specify these values means the control will not be rendered.

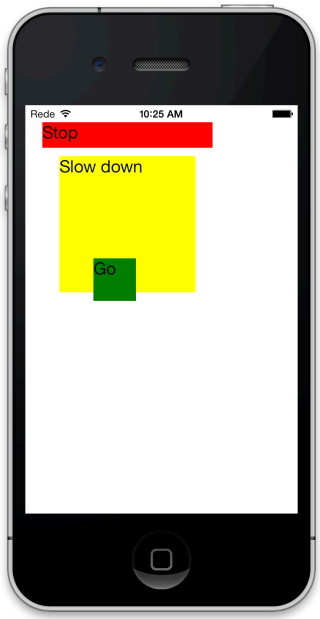
Unlike the StackLayout, AbsoluteLayout lets you set the WidthRequest and HeightRequest properties to accurately size each control. Then when adding the controls to the Children collection use the position parameter to set its location in the layout. You can also supply an optional AbsoluteLayoutFlags value to specify the values are proportional rather than absolute.

A simple AbsoluteLayout might look like this:

```
public class MyAbsoluteLayout : ContentPage
{
	public MyAbsoluteLayout ()
	{
		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 (yellow, new Point (40, 60));
            absLayout.Children.Add (green, new Point (80, 180));

            Content = absLayout;
        }
}
```

The resulting screen is shown below. Each control is positioned and sized exactly according to the values specified. This allows for very precise positioning of controls on the screen, however you must then be careful to test this screen on all target screensizes.



AbsoluteLayout lets you overlap controls, which is impossible with the StackLayout.

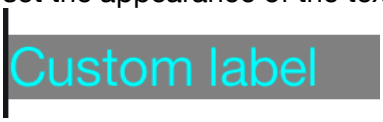
Note that the order the controls are added to the `Children` collection affects the Z-index of elements on screen – the first control appears at the 'bottom' of the Z-index and subsequent controls are added higher, meaning they can overlap (as the green label does in this example). Care must be taken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

Control Basics

In earlier examples we were introduced to the Label and Button controls so that we could learn about Pages and Layouts. This section will cover those two controls and introduce a few more to help build simple Xamarin.Forms apps – many more controls are available and covered in the controls reference documentation.
Note: the X, Y, Width, and Height properties are not discussed in this section. Remember you must specify these values if using an `AbsoluteLayout`; or one of the Relative-To methods if using a `RelativeLayout`. Use `WidthRequest` and `HeightRequest` in managed layouts like `StackLayout`.

Label

The label is a read-only text display control with a number of optional properties to set the appearance of the text.




```
var label = new Label {
    Text = "Custom label",
    Font = Font.SystemFontOfSize (20),
    TextColor = Color.Aqua,
    BackgroundColor = Color.Gray,
    IsVisible = true,
    LineBreakMode = LineBreakMode.WordWrap
};
```

Button

Buttons are an important way of receiving user input. In addition to altering the display attributes you can also wire-up the `Clicked` event for when the button is touched.


```
var button = new Button { Text = "Click Me!" };
int i = 1;
button.Clicked += (s, e) => button.Text = "You clicked me: " + i++;
```

You should keep in mind that buttons render very differently on each platform

Entry

The Entry control is a simple text-input. You can set the display attributes of the text content as well as the initial value for the control, or else a placeholder value prior to any user input.


```
var nameEntry = new Entry {
    Text = "",
    BackgroundColor = Color.Gray,
    IsPassword = false,
    Keyboard = Keyboard.Default,
    Placeholder = "task name",
    TextColor = Color.Black
};
```

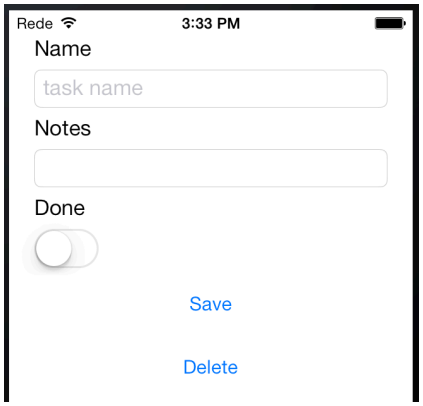
Switch

The switch control is a visual representation of a boolean – it can be in either an on or off state.

```
var doneSwitch = new Switch ();
```

Example Screen

We can combine the controls we've seen so far into this example ContentPage:



The code uses a StackLayout as shown below:

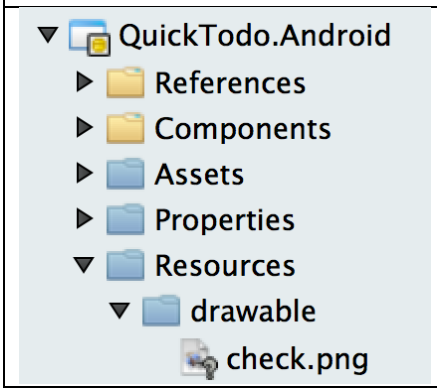
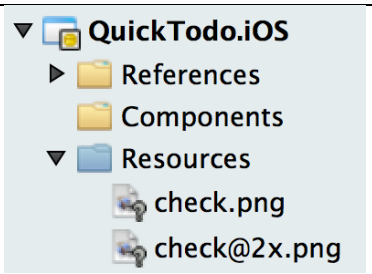
```
public class TodoItemPage : ContentPage
{
    string name, description;

    public TodoItemPage ()
    {
        var nameLabel = new Label { Text = "Name" };
        var nameEntry = new Entry { Placeholder = "task name" };

        var notesLabel = new Label { Text = "Notes" };
        var notesEntry = new Entry ();

        var doneLabel = new Label { Text = "Done" };
        var doneSwitch = new Switch ();

        var saveButton = new Button { Text = "Save" };
        var deleteButton = new Button { Text = "Delete" };

        Content = new StackLayout {
            VerticalOptions = LayoutOptions.StartAndExpand,
            Padding = new Thickness(20),
            Children = {nameLabel, nameEntry,
                notesLabel, notesEntry,
                doneLabel, doneSwitch,
                saveButton, deleteButton}
        };
    }
}
```

Image

There are a number of different ways to bring an image into your Xamarin.Forms app. The simplest way is to add an image file to your platform projects and reference it in the Xamarin.Forms. This mechanism also means you can actually use a different image on each platform (if desired).

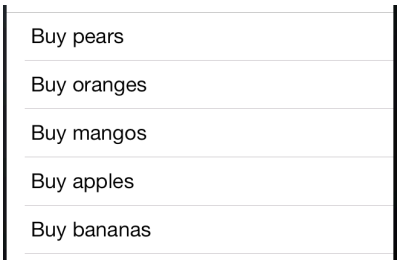
```
var tick = new Image {
    Source = FileImageSource.FromFile ("check.png"),
};
```

This image file exists in each platform project, including a retina version in iOS. Remember that some of the target filesystems are case-sensitive

Build Action: Android Resource	Build Action: Bundle Resource
	

ListView

ListView presents a scrolling list of items – it has a few properties you can set (most importantly the list of objects to display) but the actual appearance of the list depends mainly on the ItemTemplate. By default each cell will use a simple text rendering (the default row style on each underlying platform).
This is an example of a simple ListView using the default ItemTemplate (ie. no ItemTemplate was specified). Just for interest, the default is a TextCellTemplate class.



The code for this ListView is shown below, using a simple array of strings to populate the data.

```
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}
};
```

Built-in Cell Layouts

There are a number of built-in ViewCells that you can use in conjunction with a ListView. To specify one of the built-in options and databind to it, use the following syntax:

```
listView.ItemSource = new string [] { "Buy pears", "Buy oranges", "Buy
mangos", "Buy apples", "Buy bananas" };
listView.ItemTemplate = new DataTemplate (typeof (TextCell)){
    Bindings = {
        { TextCell.TextProperty, new Binding (".") }
```

```
    }
};
```

Because we've set the ItemSource to an array of strings, when the list view is being rendered each string is passed to a cell as its BindingContext. We can create a binding to the string like this `SetBinding (Label.TextProperty, ".")` – the dot-syntax means "use the bound object as the data source, do not query for a property on the object".

Other built-in cells include EntryCell, ImageCell and TextCellStyle. The available property bindings are easily explored using autocomplete on each Type.

Custom ViewCell

To customize the appearance of a `ListView`, create `ViewCell` subclasses to set as the `ItemTemplate`.
Creating a cell is very similar to creating a page – create the controls and add them to a layout. Finally set the layout to `ViewCell.View` property (as opposed to the `Content` property of `ContentPage`).

This simple cell contains a label and an image of a 'tick'. To create this cell use the following code (the SetBinding method will be explained in the next section):

```
public class TodoItemCell : ViewCell
{
    public TodoItemCell ()
    {
        var label = new Label {
            YAlign = TextAlignment.Center
        };
        label.SetBinding (Label.TextProperty, "."); // show the object
that's the data source for the row
        var tick = new Image {
            Source = FileImageSource.FromFile ("check"),
        };
        var layout = new StackLayout {
            Padding = new Thickness(20, 0, 0, 0),
            Orientation = StackOrientation.Horizontal,
            HorizontalOptions = LayoutOptions.StartAndExpand,
            Children = {label, tick}
        };
        View = layout;
    }
}
```

Once you've created a cell, you can use it in your ListView by creating a Data Template using your ViewCell subclass as shown in the following code. This snippet creates a new list view, sets the ItemSource to an array of strings, and then sets the ItemTemplate to be used for each cell to the class listed above.

```
var listView = new ListView {
    RowHeight = 40
};
```

```
listView.ItemSource = new string [] { "Buy pears", "Buy oranges", "Buy
mangos", "Buy apples", "Buy bananas" };
listView.ItemTemplate = new DataTemplate (typeof (TodoItemCell));
```

Because we've set the ItemSource to an array of strings, when the list view is being rendered each string is passed to a cell as its BindingContext. Within the cell definition we created a binding to the string like this `SetBinding (Label.TextProperty, ".")`.

More Data Binding

If the collection of objects that you assign to a list view's ItemSource, you can bind to different properties in the cell. For example, if we consider this simple TodoItem class

```
public class TodoItem
{
    public TodoItem () {}
    public int ID { get; set; }
    public string Name { get; set; }
    public string Notes { get; set; }
    public bool Done { get; set; }
}
```

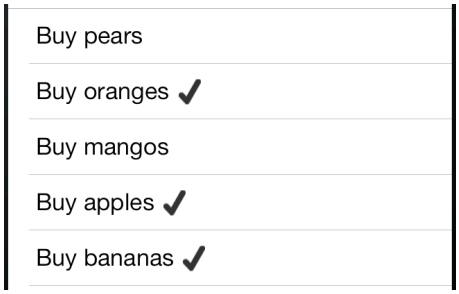
We could then initialize the `ItemSource` with a collection of `TodoItem` objects (instead of an array of strings)

```
listView.ItemSource = new TodoItem [] {
    new TodoItem {Name = "Buy pears"},
    new TodoItem {Name = "Buy oranges", Done=true},
    new TodoItem {Name = "Buy mangos"},
    new TodoItem {Name = "Buy apples", Done=true},
    new TodoItem {Name = "Buy bananas", Done=true}
}
```

In the `TodoItemCell`, instead of binding the Label to "." each property is bound to a specific TodoItem property (Name or Done). To create a binding you call the `SetBinding` method on a control, specifying the property whose value should change, and the name of the source-data

```
var label = new Label {
    YAlign = TextAlignment.Center
};
label.SetBinding (Label.TextProperty, "Name");
var tick = new Image {
    Source = FileImageSource.FromFile ("check"),
};
tick.SetBinding (Image.IsVisibleProperty, "Done");
```

The list will now appear like this – notice that the 'tick' image is only visible where the `Done` property is set to true.



You can bind to a variety different properties on Xamarin.Forms controls, including TextProperty, IsVisibleProperty, XProperty, YProperty and many more.

Navigating from a ListView

So far we've built two screens: a list and a data entry screen. A button can push a new screen inside a NavigationPage using this code – a simple lambda that calls the Push method

```
button.Clicked += (s, e) => Navigation.Push(new MySecondPage());
```

To get the list view to open the data entry page (displaying information from the selected cell) implement the `ItemSelected` event as shown below. The second parameter (e) has a Data property that contains the object bound to the selected cell. The event handler casts this object to the correct type; instantiates an instance of the data entry page (TodoItemPage) then sets its `BindingContext` and pushes the page onto the navigation stack.

```
listView.ItemSelected += (sender, e) => {
    var todoItem = (TodoItem)e.Data;     // gets the object for the row
that was touched
    var todoPage = new TodoItemPage();
    todoPage.BindingContext = todoItem;     // binds the object to the UI
    Navigation.Push(todoPage);         // shows the next page
};
```

Because we already set-up data binding on the `TodoItemPage`, this code will let the list view open the detail view populated with data from the object associated with the cell we touched.

Debugging

Of course Xamarin.Forms lets you debug with breakpoints, watches and the Immediate window just like any Xamarin or .NET application. If you want to dump information to the console while you are running the app you can use the System.Diagnostics.Debug.WriteLine method, as shown:

```
using System.Diagnostics;
// some code
Debug.WriteLine("written to console");
```

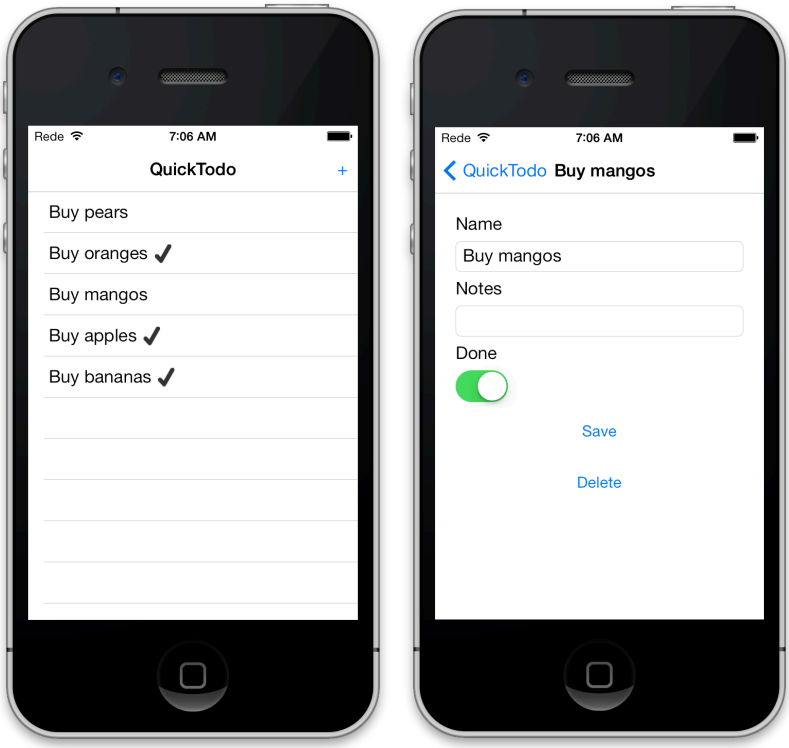
Writing platform-specific code in Xamarin.Forms

For example, the Todo app needs a button that lets the user create *new* todo items, which can be done by adding a toolbar item to the top of the screen. The following code shows how a ToolbarItem is created and added to the ToolBarItems collection on the page.

The highlighted if statement creates a different ToolbarItem for the Android platform. On the Android platform an image called “plus” will be displayed for the add button – we learned earlier how to include an image in the platform specific projects.

```
var tbi = new ToolbarItem ("+", null, () => { // by default, add button
will be a plus "+"
    var todoItem = new TodoItem();
    var todoPage = new TodoItemPage();
    todoPage.BindingContext = todoItem;
    Navigation.Push(todoPage);
}, 0, 0);
if (Device.OS == TargetPlatform.Android) { // Android will use image called
"plus"
    tbi = new ToolbarItem ("+", "plus", () => {
        var todoItem = new TodoItem();
        var todoPage = new TodoItemPage();
        todoPage.BindingContext = todoItem;
        Navigation.Push(todoPage);
    }, 0, 0);
}
ToolbarItems.Add (tbi); // adds the button to the page
```

This screenshot shows the working screens – although we do not have a database yet, the code lets use add and edit todo items that are stored in an in-memory list.



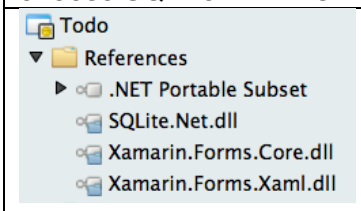
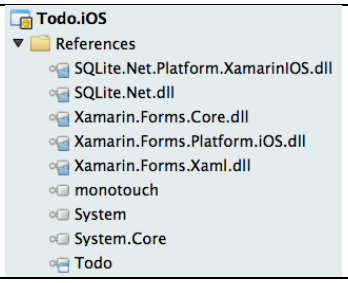
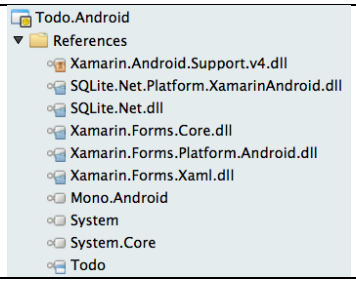
Adding a database

Replacing the in-memory List<TodoItem> storage with a database requires a few steps:

1) add the SQLite.NET PCL NuGet so that we can write data-access code using the lightweight ORM syntax of SQLite.net
2) annotate our TodoItem class so it can be stored in a SQLite database
3) add a TodoItemManager class with CRUD methods
4) instantiate the SQLiteConnection (different on each platform!)

Include SQLite.NET NuGet

For each project in the solution right-click and select **Manage NuGet Packages...** and search for the relevant NuGet to install. Refer to the Using NuGet in Xamarin Studio documentation for information on adding the NuGet add-in.

Xamarin.Forms PCL project search for SQLite.NET choose **SQLite.NET PCL**	*iOS project* search for SQLite.NET choose **SQLite.NET PCL xamarinios**	*Android project* search for SQLite.NET choose **SQLite.NET PCL xamarinandroid**
		

The NuGet packages will add the relevant PCL and platform-specific assemblies to each project.

Annotate the model

To store objects using SQLite.NET, annotate the class with SQLite.NET attributes such as `PrimaryKey` and `AutoIncrement` as shown below.

```
using SQLite.Net.Attributes;
public class TodoItem
{
    public TodoItem (){}
    [PrimaryKey, AutoIncrement]
    public int ID { get; set; }
    public string Name { get; set; }
    public string Notes { get; set; }
    public bool Done { get; set; }
}
```

There are many optional attributes that let you set the column name and other database settings. See the SQLite.NET docs for more info.

Build the data access

Now let's create the data access methods required to create, read, update and delete. This class exists in the Xamarin.Forms PCL along with all our other code – however it's constructor requires an SQLiteConnection which we have to instantiate before the actual database interactions can work: we'll cover that a bit later.

```
using SQLite.Net;
public class TodoItemDatabase
{
    static object locker = new object ();
    SQLiteConnection database;
    public TodoItemDatabase (SQLiteConnection conn)
    {
        database = conn;
        // create the tables
        database.CreateTable<TodoItem>();
    }

    public IEnumerable<TodoItem> GetItems ()
    {
        lock (locker) {
            return (from i in database.Table<TodoItem>() select
i).ToList();
        }
    }

    public TodoItem GetItem (int id)
    {
        lock (locker) {
            return database.Table<TodoItem>().FirstOrDefault(x => x.ID ==
id);
        }
    }

    public int SaveItem (TodoItem item)
    {
        lock (locker) {
            if (item.ID != 0) {
                database.Update(item);
                return item.ID;
            } else {
                return database.Insert(item);
            }
        }
    }

    public int DeleteItem(int id)
    {
        lock (locker) {
            return database.Delete<TodoItem>(id);
        }
    }
}
```

Wire up the data access

To use the TodoItemDatabase we're going to create a single instance of it that can be accessed from the rest of the application. This will allow our list and detail screens to access the database to display and save objects. The following code in the Xamarin.Forms App class implements a `SetDatabaseConnection` that allows a SQLiteConnection to be passed to the App and stored for later use.

```
static SQLite.Net.SQLiteConnection conn;
static TodoItemDatabase database;
public static void SetDatabaseConnection (SQLite.Net.SQLiteConnection
connection)
{
    conn = connection;
    database = new TodoItemDatabase (conn);
}
public static TodoItemDatabase Database {
    get { return database; }
}
```

Now we can add the following code to our list page to retrieve all the TodoItems from the database in the list page, override the OnAppearing method

```
protected override void OnAppearing ()
{
    base.OnAppearing ();
    listView.ItemSource = App.Database.GetItems ();
}
```

In the detail page (TodoItemPage) we'll implement the button Clicked event to retrieve the object that is data-boudn to the user interface and save or delete it, as shown here:

```
var saveButton = new Button { Text = "Save" };
saveButton.Clicked += (sender, e) => {
    var todoItem = (TodoItem)BindingContext;
    App.Database.SaveItem(todoItem);
    Navigation.Pop();
};

var deleteButton = new Button { Text = "Delete" };
saveButton.Clicked += (sender, e) => {
    var todoItem = (TodoItem)BindingContext;
    App.Database.DeleteItem(todoItem.ID);
    Navigation.Pop();
};
```

These few lines of code let us read, save and delete from the device's SQLite database via SQLite.Net.

Create the SQLite connection

Until now, every line of code we've written since the bootstrap code has been in the Xamarin.Forms PCL – we've built the user interface and the data access code using SQLite.NET in completely sharable code.

However the SQLite database engine that is present on each device requires a file-path that specifies where the database file is saved. We cannot write this code in the PCL because each platform has slightly different filesystem characteristics. The SQLite.Net code itself also needs some platform-specific implementation to be passed to the PCL library.

iOS AppDelegate class

The new lines of code are highlighted in the iOS app's Finished Launching method below:

```
public override bool FinishedLaunching (UIApplication app, NSDictionary
options)
{
    Xamarin.Forms.Init ();
    // create a new window instance based on the screen size
    window = new UIWindow (UIScreen.MainScreen.Bounds);

    var sqliteFilename = "TodoSQLite.db3";
    string documentsPath = Environment.GetFolderPath
(Environment.SpecialFolder.Personal); // Documents folder
    string libraryPath = Path.Combine (documentsPath, "..", "Library"); //
Library folder
    var path = Path.Combine(libraryPath, sqliteFilename);

    // configure the platform specific implementation for SQLite.NET
    var plat = new SQLite.Net.Platform.XamarinIOS.SQLitePlatformIOS();
    var conn = new SQLite.Net.SQLiteConnection(plat, path);

    // Set the database connection string
    App.SetDatabaseConnection (conn);

    window.RootViewController = App.GetMainPage ().CreateViewController ();
    window.MakeKeyAndVisible ();
    return true;
}
```

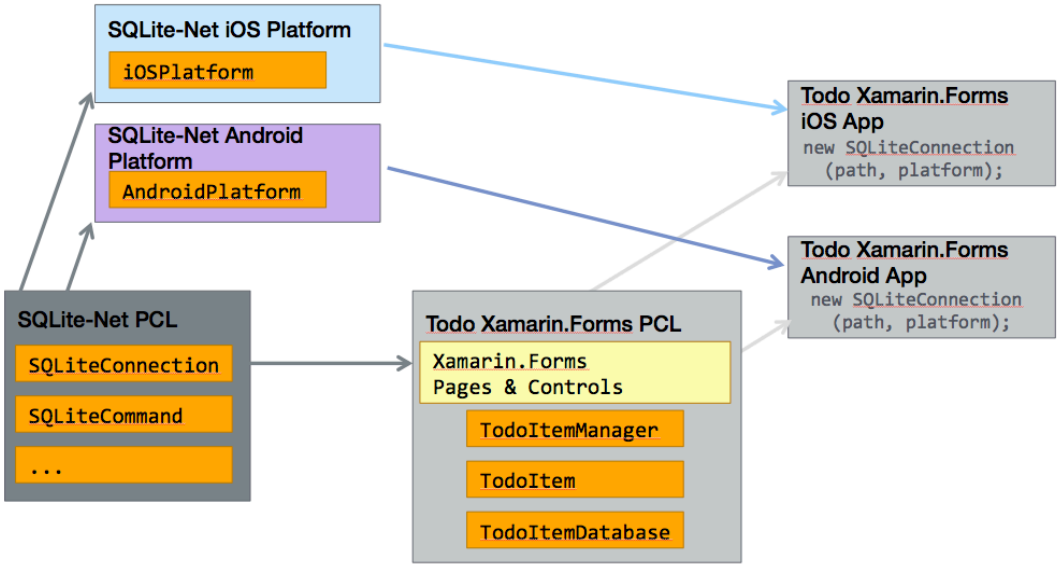
Android Main activity class

The changes to the main Android activity are similar, as shown here:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

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

    var sqliteFilename = "TodoSQLite.db3";
    string documentsPath = System.Environment.GetFolderPath
(System.Environment.SpecialFolder.Personal); // Documents folder
    var path = Path.Combine(documentsPath, sqliteFilename);

    // configure the platform specific implementation for SQLite.NET
    var plat = new
SQLite.Net.Platform.XamarinAndroid.SQLitePlatformAndroid();
    var conn = new SQLite.Net.SQLiteConnection(plat, path);

    // Set the database connection string
    App.SetDatabaseConnection (conn);

    SetPage (App.GetMainPage ());
}
```

This diagram visualizes the assembly references that have been set-up to get SQLite.Net working in the Xamarin.Forms app. The SQLite-Net PCL is referenced by the Xamarin.Forms App PCL, and each application project (in addition to referencing

the Xamarin.Forms app) also references the platform-specific SQLite-Net assembly. These things are wired up by creating the SQLiteConnection object in the platform project and passing it into the Xamarin.Forms App project.



Taking advantage of platform-features

Xamarin.Forms is a scaffolding framework that enables developers to quickly and easily build cross-platform applications that are fully native apps, with native UI widgets, native performance and access to all the underlying features of each platform's SDK.

First define an interface the expresses the API that the feature should have. You could also build an abstract class to provide some shared behavior.

```
public interface ITextToSpeech
{
    void Speak (string text);
}
```

You then need to provide a way for the platform-specific application code to provide an implementation. This can be as simple as passing an implementation to the Xamarin.Forms App class as shown here:

```
static ITextToSpeech TextToSpeech;
public static void SetTextToSpeech (ITextToSpeech speech)
{
    TextToSpeech = speech;
}
public static ITextToSpeech Speech {
    get { return TextToSpeech; }
}
```

The iOS and Android apps will need to call the SetTextToSpeech method with before attempting to use this feature.

Now we can use this method in our code - we can read an individual todo item like this on the TodoItem page:

```
var speakButton = new Button { Text = "Speak" };
speakButton.Clicked += (sender, e) => {
    var todoItem = (TodoItem)BindingContext;
    App.Speech.Speak(todoItem.Name + " " + todoItem.Notes);
};
```

It could be even more useful if we added speech to the main todo list. By adding a new data access method called GetItemsNotDone we can have the app read out only the tasks in the list that haven't been marked as done. This method also demonstrates that it's possible to use SQL syntax to create queries against the SQLite database.

```
public IEnumerable<TodoItem> GetItemsNotDone ()
{
    lock (locker) {
        return database.Query<TodoItem>("SELECT * FROM [TodoItem] WHERE
[Done] = 0");
    }
}
```

The implementation of the "speak tasks that haven't been done" button looks like this (it uses a question mark as the button icon):

```
var tbi2 = new ToolbarItem ("?", null, () => {
    var todos = App.Database.GetItemsNotDone();
    var tospeak = "";
    foreach (var t in todos)
        tospeak += t.Name + " ";
    if (tospeak == "") tospeak = "there are no tasks to do";
    App.Speech.Speak(tospeak);
}, 0, 0);
ToolbarItems.Add (tbi2);
```

That's all we can do in the shared Xamarin.Forms App PCL.

Xamarin.iOS

Implement this class in the iOS project

```
public class Speech : ITextToSpeech
{
    public Speech () {}
    float volume = 0.5f;
    float pitch = 1.0f;
    public void Speak (string text)
    {
        var speechSynthesizer = new AVSpeechSynthesizer ();
        var speechUtterance = new AVSpeechUtterance (text) {
            Rate = AVSpeechUtterance.MaximumSpeechRate/4,
            Voice = AVSpeechSynthesisVoice.FromLanguage ("en-US"),
```

```
            Volume = volume,
            PitchMultiplier = pitch
        };
        speechSynthesizer.SpeakUtterance (speechUtterance);
    }
}
```

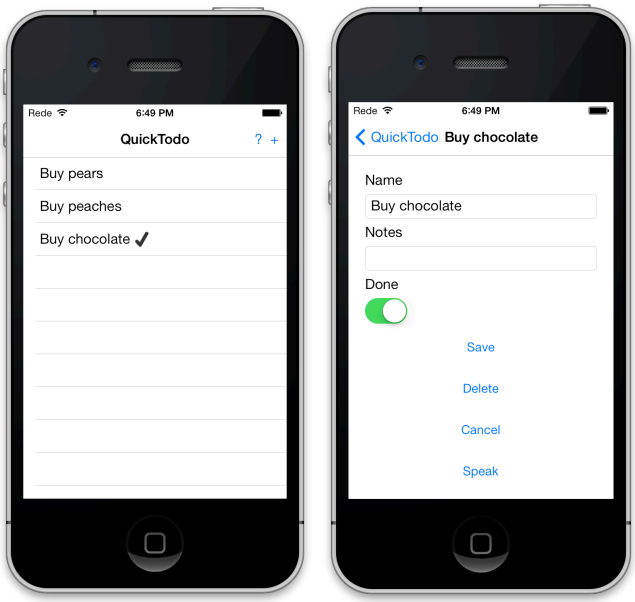
and then instantiate it in the AppDelegate and pass to the Xamarin.Forms app.

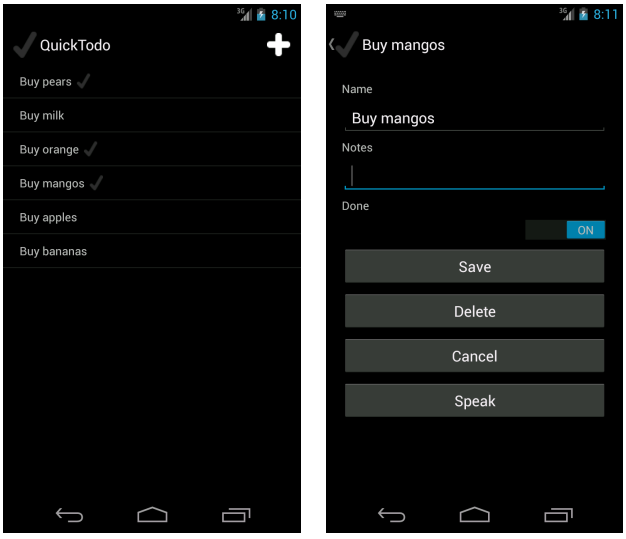
```
App.SetTextToSpeech (new Speech ());
```

Xamarin.Android

Android's speech API requires an Activity Context, so it's been implemented using a platform-specific renderer instead. These are introduced later in the document – see the code in the Android project `SpeakButtonRenderer` class.

The app now has speech buttons on both screens, using the native text-to-speech APIs offered by Apple and Google!





Clicking on the "?" button in the first screen will say "buy pears buy peaches", and clicking on the "Speak" button in the second screen will say "buy chocolate".

Using Xamarin.Forms.Maps

Maps are a special control in Xamarin.Forms – they are packaged in a separate NuGet that you only need to install if you plan to display a map. To add support for maps in Xamarin.Forms:

1) Add a reference to the Google Play Services component to your Android application project.
2) Add the Xamarin.Forms.Maps NuGet to your Xamarin.Forms PCL and all your platform projects (Android, iOS and Windows Phone).
3) In the startup method on each platform, call the maps `Init` method
   `Xamarin.FormsMaps.Android.Xamarin.FormsMaps.Init (this, bundle);  // Android`
   `Xamarin.FormsMaps.iOS.Xamarin.FormsMaps.Init (); // iOS`

The code for a simple map display requires the following classes:

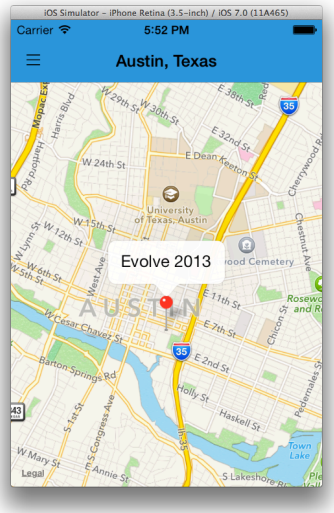
- **Map** – a Map user interface control.
- **MapSpan** – a way to represent the viewport for the map, using center coordinates and degrees of view.
- **Position** – a class to represent latitude and longitude coordinates.
- **MapType** – an enum to select whether to show street maps, satellite imagery or a composite of both.
- **Pin** – you can optionally add pins to a map to highlight specific locations.

This isn't an exhaustive list of the Map control, but the code below will produce a working map display with a pin.

```
public class MapPage : ContentPage
{
	public MapPage ()
	{
		NavigationPage.SetHasNavigationBar (this, true);

		Title = "Austin, Texas";

		/* DON'T FORGET
		 * Xamarin.FormsMaps.Init ();
		 */
		var map = new Map(new MapSpan(new Position(30.26535, -
97.738613), 0.05, 0.05))
		{
			MapType = MapType.Street,
			HeightRequest = 508
		};

		Pin pin;
		map.Pins.Add(pin = new Pin()
			{
				Label = "Evolve 2013",
```

```
                        Position = new Position(30.26535, -
97.738613),
                        Type = PinType.Place
                    });

            Content = new StackLayout {
                  VerticalOptions = LayoutOptions.StartAndExpand,
                  Children = {map}
            };
      }
}
```

The map sample is part of the **Evolve13** sample app, shown here:



Setting up Maps in Android

To come… Requires key and Google Play Services Component

File Operations in the Xamarin.Forms PCL

To come… use PCLStorage Nuget.

Accessing Platform-Specific APIs

There are a number of different options to access platform-specific APIs from within Xamarin.Forms code.

- **Dependency Injection** – the simplest mechanism is simply
- **Platform Renderers** – Xamarin.Forms allows you to write your own platform-specific code to render controls defined and used in the Xamarin.Forms PCL.

These two methods are discussed below.

Dependency Injection

A simple example of accessing a platform-specific API is the text-to-speech capability in the **Todo** sample.

Firstly, create an Interface in the PCL

```
public interface ITextToSpeech
{
    void Speak (string text);
}
```

and then add a mechanism to inject and store a reference to an implementation of this Interface. The following code is added to the App class – it provides a method for the ITextToSpeech implementation to be passed in from the application

```
static ITextToSpeech TextToSpeech;
public static void SetTextToSpeech (ITextToSpeech speech)
{
    TextToSpeech = speech;
}
public static ITextToSpeech Speech {
    get { return TextToSpeech; }
}
```

Now it's really easy to code against the interface anywhere in the Xamarin.Forms PCL app,

```
if (tospeak == "") tospeak = "there are no tasks to do";
App.Speech.Speak(tospeak);
```

The interface is then implemented for each platform (where possible – sometimes features are not available on all devices). For iOS, the implementation is shown below:

```
public class Speech : ITextToSpeech
{
       public Speech ()
       {
       }

       float volume = 0.5f;
       float pitch = 1.0f;
       /// <summary>
       /// Speak example from:
```

```
        /// http://blog.xamarin.com/make-your-ios-7-app-speak/
        /// </summary>
        public void Speak (string text)
        {
                var speechSynthesizer = new AVSpeechSynthesizer ();

                var speechUtterance = new AVSpeechUtterance (text) {
                        Rate = AVSpeechUtterance.MaximumSpeechRate/4,
                        Voice = AVSpeechSynthesisVoice.FromLanguage ("en-US"),
                        Volume = volume,
                        PitchMultiplier = pitch
                };

                speechSynthesizer.SpeakUtterance (speechUtterance);
        }
}
```

And the final step – create and set the implementation inside AppDelegate:

```
App.SetTextToSpeech (new Speech ());
```

The **Speak** button can now trigger the iOS-specific text-to-speech engine.

Platform-specific Renderers for Xamarin.Forms Controls

The Evolve13 sample app has a simple example of how to build a platform-specific renderer. It overrides the rendering of a Button control so that it can interact with the iOS-specific Twitter API, allowing the user to send a Tweet from Xamarin.Forms code.

On the `SpeakerPage` (where the Twitter button is displayed) we create a subclass of Xamarin.Forms.Button:

```
public class TweetButton : Button
{
    public string Tweet;
    public static string TwitterHashtag = "#xamarin";
}
```

and then we reference that new subclass in our Xamarin.Forms code

```
tweetButton = new TweetButton {
    BackgroundImage = "tweet.png",
    Text = "Tweet",
};
tweetButton.SetBinding (Button.IsVisibleProperty, "HasTwitter");
```

The last step on the `SpeakerPage` is to override `OnBindingContextChanged` so the subclassed Button's `Tweet` property can be set to the text to be tweeted.

```
protected override void OnBindingContextChanged ()
{
    base.OnBindingContextChanged ();
    var speaker = BindingContext as Speaker;
```

```
    tweetButton.Tweet = speaker.TwitterHandle + " " +
TweetButton.TwitterHashtag;
    Debug.WriteLine ("Tweet: "+tweetButton.Tweet);
}
```

Now we need to add the platform-specific code to alter (or re-implement) how this specific Button class is rendered.

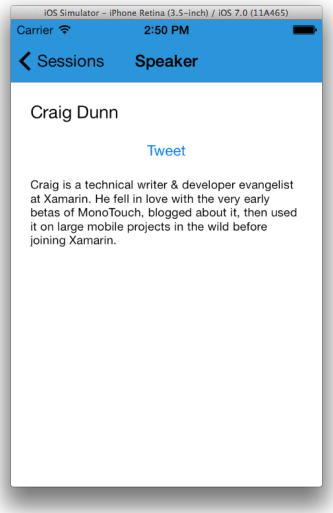
In the iOS project create a class for the TweetButtonRenderer. The complete code is shown below, followed by an explanation of what's happening:

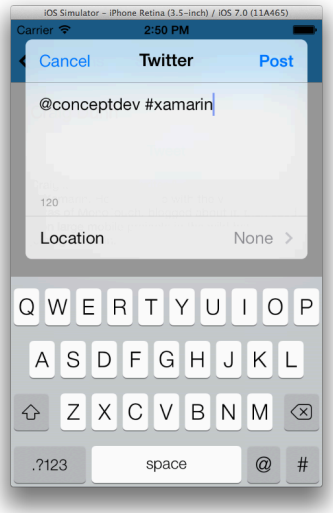
```
[assembly:ExportRenderer(typeof(Evolve13.TweetButton),
typeof(Evolve13.iOS.TweetButtonRenderer))]


public class TweetButtonRenderer : ButtonRenderer
{
	protected override void OnModelSet (VisualElement view)
	{
		base.OnModelSet (view);

		var tweetButton = view as TweetButton;

		var button = Control as UIButton;

		button.TouchUpInside += (object sender, EventArgs e) => {
			var tweetController = new
TWTweetComposeViewController();
			tweetController.SetInitialText (tweetButton.Tweet);

			var parentview = button.Superview;

	parentview.Window.RootViewController.PresentModalViewController(twee
tController, true);
		};
	}
}
```

Firstly, notice the `assembly:ExportRenderer` attribute – this is how we notify the Xamarin.Forms engine that the `Evolve13.TweetButton` (the subclass we created in the PCL) should be rendered with *this* class `Evolve13.iOS.TweetButtonRenderer` (instead of the built-in ButtonRenderer).

The `TweetButtonRenderer` class subclasses `ButtonRenderer`, allowing certain methods to be overridden to customize how the control is rendered for this platform. In this example we are not modifying the rendering of the button, merely implementing its `TouchUpInside` event so that we can call some native iOS APIs – the `TWTweetComposeViewController`.

The `TouchUpInside` method itself presents a native iOS Tweet view, thus allowing the Xamarin.Forms app to perform custom, platform-specific behavior on iOS. The Tweet button and the native iOS Twitter input are shown below:


iOS Simulator - iPhone Retina (3.5-inch) / iOS 7.0 (11A465)
Carrier
2:50 PM
Sessions
Speaker
Craig Dunn
Tweet
Craig is a technical writer & developer evangelist at Xamarin. He fell in love with the very early betas of MonoTouch, blogged about it, then used it on large mobile projects in the wild before joining Xamarin.


iOS Simulator - iPhone Retina (3.5-inch) / iOS 7.0 (11A465)
Carrier
2:50 PM
Cancel
Twitter
Post
@conceptdev #xamarin
120
Location
None
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
.?123
space
@
#

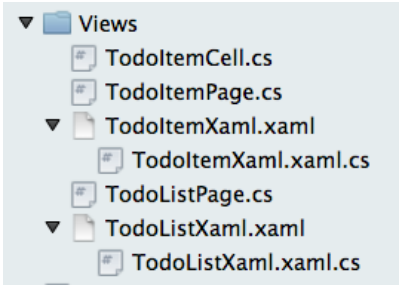
Using XAML for layout

Xamarin.Forms also supports XAML markup as a way to express the layout of a page. XAML works much the same way as it does in WPF and Windows RT.

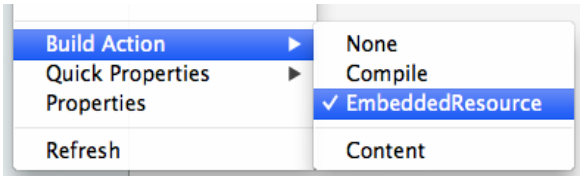
XAML pages consist of a

- **.xaml file** that contains the markup.
- **.xaml.cs** file that contains a partial class/code with user-generated (ie. your) code. You MUST call `InitializeComponent` in the constructor – this method gets generated from the XAML markup and stored in the .xaml.g.cs file.
- **.xaml.g.cs** file (that you can't see, because it's generated in the /obj/ folder) that is the rest of the partial class, containing the declarations of the objects in the XAML markup as well as the implementation of the `InitializeComponent` method. This class is generated by the compiler and should not be edited.

The xaml files can live with your other code in your Xamarin.Forms PCL project (this screenshot is from the **TodoXaml** sample app).



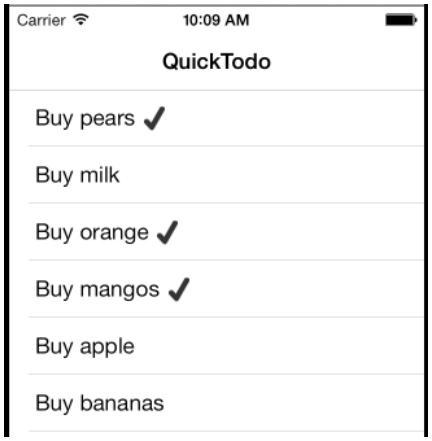
The .xaml file MUST be set to have **Build Action: EmbeddedResource**.



The Todo application we built earlier in the document is also available with Xaml views. The C# code in the Xaml version of the application is much the same, however the layout is now done using familiar XML-based syntax with databinding support.

ListView XAML

This list view in the TodoXaml sample app has been defined with XAML rather than C# code as shown previously – as you'd expect it is indistinguishable from the hand-coded version.



The XAML that creates this screen is shown below. The `x:Class` attribute is important – this is how the compiler knows where to look for the partial class that defines the behavior for this XAML layout.

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/Xamarin.Forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Todo.TodoListXaml"
    Title="Todo">
    <ContentPage.Content>
        <ListView x:Name="listView" ItemSelected="OnItemSelected">
               <ListView.ItemTemplate>
                 <DataTemplate>
                   <ViewCell>
                     <StackLayout Padding="20,0,0,0"
HorizontalOptions="StartAndExpand" Orientation="Horizontal">
                           <Label Text="{Binding Name}" YAlign="Center" />
                           <Image Source="check" IsVisible="{Binding Done}"
/>
                       </StackLayout>
                   </ViewCell>
                 </DataTemplate>
               </ListView.ItemTemplate>
             </ListView>
    </ContentPage.Content>
</ContentPage>
```

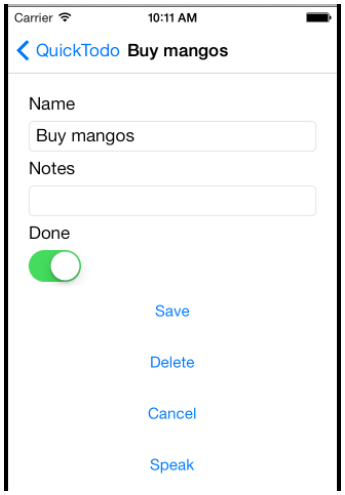
The associated ‘code behind’ is shown below – it must be a partial class whose name matches the value used in the XAML `x:Class` attribute. The `OnAppearing` method is identical to the earlier version of the app, however we have moved the ‘row touched’ code from an `ItemSelected` event handler into a method that has been wired up in the XAML. Note the method signature is important when implementing methods that are referenced in XAML – the compiler will only wire-up methods that have the exact signature that it expects.

```
public partial class TodoListXaml : ContentPage
{
       public TodoListXaml ()
       {
              InitializeComponent (); // required
       }
```

```
        protected override void OnAppearing ()
        {
                base.OnAppearing ();
                listView.ItemSource = App.Database.GetItems ();
        }
        public void OnItemSelected (object sender, EventArgs e) {
                var todoItem = ((EventArg<object>)e).Data as TodoItem;
                var todoPage = new TodoItemPage();
                todoPage.BindingContext = todoItem;
                Navigation.Push(todoPage);
        }
}
```

Input form XAML

The screen that consists of user-input fields also looks the same whether defined in XAML or C#



The XAML for this layout is shown here:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/Xamarin.Forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Todo.TodoItemXaml">
    <ContentPage.Content>
       <StackLayout VerticalOptions="StartAndExpand">
               <Label Text="Name" />
               <Entry Text="{Binding Path=Name}" x:Name="nameEntry"
Placeholder="task name" />

        <Label Text="Notes" />
               <Entry Text="{Binding Path=Notes}" x:Name="notesEntry" />

               <Label Text="Done" />
               <Switch IsToggled="{Binding Path=Done}" x:Name="DoneSwitch"
/>

               <Button Text="Save" Clicked="OnSaveClicked" />

                <Button Text="Delete" Clicked="OnDeleteClicked" />
```

```
            <Button Text="Cancel" Clicked="OnCancelClicked" />

            <Button Text="Speak" Clicked="OnSpeakClicked" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The code-behind differs mainly in the way the button Clicked events are wired up – instead of using an anonymous method each method is explicitly declared and referenced in the Xaml.

```
public partial class TodoItemXaml : ContentPage
{
	public TodoItemXaml ()
	{
		InitializeComponent (); // required
	}
	public void OnSaveClicked (object sender, EventArgs e)
	{
		var todoItem = (TodoItem)BindingContext;
		App.Database.SaveItem(todoItem);
		this.Navigation.Pop();
	}
	public void OnDeleteClicked (object sender, EventArgs e)
	{
		var todoItem = (TodoItem)BindingContext;
		App.Database.DeleteItem(todoItem.ID);
		this.Navigation.Pop();
	}
	public void OnCancelClicked (object sender, EventArgs e)
	{
		this.Navigation.Pop();
	}
	public void OnSpeakClicked (object sender, EventArgs e)
	{
		var todoItem = (TodoItem)BindingContext;
		App.Speech.Speak(todoItem.Name + " " + todoItem.Notes);
	}
}
```

See the Forums thread
http://forums.xamarin.com/discussion/11279/Xamarin.Forms-xaml-gets-started/p1
for more information on Xaml in Xamarin.Forms.

You can use either Xaml or code to define your layouts with Xamarin.Forms. All of Xamarin.Forms's features, including databinding, are available regardless of which method you use.

Adding XAML to a Xamarin.Forms Project

The beta versions of Xamarin.Forms do not yet have full IDE support for some features… including adding new XAML files to a Xamarin.Forms project (there is a Xamarin Studio Add-in in the works – stay tuned).

Until then, you'll need to add the XAML file and its associated 'code-behind' manually. Add two new files, one with a .xaml extension and one with a .xaml.cs extension, with content similar to that shown below:

MyPage.xaml

```
<?xml version="1.0" encoding="UTF-8" ?>
 <ContentPage
     xmlns="http://xamarin.com/Xamarin.Forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     x:Class="Demo.MyPage">
     <ContentPage.Content>
         <Label Text="Hello, World!" x:Name="mylabel"/>
     </ContentPage.Content>
 </ContentPage>
```

MyPage.xaml.cs

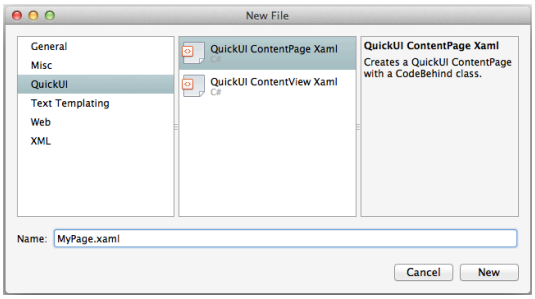
```
namespace Demo {
     public partial class MyPage : ContentPage
     {
         public MyPage ()
         {
             InitializeComponent ();
         }
     }
}
```

MAKE SURE you right-click on the .xaml file and choose **Build Action: EmbeddedResource**.

The `x:Class` namespace and classname (eg. `Demo.MyPage`) must match the C# file's namespace and class name.

The IDE will probably highlight the `InitializeComponent` method as an error (since the implementation won't exist until after you first attempt to compile). After a successful compile the `.xaml.g.cs` generated file will have been created in the /obj/ folder and the warning should disappear.

In future you'll easily be able to add Xaml to your project via **Add > New File…**



(to come)

What if you want to do Mvvm?

Xamarin.Forms supports both the MVVM and the MVC model for creating applications. Both MVVM and MVC provide a series of pros and cons.

MVC:

- More commonly understood
- Simpler application logic flow
- Can require considerable effort to maintain views in sync with backend services

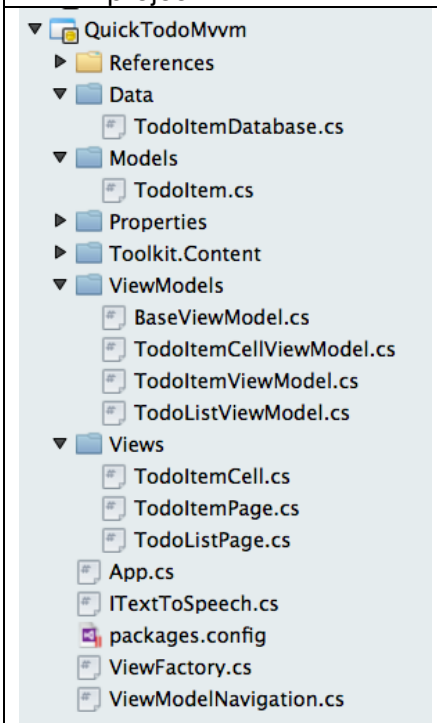
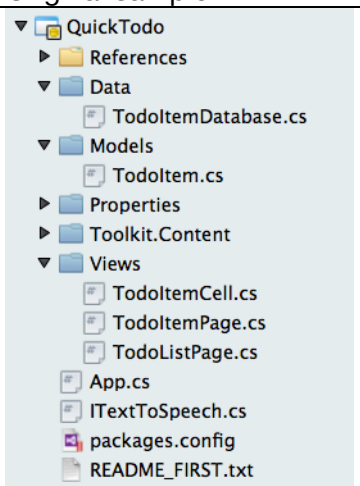
MVVM:

- Less commonly understood
- More complex application logic
- Implementation of navigation is non-trivial
- Automatic and incredibly simple synchronization between view and backend

In general the approach we tend to take is MVVM with a slight leak of the abstraction around Navigation to help simplify.

- Views are subclasses of a ContentPage
- ViewModels are subclasses of a common BaseViewModel
  - BaseViewModel implements Navigation and INPC
  - INavigation is bound into VM OneWayToSource
- Standard naming is Foo and FooViewModel

A basic Mvvm pattern has been implemented in the **TodoMvvm** sample. The project structure is shown here, compared to the original **Todo** sample – the key difference is the addition of the ViewModel classes.

Mvvm project	Original sample
	

How does the ViewModel affect the code?

In the original sample (*without Mvvm*), the following code appears in the `TodoListPage` – we assign data and wire up event handlers directly in the view (Page class).

```
listView.ItemSource = App.Database.GetItems ();
listView.ItemSelected += (sender, e) => {
	var todoItem = (TodoItem)e.Data;
	var todoPage = new TodoItemPage();
	todoPage.BindingContext = todoItem;
	Navigation.Push(todoPage);
};
```

In the Mvvm solution, rather than assigning the `ItemSource` and `ItemSelected` directly, they are bound to properties of a `ViewModel` class

```
listView.SetBinding (ListView.ItemSourceProperty, "Contents");
listView.SetBinding (ListView.SelectedItemProperty, new Binding
("SelectedItem", BindingMode.TwoWay));
```

And the ViewModel class implements them (along with property changed notifications) like this:

```
ObservableCollection<TodoItemCellViewModel> contents = new
ObservableCollection<TodoItemCellViewModel> ();
public ObservableCollection<TodoItemCellViewModel> Contents {
	get { return contents; }
	set
	{
		if (contents == value)
			return;
		contents = value;
		OnPropertyChanged ();
	}
}
object selectedItem;
public object SelectedItem
{
	get { return selectedItem; }
	set
	{
		if (selectedItem == value)
			return;
		// something was selected
		selectedItem = value;
		OnPropertyChanged ();
		if (selectedItem != null) {
		var todovm = new TodoItemViewModel
(((TodoItemCellViewModel)selectedItem).Item);
			Navigation.Push (ViewFactory.CreatePage (todovm));
			selectedItem = null;
		}
	}
```

```
}
```

The view model implements logic without being tied to any specific user interface elements. It exposes properties and commands that views (ie. Xamarin.Forms Pages) can bind to in order to access functionality. The code for a view should therefore be minimal and limited to creating and arranging user interface elements – with all logic being called via bindings to the view model.

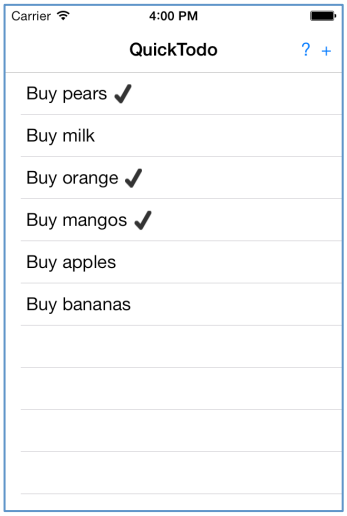
This difference is even more pronounced if you are building your views with Xaml, since almost no view code should be required at all – the Xaml syntax supports binding controls and commands directly to view model classes.

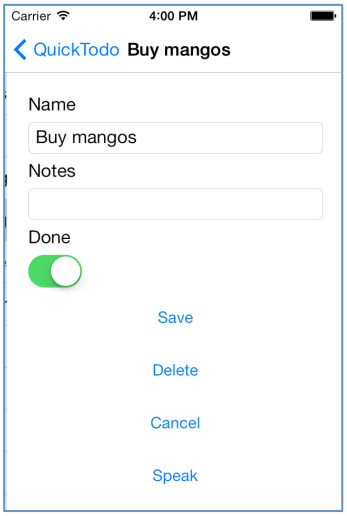
Stats & Screenshots (for fun)

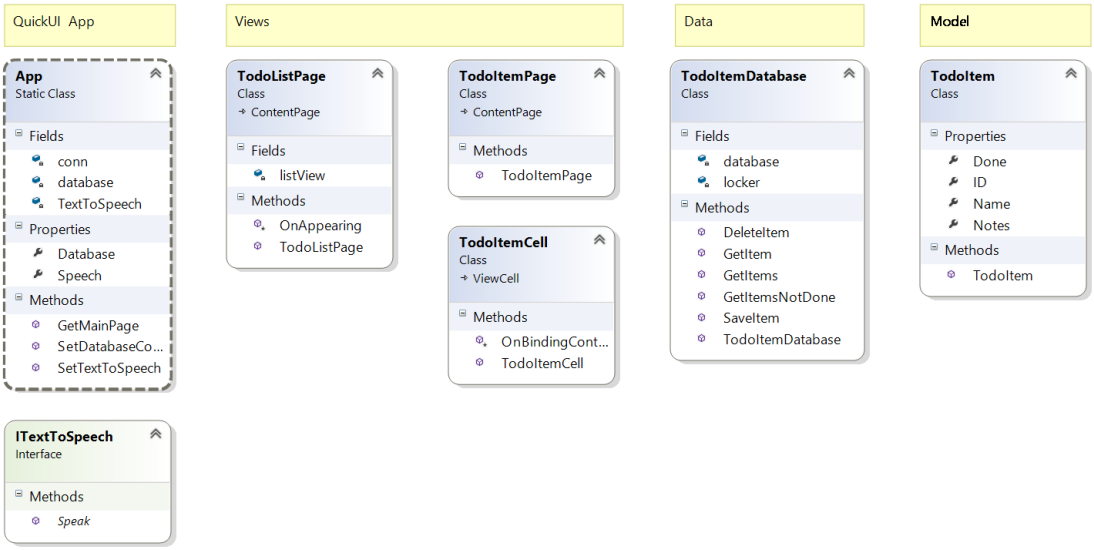
Todo

(85% is common, even taking boilerplate into account)

PCL 153 lines of C# (obviously a small app)
iOS 26 lines of C# (bootstrap boilerplate + speech API)
Android 26 lines of C# (bootstrap boilerplate + speech API)

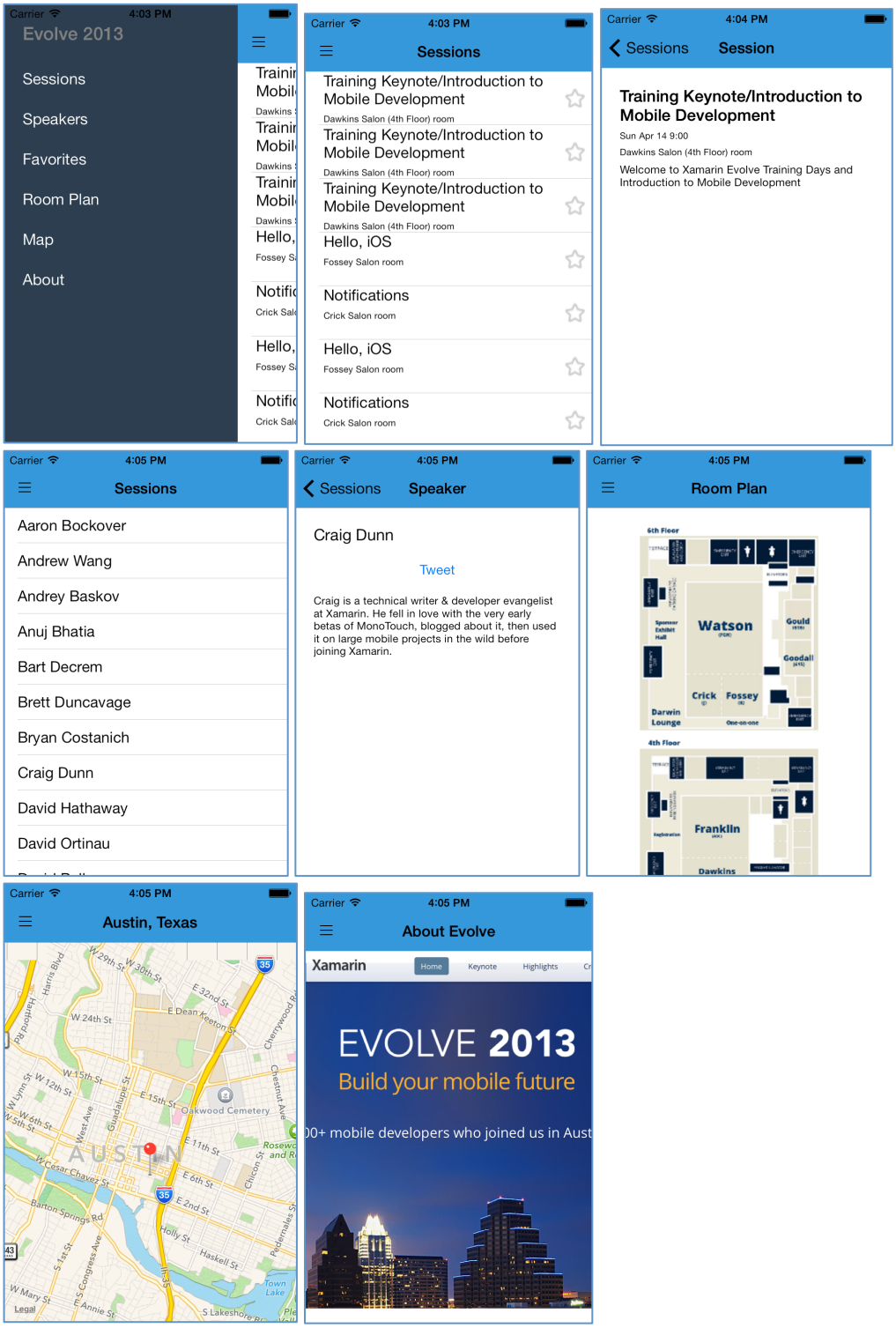






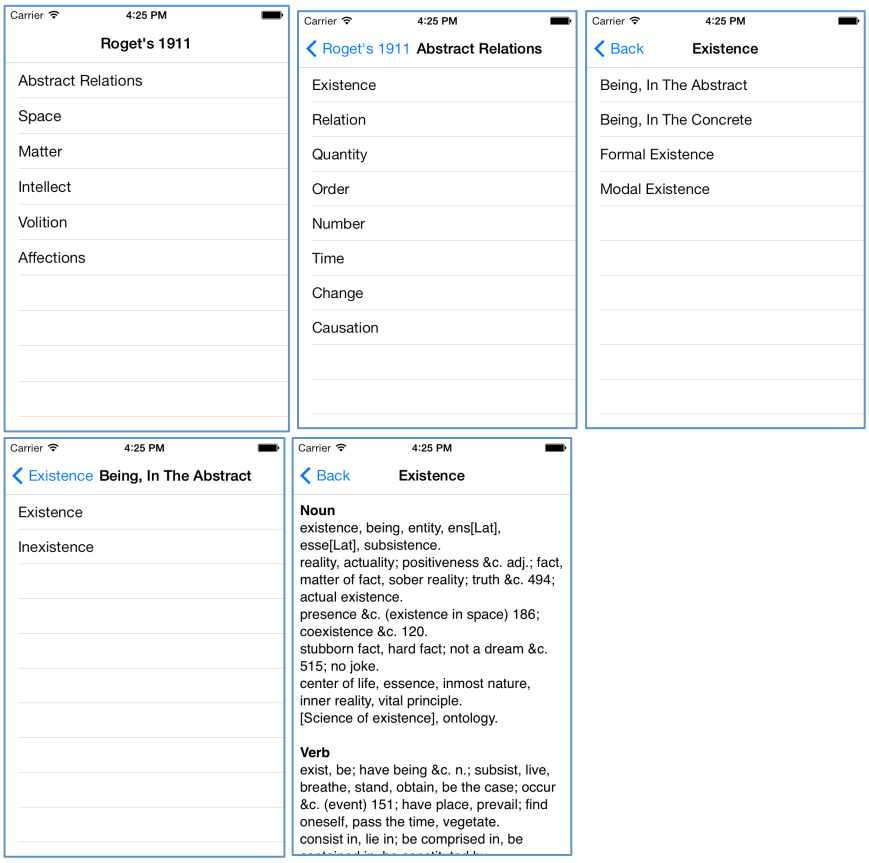
Evolve13

(83% is common, even taking boilerplate into account)
PCL 230 lines of C# (obviously a small app)
iOS 44 lines of C# (bootstrap boilerplate + twitter & ListViewRenderer)
Android 34 lines of C# (bootstrap boilerplate + twitter & UITableViewRenderer)



Roget1911

(87% is common, even taking boilerplate into account)
PCL 208 lines of C# (obviously a small app)
iOS 30 lines of C# (bootstrap boilerplate + XML filecopy)
Android 30 lines of C# (bootstrap boilerplate + XML filecopy)



References

Introduction to Cross Platform Mobile Development
Introduction to Portable Class Libraries
Cross Platform Data
Using NuGet with Xamarin Studio