Xamarin.Forms

Succinctly

by Derek Jensen

Syncfusion | Technology Resource Portal

Xamarin.Forms Succinctly

By
Derek Jensen

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Jeff Boenig
Copy Editor: Graham High, content producer, Syncfusion, Inc.
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Derek Jensen is a lifetime nerd who calls the Chicagoland area home. He is your typical family man who loves activities with his wonderful wife and three amazing children. He is a huge sports buff (both playing and coaching) and loves football, baseball, and basketball. He loves just about any technology he can get his hands on, but spends most of his time on the server side of the world. Over the last several years he has been blessed to have the time to dig deeply into Xamarin and create a number of written tutorials as well as online video course material. His day job consists of writing highly transactional backend web services, data access layers, and n-tier applications using the .NET Framework in C# for Redbox. During the evenings he enjoys helping others write cross-platform mobile applications with Xamarin, learning more about mobile technology, and sharing his findings with anyone who will listen via numerous online avenues.

Chapter 1 Introduction

In this chapter, we will be setting the foundational concepts as to why Xamarin, and ultimately Xamarin.Forms, is such a monumental step forward in the creation of cross-platform mobile applications. We'll learn about how Xamarin's approach to creating a framework differs from some of the other cross-platform frameworks on the market. We'll also discuss the myth of write once, run everywhere, and how that is truly not the goal we should be striving for. Finally, we will lay out the requirements needed to install Xamarin and begin creating applications using Xamarin.Forms.

In the beginning, there were several different manufacturers and carriers of mobile devices. They all had their view of what the mobile world should look like and consequently their own proprietary operating systems. As these manufactures began to release software development kits (SDKs) for their hardware and software, it became apparent to developers that this was going to be the future. The problem, inevitably, was choice. In order to target all of these platforms, developers needed to learn about the capabilities of different devices. They also needed to learn about different SDKs, development tools, as well as languages. While many were up for the adventure, the challenge proved too much for others.

Then there was a beacon of light. An idea that was so profound, that many believed it to be impossible. As time goes on and more and more people join the revolution, we find that it is not only possible; in many instances it is actually preferred. That beacon is Xamarin. And the light is Xamarin.Forms. This is where our story begins.

Before the revolution

Before Xamarin arrived at the party, there was a very clear demarcation of where mobile developers belonged. If you wanted to target the Android platform, you had to learn Java. If you wanted to target the iOS platform, you had to learn Objective-C. And if you dared to target the Windows Phone platform, you had to learn C#. There was no way around this separation. Breaking into this market was very exciting and daunting all at the same time. As a new mobile developer seeking guidance, you could ask around in person or check forums and blogs to see where you should start. It's at this point you begin to realize that the world of mobile development is very territorial. Not only are most people biased about the languages they write applications in, but they also hold biased opinions about platforms as well as device manufacturers and form factors. At this point, most people duck for cover and run the other way. Those who choose to stick around for a while are faced with one of the most daunting questions of all: Where to start?

When it comes to getting started in mobile development, most choose to start with a platform or language in which they feel some comfort or allegiance. Regardless of your choice, if you have never done mobile development in the past, this is an arduous uphill struggle. Creating applications for mobile devices is not like writing software to run on servers or even desktops for that matter. You are forced to work within the constraints of a smaller form factor as well as limited CPU, memory, I/O, and performance in general. Realizing that these constraints may seem like a step backwards in the evolution of technology, many developers choose to bail.

Sticking with the process of creating mobile applications is an incredibly rewarding experience. There is nothing like coming up with a “great” idea, starting on the journey of mobile development, and finally seeing your app in an app store available to the general public. Most would simply be happy with this and move on. What happens, though, if your app gets popular on that platform and it starts to generate some buzz? What do you do then? You can’t simply limit this wonderful creation to live on only one platform. But you just spent the last several months learning a new language and platform. You haven’t seen much sunlight and your family hardly recognizes you anymore. Wait, you did have a family right? For some, embarking on another adventure to learn another new language and platform is just too much to stomach again.

When it comes to attempting to take your mobile application to multiple platforms, the process is typically broken up into two separate camps. There are those who will once again lock themselves in the basement and battle through another agonizing spell of late night coding sessions and constant Internet searching through blogs, forums, SDK documents, tutorials, and maybe even some online training. While this may seem silly to some, it is definitely the way to go if you are more interested in expanding your knowledge and making yourself more marketable. The downside is that it can be a very long process. There are others who will pool together some money to hire someone with specific knowledge of the new platform who can probably get the job done much faster. This works well because the time to market is much quicker, but it costs money and ultimately you won’t know the quality of code in the application or if the platform-specific APIs are being used correctly. There is definitely a trade-off in either case.

This entire process can be even more daunting if you come from a Microsoft .NET background as I do. While the Windows Phone platform is gaining popularity, it is still in a distant third place in a three-horse race with iOS and Android. Outside of the platform view, the languages themselves are very different. C# and Java, while they share some similarities, have some very large differences in basic language functionality. At least they share some commonality with respect to garbage collection, so you don’t have to worry explicitly about memory management. And then there is Objective-C, which I’m convinced was created to simply be different from everything else, not to mention it takes a step backwards and requires you to handle all the memory management yourself. In later versions of the iOS platform, there is some semblance of garbage collection, but in the earlier days, this was not the case. Taking these differences into consideration, beginning with C# and the Windows Phone platform has definitely not been the popular choice whether you come from that world or not.

Xamarin enters the party

In mid-2011, Xamarin was formed and officially signed an agreement with Novell that stated Xamarin would fully retain rights to the Mono framework, as well as the new up-and-coming frameworks for writing iOS and Android apps using C#, MonoTouch and Mono for Android. At this point, not only did the world of mobile development open up for .NET developers around the world, but it also introduced a very exciting concept for mobile developers in general: cross-platform mobile applications.

The idea of creating mobile applications using a single language and framework was not necessarily a new one. Other frameworks emerging onto this scene were creating an abstraction layer on top of the platforms that hid the native APIs from the developer. This methodology had both positives and negatives. The positives included the ability to create mobile applications for multiple platforms in a single language (including HTML, JavaScript, CSS, Lua, etc.). This smoothed out the learning curve from a language perspective at least. The problem was, as developers, we didn't have direct access to the platform-specific APIs. We were at the mercy of the framework developers to further abstract the APIs within their frameworks and follow their guidelines. While this may not be a big issue in basic applications, when creating mobile applications of any size and complexity, this can be very limiting, especially when you want to target some specific functionality or even the hardware of the platform.

Xamarin took a distinctively different approach. Instead of creating its own APIs and translating them into the platform APIs, it simply created a mapping from its C# classes directly to the native platform APIs. That means that there is a **`UITableViewController`** class in C# that maps directly to the **`UITableViewController`** class in the iOS SDK. While this method of mapping existing platform-specific concepts to C# constructs requires a little more up-front effort for developers who have little or no knowledge of non-Windows mobile platforms, in the long run it provides a better understanding of the individual native platforms and provides the opportunity to write native applications in other languages should that be necessary.

Obviously, the advent of this technology that allowed developers to create applications that targeted multiple platforms using a single language was very exciting. Developers who were familiar with C# and the .NET Framework could now make a few changes to their existing mobile applications that targeted Windows Phone and be able to run that app on iOS and Android as well. Unfortunately, that isn't the case. This concept introduces one of the largest falsities in the realm of cross-platform mobile development tools: write once, run everywhere (WORE).

Write once doesn't run everywhere

One of the biggest misconceptions early on in the world of Xamarin, as well as other cross-platform development options, was that you would now be capable of writing a single application, in a single language, and just have it magically run on all the major mobile platforms. Unfortunately that is not the case. While Xamarin and other platforms allow you to write mobile applications in a language you are familiar with and target multiple platforms, there will always need to be customization. The reason is that because the major mobile platforms are so different, in order to take advantage of all the amazing features they have to offer, you will need to stray from your common code base.

Writing custom platform code within a cross-platform application is not a bad thing. This concept allows you, as a developer, to take advantage of not only the ability to share common code, but also to target those really cool platform-specific features that end users have come to expect. This gives you the freedom to make your application stand out in the vast sea of app stores and have a chance to gain some share in the marketplace. And as developers of mobile applications, that is what we strive for.

While customization of cross-platform applications is extremely important, it is a fine line to walk. On one hand, you need to be able to target those nice platform-specific features, but on the other hand you don't want to negate the fact that you are using Xamarin for the purpose of being able to share code across multiple platforms. To this point, you will often hear people talk about the percentage of shared code in their cross-platform applications. Depending on the type of application and how well it is written, it is not uncommon to hear shared code percentages in the 50%–60% ranges. Translated into a simple example, that means that instead of writing two different 10,000 line applications that total around 20,000 lines of code, you are now writing a single application that totals around 15,000 lines of code. In a small application such as this, that difference may not seem like a big deal, but in the typical software development lifecycle, you are saving development and testing time and ultimately getting your product to market faster. These are great benefits, but ultimately we are always trying to get better.

As software developers, and human beings in general, we are a competitive bunch. We always want to push the envelope and see how far we can go. Sure 50%–60% shared code is good, but how do we push it up to 90%? Can we get that far? How do we go further? These are natural questions that are raised and investigated by many as Xamarin is chosen as a platform for creating mobile applications. Until Xamarin 3 was publicly announced on May 28, 2014, the only options typically involved taking shortcuts to get away from some of the platform-specific features and become more generic.

Xamarin.Forms make a splash

The announcement of Xamarin 3 came with several very exciting features being added to the Xamarin platform. One of these features was Xamarin.Forms. Xamarin.Forms touted the ability to enable developers to create shared domain-level code as well as create shared UI code. This idea spat in the face of critics who said that there is no way to achieve true WORE applications. Up until this point, the only way to do that was to create a full abstraction layer that removed the developers' ability to write applications that run natively on devices and use native controls. While Xamarin.Forms doesn't provide full WORE capabilities, it takes a large portion of the issue out of the picture, the UI.

Xamarin.Forms continued to build on the solid foundation that Xamarin had already created. Since Xamarin.iOS and Xamarin.Android allow code to be written in C# and then mapped to native classes at runtime, Xamarin applications can be described as running natively on devices. Xamarin.Forms took this concept further by also allowing controls written in C# to be mapped to native controls at runtime as well. On top of that, at launch, Xamarin.Forms contained more than 40 out-of-the-box controls that could be shared across platforms, including iOS, Android, and Windows Phone. These 40 controls are broken down into four main categories:

- **`Page`** controls that represent an entire screen.
- **`Layout`** controls that are used to place UI controls into logical structures.
- **`View`** controls that are common UI controls.
- **`Cell`** controls that contain data placed in **`ListView`** and **`TableView`** controls.

These controls can be placed together in any configuration to provide a very flexible and customized UI to suit many needs. One concern may be that you need more than just these 40 controls. Or, what about customization? When it comes to creating your own controls or customizing those that come with Xamarin.Forms, you have complete control. You can create, customize, and tweak these controls until you have reached fully customized platform-specific UI control nirvana.

The development tools

Like learning any other skill, technology or otherwise, you will need to have the proper tools and Xamarin.Forms is no different. Before you can begin creating Xamarin apps in general, you will need to create an account at Xamarin.com and then proceed to download and install the software. In order to utilize the Xamarin.Forms toolset, you will need to meet both the development system requirements as well as the mobile operating system requirements.

Development system requirements

Xamarin.Forms applications, as well as Xamarin applications in general, can be written on both Mac systems and Windows systems. Depending on the development system you choose, there may be additional requirements or SDKs needed to fully support your applications.

Mac requirements

In order to develop with Xamarin.Forms on a Mac, you will need to use at least Xamarin Studio 5. If you plan on targeting the iOS platform for your Xamarin.Forms application, you will also need to install Xcode 5 or greater and OS X 10.8 (Mountain Lion) or newer.

If you are unfamiliar with Xcode at this point, don't worry as it isn't required to follow along with the examples in this book. Xcode is simply the integrated development environment (IDE) you would typically use to write Objective-C or Swift code to create applications that target the Apple platforms: iOS and Mac OS X. Since we are using Xamarin to create mobile applications, we only need access to Xcode for the compilation process.

Note: Windows Phone apps cannot be developed on OS X. Xamarin Studio solution templates for Xamarin.Forms do not include a Windows Phone project.

Windows requirements

To create Xamarin.Forms applications on a Windows machine, you will need to use either Xamarin Studio 5 or Visual Studio 2012 or later. Using the Portable Class Library (PCL) version of the Xamarin.Forms solution template requires Profile 78 that targets .NET 4.5, which can be installed for Visual Studio 2012. Visual Studio 2013 already supports Profile 78 and no further installation is required. To use the Shared Project solution template for Xamarin.Forms, you will need to use at least Visual Studio 2013 Update 2.

In the world of PCLs, the concept of profiles can be a little confusing. Each profile describes a certain combination of platforms that are supported by a PCL. In the case of Profile 78, support is built-in for .NET Framework 4.5, Windows 8, and Windows Phone Silverlight 8.

Note: Targeting Windows Phone will require you to install the Windows Phone SDK.

Note: Xamarin.Forms targets Windows Phone 8 and above. In order to develop for Windows Phone 8, you must use a Windows 8 PC.

Note: Targeting the iOS platform from a Windows machine requires local network access to a Mac with Xamarin and Xcode installed and configured as a Xamarin.iOS Build Host.

Mobile system requirements

When creating mobile applications with Xamarin.Forms, you must target one or more of the following mobile operating systems:

- Android 4.0 or higher.
- iOS 6.1 or higher.
- Windows Phone 8 or higher.

Note: Targeting Windows Phone will require you to install the Windows Phone Toolkit.

Summary

Over the course of this first chapter we covered a number of the non-technical aspects of cross-platform mobile development as well as Xamarin.Forms, including the following:

- Cross-platform mobile development before Xamarin.
- The different approaches of Xamarin and other frameworks.
- The introduction of Xamarin.Forms.
- Requirements for using Xamarin and Xamarin.Forms.

Chapter 2 Code Sharing

This chapter is going to set the foundation for being able to share the same code across multiple platforms without the headache of duplicated code. In order to truly take advantage of all that Xamarin and Xamarin.Forms has to offer, we need to understand not only what code sharing means, but also how to take advantage of the mechanisms that are offered. Yes, there will be trade-offs, but in the end we will possess the ability to make the all-important choice of code sharing solutions to optimize our code for cross-platform applications.

Overview

The concept of code sharing is vital to using Xamarin and Xamarin.Forms. Without the ability to share code effectively you are not fully realizing the value that Xamarin brings to the table. Structuring your code in such a way that enables you to maximize your percentage of shared code is what will ultimately allow you to reap the benefits of using Xamarin and Xamarin.Forms. Unfortunately, wanting to increase your percentage of shared code and actually doing so are two completely different things. Without a firm grasp of your options in the beginning, you can cause added complexity later in the development cycle.

At first glance, you may wonder just what the big deal is when it comes to sharing code. After all, you do it all the time in your day-to-day work in the .NET Framework. You create Class Library projects, put a bunch of code in there, and use this library not only in this project, but then share it with others for other uses. Unfortunately, that way of sharing code isn't going to work exactly the same when it comes to Xamarin.Forms applications.

If you have ever tried to reference a normal class library from a Xamarin.iOS or Xamarin.Android project, you are no doubt familiar with the following issue:

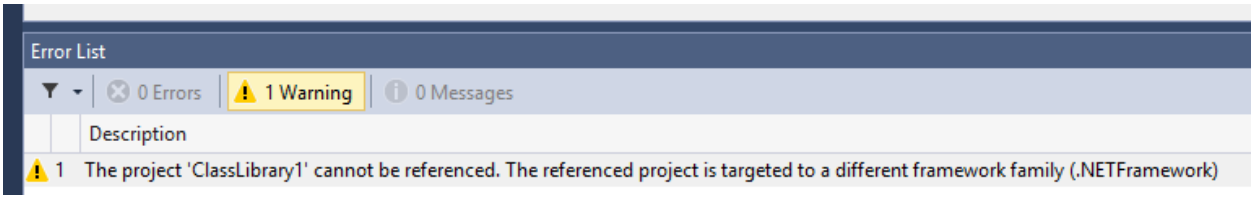


Figure 1: Incompatible target framework warning in Visual Studio 2013

The reason for this is that the versions of the .NET Framework that are used in Xamarin.iOS, Xamarin.Android, and Windows Phone are not full versions of the .NET Framework. They are smaller subsets designed to work more efficiently on mobile platforms. That being the case, we need a different way to add shared code from a single (or multiple) shared assemblies to these projects without the need to duplicate the code in a special project for each platform.

When it comes to sharing code within Xamarin.Forms applications, you have two primarily accepted methods:

- Portable Class Libraries
- Shared Projects

When choosing a shared code option, it is important to understand that neither one is “right”. Through the rest of this chapter you will come to better understand the differences between the two and begin to formulate an opinion on the benefits of one over the other in certain scenarios. Let’s take a look at these two options and see how they play a role in the use of Xamarin.Forms.

Portable Class Libraries

Portable Class Libraries (PCLs) are specialized versions of the normal **Class Library** project. PCLs are designed with the idea of sharing code across multiple projects that target different versions of the .NET Framework. When you are working with a PCL, you have the ability to configure it to target the frameworks that are appropriate for your application. To accomplish this, when creating a PCL, you are faced with some options that allow you to specify the lowest common denominator of functionality you would like it to support.

Creating a PCL

The process of creating a PCL is not overly difficult, but there are a few things to take into consideration. Let’s walk through the process.

Start by creating a new project in Visual Studio or Xamarin Studio and selecting the **Class Library (Portable)** template under **Visual C#** family. Give your new project a name of **Portable** and click **OK**.

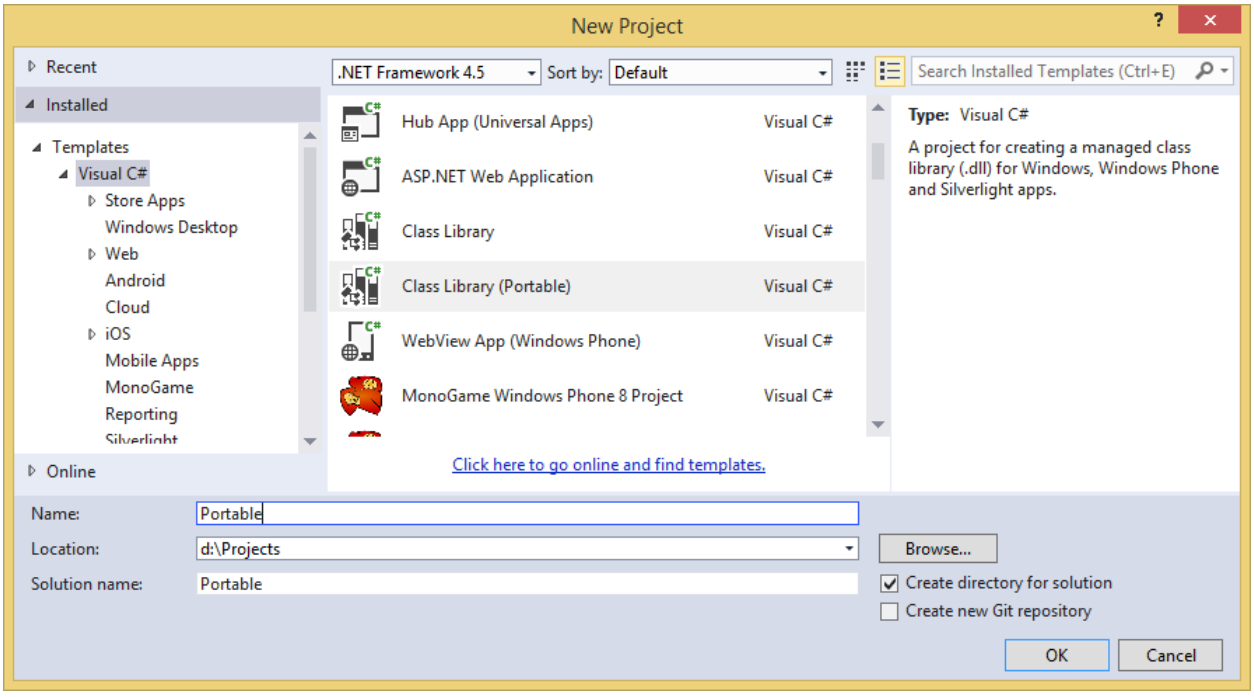


Figure 2: Creating a new PCL project in Visual Studio 2013

You will be presented with a new dialog box that will allow you to specify which platforms you would like this PCL to be compatible with.

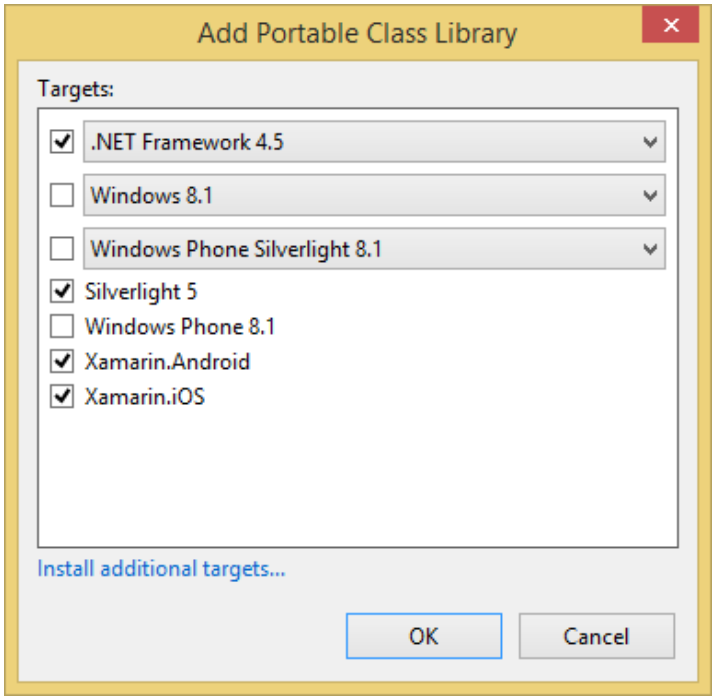


Figure 3: Dialog box for choosing PCL target frameworks

Note: Xamarin Studio does not present this dialog. You can specify which frameworks you are targeting under the Project Properties.

Depending on what version of Visual Studio you are using and what SDKs you have installed, your dialog box might look a little different, but the concepts are the same. From this dialog box you are able to choose which versions of the .NET Framework (and ultimately the functionality) you would like to support.

Tip: To get a full list of supported versions of the .NET Framework as well as functionality, visit the Cross-Platform Development with the PCL page on MSDN.

At first glance, you may want to select all of them just to be safe. An important concept to keep in mind when using PCLs is that the more target platforms you wish to support, the less functionality you receive. The reason for this is that in order to support more platforms, the lowest common denominator amongst them gets smaller. A good example of this is the **`HttpClient`** class that is used rather extensively for its ability to make HTTP requests over the network in a fully asynchronous manner. Since it is not supported in all versions of Windows Phone Silverlight, it will be left out of your PCL. This will require you to either write your own or obtain another version from the Internet or via NuGet.

At this point you may think that you will just not include any of the Windows Phone Silverlight platforms and go about your merry way. Unfortunately, that is impossible. As mentioned earlier, the current version of Xamarin.Forms requires Profile 78 which includes support for the following:

- .NET 4.5
- Windows 8.0
- Windows Phone Silverlight 8.0

Keeping with the prescribed profile for Xamarin.Forms means that you will need to look elsewhere for **`HttpClient`** support if your application needs it. Luckily there is a package on NuGet that will add support for this class in your PCL in Xamarin.Forms projects. With it, you can continue writing shared code for your Xamarin.Forms application.

Now that you have a grasp of what PCLs are and how they can be used in your Xamarin.Forms project, let's take a look at one.

Using a PCL

When it comes to actually using a PCL, the process is not that different from using a typical class library. To use the PCL you created in the last step, let's add a new project to the solution by right-clicking the solution name and selecting **Add > New Project**. From the **Add New Project** dialog, select the **Android** family on the left, choose the **Android Application** template, and click **OK**.

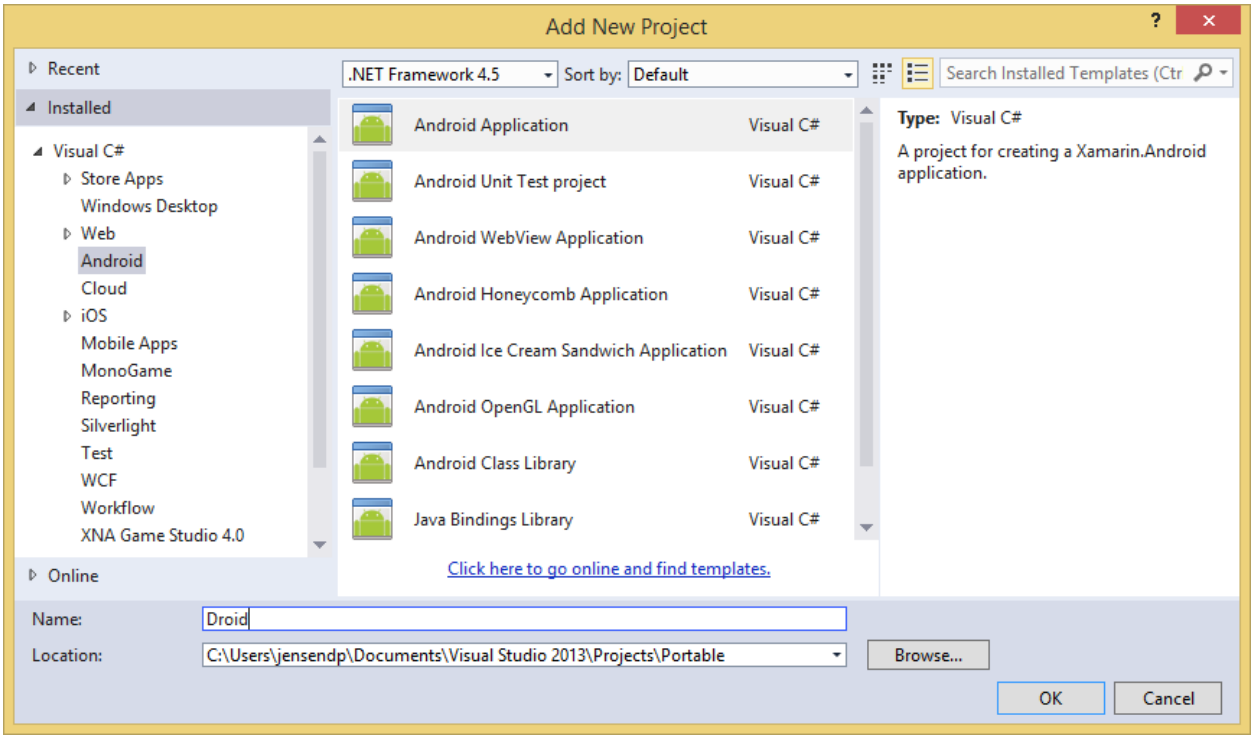


Figure 4: Adding a new Android project to the solution

Now you can add a reference from your Android project to your PCL. To do so, you will simply right-click on the **References** folder in your Android project and select **Add Reference** from the context menu. Once the **Add Reference** dialog box appears, select the **Solution** category on the left, select the check box next to the name of your PCL project, and click **OK**.

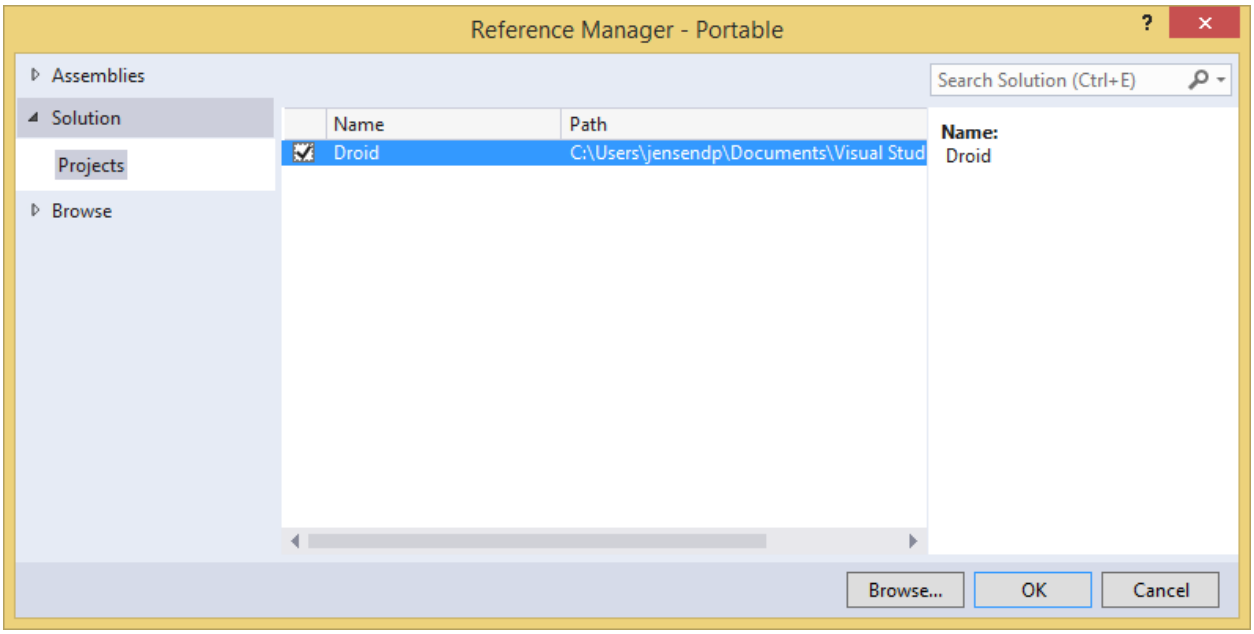


Figure 5: Adding a reference from the Android project to the PCL project

If you receive an error dialog when you click the **OK** button, or see a warning when you build your solution, you may have a mismatch in the versions of the .NET Framework you are targeting between your Android application and the PCL. To verify which versions you are targeting, you can right-click the PCL project name and select **Properties** from the context menu. At the bottom of the **Library** tab of the project properties, you will see a section named **Targeting**.

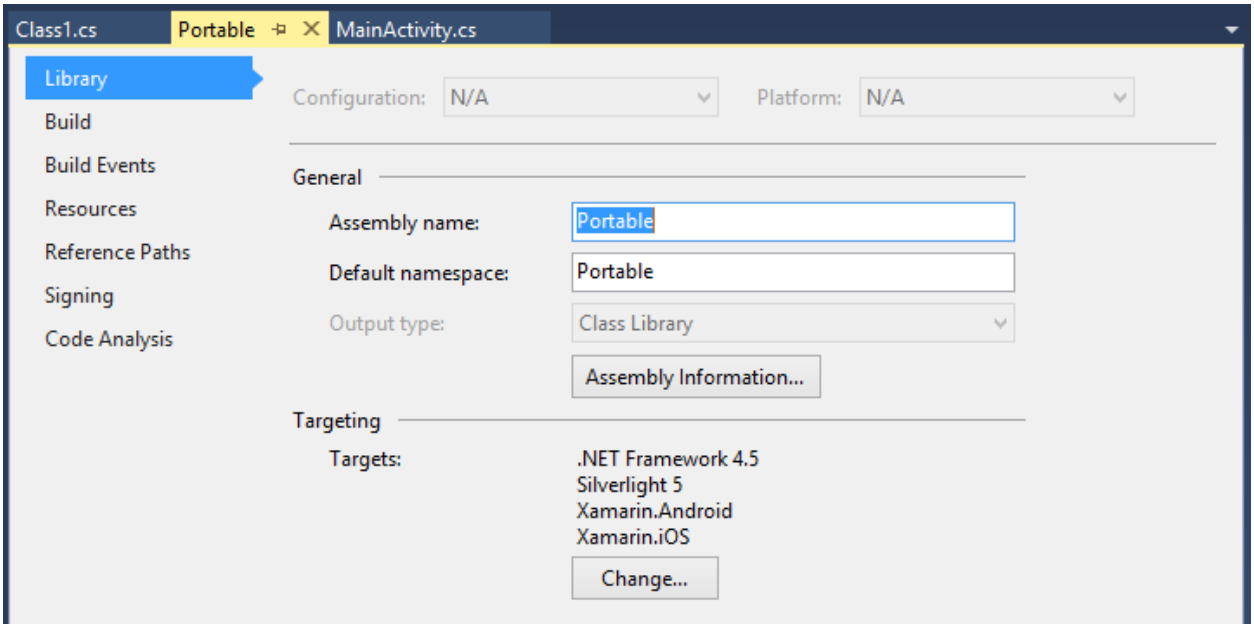


Figure 6: Properties view of the PCL project

Here is where you can see the versions you are targeting. To change the selections, simply click the **Change** button and you will see the **Change Targets** dialog box that looks strikingly similar to the **Add Portable Class Library** dialog box you saw in Figure 3.

Xamarin.Forms and PCLs

Just like any choice you are faced with in software development, there are always pros and cons to take into consideration. Using PCLs within your Xamarin.Forms application is no different. While PCLs work very well in Xamarin.Forms applications, there are a few very important things to consider when choosing them in your projects:

Pros

- Can access other PCLs targeting similar frameworks.
- Allow you to use XAML-based views (more on this in Chapters 4 and 5).
- Easy to write unit tests to verify functionality.
- Generate an assembly that can be shared in other projects.

Cons

- Can't contain platform-specific code.
- Can't reference any projects or libraries that contain platform-specific code.
- Only contain access to a subset of the .NET Framework.

Shared Projects

The **Shared Project** template was introduced with the release of Visual Studio 2013 Update 2, and Xamarin added support for these specialized projects in its release of Xamarin Studio 5. From a Microsoft perspective, they were created to support the concept of universal applications in which you can create a single code base that will run on both Windows 8.1 and Windows Phone 8.1. Because of the simplistic way that it is designed, it can be used for more platforms than just those two.

Creating a Shared Project

Let's investigate Shared Projects by creating a simple example. Start by creating a new project in Visual Studio 2013 Update 2 or Xamarin Studio 5 using the **Shared Project (Empty)** project template under the **Visual C#** family.

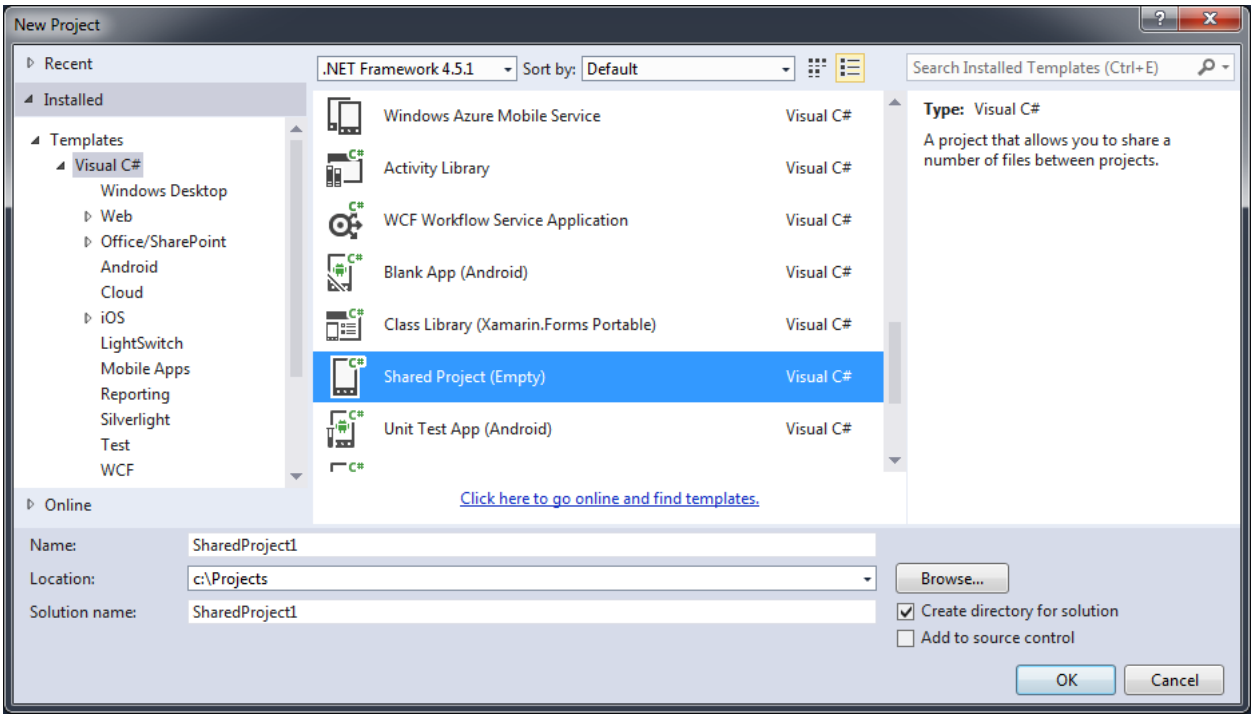


Figure 7: Creating a new shared project in Visual Studio 2013

Give the project a name and click **OK**.

Note: If you are using Visual Studio and don't see the Shared Project template, you'll need to download and install the Shared Project Reference Manager extension and restart Visual Studio.

Once the solution and project are created, you will notice something very different in the Solution Explorer. The project is completely empty. There are no Resources or Components folders. There are no Properties or default classes created. You are all alone.

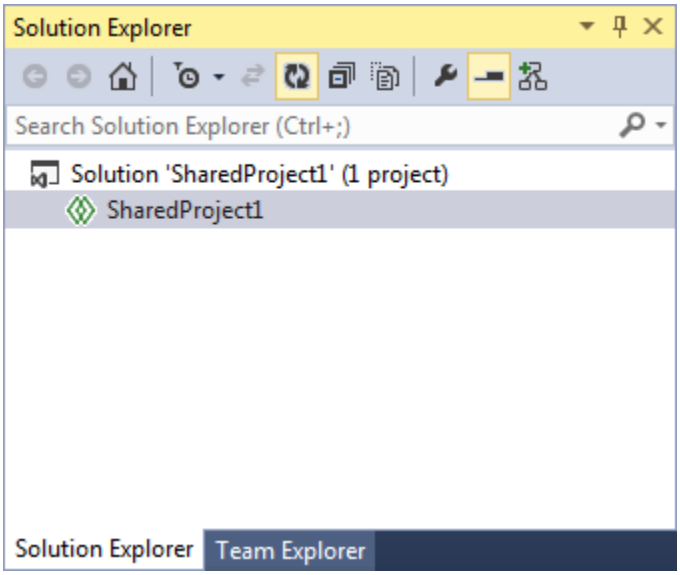


Figure 8: Empty shared project in Visual Studio 2013

Another drastic difference between a Shared Project and most other project templates is the minimal amount of properties available if you were to right-click your project and select **Properties**.

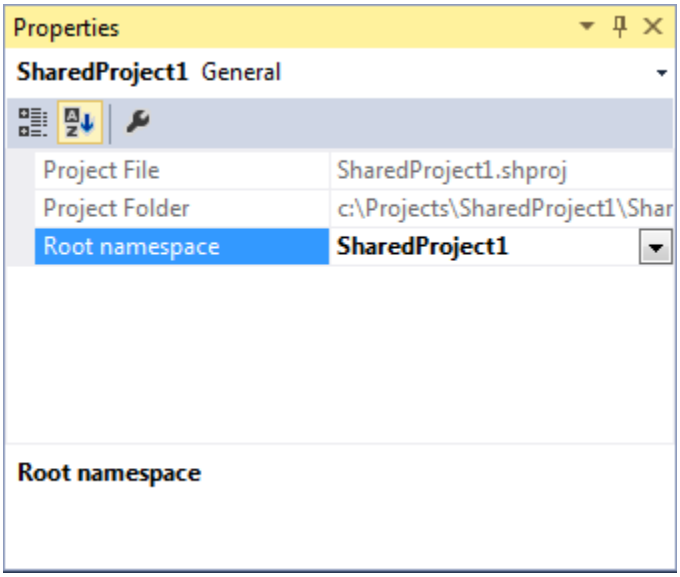


Figure 9: Shared project properties in Visual Studio 2013

From this view it is clear to see that you have very few options to configure a Shared Project. In fact, the only property that you can change is the root namespace. This property will automatically set the root namespace of any files added to this project that use namespaces.

The lack of properties may come as a shock because this is where you would see many options, including the ability to set the output type of a project to Windows Application, Console Application, or Class Library. Since this option is not available, you may wonder what you should expect from the result of building a Shared Project. The answer is nothing.

A Shared Project doesn't actually build on its own. It is merely a shell that contains code that will be included in other projects that reference it. That means that if there is no code that references anything within this project, none of it will be compiled into the final assembly (or group of assemblies). To further increase your control of what code in this project is included in other referencing assemblies, you have access to conditional compilation.

Conditional Compilation

Conditional compilation is one of those often overlooked features of C# and the .NET Framework. If you are unfamiliar with conditional compilation, here is a simple explanation. Take a look at the following code.

```
public int MyMethod()
{
     var result = 0;

     #if MY_SYMBOL
          result = 10;
     #else
          result = 20;
     #endif

     return result
}
```

Code Listing 1

Any code that has a number sign (#) prepended to it is considered a **preprocessor directive**. These directives are used by the compiler to determine which lines of code should be included in the compilation process. The following are fairly common to see when dealing with conditional compilation:

- #if
- #else
- #elif
- #endif

These directives are used to check for the existence of other symbols that have been defined using either the **#define** directive or by specifying conditional compilation symbols in the project properties. In the previous code, the symbol **MY_SYMBOL** is being checked for but nowhere in the code has it been defined. In order to define a symbol, you need to use the **#define** directive like this: **#define MY_SYMBOL**. Since that line doesn't exist, the compiler determines that the **#else** block should be compiled into the source. If this code were to be referenced by a console application that output the result of a call to **MyMethod()**, the console would write 20. On the other hand, if you were to place the **#define MY_SYMBOL** statement at the top of the file and recompile the code, the compiler would see that **MY_SYMBOL** has been defined, and then compile the **#if** block and the same console application would write 10.

Symbols found in Xamarin.Forms

With an understanding of conditional compilation, directives, and symbols fresh in your mind, now is a good time to mention why this is important. In the previous example, you saw that depending on the existence of a symbol, the code that is being used can be changed at compile time. This is an important feature used with Shared Projects in Xamarin.Forms.

There are several symbols defined by the different platform-specific Xamarin projects, as well as the Windows Phone projects, that will allow you to target the code written in a Shared Project to a specific platform. Some of the important symbols to note are:

- **__MOBILE__**: Defined in both Xamarin.iOS and Xamarin.Android projects.
- **__IOS__**: Defined in Xamarin.iOS projects.
- **__ANDROID__**: Defined in Xamarin.Android projects.
- **WINDOWS_PHONE** Defined in Windows Phone 7 and 8 projects.

In addition to these symbols, when it comes to Android applications, it is possible to get an even finer granularity. Depending on which version of the Android SDK you are targeting, there will be another symbol defined with the following format: **__ANDROID_xx__**. In this case, the **xx** represents the version of the Android API that is being used. For example, if your application is targeting API level 11, then there will be an **__ANDROID_11__** symbol defined. This allows you to target not only specific platform features at the iOS, Android, and Windows Phone level, but also features that are available in different levels of the Android SDK.

This access to conditional compilation and a set of predefined symbols opens up a whole new world to developers that doesn't exist in PCLs. By using conditional compilation and checking for a few of these defined symbols, you can now include platform-specific code that targets the multiple platforms in a single project, and you can also mix them in a single file.

Using a Shared Library

Using a Shared Library from another project is slightly different than referencing a PCL. Let's start by once again adding another Android project to our solution as we did before. This time, if you right-click the **References** folder in the Android project, select **Add Reference**, and look in the **Solution** section of the **Add Reference** dialog box, you will not see your Shared Library there.

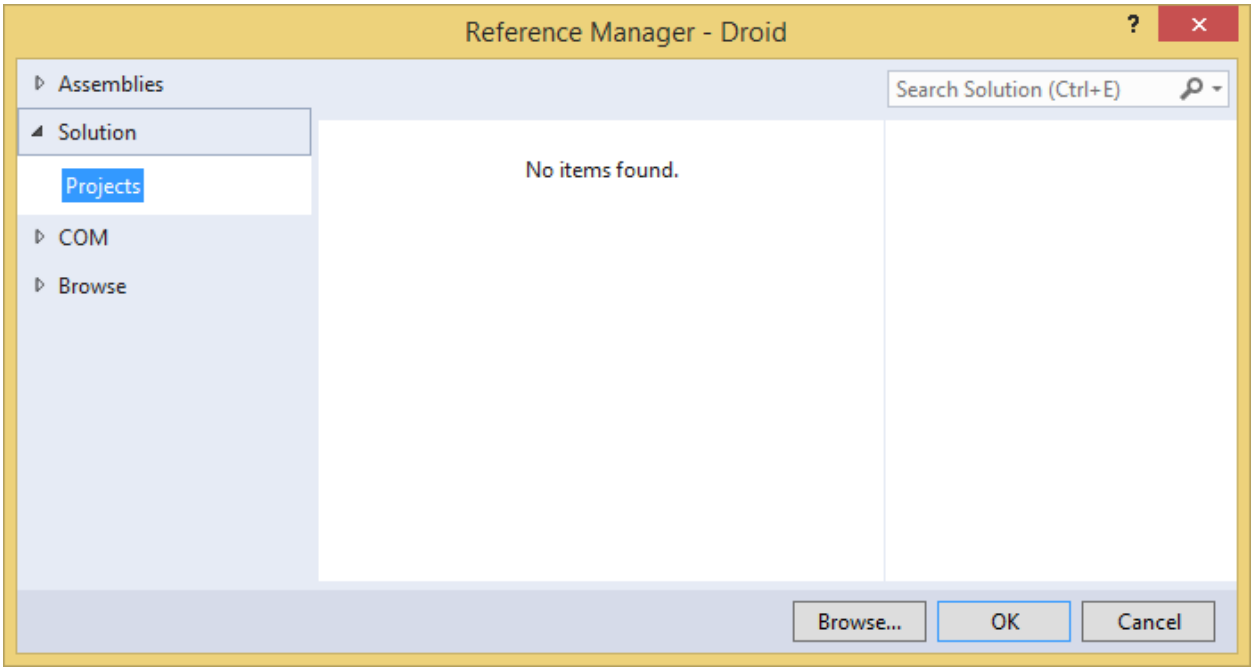


Figure 10: Trying to add a reference to a Shared Project via Add Reference

Instead, you are going to have to use another route to add a reference to the Shared Project. This time, when you right-click the Android project, you need to select the **Add Shared Project Reference** option in the context menu.

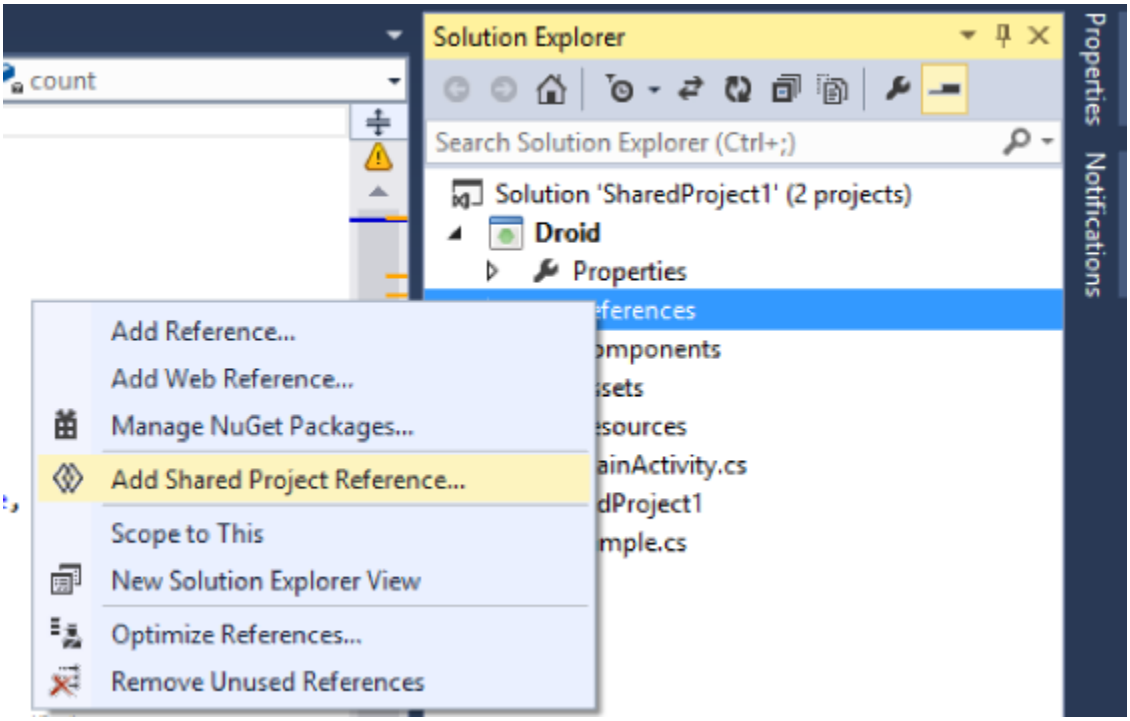


Figure 11: Adding a shared project reference

This will open a dialog box very similar to the normal Add Reference dialog, but this one is designed specifically to find and add Shared Projects that are found within this solution to your specific project.

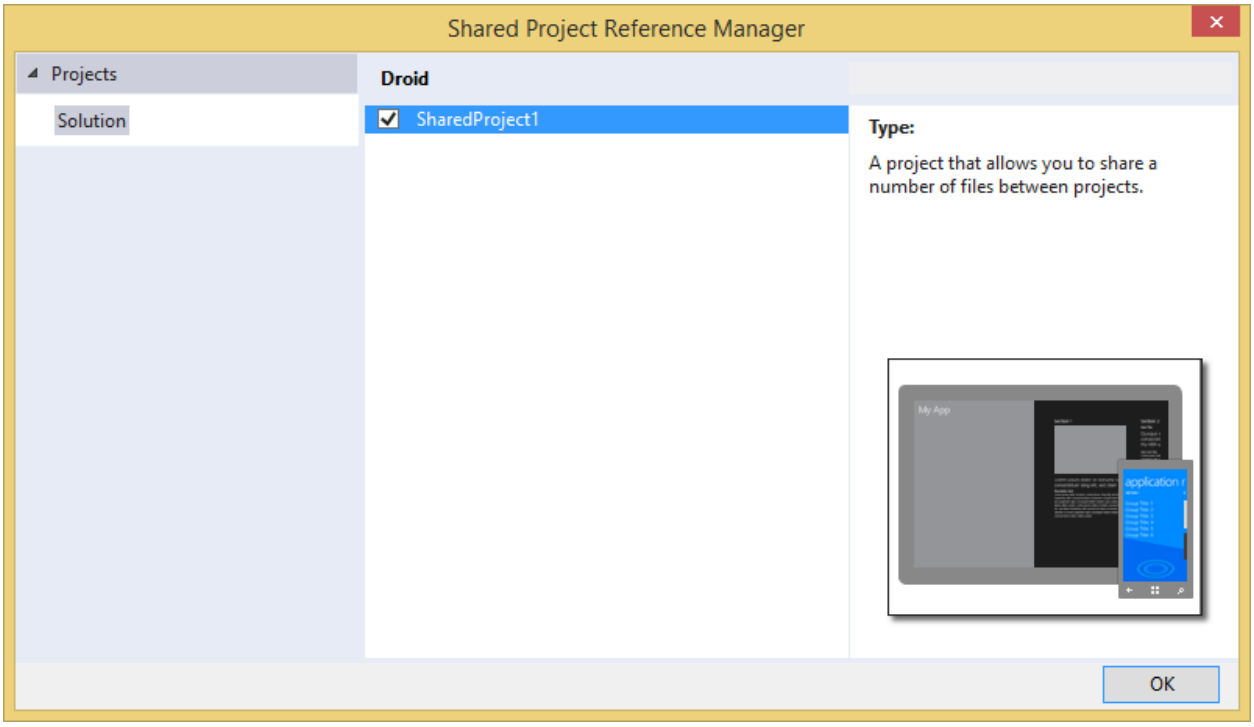


Figure 12: Shared project reference manager

Once you select the appropriate Shared Project and click **OK**, you will see a reference to the Shared Project within your Reference folder, with a special symbol next to it to indicate that it is a Shared Project.

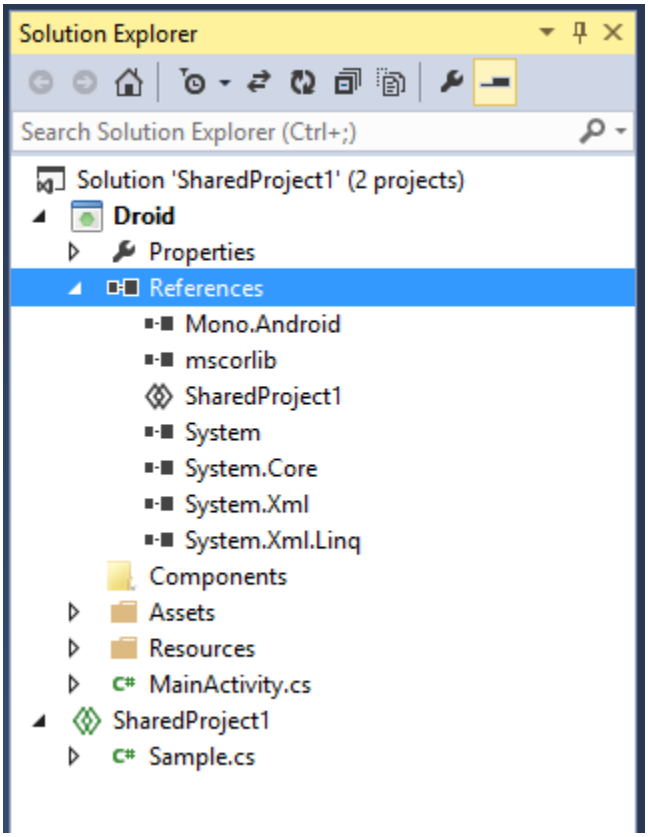


Figure 13: Shared project reference from Android project

Note: If you are using Visual Studio and don't see the Add Shared Project Reference option, you'll need to download and install the Shared Project Reference Manager extension and restart Visual Studio.

Context Switcher

When working with Shared Projects, it will be important for you to be able to easily see what the code looks like from the context of different projects within your solution that reference the shared code. This can be a little difficult, as well as intimidating, when you are looking at a screen full of **`#if`**, **`#else`**, and **`#endif`** directives and trying to make sense of what this code would compile when referenced by a Xamarin.iOS, Xamarin.Android, or Windows Phone project. Thanks to Microsoft and the Visual Studio team, that process is made a little easier.

You will notice that at the top of the code editor you still have the navigation bar that lets you jump to different types within the current file as well as different members within that type. A third drop-down menu has been added on the far left that is known as the **context switcher**. This drop-down lets you look at the code in the current file from the eyes of the referencing project and all of its defined symbols.

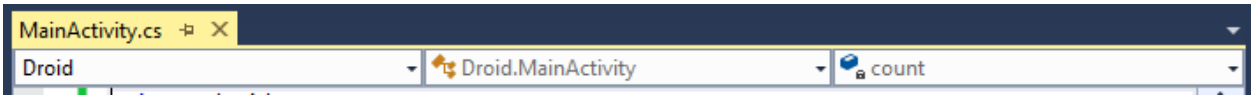


Figure 14: Context switcher

Note: If your context switcher only contains (Miscellaneous Files), you may need to remove and re-add the references to the Shared Project to correctly populate the drop-down.

Xamarin.Forms and Shared Projects

Choosing to use Shared Projects with your Xamarin.Forms application can provide you with further opportunities to consolidate shared code used by your application. This further consolidation comes at a bit of a cost, but depending on the type of application you are building, this cost may be acceptable. In order to make an educated decision about whether or not you should use Shared Projects, consider their pros and cons:

Pros

- Have access to all functionality in referencing projects.
- Can contain platform-specific code.
- Support conditional compilation to allow multiple platform-specific code in a single file.

Cons

- Doesn't create an assembly, so code can't be shared to other applications.
- Doesn't support creating views using XAML.
- Unit testing is more complicated when platform-specific code exists.

Summary

In this chapter, we covered the two primary code sharing strategies when creating applications using Xamarin.Forms, Portable Class Libraries and Shared Projects.

Along the way, we learned how to create both types of projects as well as how to utilize them from other projects. We also learned about conditional compilation when using Shared Projects, and how Visual Studio can help to clarify what code will be compiled in a Shared Project by using the context switcher. Finally, we learned about the pros and cons of each code sharing strategy to help understand the trade-offs when choosing one to use.

Chapter 3 Hello, Xamarin.Forms

In this chapter, we are finally going to introduce Xamarin.Forms. We will walk through the process of creating both types of Xamarin.Forms projects and investigate the makeup and structure of each. By the end of this chapter you should have the ability to not only create a Xamarin.Forms project, but also know how the application works, as well as be able to run the application on all the emulators and simulators for iOS, Android, and Windows Phone.

Getting Started with Xamarin.Forms

The first step to understanding Xamarin.Forms is to create an application and study it, so that is exactly what we are going to do. To begin, let's create a new Xamarin.Forms application. First, let's open Visual Studio or Xamarin Studio and select **File > New > Project**. In the **New Project** dialog box, select the **Mobile Apps** category in the list of templates on the left and you will be presented with three project template options:

- Blank App (Xamarin.Forms Portable): PCL version.
- Blank App (Xamarin.Forms Shared): Shared Project version.
- Class Library (Xamarin.Forms Portable): Additional library for PCL version.

For this example, we will only create the PCL version to see the structure and layout of the projects. You may choose to use the Shared Project version if you wish, but the resulting code and application will be the same regardless of which template you use.

Creating the Application

First, select the **Blank App (Xamarin.Forms Portable)** project template; give it a name of **HelloXamarinFormsPCL**, and click **OK**. Your screen should look similar to the following figure if you are using Visual Studio. If you are using Xamarin Studio, the process will be very similar, but some of the wording and buttons may be slightly different.

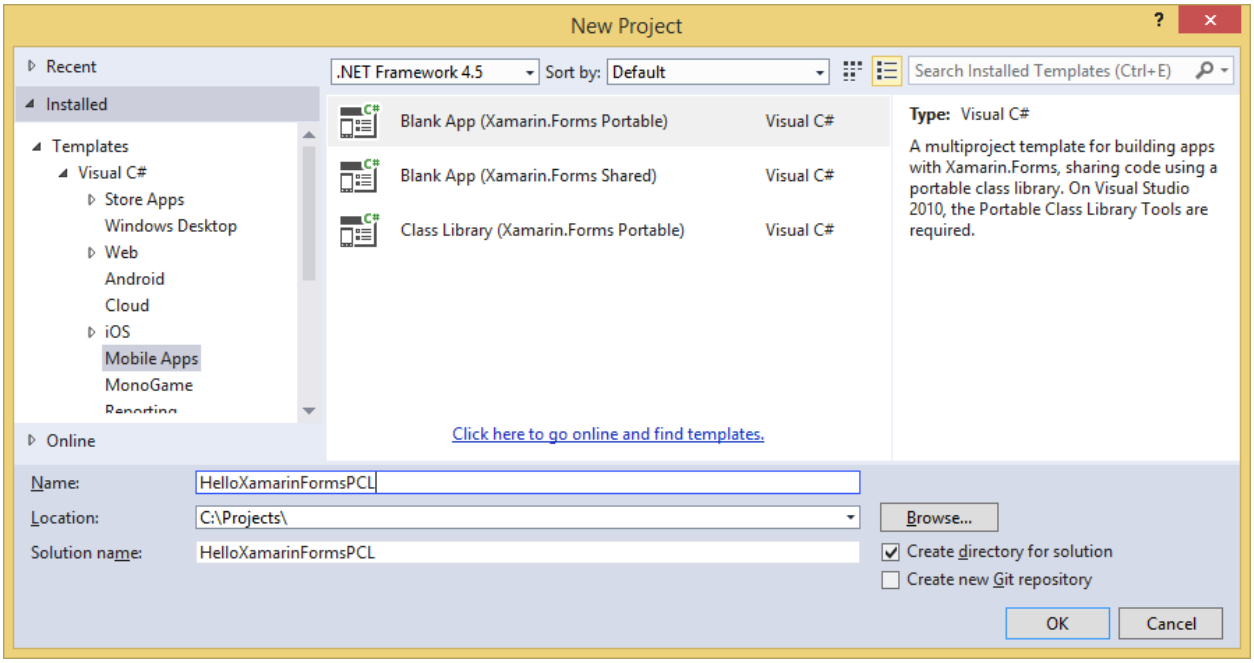


Figure 15: Visual Studio 2013 mobile apps templates

Once your Xamarin.Forms application is created, open the **Solution Explorer** and you will see that the Xamarin.Forms project template has created four projects for you:

- HelloXamarinFormsPCL (Portable)
- HelloXamarinFormsPCL.Android
- HelloXamarinFormsPCL.iOS
- HelloXamarinFormsPCL.WinPhone (Windows Phone 8.0)

The naming convention makes it quite easy to see which project corresponds to which platform, and the **Portable** project is used to contain the shared code. You are free to put all of your shared code in this single PCL if you choose. You can also split your shared code among multiple PCLs if you wish.

Exploring the project structure

For the most part, the platform-specific projects are the same as the normal Xamarin project templates with a small exception in each. In this section, we will only be covering what is new or different in the Xamarin.Forms project structure.

HelloXamarinFormsPCL (Portable)

In the PCL project of the Xamarin.Forms solution, you get the basic structure of an app. You are given a very basic project to contain your shared code. Since one of the main points of Xamarin.Forms is to share your UI code, that is all that is contained here by default. If you open the **App.cs** file, you will see something that looks like the following code example.

```
public class App : Application
```

```
{
    public App()
    {
        // The root page of your application
        MainPage = new ContentPage
        {
            Content = new StackLayout
            {
                VerticalOptions = LayoutOptions.Center,
                Children = {
                  new Label {
                     XAlign = TextAlignment.Center,
                     Text = "Hello, Forms!"
                  }
              }
            }
        };
    }

    protected override void OnStart()
    {
        // Handle when your app starts
    }

    protected override void OnSleep()
    {
        // Handle when your app sleeps
    }

    protected override void OnResume()
    {
        // Handle when your app resumes
    }
 }
```

Code Listing 2

The purpose of the **App** class is to provide a bootstrap point into the Xamarin.Forms UI code as well as give you access to the lifecycle of a mobile application. The constructor of the **App** class is used to set the **MainPage** property of the **Application** base class that will set the initial screen presented to the user. In this case, it simply sets this property to a new instance of a **ContentPage** that contains a **StackLayout** with a child **Label.** These different UI components will be covered in more detail in the next chapter, but for now think of them as generic representations of controls that will map to native controls at runtime.

This seems innocent enough, but alone this code really doesn't do anything. Sure it sets up a new main page for your application, but how do the different platform-specific projects know to use this **App** class and its **MainPage** property? All of the other projects that reference this library add a couple lines to call this method and use some Xamarin.Forms magic to create native UI controls from the Xamarin.Forms controls that are returned. Let's take a look at them.

HelloXamarinFormsPCL.Android

The entry point into Xamarin.Android projects is in the **MainActivity** class, or more specifically the **Activity** that is marked as being the **MainLauncher**. In our example, that is the **MainActivity** class and the **OnCreate** method. If you take a look into this method, you are going to see the following code.

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

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


    LoadApplication(new App());
}
```

Code Listing 3

As you will notice in the other platform-specific projects, as well as this one, the process of getting Xamarin.Forms controls to show up in the UI is made up of two phases: initialization and displaying. In the Android implementation of Xamarin.Forms, there is a helper method on the **Forms** class named **Init()**. This method has the job of taking the information about the **MainLauncher Activity**, as well as a **Bundle** object passed in to the **OnCreate()** method, and informing Xamarin.Forms about them and ultimately giving them access to the world of the Android project.

If you are unfamiliar with typical Android applications, whether native or Xamarin, this **Bundle** concept may be a little confusing. In our example, it is not crucial to the understanding of Xamarin.Forms, but for the sake of completeness, here is a simple explanation.

In the lifecycle of Android screens, known as activities, there are several times when the screen, and the application in general, can be in danger of being shut down. Regardless of what causes this closing of the **Activity**, the **OnCreate** method is typically one of the first methods to be called once the **Activity** is brought back to the user. The **Bundle** object that is passed into the **OnCreate()** method can be used to save the state of the application before it is closed if the developer takes advantage of it in other lifecycle events of the **Activity**. If the developer took the time to save the application state into this **Bundle** before it closed, there is an opportunity here to restore the application state to save the user from starting all over in the application.

The final step in the **`OnCreate()`** method is to call a Xamarin.Forms helper method named **`LoadApplication()`**, pass to it a new instance of our **`App`** class from the shared code project (PCL in this case), and display it to the user. At this point, from a developer's perspective, all the logic is now taking place within the Xamarin.Forms shared code project.

HelloXamarinFormsPCL.iOS

Similarly to the Android-specific project, the iOS version has the job of calling some Xamarin.Forms initialization code and passing on the control of everything UI-related to Xamarin.Forms itself. While the process is similar, the implementation is slightly different.

Like any iOS application, the entry point into this world is via the **`AppDelegate`** class, and it typically takes place in the **`FinishedLaunching()`** method. If you were to open the **AppDelegate.cs** file in the **HelloXamarinFormsPCL.iOS** project, you will see something similar to the following code sample.

```
public override bool FinishedLaunching (UIApplication app,
                                        NSDictionary options)
{
      global::Xamarin.Forms.Forms.Init ();


      LoadApplication(new App());

      return base.FinishedLaunching(app, options);
}
```

Code Listing 4

In this scenario, we see some similar initialization code. Here, it's in the form of a simple method named **`Init()`** on the **`Forms`** class without any parameters. Then, we see a similar method call as the Android counterpart to an iOS version of the **`LoadApplication`** method, passing to it a new instance of our **`App`** class. Finally it will return the result from the base **`FinishedLaunching`** method to finalize any setup that is needed. From this point on, the UI flow is controlled from within Xamarin.Forms.

HelloXamarinFormsPCL.WinPhone

Finally, in the Windows Phone application, we see the same concepts again. The main entry point of this application is the constructor of the **`MainPage`** class. This can be found in the **MainPage.xaml.cs** code-behind file under the **MainPage.xaml** file in the **Solution Explorer**. The constructor looks like the following code.

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

```
    SupportedOrientations =
SupportedPageOrientations.PortraitOrLandscape;

    global:: Xamarin.Forms.Forms.Init();
        LoadApplication(new App());
}
```

Code Listing 5

Once again, in this scenario, we see in the Windows Phone implementation of Xamarin.Forms there is an **`Init()`** method on the **`Forms`** class to initialize the use of Xamarin.Forms. After that, there is a call to the **`LoadApplication`** method passing in a new instance of our **`App`** class again on the **`Application`** class:

- OnStart
- OnSleep
- OnResume

While these methods may be an addition to Xamarin.Forms, these concepts are not new to the mobile development world. These methods provide opportunities for you as a developer to recognize and react when your application starts, is sent to the background, or resumes after being sent to the background. At this point, you may be wondering why you would care about these lifecycle methods at all. There are a number of situations in which these are useful, but the main one has to do with the persistence of data.

Nothing is more annoying to a user than having to repeat an operation. Imagine a user opening an application, filling out some data within a form, and then having to switch to another app or having the app close for some reason. Typically what will happen is when the user opens the app again, he or she will have to re-enter all of that information. These lifecycle methods can help with that situation greatly.

Now with the **`OnSleep`** method, you can notice when your app is going to be sent to the background, or even closed, and quickly store some important data on behalf of the user. Then, when the **`OnStart`** or **`OnResume`** methods fire, you can retrieve that information and use it going forward. The question in these types of scenarios quickly becomes where do I store this data?

Sure, you can take this opportunity to store information in a database or call out to some web service to save the data for you, but there is a common issue with these approaches: time. When these lifecycle events occur on devices, you don't have a lot of time to act. The devices themselves only give you a brief amount of time to do something, on the **`OnSleep`** method for instance, before your app is sent to the background or shut down completely. That being the case, wouldn't it be nice if we had a simple and quick way to store information in these types of situations? It just so happens that we do.

Also included in the Xamarin.Forms 1.3 release is a new static **`Properties`** dictionary that allows you to quickly store and retrieve data at any point in your application's lifecycle. You can access it from the **`Current`** property of the **`Application`** class, **`Application.Current.Properties["name"] = "Some Value"`**. Using this dictionary is like using any other dictionary in that you access and set data via the key name. One thing to note about this dictionary is that its keys are **`string`** type and the values are **`object`** types. So after you retrieve a value, you will want to cast it to whatever type it truly is. Of course, you should always exercise caution when using dictionaries by checking to make sure that a dictionary contains a key before actually trying to access it.

Finally is the idea of persistence. And luckily for us, we don't actually have to worry about that. The **`Properties`** dictionary is automatically saved to the device and is available as soon as the application returns from the background or after it is restarted.

Running the Application

Now that you are familiar with the project structure, let's see this application in action. Unless you are interested in taking a look at each individual platform running in its own emulator or simulator one at a time, the easiest way to see them all in action is to start them all at the same time. To do this, right-click the solution, **HelloXamarinFormsPCL**, and select **Properties**. In the **Solution Properties** dialog box that appears, expand **Common Properties** on the left and select **Startup Project**.

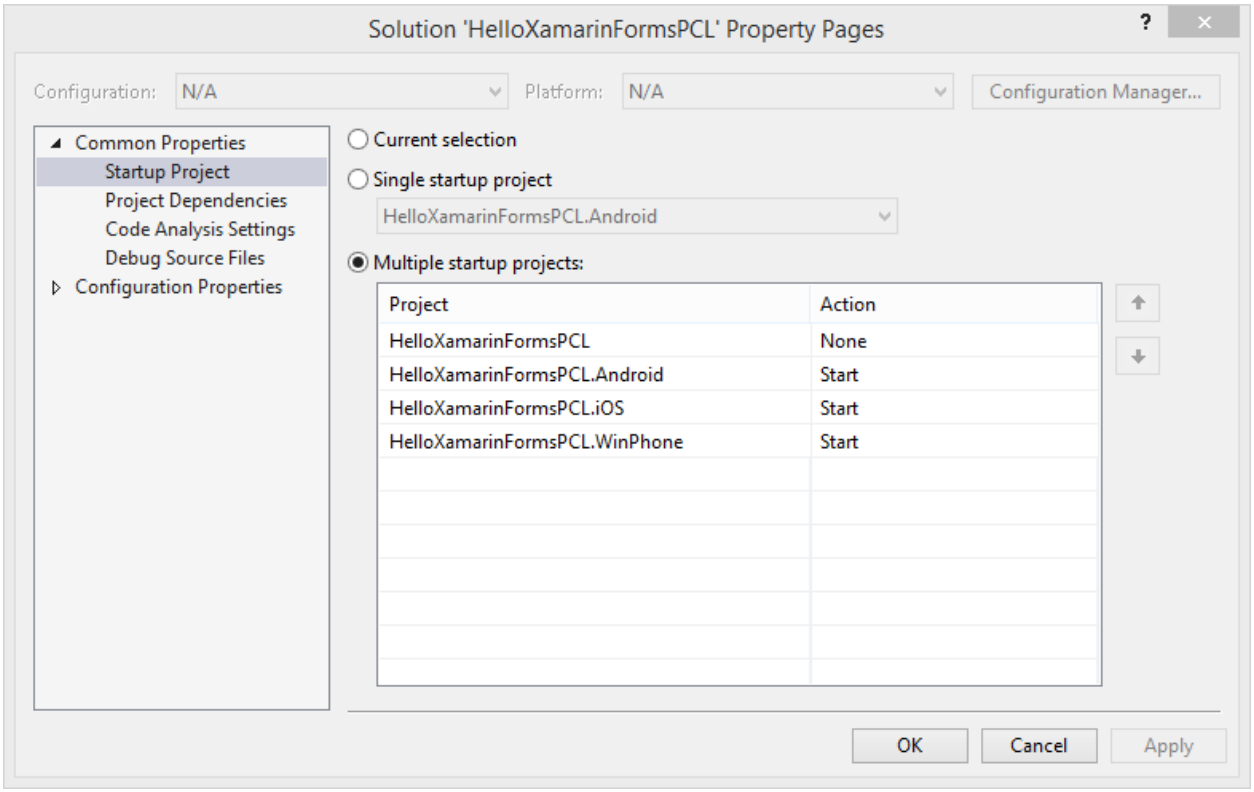


Figure 16: Multiple startup projects in Visual Studio 2013 solution properties

From this view, you can select a single or multiple projects to start up when you run your application. If you would like to start multiple projects, simply select the **Multiple startup projects** option. In the box below the radio button, set the **Action** for each project you want to run to **Start**. Leave the shared code library project action as **None**, because class library projects cannot be a startup project. Also, if you choose to run the iOS application from Visual Studio on Windows, you will need to have network access to a Mac with Xamarin.iOS and Xcode installed and configured to run as a Xamarin.iOS Build Host.

After you have selected which projects you wish to start when you run your code, start your application and you should see three applications running in different emulators or simulators. These applications are running from the same shared UI code found in the **HelloXamarinFormsPCL** project. All three of the applications look remarkably similar in structure, but it's also at least somewhat obvious that they still maintain their own platforms' look and feel. With a simple example like this it's difficult to see that the controls are truly native, so let's modify this example just a little bit.

Figure 17: Hello on Android

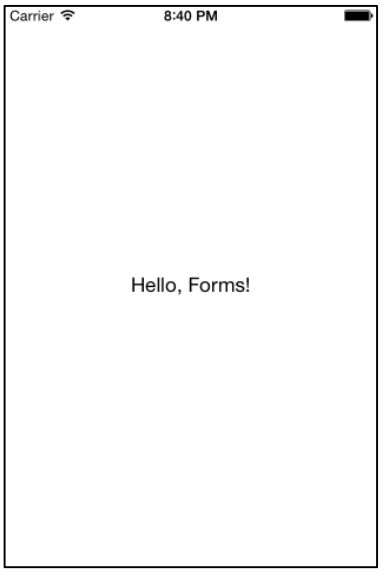


Figure 18: Hello on iOS

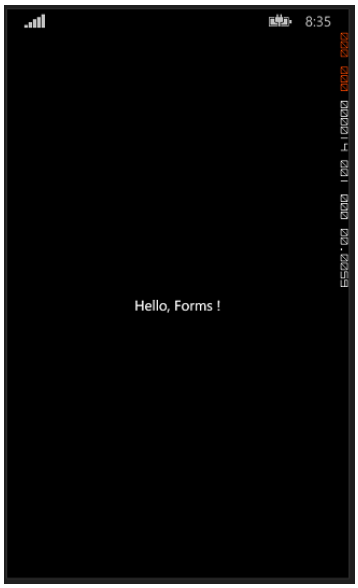


Figure 19: Hello on Windows Phone

Modify the Application

To further prove the point that Xamarin.Forms allows you to write shared UI code in a simple and concise manner with a fully native resulting application, let's tweak our application just a little.

Start by opening the **HelloXamarinFormsPCL** project and taking a look at the **App.cs** file. Modify the **App** class to look like the following code example.

```
public class App : Application
{
    public   App() {
        var label = new Label {
            Text = "Hello, Xamarin.Forms !",
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        var button = new Button {
            Text = "I'm a button",
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        var entry = new Entry {
            Placeholder = "Enter some text",
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        var datePicker = new DatePicker {
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        MainPage = new ContentPage
        {
            Content = new StackLayout {
                Children = { label, button, entry, datePicker}
            }
        };
    }
}
```

Code Listing 6

As you can see, in the end, we are still using a **`ContentPage`** object. This time we have set its **`Content`** property to a new instance of a **`StackLayout`** control. We will go into more detail on **`StackLayout`** controls as well as many others in the next chapter. For now, think of a **`StackLayout`** as a control that will automatically lay out other controls, via its **`Children`** property, on the UI in either a vertical or horizontal orientation. The child controls will be placed in such a way that they flow nicely in the UI based on their sizes.

In this example, the **`Children`** property of the **`StackLayout`** is populated with four different controls:

- **`Label`**: Displays a simple string of text.
- **`Button`**: Allows the user to click and generate an event.
- **`Entry`**: Allows the user to enter a single line of text.
- **`DatePicker`**: Allows the user to select a date.

Once all of these controls are added to the UI and the application is run, the controls are mapped to native controls and create three very similar, yet uniquely native, applications across all three different platforms.

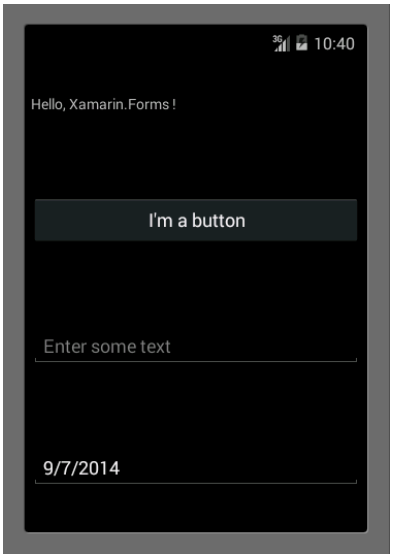


Figure 20: Updated Android app

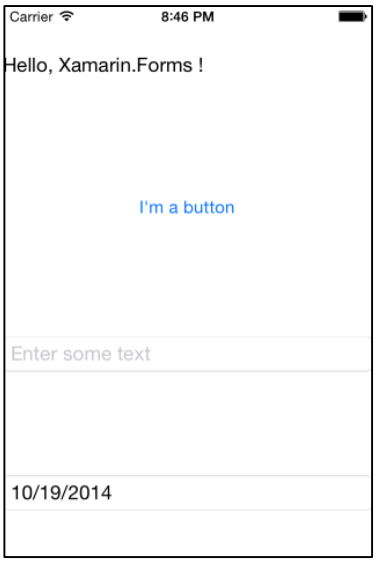


Figure 21: Updated iOS app

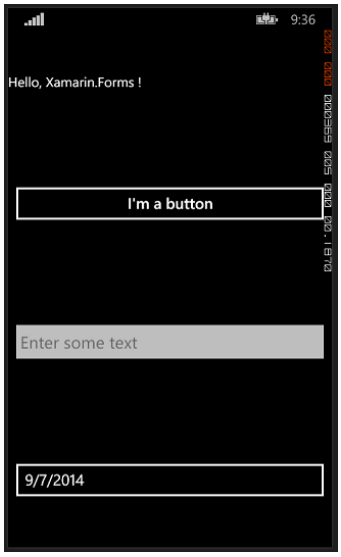


Figure 22: Updated Windows Phone app

Summary

In this chapter we have covered the basics of creating a simple Xamarin.Forms application. We've discussed the project structure that is created out of the box using the Xamarin.Forms project templates. After that, we successfully ran the boilerplate application in all three emulators to see what Xamarin.Forms looks like in action. Finally, we customized the application with some basic out-of-the box controls to see just what it feels like to write cross-platform UI code in a simple and concise manner, and then we were able to once again run our application and see the fruits of our labor.

Chapter 4 Introduction to XAML

When it comes to creating user interfaces in Xamarin.Forms applications, you have two primary options: code and XAML. The code aspect of the equation is relatively easy to understand. Most programming languages and platform SDKs have some mechanism for developers to programmatically create user interfaces and tap into the events and interactions with these user interfaces. XAML, on the other hand, is a little different.

XAML, an acronym that stands for **eXtensible Application Markup Language**, is a declarative markup language in the dialect of XML. XAML was originally created by Microsoft as a succinct, declarative language that when applied to the .NET Framework programming model, could instantiate and initialize objects as well as their properties. Microsoft introduced this language along with Windows Presentation Foundation (WPF) as a mechanism to separate the UI definition of an application from the runtime logic by using code-behind files. As we will see later, these code-behind files may be useful, but there is a better way to interact with the UI controls from code.

This chapter will walk through the process of understanding the basics of XAML from its syntax to data binding. This is not meant to be an exhaustive explanation, rather a good starting point to understand the XAML enough to be able to use it in your Xamarin.Forms applications.

Basics of XAML

Before diving into the different types of `Page` objects available and using `Layout` classes to structure them, it's important to have a basic understand of XAML. XAML is really just XML with a few features added to it. This section on XAML is in no way meant to be exhaustive. There are entire books written on the topic of XAML. This is merely here to provide a simple overview of XAML and some of the common characteristics that you will need to understand in order to effectively use it within your Xamarin.Forms applications.

Let's begin learning the basics of XAML by working through an example that is going to touch on some of the more important characteristics.

The Syntax

For this example you can either use the sample application you created in the previous chapter or create a new one. When creating the new solution that will support XAML, you can choose either a PCL or Shared Library project. That being the case, the official Xamarin-supported way to create XAML code in your Xamarin.Forms project is through the PCL template, so that is what I will use in this section.

Let's start by creating a new project using the **Blank App (Xamarin.Forms Portable)** template. Once you have a new project, right-click on your PCL project and select **Add > New Item.** From the **Add New Item** dialog box, search for the **Forms Xaml Page** file template. Give the new file a name and then click **Add**.

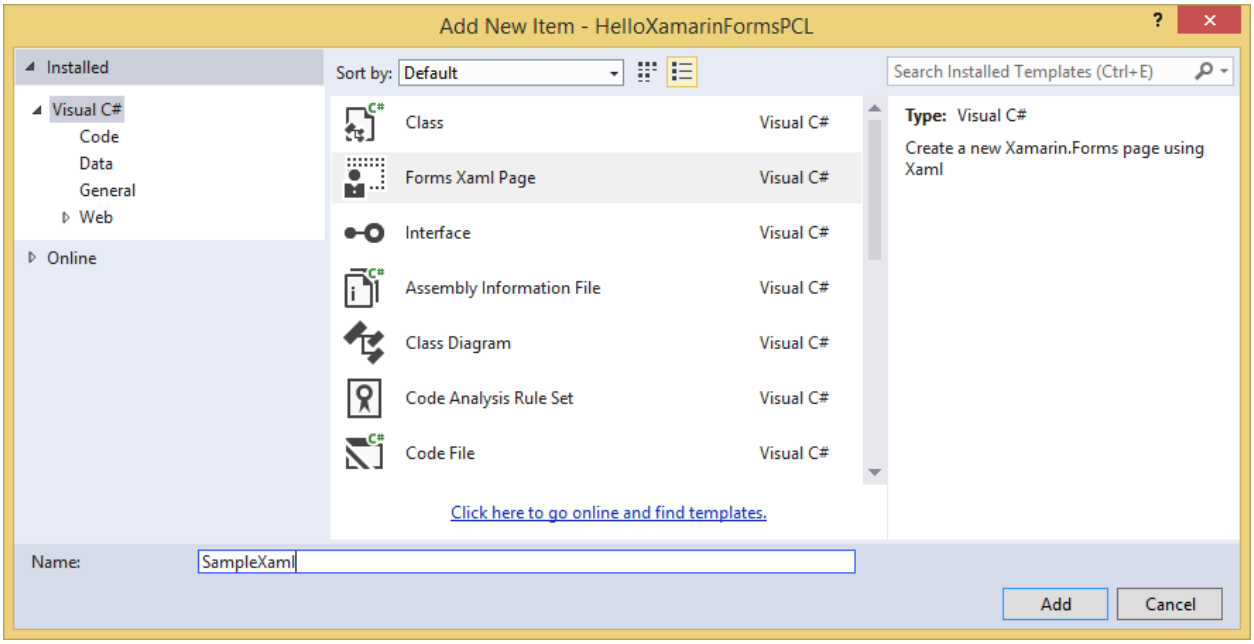


Figure 23: Add a new XAML page

Note: If you are presented with a dialog that asks whether or not you trust this template, simply click the Trust button to continue.

After you have added the new XAML page to your project, open it and remove everything between the opening and closing tags of the **`ContentPage`** element so you have only the following code.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="HelloXamarinFormsPCL.SampleXaml">
</ContentPage>
```

Code Listing 7

In this example, you have the basic building blocks of a XAML class. The first line of this block of code is considered the **prolog** in XML. This is an optional element that contains the XML **`version`** number as well as the **`encoding`** used for the file as attributes of the **`<?xml ?>`** element.

The second line contains all that is needed to create an instance of the **`ContentPage`** class. Within the **`ContentPage`** element, the first two attributes are namespace declarations that point to URLs. You can try navigating to them, but I assure you there is nothing there. They are simply used to provide a mechanism for versioning the functionality available to this class.

One interesting thing to note is that the second namespace declaration uses a prefix of **x**. In this case, the **x** is used before any other attribute within this element to associate the particular attribute to that namespace. This means the attribute name it prefixes should be understood to be associated with that specific namespace to avoid naming collisions between the properties found in different namespaces.

In the final attribute of the **ContentPage** element, the **x** prefix is used to state that the **Class** attribute is specific to the second namespace. This attribute is used to specify the fully qualified class name. This means the **SampleXaml** class is in the **HelloXamarinFormsPCL** namespace. At this point, if you were to run this application, you would see nothing but a blank screen on all three platforms. That's a little boring, so let's add something to the screen.

Adding Content

It's very simple to add content and elements to a Xamarin.Forms **Page**. Since we are using a **ContentPage** in this example, we will be dealing with its **Content** property to add a UI element to the screen. From an XML perspective, the **Content** property is mapped to the inner XML of the **ContentPage** node. One of the drawbacks with this **Page** type is that it only allows for a single view to be added to its **Content** property. There is a way around that using **Layout** controls however, and we will cover that later in this chapter. For this example, we will keep things simple and just add a **Button** to our **ContentPage**.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="HelloXamarinFormsPCL.SampleXaml">
  <Button Text="Click Me"
          HorizontalOptions="Center"
          VerticalOptions="Center" />
</ContentPage>
```

Code Listing 8

To add a **Button** to our page, we have simply added a **Button** element to the **ContentPage** element, thus creating a hierarchy on our **Page**. Once again, you can see that to instantiate a new instance of a **Button**, you simply need to declare it as an element and initialize any of its properties via attributes on that element.

You will notice very quickly when it comes to working with these classes that a number of their properties are simple types. In the previous example, the **Text** property is just a **string**. Other properties will just be simple integers or other numeric values. But how does something like **TextColor** or **HorizontalOptions** work? Those are surely more complex types or enumerations. How does the XAML know how to interpret those and get an actual value for those string representations and assign it to those properties? The answer lies in **TypeConverters**.

TypeConverters

Xamarin takes advantage of several classes within the .NET Framework with XAML to allow you to assign simple string values to enumeration properties or complex types and just have them work. These classes are known as converters and all derive from the **`TypeConverter`** base class. These classes have the specific job of mapping the string value associated with these complex type properties to the actual types that the property expects. In the previous example, the **`Text`** property expects a string so there is no converter needed. The other three properties do need converters. Specifically, the **`TextColor`** property will take advantage of the **`ColorTypeConverter`** and the **`HorizontalOptions`** and **`VerticalOptions`** properties will use the **`LayoutOptionsConverter`**.

Based on the specific converter, the implementations can be quite different. For the **`ColorTypeConverter`**, you can specify either the name of a color defined within the **`Color`** structure or a hexadecimal RGB value. The **`LayoutOptionsConverter`**, on the other hand, only accepts string names that map to the **`LayoutOptions`** structure.

The code-behind file

Whether you are creating WPF, Silverlight, or Xamarin.Forms applications, when you create a XAML page, you also get a code-behind file. If you look in the Solution Explorer for your PCL project and find your **SampleXaml.xaml** file, you will notice a small arrow next to it. If you click the arrow you will reveal the code-behind file.

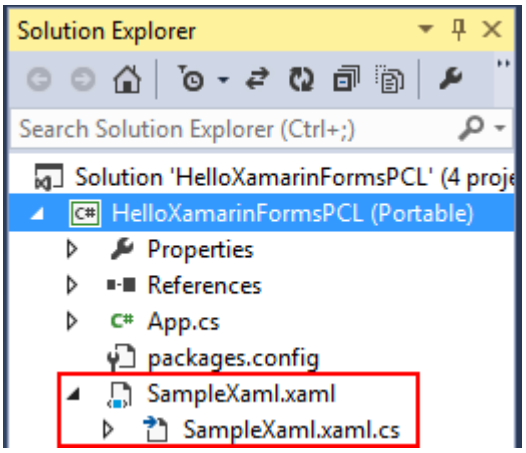


Figure 24: Finding the XAML code-behind file in Visual Studio

You will know that it's a code-behind file because of the naming convention as well. Code-behind files always share the same name as their XAML pages with the addition of the .xaml.cs extension. Open the code-behind file to take a look.

```
public partial class SampleXaml
{
    public SampleXaml()
```

```
    {
        InitializeComponent();
    }
}
```

Code Listing 9

In the beginning, the code-behind is rather dull. There isn't much code, only a simple default constructor that calls a method named **`InitializeComponent()`** that doesn't exist yet. Another interesting thing to note is that the class is also declared as **`partial`**. This is by design.

During the build process, an automatically generated file will be created that will be the other part of the **`partial SampleXaml`** class and contain the declaration and implementation of the **`InitializeComponent()`** method. If you navigate into your file system to your project directory and find the **obj\Debug** directory, you will find a file named **SampleXaml.xaml.g.cs**. This is the automatically generated file that will finish this **`partial`** implementation.

```
namespace HelloXamarinFormsPCL
{
    using System;
    using Xamarin.Forms;
    using Xamarin.Forms.Xaml;


    public partial class SampleXaml : ContentPage
    {

        private void InitializeComponent()
        {
            this.LoadFromXaml(typeof(SampleXaml));
        }
    }
}
```

Code Listing 10

This implementation of the **`SampleXaml`** class specifies that it derives from the **`ContentPage`** class. It contains a private implementation of the **`InitializeComponent()`** method that uses the **`LoadFromXaml()`** method to create a new instance of our **`SampleXaml`** class using our XAML definition.

Note: This is an automatically generated file. Any modifications made to it will be overwritten during the next build. Do not modify this file.

Later in this chapter, you will be introduced to a very common pattern used in conjunction with XAML that will encourage you to avoid the code-behind file as much as possible. This doesn't mean that this file is evil. It just means that there may be more flexible ways of attaching code to your UI. Until then, we will use the code-behind file to demonstrate some common functionality when building UIs and handling events.

Before you run this application, you will need to make one final change. By default, Xamarin.Forms applications come with some boilerplate code in the **App.cs** file. In order to run your XAML code you will need to modify the constructor in the **`App`** class to set the **`MainPage`** property to a new instance of your **`SampleXaml`** class with the line: **`MainPage = new SampleXaml()`**. If you were to run this application now, you would see the following.

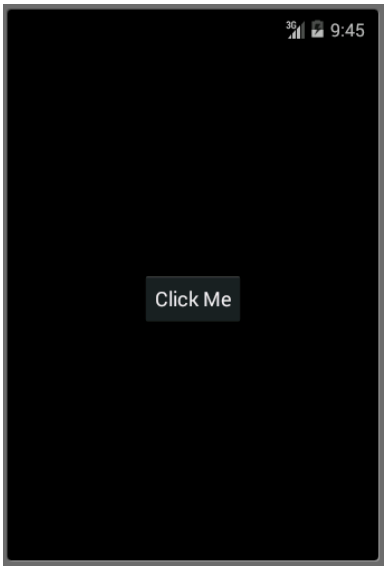


Figure 25: Button on Android

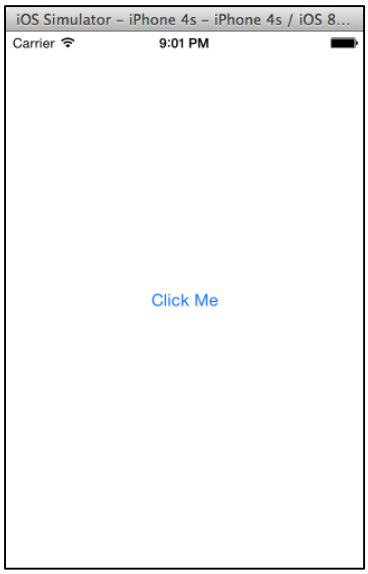


Figure 26: Button on iOS

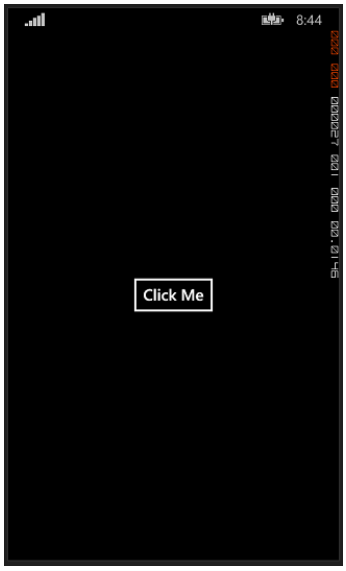


Figure 27: Button on Windows Phone

As you can see, this is nothing overly exciting and if you were to click the button, nothing would happen. Let's fix that.

Handling Events

A common piece of functionality that goes hand in hand when working with a UI is the ability to present users with something that they can interact with. When users are given a button for example, you are going to want to know when they click on the button and respond to that event. The clicking of a button in the world of XAML, much like any other .NET languages, is an event.

Let's once again update our sample XAML to be able to handle the event when a user clicks on the button.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="HelloXamarinFormsPCL.SampleXaml">
   <Button Text="Click Me"
           HorizontalOptions="Center"
           VerticalOptions="Center"
           Clicked="Button_Clicked" />
</ContentPage>
```

Code Listing 11

As you can see in the previous code example, event handlers in XAML are created the same as properties. They are attributes that are assigned to elements that represent the class. The value that's assigned to these attributes is the name of the event handler method that is expected to be found in the code-behind file. Let's go into the code-behind file and update it for the **`Button_Clicked`** event handler.

```
public partial class SampleXaml
{
    public SampleXaml()
    {
        InitializeComponent();
    }

    public void Button_Clicked(object sender, EventArgs e)
    {
        var button = (Button)sender;
        button.Text = "Ouch!";
    }
}
```

Code Listing 12

The **`Button_Clicked`** method here has the same standard event handler signature as seen in other places around the .NET Framework. It takes two parameters. The **`sender`** parameter contains a reference to the object that sent the event (the **`Button`**). The **`e`** parameter contains any additional information about the event in the form of **`EventArgs`**.

To illustrate that something is truly happening within this event, once it's fired we cast the sender to an instance of the **`Button`** class, and then we change the **`Text`** property on the **`Button`** to something else. Once we run this application we should see each of the three platforms responding in a similar fashion to the user clicking the button.

Figure 28: Clicked button on Android

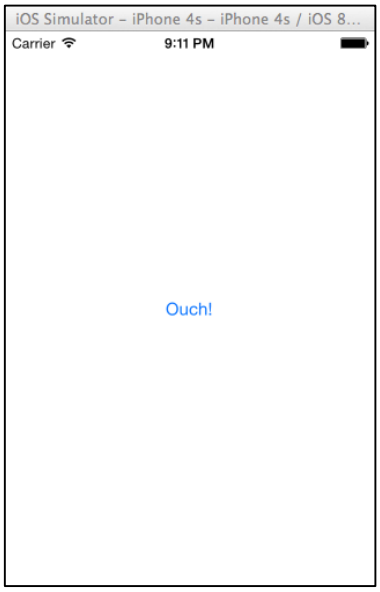


Figure 29: Clicked button on iOS

Figure 30: Clicked button on Windows Phone

Binding

If you have never seen XAML before, this will probably strike you as a somewhat complicated topic. Binding is simply a mechanism for a developer to create UI elements that display and interact with data. When it comes to displaying data in the previous examples, most properties that are being assigned are done explicitly in the editor. The values are being hard coded. If at any point we want to update the **`Text`** property on a **`Button`**, we will either have to do it manually in the XAML file itself, or change it during some event that we are handling in the code-behind file. While this is typically fine, when your application begins to grow and become more complex, you will quickly realize you will need to find a better way.

Binding, or data binding as it's commonly called, is the process of making a connection between the UI (XAML) and a data object that allows information to flow between the two. This means that instead of designing a XAML screen to suit a specific purpose, we can create a screen a little more generically, and allow some data behind the scenes to populate the properties with values, even if it doesn't realize it is happening. The best way to understand data binding is through an example.

Let's start with taking our previous example and updating it just a bit.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="HelloXamarinFormsPCL.SampleXaml">
    <StackLayout>
```

```
            <Button Text="Click Me"
                    Clicked="Button_Clicked" />
            <Label Text="{Binding LabelText}" />
            <Entry Text="{Binding EntryText}" />
        </StackLayout>
</ContentPage>
```

Code Listing 13

We have just introduced a concept that is known as a **markup extension**. A markup extension, typically denoted by the use of opening and closing curly braces, is a special way to extend the functionality of XAML.

In general, the way that XAML works is that a parser will go through your XAML file and interpret attribute values as a string that can be converted into something that the attribute understands. One way this is done is through type converters that assign literal values to attributes. There are other scenarios, though, where additional functionality is required. One of the times where different functionality is required is during data binding. In this case, we tell the XAML parser that the values being assigned to the **`Text`** properties of both the **`Label`** and **`Entry`** are going to use the **`Binding`** markup extension.

The **`Binding`** markup extension provides a special data-bound property value that is deferred until runtime. This means that at the time you are writing your code and during compile time, the values for these specific properties aren't known. It's assumed that during runtime a value is going to be assigned to the properties in some manner. That manner is through a **`BindingContext`** in Xamarin.Forms or **`DataContext`** in other technologies.

BindingContext

A **`BindingContext`** is a special property found in the **`BindableObject`** class that allows for such data binding to occur. The **`BindableObject`** class is a few layers up in the inheritance tree when it comes to Xamarin.Forms. This class is used as a base class for all classes in Xamarin.Forms that represent visual elements. This means that all of these derived classes have access to the **`BindingContext`** property. At some point during the life of your XAML page (if you are using data binding), you are going to need to tell it where to look for the data. The most common way to do this is to assign the **`BindingContext`** property a value during its construction. So let's see how that happens.

Open your code-behind file for the example we are working on and modify it to look like the following.

```
public partial class SampleXaml : ContentPage
{
    public SampleXaml()
    {

        InitializeComponent();
        BindingContext = new SampleContext {
             LabelText = "Sample Label",
```

```
            EntryText = "Sample Entry"
        };
    }

    public void Button_Clicked(object sender, EventArgs e)
    {
        var button = (Button)sender;
        button.Text = "Ouch!";
    }
}
```

Code Listing 14

As you can see here, we are assigning a new instance of the **`SampleContext`** class to the **`BindingContext`** property of our **`SampleXaml`** class. Of course that doesn't exist yet, so let's go ahead and add a new class to our PCL project and name it **SampleXaml**.

```
public class SampleContext
{
    public string LabelText { get; set; }
    public string EntryText { get; set; }
}
```

Code Listing 15

This is a very simple class that has no functionality at the moment and only two properties. You may notice those two properties have the same names that are referenced in the **`Binding`** markup extension in the XAML file. That is no coincidence and you will soon see why.

If you were now to run this application, you should see the following.

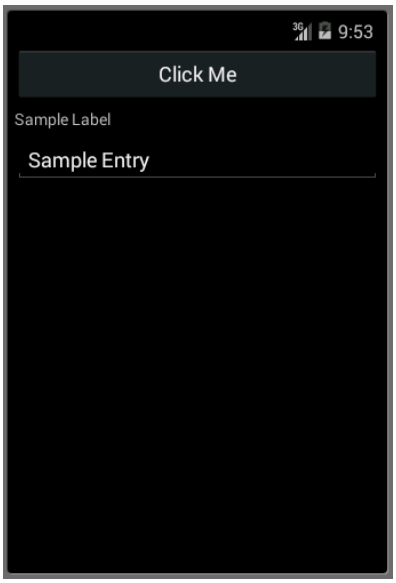


Figure 31: Binding on Android

Figure 32: Binding on iOS

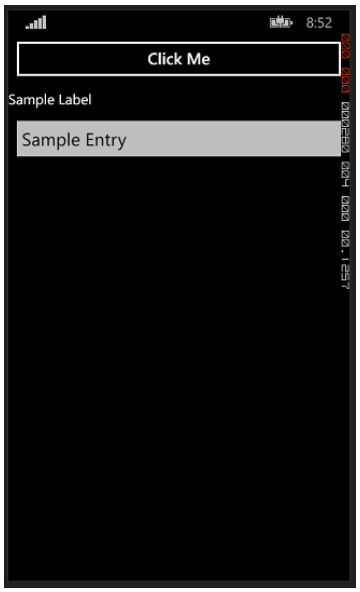


Figure 33: Binding on Windows Phone

Success! You have now assigned a value to a XAML property without explicitly setting it. You can now control the values of properties in our UI using the **`Binding`** markup extension at runtime. This is a great step forward, but poses a couple additional questions. How can you update the values in the UI without explicitly assigning it in the code-behind, and what about handling events? Don't worry; both of these scenarios will be handled by the implementation of a couple simple interfaces: **`INotifyPropertyChanged`** and **`ICommand`**.

INotifyPropertyChanged

`INotifyProptertyChanged` is a simple, yet very powerful interface that helps facilitate the automatic updating of UI elements from the **`BindingContext`** and vice versa. The **`INotifyPropertyChanged`** definition is found in the **`System.ComponentModel`** namespace and looks like the following.

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}
```

Code Listing 16

As you can see, it only contains a handler for the **`PropertyChanged`** event. While this seems relatively unassuming, this is extremely important. By taking advantage of this interface within our **`SampleContext`**, the UI can subscribe to **`PropertyChanged`** events for our **`LabelText`** and **`EntryText`** properties so the UI always matches the data in the **`BindingContext`**. Let's update our code to take advantage of this functionality.

```
public class SampleContext : INotifyPropertyChanged
{
    private string _labelText;
    public string LabelText {
         get { return _labelText; }
         set {
              if(_labelText == value)
                 return;

              _labelText = value;
              OnPropertyChanged("LabelText");
             }
         }

    private string _entryText;
    public string EntryText {
         get { return _entryText; }
         set {
               if(_entryText == value)
                   return;

                _entryText = value;
                OnPropertyChanged("EntryText");
         }
     }

     public event PropertyChangedEventHandler PropertyChanged;
     protected virtual void OnPropertyChanged(string propertyName)
     {
         PropertyChangedEventHandler handler = PropertyChanged;
         if (handler != null)
             handler(this, new PropertyChangedEventArgs(propertyName));
     }

}
```

Code Listing 17

Now, we have added a few things to our class. First, there is now an **`OnPropertyChanged`** method that takes a **`propertyName`** string as an argument. This method will check to see if anything is subscribed to these changes, and if so, execute the handler to inform it which property has changed.

Also, in support of the new method, we have added two backing fields to contain the actual data associated with the properties. This now allows our property getter to simply return this value, and the setter to check to see if the value has actually changed. If the value has changed, we will then call the new **OnPropertyChanged** method which will in turn notify all the subscribers that a property has changed and that they must get the new value. This is how the XAML will ultimately know that something has changed and update the properties that are using the **Binding** markup extension.

If you run your application again, you will see that everything still works, but nothing is changing. In this case, we can't see our **INotifyPropertyChanged** actually working. To illustrate that it is truly functioning properly, we need to update one of our properties. To do that, we will introduce some interactivity.

ICommand

To wrap up this example and this section on XAML, we need to understand how to handle user activity through our UI. Maybe that activity is through editing some text, selecting a radio button, or clicking a button. In our example, we have a button, so let's add some event handling to that to update our properties. Under normal circumstances, you would probably want to add a button click event handler like we did in a previous exercise. Instead, now that we understand binding, wouldn't it be nice if we could do something similar to the property binding that we did earlier and apply it to events? Actually, we can do just that.

In order to attach event handling to XAML UI components, we need to introduce a new concept similar to the **INotifyPropertyChanged** interface. To handle these types of interactions we need to use a new interface found in the **System.ObjectModel** namespace called **ICommand**. The **ICommand** interface, similar to the **INotifyPropertyChanged** interface, is going to allow us to add bindings to UI elements that receive interaction from the user. First, let's take a look at the **ICommand** interface.

```
public interface ICommand
{
    event EventHandler CanExecuteChanged;

    bool CanExecute(object parameter);

    void Execute(object parameter);
}
```

Code Listing 18

As you can see, the **ICommand** interface has three members:

- **CanExecute**: Determines whether or not this command can execute.
- **CanExecuteChanged**: The event fired when the **CanExecute** flag has changed.
- **Execute**: The method that executes the desired functionality of the command.

By using this interface to our advantage, we can translate actions through the UI to some relevant handling mechanism in our code in a very similar fashion to a normal event handler. Now that we have this **ICommand** interface, let's create a sample command to see how we would handle the click of our button.

```
public class SampleCommand : ICommand
{
    private Action _toExecute;

    public SampleCommand( Action toExecute ) {
        _toExecute = toExecute;
    }

    public bool CanExecute( object parameter ) {
        return true;
    }

    public void Execute( object parameter ) {
        _toExecute( );
    }

    public event EventHandler CanExecuteChanged;
}
```

Code Listing 19

This is a very simple implementation of the **ICommand** interface. You can get as complex as you want, but this is sufficient to understand the basic concept. In this case, I want to make sure that the button is always executable, so the **CanExecute** method always returns true. As part of the constructor, I am passing in a generic delegate that will contain the logic I want executed every time the button is clicked. This way, when the **Execute** method is called, we simply execute this delegate. And that's it. Now let's take a look at how this gets wired up in **SampleContext**.

```
public class SampleContext : INotifyPropertyChanged
{
    public SampleContext( )
    {
        _buttonCommand = new SampleCommand(() => LabelText = DateTime
                                                             .Now
                                                             .ToString());
    }

    private ICommand _buttonCommand;
    public ICommand ButtonCommand {
        get { return _buttonCommand; }
    }
```

```
    // Rest of code removed for brevity
}
```

Code Listing 20

As you can see, this looks remarkably similar to the property bindings we did before. We create a private instance of an **ICommand** interface that is initialized during the constructor. In the constructor, the command is set to a new instance of the **SampleCommand** class, and the generic delegate is assigned as setting the **LabelText** property to **DateTime.Now.ToString()**. Finally, we have a public **ButtonCommand** property that simply returns our private instance.

The very last thing we need to do is wire up this command to our XAML page.

```
<!-- Rest removed for brevity -->

<Button Text="Click Me"
        Command="{Binding ButtonCommand}"/>

<!-- Rest removed for brevity -->
```

Code Listing 21

The only thing that changes is the **Button** element. We simply add a **Command** element and set its **Binding** markup extension to the public property **ButtonCommand**. Now when we execute this code, the XAML parser will use the **Binding** to the **ButtonCommand** to determine if it **CanExecute**. Then, when the button is clicked, the **Execute** method is called, and the **LabelText** is updated and shown on the screen.

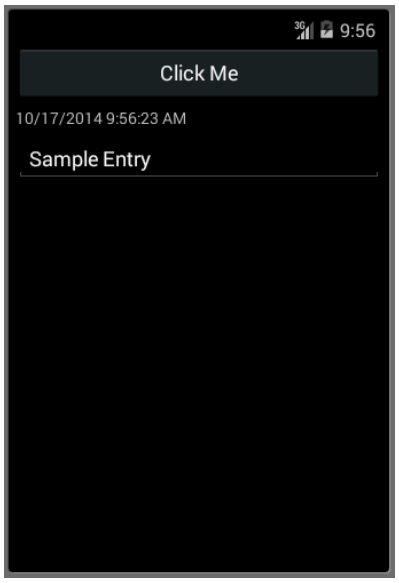


Figure 34: Updated binding on Android

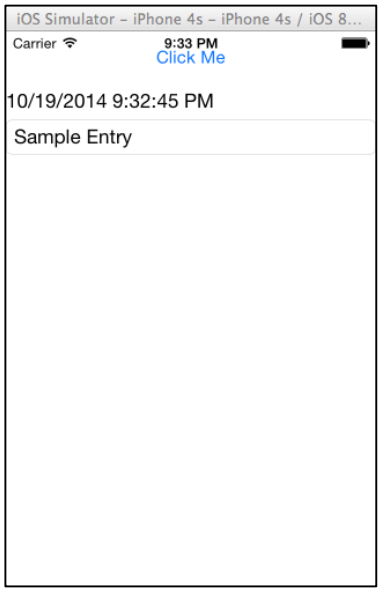


Figure 35: Updated binding on iOS

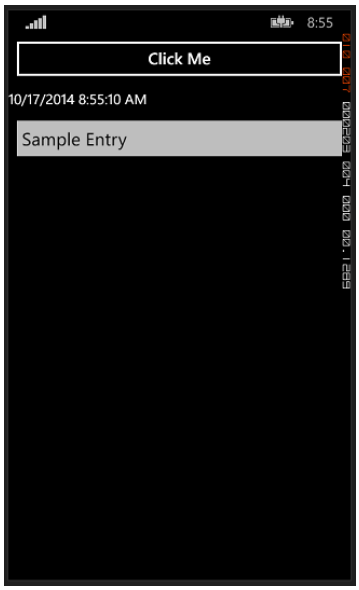


Figure 36: Updated binding on Windows Phone

Summary

In this chapter, we laid the groundwork for understanding the basic concepts behind XAML. Using this knowledge, you should now be able to go through the next chapters on the different UI aspects found within Xamarin.Forms and fully understand the differences between creating them in code and using XAML.

Chapter 5 Building User Interfaces

Now that you have a grasp on the basics of XAML, it's time to start digging into the building blocks of Xamarin.Forms applications. One of the first things to truly understand when building applications using Xamarin.Forms is how to create and lay out screens. In this chapter, we will discuss both of these concepts by walking through the different types of screens, or pages, that can be created out of the box with Xamarin.Forms, as well as the basic need for layouts and how they can be utilized.

If you are using the XAML versions of the code described in this chapter, you may need to change the **`x:Class`** attribute to reflect the name of the PCL project in your solution. In my samples, I ran the code from a project named **`XamlBasics`** in a XAML class named **`SampleXaml`**. If your naming convention doesn't match this, simply update the **`x:Class`** attribute to reflect your project name and class name respectively.

Creating Pages

When it comes to creating screens within your Xamarin.Forms applications, the foundational UI element is the **`Page`**. Think of a **`Page`** as a screen within your application to which you can add additional elements. From an implementation perspective though, a **`Page`** is merely a base class from which there are five options to choose from:

- **`ContentPage`**: Displays a single **`View`**.
- **`MasterDetailPage`**: Manages two separate **`Pages`** (panes) of information.
- **`NavigationPage`**: Manages the navigation of a stack of other **`Pages`**.
- **`TabbedPage`**: Allows navigation between several child **`Pages`** using tabs.
- **`CarouselPage`**: Allows swipe gestures between several sub-**`Pages`**.

ContentPage

The **`ContentPage`** is the simplest of all the **`Page`** classes. It is meant to only contain a single **`View`** to display to the user. A **`View`** is nothing more than the Xamarin.Forms term for a UI control or widget. At this point, the restriction to a single **`View`** or UI element may seem a little restrictive, but later on in this chapter, we will discuss a way around this using **`Layout`** controls that will allow us to add multiple elements to the screen.

`ContentPages` are typically used for simple screens within your application that use **`Layout`** controls to organize your UI elements.

In the examples presented so far in the book, we have primarily been using **`ContentPage`** elements as examples. We will create one more here just to illustrate a simple example of a **`ContentPage`**.

In the **`ContentPage`** class, there is a property named **`Content`** that can be assigned to via code or by placing the child (content) UI element within the **`ContentPage`** element.

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlBasics.SampleXaml">
    <Label Text="This is a simple ContentPage"
           HorizontalOptions="Center"
           VerticalOptions="Center" />
</ContentPage>
```

Code Listing 22

Code

```
var label = new Label {
    Text = "This is a simple ContentPage",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
};

var contentPage = new ContentPage {
    Content = label
};
```

Code Listing 23

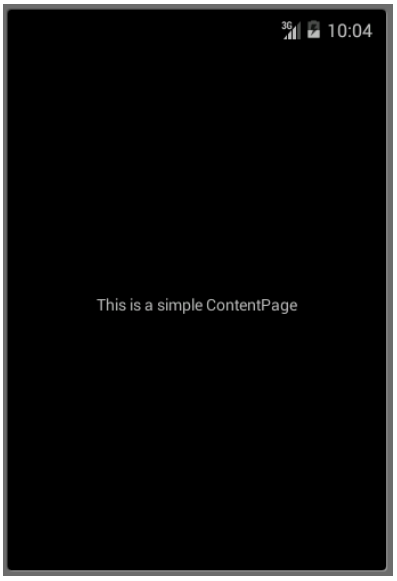


Figure 37: ContentPage on Android

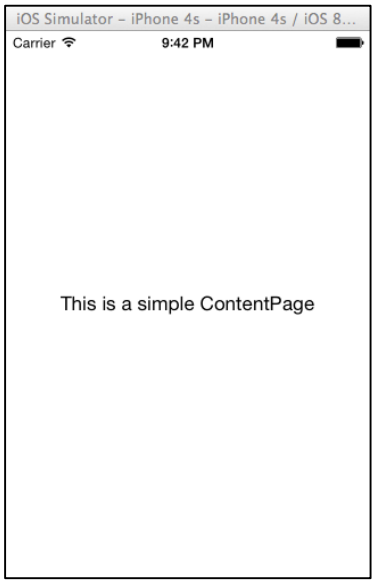


Figure 38: ContentPage on iOS

Figure 39: ContentPage on Windows Phone

MasterDetailPage

The **MasterDetailPage** within Xamarin.Forms is a very useful **Page** when you wish to split your content into two separate categories: generic and detailed. You will typically find this type of **Page** in a section of an application where you don't want to inundate users with more data than they want. This may or may not be presented in the form of a list of data, or you may start users on a page that contains the summary of a product and if they want more detail, they can swipe left to show more information.

This **Page** can also be used to create those very popular tray-style navigation views where the menu is hidden to the left or right and upon clicking a button, the menu can show itself and present users with some navigation options.

At its core, the **MasterDetailPage** consists of two **Page** objects that can easily be navigated between. The first **Page** is assigned to the **Master** property of the **MasterDetailPage** and the second **Page** is assigned to the **Detail** property. From there, the basic swipe gesture support is inherent in the **Page** itself so you don't have to implement it manually.

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  x:Class="XamlBasics.SampleXaml">
     <MasterDetailPage.Master>
          <ContentPage Title = "Master" BackgroundColor = "Silver">
               <Label Text="This is the Master page."
                      TextColor = "Black"
                      HorizontalOptions="Center"
                      VerticalOptions="Center" />
          </ContentPage>
     </MasterDetailPage.Master>
     <MasterDetailPage.Detail>
          <ContentPage>
               <Label Text="This is the Detail page."
                      HorizontalOptions="Center"
                      VerticalOptions="Center" />
          </ContentPage>
     </MasterDetailPage.Detail>
</MasterDetailPage>
```

Code Listing 24

Code

```
var masterDetailPage = new MasterDetailPage {
     Master = new ContentPage {
          Content = new Label {
                              Title = "Master",
               BackgroundColor = Color.Silver,
```

```
                TextColor = Color.Black,
                Text = "This is the Master page.",
                HorizontalOptions = LayoutOptions.Center,
                VerticalOptions = LayoutOptions.Center
            }
        },
        Detail = new ContentPage {
            Content = new Label {
                Title = "Detail",
                Text = "This is the Detail page.",
                HorizontalOptions = LayoutOptions.Center,
                VerticalOptions = LayoutOptions.Center
            }
        }
};
```

Code Listing 25

When you run this application, you will first be presented with a screen that contains a label stating that it is the **`Detail`** page.

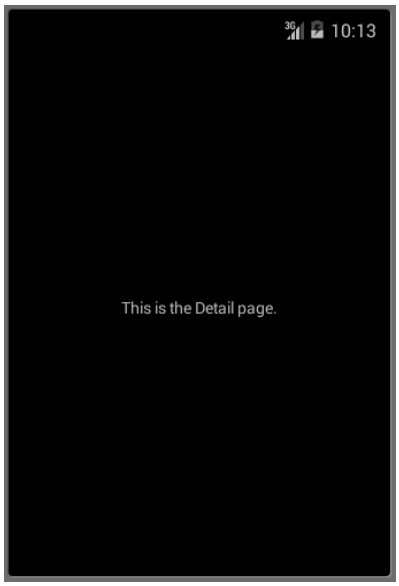


Figure 40: Detail page of MasterDetailPage on Android

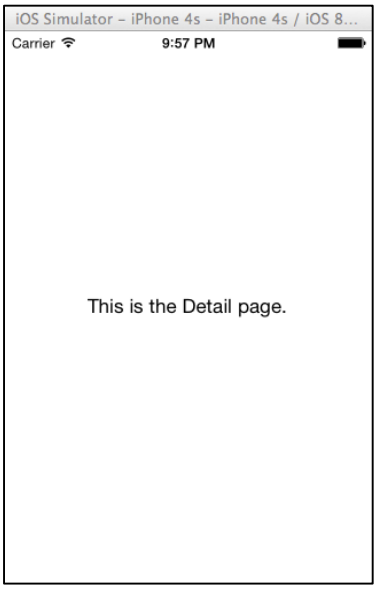


Figure 41: Detail page of MasterDetailPage on iOS

Figure 42: Detail page of MasterDetailPage on Windows Phone

After you swipe the screen, or click the button at the bottom in the Windows Phone example, you will be presented with the **`Master`** page and a label that confirms it.

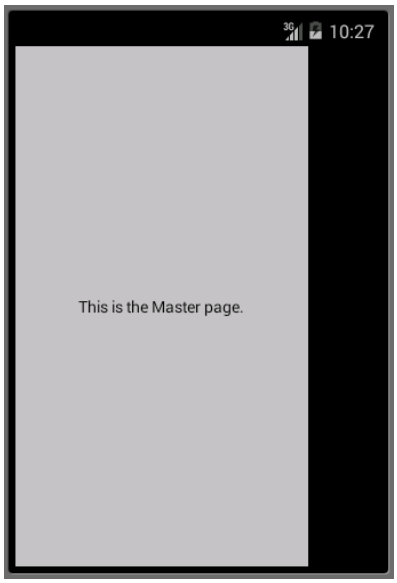


Figure 43: Master page of MasterDetailPage on Android

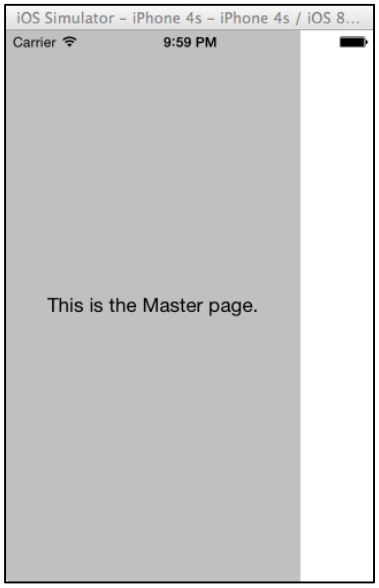


Figure 44: Master page of MasterDetailPage on iOS

Figure 45: Master page of MasterDetailPage on Windows Phone

NavigationPage

A **NavigationPage** in the world of Xamarin.Forms isn't much different than most navigation controls within the mobile space. The idea behind a **NavigationPage** is that it serves as a manager for a stack of **Pages** that the user can navigate between. Referring to this **Page** as a stack is not a coincidence. It truly is a stack. It has methods on it to **Push** and **Pop Page** objects to and from the stack.

The **NavigationPage** is a good option within your application if you are dealing with a decent number of individual screens that the user has the option to navigate between. The built-in functionality of the **NavigationPage** is a big help when handling these navigational concepts so you don't have to.

XAML

While the **NavigationPage** is, technically speaking, a **Page**, it doesn't act exactly like the rest of the **Page** classes. Although it is possible to create an instance of a XAML **NavigationPage**, you aren't going to see many, if any, created that way in the wild. That being the case, I will save you from yourself here. When it comes to creating a **NavigationPage**, you will want to stick with the code version as it is easier to understand and much more useful. This is typically due to the dynamic addition and removal of pages. At design time you don't know all the combinations of ways to navigate to and from each page.

Code

```
var rootLabel = new Label {Text = “Root Page”};
var anotherLabel = new Label {Text = “Another Label”};
var button = new Button {
    Text = “Next Page”
};

var stackLayout = new StackLayout {
    Children = {rootLabel, button}
};

var newPage = new ContentPage {
    Content = anotherLabel
};

var navPage = new NavigationPage(new ContentPage
{
    Title = “Root Nav Page”,
    Content = stackLayout
});

button.Clicked += ( sender, args ) => navPage.PushAsync( newPage );
```

Code Listing 26

When using a **NavigationPage** as a root **Page** in your application, it gives you access to a navigation bar at the top of the screen. In order to set the text on the navigation bar, the **NavigationPage** uses the **Title** property on its **CurrentPage**. This **CurrentPage** property is a **read-only** property, so in order to set it you will need to make sure that you have set the **Title** property on the **Page** that you are pushing onto the stack as in the previous code example.

When you run your application, you will initially see a screen similar to the following figures.

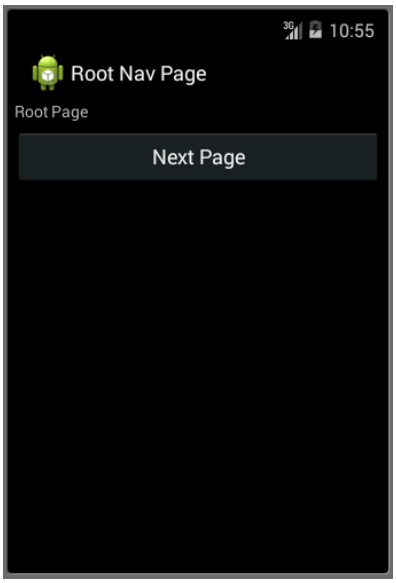


Figure 46: NavigationPage on Android

Figure 47: NavigationPage on iOS

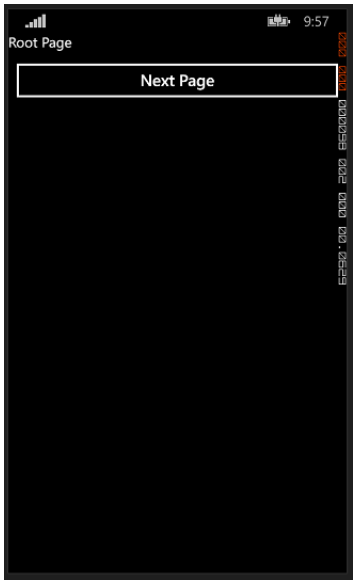


Figure 48: NavigationPage on Windows Phone

After clicking on the button, you will transition to the next page which will look like the following figures.

Figure 49: Next page of NavigationPage on Android

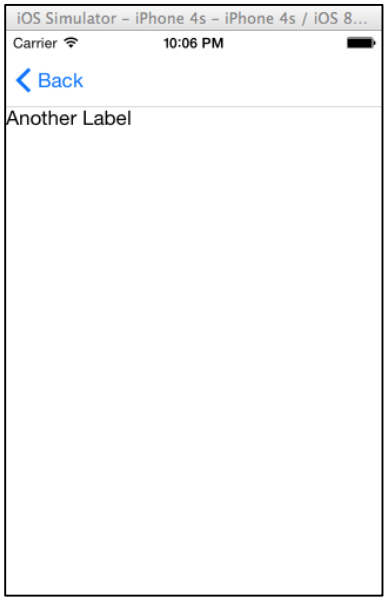


Figure 50: Next page of NavigationPage on iOS

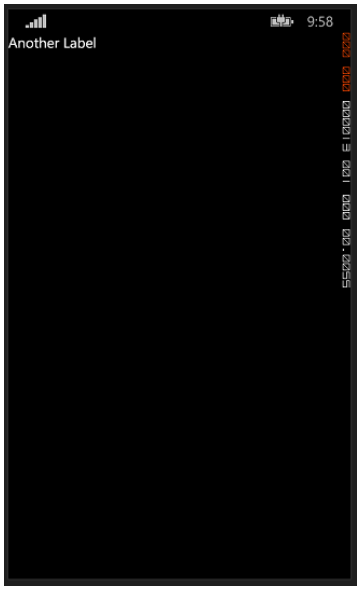


Figure 51: Next page of NavigationPage on Windows Phone

TabbedPage

A **TabbedPage** is similar to a **NavigationPage** in that it allows for and manages simple navigation between several child **Page** objects. The difference is that generally speaking, each platform displays some sort of bar at the top or bottom of the screen that displays most, if not all, of the available child **Page** objects.

In Xamarin.Forms applications, a **TabbedPage** is generally useful when you have a small predefined number of pages that users can navigate between, such as a menu or a simple wizard that can be positioned at the top or bottom of the screen.

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            x:Class="XamlBasics.SampleXaml">
  <TabbedPage.Children>
    <ContentPage Title="Tab1">
      <Label Text="I'm the Tab1 Page"
             HorizontalOptions="Center"
             VerticalOptions="Center"/>
    </ContentPage>
    <ContentPage Title="Tab2">
    <Label Text="I'm the Tab2 Page"
           HorizontalOptions="Center"
           VerticalOptions="Center"/>
    </ContentPage>
  </TabbedPage.Children>
</TabbedPage>
```

Code Listing 27

Code

```
var page1 = new ContentPage {
    Title = "Tab1",
    Content = new Label {
        Text = "I'm the Tab1 Page",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    }
};

var page2 = new ContentPage {
    Title = "Tab2",
    Content = new Label {
        Text = "I'm the Tab2 Page",
        HorizontalOptions = LayoutOptions.Center,
```

```
        VerticalOptions = LayoutOptions.Center
    }
};

var tabbedPage = new TabbedPage {
     Children = { page1, page2 }
};
```

Code Listing 28

Another similarity between the **NavigationPage** and the **TabbedPage** lies in the use of the child **Page** **Title** property. In the **NavigationPage** example, the **Title** property of the child **Page** is used to populate the navigation bar title. When it comes to the **TabbedPage**, the child **Page** **Title** property is used for the tab text. Once again, if you wish to populate the tab buttons with custom text, it is best to set the **Title** property of the child **Page** when adding it to the **Children** collection of the **TabbedPage**.

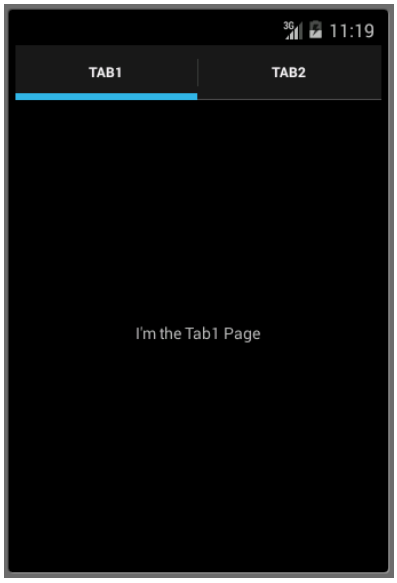


Figure 52: TabbedPage on Android

Figure 53: TabbedPage on iOS

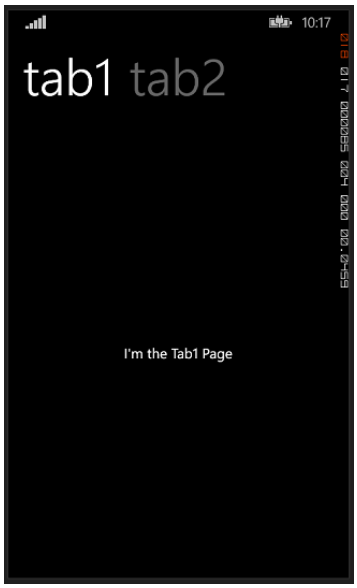


Figure 54: TabbedPage on Windows Phone

Clicking on **Tab2** on the interface will transition to the next tab.

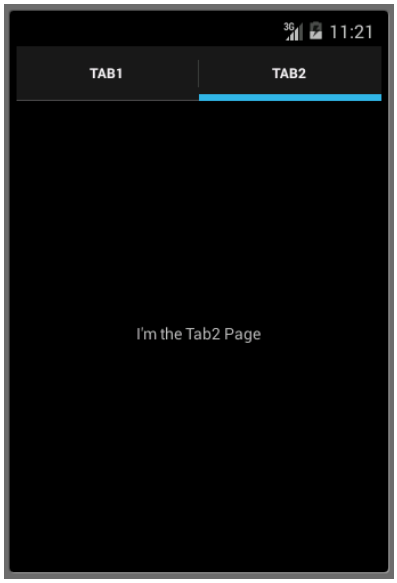


Figure 55: Second tab of TabbedPage on Android

Figure 56: Second tab of TabbedPage on iOS

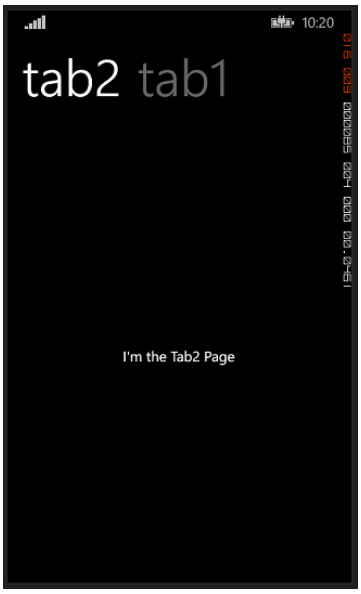


Figure 57: Second tab of TabbedPage on Windows Phone

CarouselPage

The final predefined **Page** in the Xamarin.Forms world comes in the form of the **CarouselPage**. Once again, the **CarouselPage** is a way to manage a collection of child **Page** objects that provides the ability to switch between them with a simple swipe gesture.

A very common scenario that lends itself well to the use of the **CarouselPage** is a gallery. A gallery is typically considered a collection of similar content, such as photos, that can be displayed one at a time to users.

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class="XamlBasics.SampleXaml"
              Title="Carousel Page">
  <CarouselPage.Children>
    <ContentPage>
      <Label Text="Hi, I'm Page1!"
             HorizontalOptions="Center"
             VerticalOptions="Center"/>
    </ContentPage>
    <ContentPage>
```

```
    <Label Text="Hi, I'm Page2"
           HorizontalOptions="Center"
           VerticalOptions="Center"/>
    </ContentPage>
  </CarouselPage.Children>
</CarouselPage>
```

Code Listing 29

Code

```
var page1 = new ContentPage {
        Content = new Label {
        Text = "Hi, I'm Page1!",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    }
};
var page2 = new ContentPage {
        Content = new Label {
        Text = "Hi, I'm Page2!",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    }
};

var carPage = new CarouselPage {
    Title = "Carousel",
    Children = {page1, page2}
};
```

Code Listing 30

A **CarouselPage** has its roots in the Windows Phone world. The Windows Phone platform has a specific look and feel when displaying a **CarouselPage** that includes a title that spans several child pages at the top. There is not a control on the iOS and Android platforms that emulate that concept. The **CarouselPage** will still work on those platforms, but will not have the exact same look and feel.

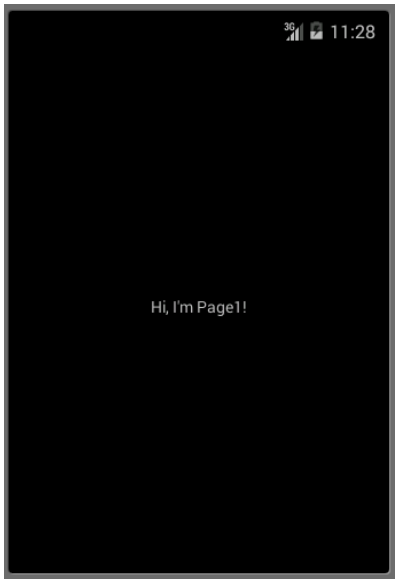


Figure 58: CarouselPage on Android

Figure 59: CarouselPage on iOS

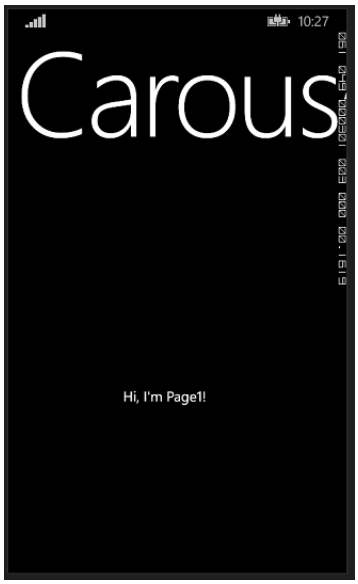


Figure 60: CarouselPage on Windows Phone

Scrolling to the left or right will transition to the next **Page**. In the emulators, to perform a scrolling action, you will need to click and drag the pointer left or right.

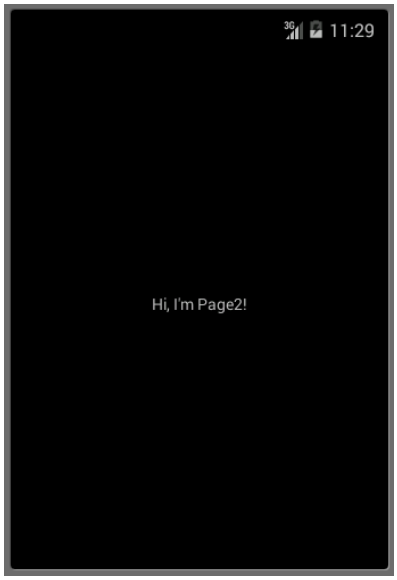


Figure 61: Second page of CarouselPage on Android

Figure 62: Second page of CarouselPage on iOS

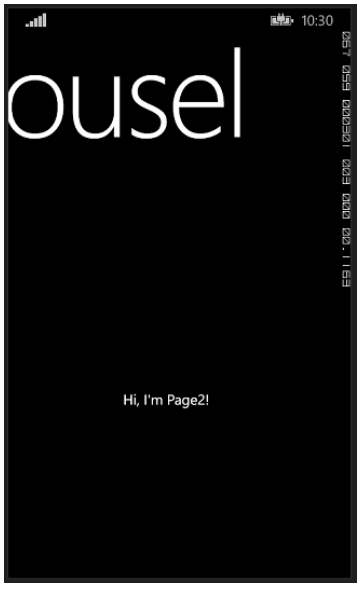


Figure 63: Second page of CarouselPage on Windows Phone

Adding Layouts

Earlier in this chapter, you were introduced the concept of a **Page**. During that introduction, you learned about the simplest built-in **Page** class in Xamarin.Forms: the **ContentPage**. If you recall, that **Page** can only contain a single UI element, or **View**. Initially, that may have seemed rather limiting, but as promised I am going to show you a way to get around that limitation.

If you look more closely at the **Content** property on a **ContentPage**, you will see that it accepts an object of type **View**. In the world of Xamarin.Forms, a **View** is another name for a control. In the next chapter we will go into more detail about some of the different **View** classes, but we are also going to introduce a few of them now.

There is a specialized kind of **View** object within Xamarin.Forms known as a **Layout**. The **Layout** controls act as containers for a collection of child elements, or **View** objects. Built into Xamarin.Forms are seven **Layout** classes:

- **StackLayout**: Automatically organizes child elements horizontally or vertically.
- **AbsoluteLayout**: Positions child elements at absolute coordinates.
- **RelativeLayout**: Positions child elements based on relative constraints.
- **Grid**: Contains a series of rows and columns to contain elements.
- **ContentView**: Used as a base for user-defined elements.
- **ScrollView**: Allows for scrolling through its content if necessary.
- **Frame**: Contains a single child element and options for creating a border or frame around it.

Layout controls not only help contain multiple elements, but also provide different levels of positioning to those elements. Depending on the type of application you are building, there is bound to be a built-in **Layout** type that will prove helpful.

Over the rest of this section, I will be providing samples of these **Layout** classes and their basic usage. To reduce the amount of XAML and C# code for you to view, always assume that these **Layout** controls are being added to the **Content** property of a **ContentPage** unless otherwise noted. In your application development, feel free to use these **Layout** controls in any of the different **Page** types that you wish.

StackLayout

A **StackLayout** is used to automatically position all of its child elements in a single line. The direction of the line is based on the **Orientation** property. The valid options for this property are **Horizontal** or **Vertical**. The default value, if not explicitly set, is **Vertical**.

When it comes to adjusting the amount of space between the child elements in the **StackLayout**, this should be done using the **Spacing** property and not at the individual child element level. During the layout process of a **StackLayout**, the user-assigned bounds will be overwritten and set to the value associated with the **StackLayout** control.

XAML

```
<StackLayout Orientation="Vertical" Spacing="25">
    <Button Text="I'm a Button"
            HorizontalOptions="Center"/>
    <Label Text="I'm a Label"
           HorizontalOptions="Center"/>
</StackLayout>
```

Code Listing 31

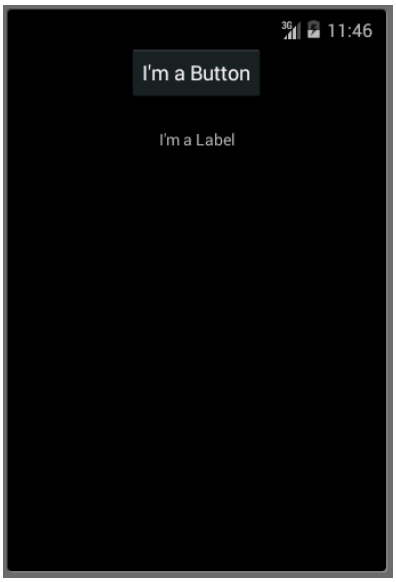
Code

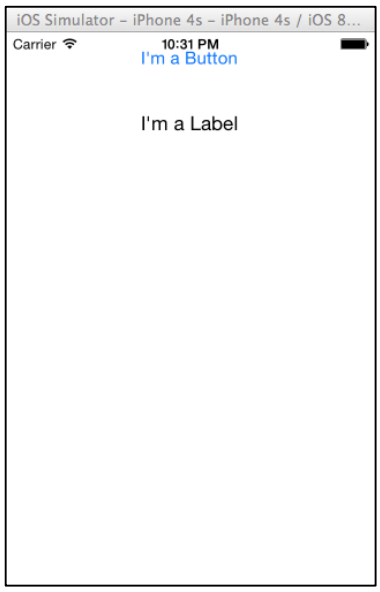
```
var button = new Button{
     Text = "I'm a Button",
      HorizontalOptions = LayoutOptions.Center
};

var label = new Label{
     Text = "I'm a Label",
     HorizontalOptions = LayoutOptions.Center
};

var stackLayout = new StackLayout {
     Spacing = 25,
     Children = { button, label }
};
```

Code Listing 32





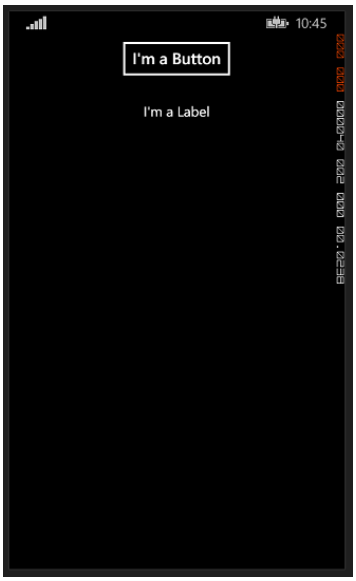


Figure 64: StackLayout on Android

Figure 65: StackLayout on iOS

Figure 66: StackLayout on Windows Phone

AbsoluteLayout

The **AbsoluteLayout** control in Xamarin.Forms allows you to specify where exactly on the screen you want the child elements to appear, as well as their size and shape (bounds). There are a few different ways to set the bounds of the child elements based on the **AbsoluteLayoutFlags** enumeration that are used during this process.

The **AbsoluteLayoutFlags** enumeration contains the following values:

- **All**: All dimensions are proportional.
- **HeightProportional**: Height is proportional to the layout.
- **None**: No interpretation is done.
- **PositionProportional**: Combines **XProportional** and **YProportional**.
- **SizeProportional**: Combines **WidthProportional** and **HeightProportional**.
- **WidthProportional**: Width is proportional to the layout.
- **XProportional**: X property is proportional to the layout.
- **YProportional**: Y property is proportional to the layout.

The process of working with the layout of the **AbsoluteLayout** container may seem a little counterintuitive at first, but with a little use it will become familiar. Once you have created your child elements, to set them at an absolute position within the container you will need to follow three steps. You will want to set the flags assigned to the elements using the **AbsoluteLayout.SetLayoutFlags()** method. You will also want to use the **AbsoluteLayout.SetLayoutBounds()** method to give the elements their bounds. Finally, you will want to add the child elements to the **Children** collection. Since Xamarin.Forms is an abstraction layer between Xamarin and the device-specific implementations, the positional values can be independent of the device pixels. This is where the layout flags mentioned previously come into play. You can choose how the layout process of the Xamarin.Forms controls should interpret the values you define.

XAML

```
<AbsoluteLayout>
    <BoxView Color="Red"
             AbsoluteLayout.LayoutBounds="0, 0, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />
    <BoxView Color="Blue"
             AbsoluteLayout.LayoutBounds="0.15, 0.15, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />
    <BoxView Color="Yellow"
             AbsoluteLayout.LayoutBounds="0.30, 0.30, 0.25, 0.25"
             AbsoluteLayout.LayoutFlags="All" />
    <BoxView Color="Purple"
```

```
            AbsoluteLayout.LayoutBounds=”0.45, 0.45, 0.25, 0.25”
            AbsoluteLayout.LayoutFlags=”All” />
    <BoxView Color=”Green”
            AbsoluteLayout.LayoutBounds=”0.60, 0.60, 0.25, 0.25”
            AbsoluteLayout.LayoutFlags=”All” />
</AbsoluteLayout>
```

Code Listing 33

Code

```
var redBox = new BoxView {
    Color = Color.Red
};
var blueBox = new BoxView {
    Color = Color.Blue
};
var yellowBox = new BoxView {
    Color = Color.Yellow
};
var purpleBox = new BoxView {
    Color = Color.Purple
};
var greenBox = new BoxView {
    Color = Color.Green
};

AbsoluteLayout.SetLayoutFlags(redBox, AbsoluteLayoutFlags.All);
AbsoluteLayout.SetLayoutBounds(redBox, new Rectangle(0, 0, 0.25, 0.25));

AbsoluteLayout.SetLayoutFlags(blueBox, AbsoluteLayoutFlags.All);
AbsoluteLayout.SetLayoutBounds(blueBox, new Rectangle(0.15, 0.15, 0.25, 0
.25));

AbsoluteLayout.SetLayoutFlags(yellowBox, AbsoluteLayoutFlags.All);
AbsoluteLayout.SetLayoutBounds(yellowBox, new Rectangle(0.30, 0.30, 0.25,
 0.25));

AbsoluteLayout.SetLayoutFlags(purpleBox, AbsoluteLayoutFlags.All);
AbsoluteLayout.SetLayoutBounds(purpleBox, new Rectangle(0.45, 0.45, 0.25,
 0.25));

AbsoluteLayout.SetLayoutFlags(greenBox, AbsoluteLayoutFlags.All);
AbsoluteLayout.SetLayoutBounds(greenBox, new Rectangle(0.60, 0.60, 0.25,
0.25));

var absolute = new AbsoluteLayout {
```

```
    Children = {redBox, blueBox, yellowBox, purpleBox, greenBox}
};
```

Code Listing 34

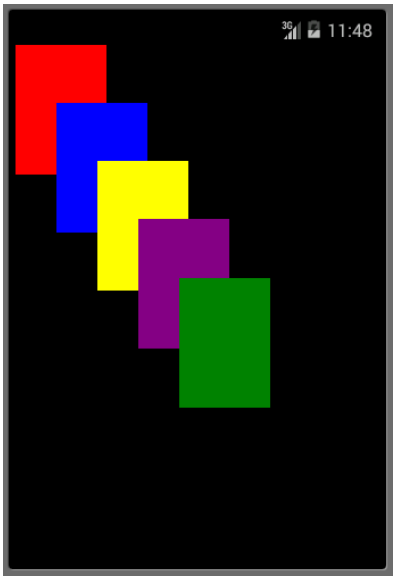


Figure 67: AbsoluteLayout on Android

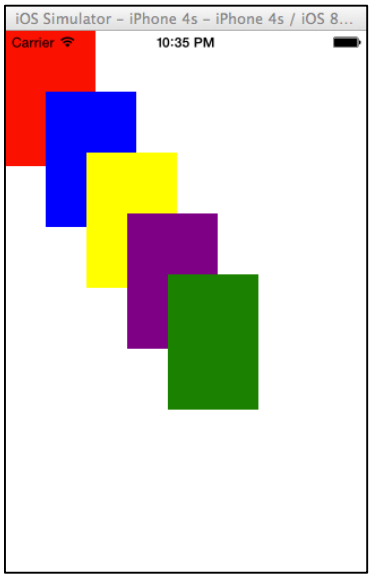


Figure 68: AbsolutionLayout on iOS

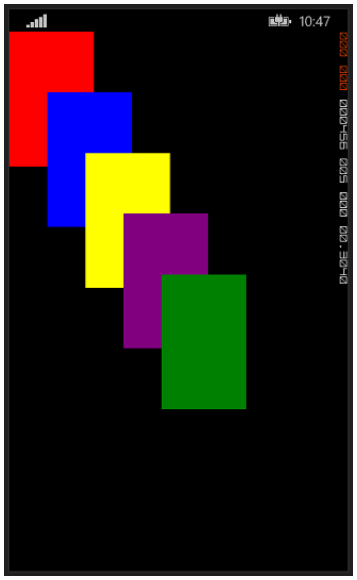


Figure 69: AbsoluteLayout on Windows Phone

This is a very good example of the succinctness of using XAML over using C# code. Either way is acceptable and will reach the desired result, but with XAML you can accomplish the same effect in less than half the code.

RelativeLayout

The **`RelativeLayout`** container is a mechanism for specifying the location of child elements relative either to each other or to the parent control. In order to achieve the relative locations, you need to create a series of **`Constraint`** objects that define each particular child element's relative position to another.

When working with a **`RelativeLayout`** container in XAML, you will quickly notice some differences from what you have worked with up to this point. In most cases, you are able to set the properties of an object using a simple assignment to an attribute, but there are instances where that syntax doesn't offer enough. That is where we require more functionality within the markup that is ultimately backed by code somewhere. This additional functionality within XAML is referred to as a **markup extension**. Markup extensions are indicated within XAML by attribute settings that are delimited by curly braces.

There are several different types of markup extensions built into XAML such as **StaticResource**, **x:Static**, and **Binding** (as seen in Chapter 4). Within the XAML example of a **RelativeLayout**, we will be using the **ConstraintExpression** extension. The **ConstraintExpression** is used to specify the location or size of a child view as a constant, or relative to a parent or another named view.

In the following example, you will be introduced to a new feature of XAML: attached properties. Attached properties are a type of global property that can be accessed by any object. In the **RelativeLayout** class, there are properties named **XConstraint** and **YConstraint**. In the following example, you will want to assign a value to these properties from within another XAML element. To get access to these properties from a nested element, you simply prefix the property name with the class that it comes from, as in **RelativeLayout.XConstraint**.

XAML

```
<RelativeLayout>
    <BoxView Color="Red"
             RelativeLayout.XConstraint="{ConstraintExpression Type=Const
ant, Constant=0}"
             RelativeLayout.YConstraint="{ConstraintExpression Type=Const
ant, Constant=0}"/>
    <BoxView Color="Blue"
             RelativeLayout.XConstraint="{ConstraintExpression Type=Relat
iveToParent, Property=Width, Factor=1, Constant=-40}"
             RelativeLayout.YConstraint="{ConstraintExpression Type=Relat
iveToParent, Property=Height, Factor=1, Constant=-40}"/>
    <BoxView Color="Yellow"
             x:Name="center"
             RelativeLayout.XConstraint=
                 "{ConstraintExpression Type=RelativeToParent,
                                        Property=Width,
                                        Factor=0.33}"
             RelativeLayout.YConstraint=
                 "{ConstraintExpression Type=RelativeToParent,
                                        Property=Height,
                                        Factor=0.33}"
             RelativeLayout.WidthConstraint=
                 "{ConstraintExpression Type=RelativeToParent,
                                        Property=Width,
                                        Factor=0.33}"
             RelativeLayout.HeightConstraint=
                 "{ConstraintExpression Type=RelativeToParent,
                                        Property=Height,
                                        Factor=0.33}" />
    <BoxView Color="Purple"
                 RelativeLayout.XConstraint=
                 "{ConstraintExpression Type=RelativeToView,
                                        ElementName=center,
                                        Property=X}"
```

```
            RelativeLayout.YConstraint=
                "{ConstraintExpression Type=RelativeToView,
                                       ElementName=center,
                                       Property=Y}"
            RelativeLayout.WidthConstraint=
                "{ConstraintExpression Type=RelativeToView,
                                       ElementName=center,
                                       Property=Width,
                                       Factor=0.33}"
            RelativeLayout.HeightConstraint=
                "{ConstraintExpression Type=RelativeToView,
                                       ElementName=center,
                                       Property=Height,
                                       Factor=0.33}" />
  </RelativeLayout>
```

Code Listing 35

Code

```
var redBox = new BoxView {
    Color = Color.Red
};
var blueBox = new BoxView {
    Color = Color.Blue
};
var yellowBox = new BoxView {
    Color = Color.Yellow
};
var purpleBox = new BoxView {
    Color = Color.Purple
};

var relativeLayout = new RelativeLayout( );
relativeLayout.Children.Add(redBox,
                            Constraint.Constant(0),
                            Constraint.Constant(0) );
relativeLayout.Children.Add(blueBox,
                            Constraint.RelativeToParent((parent) => paren
t.Width - 40 ),
                            Constraint.RelativeToParent((parent) => paren
t.Height - 40 ));
relativeLayout.Children.Add(yellowBox,
                            Constraint.RelativeToParent(parent => parent.
Width / 3),
                            Constraint.RelativeToParent(parent => parent.
Height / 3),
                            Constraint.RelativeToParent(parent => parent.
Width / 3),
```

```
                                Constraint.RelativeToParent(parent => parent.
Height / 3) );
relativeLayout.Children.Add(purpleBox,
                                Constraint.RelativeToView(yellowBox, (parent,
 sibling) => sibling.X),
                                Constraint.RelativeToView(yellowBox, (parent,
 sibling) => sibling.Y),
                                Constraint.RelativeToView(yellowBox, (parent,
 sibling) => sibling.Width / 3),
                                Constraint.RelativeToView(yellowBox, (parent,
 sibling) => sibling.Height / 3));
```

Code Listing 36

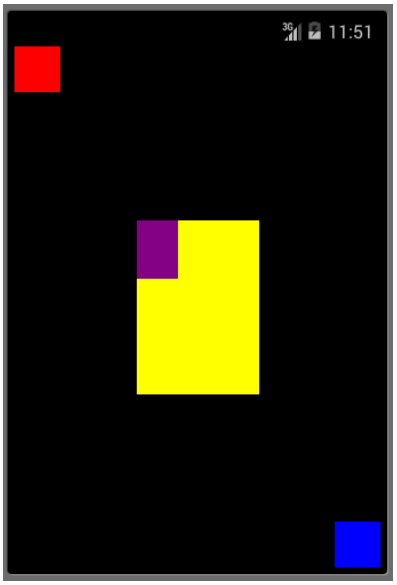


Figure 70: RelativeLayout on Android

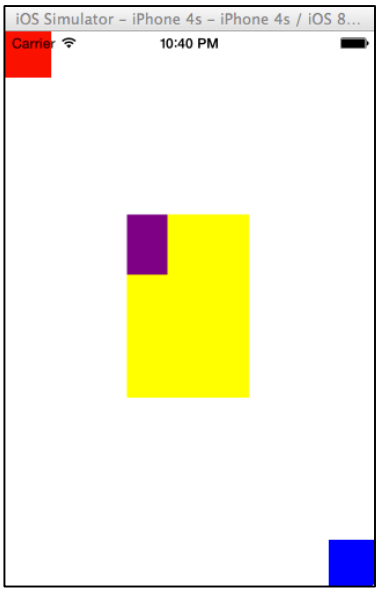


Figure 71: RelativeLayout on iOS

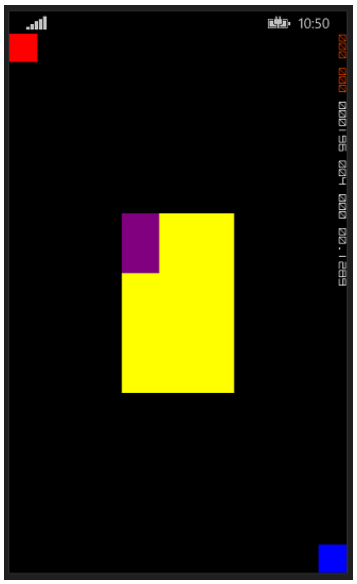


Figure 72: RelativeLayout on Windows Phone

Grid

The **`Grid`** container is a very useful option if you are looking to divide your screen into a series of rows and columns in which you can place child elements. The process of working with a **`Grid`** starts with creating a number of **`RowDefinition`** and **`ColumnDefinition`** objects. Once you have created these, you are free to assign child elements into a cell based on its coordinates. Once an element is placed within a cell, any layout properties assigned to that element will be applied to it with respect to the cell it occuppies.

When it comes to defining the **Height** and **Width** values in the **RowDefinition** and **ColumnDefinition** objects, you have three options. You can define an explicit value that the row or column will try to honor based on the amount of space available on the screen. You can choose to set the value to **auto**—this value will automatically size the row or column based on the size of its contents. Finally, you can use the star value (*****) which will allow that row or column to take all remaining space that is not used by anything else. The following example uses star sizing.

XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <Label Text="Cell 0, 0" Grid.Row="0" Grid.Column="0" BackgroundColor=
"Red" XAlign="Center" YAlign="Center"/>
    <Label Text="Cell 2, 2" Grid.Row="2" Grid.Column="2" BackgroundColor=
"Blue" XAlign="Center" YAlign="Center"/>
    <Label Text="Cell 1, 0 With Span" TextColor="Black" Grid.Row="1" Grid
.Column="0" Grid.ColumnSpan="3" BackgroundColor="Yellow" XAlign="Center"
YAlign="Center" />

</Grid>
```

Code Listing 37

Code

```
var grid = new Grid {
    RowDefinitions = new RowDefinitionCollection {
        new RowDefinition {Height = new GridLength( 1, GridUnitType.Star
)},
        new RowDefinition {Height = new GridLength( 1, GridUnitType.Star
)},
        new RowDefinition {Height = new GridLength( 1, GridUnitType.Star
)},
    },
    ColumnDefinitions = new ColumnDefinitionCollection {
        new ColumnDefinition {Width = new GridLength( 1, GridUnitType.Sta
r )},
        new ColumnDefinition {Width = new GridLength( 1, GridUnitType.Sta
```

```
r )},
        new ColumnDefinition {Width = new GridLength( 1, GridUnitType.Sta
r )},
    }
};

var label1 = new Label {
    Text = "Cell 0, 0",
    BackgroundColor = Color.Red,
    XAlign = TextAlignment.Center,
    YAlign = TextAlignment.Center
};

var label2 = new Label {
    Text = "Cell 2, 2",
    BackgroundColor = Color.Blue,
    XAlign = TextAlignment.Center,
    YAlign = TextAlignment.Center
};

var label3 = new Label
{
    Text = "Cell 1, 0 With Span",
    BackgroundColor = Color.Yellow,
    TextColor = Color.Black,
    XAlign = TextAlignment.Center,
    YAlign = TextAlignment.Center
};

grid.Children.Add(label1, 0, 0);
grid.Children.Add(label2, 2, 2);
grid.Children.Add( label3, 0, 1);
Grid.SetColumnSpan(label3, 3);
```

Code Listing 38

The previous code example will result in the following.

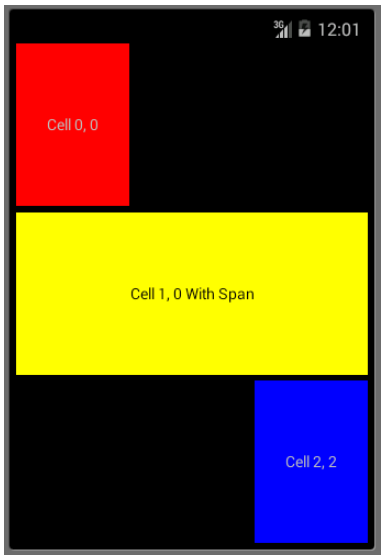


Figure 73: Grid on Android

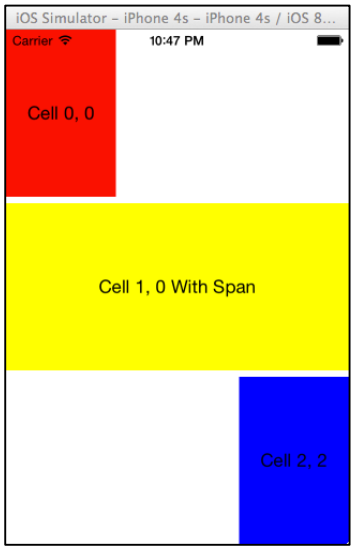


Figure 74: Grid on iOS

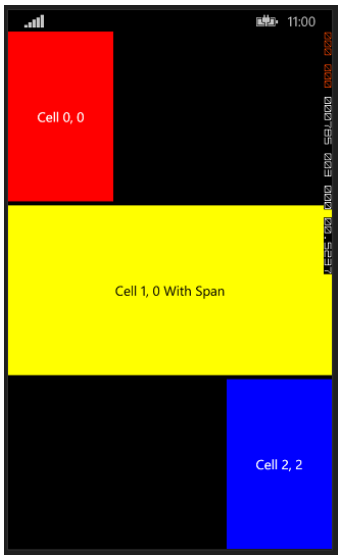


Figure 75: Grid on Windows Phone

ContentView

The **`ContentView`** container is definitely not one of the more interesting built-in **`Layout`** controls that come with Xamarin.Forms. You would typically use a **`ContentView`** for a couple of different reasons. You may want to create your own type of layout control with certain characteristics and functionality. In that case, the blank slate of a **`ContentView`** may just be the right starting point. Additionally, you may need a simple, no frills container that you can use to simply add some color or padding around another control or set of controls. In that case, a **`ContentView`** would definitely serve the purpose.

You can also think of a **`ContentView`** as a cousin to the **`ContentPage`**. Both controls only contain a single **`Content`** property that can have a single **`View`** assigned to it. So if you need to get fancy with it, you will need to use another container **`View`** to hold more than just that single **`View`**.

XAML

```
<ContentView>
    <Label Text="Hi, I'm a simple Label inside of a simple ContentView"
           HorizontalOptions="Center"
           VerticalOptions="Center"/>
</ContentView>
```

Code Listing 39

Code

```
var contentView = new ContentView {
    Content = new Label {
        Text = "Hi, I'm a simple Label inside of a simple ContentView",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    }
};
```

Code Listing 40

This code example will display the following.

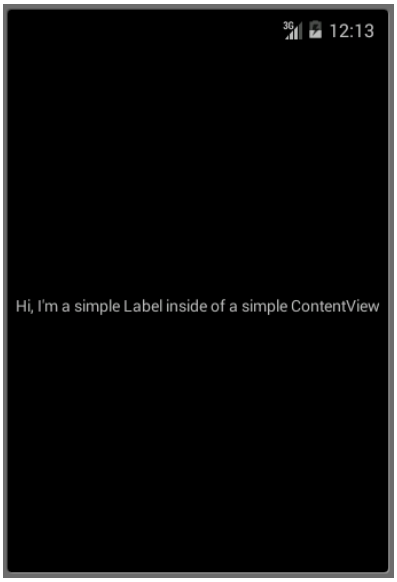


Figure 76: ContentView on Android

Figure 77: ContentView on iOS

Figure 78: ContentView on Windows Phone

ScrollView

A **`ScrollView`** is a container that has the unique capability of scrolling if its contents are too large for the screen to display. You will typically see this type of container used in situations where there is a large amount of text to be shown to the user, but it doesn't all fit within the bounds of a single **`View`**. This will allow users to scroll through the text with a swipe gesture to make the **`View`** scroll either vertically or horizontally depending on the **`Orientation`** property.

When using a **`ScrollView`**, it is not recommended to nest it inside of other elements with scrolling capabilities. The scrolling will continue to work, but will cause usability issues and confusion for users. When using a **`ScrollView`**, you will typically want to either use it by itself or if you have to nest it, nest it in something stationary.

XAML

```
<ScrollView>
     <Label Text="Very Long Lorem Ipsum" />
</ScrollView>
```

Code Listing 41

Code

```
var scrollView = new ScrollView {
     Content = new Label {
          Text = "Very Long Lorem Ipsum"
     }
};
```

Code Listing 42

An example of the previous code example is shown in the following figures.

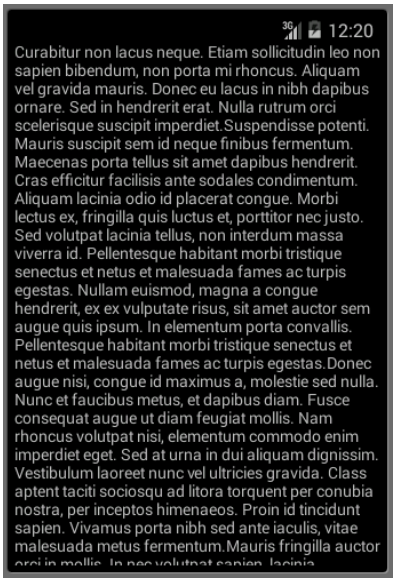


Figure 79: ScrollView on Android

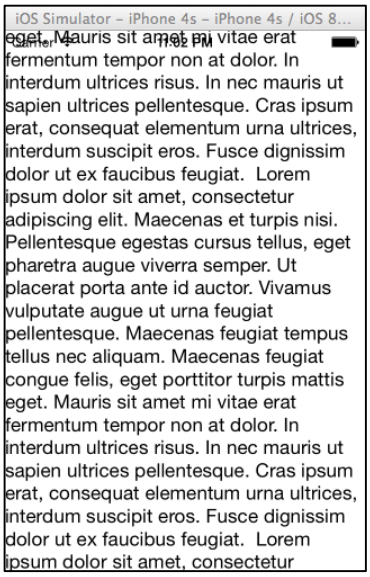


Figure 80: ScrollView on iOS

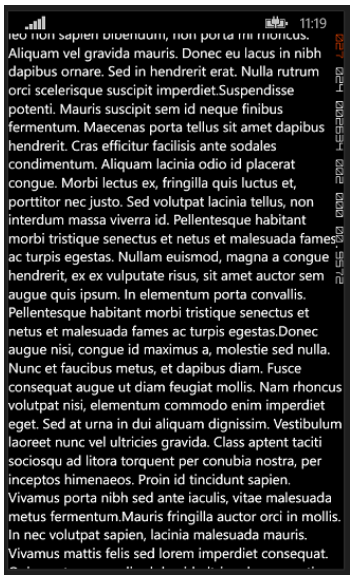


Figure 81: ScrollView on Windows Phone

Frame

The **`Frame`** is an interesting container within the Xamarin.Forms world. Its sole purpose is to provide a simple mechanism for adding some framing options around a single child element. Some of the framing options are relatively simple, like **`Padding`**. **`Padding`** is used to add some extra space around a particular element. It also contains a couple other more interesting options such as the **`OutlineColor`** of an element, and whether it has a shadow.

Although this seemingly simple layout control can make its child element seem very interesting, I advise you to use it sparingly. If you begin to give everything shadows and outline colors, nothing will stand out. Be sure to only use this in cases where a certain level of emphasis is absolutely necessary.

XAML

```
<Frame>
    <Label Text="I've been framed!"
           HorizontalOptions="Center"
           VerticalOptions="Center" />
</Frame>
```

Code Listing 43

Code

```
var frameView = new Frame {
    Content = new Label {
        Text = "I've been framed!",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    },
    OutlineColor = Color.Red
};
```

Code Listing 44

Figure 82: Frame on Android

Figure 83: Frame on iOS

Figure 84: Frame on Windows Phone

Summary

We have covered a lot of information in this chapter, but it is truly foundational when it comes to using Xamarin.Forms. We introduced the concept of **`Pages`** and the role they play in Xamarin.Forms. We explored the different built-in types of **`Pages`** as well as how to use them from both XAML and code, and some of their basic use cases.

We also discussed the concept of **`Layouts`** and how to integrate them with **`Pages`** to overcome some of the restrictions of **`Pages`**. Similarly to **`Pages`**, we went through all the built-in **`Views`** that come with Xamarin.Forms and demonstrated how to use them from both XAML and code, and looked at different scenarios where they can be useful.

Now that you are getting more familiar with the basic concepts of **`Layouts`** and **`Pages`** within Xamarin.Forms and how to use them from both XAML and code, it's time to get into some of the more noticeable controls users will be interacting with.

Chapter 6 Common Controls

Now that you have a foundational knowledge of the concepts of **Pages** and **Layouts** within Xamarin.Forms, you are ready to tackle the next step: controls. This chapter is not meant to be a comprehensive discussion of all the controls available in Xamarin.Forms for a couple reasons. Once you have a good grasp on the concepts of how to use the basic controls, you will have the necessary skills to use just about any control in Xamarin.Forms. Also, in the near future, Xamarin may introduce new controls that won't be covered here. So in my opinion, you are better off learning how to use the common controls as opposed to being spoon fed all of them.

In this chapter we will be covering the basic concepts of Xamarin.Forms controls, or **Views**. We will also be going into detail on some of the basic controls that show up in nearly every mobile application. We will be discussing how to use the controls as well as some common use cases for each.

View Basics

Before digging into examples of **View** objects, it's important to understand what a **View** actually is. In the world of Xamarin.Forms, you can think of a **View** as a UI element or widget. Some common examples of these elements are buttons, labels, and entry boxes. **View** objects are really the building blocks (from a UI perspective) of mobile applications. But these are not the only **View** classes.

In the previous chapter, you were introduced to the family of **Layout** classes. **Layout** classes are also **View** classes. If you were to take a look at the classes within the **Layout** category, you would quickly see that they are all subclasses of the **View** class. That is why they have the unique ability to be placed into a **Content** property of a **Page** to act as a container for other **View** objects.

Common Properties

When it comes to working with **View** objects, you will find that regardless of the type, you will see some very common characteristics that are shared amongst all of them. You have seen a few of these common properties in the previous chapter, but here I will cover some of them in a little more detail.

HorizontalOptions and VerticalOptions

Both the **HorizontalOptions** and **VerticalOptions** properties are members of the **View** class and are used in a very similar manner. These properties can be assigned a **LayoutOptions struct**. This **struct** has a number of values defined including **Center**, **Start**, and **End**, as well as a few others. These properties are used to position the **View** within its parent. **Center** will place the element in the center of its parent while **Start** and **End** will place it either at the beginning of its parent or at the end.

Note: Some container View objects will ignore these properties and use their own values. StackLayout is an example of this type of container which contains a custom layout cycle.

GestureRecognizers

Each subclass of the **`View`** class is given a collection that will contain all of the configured **`GestureRecognizer`** objects it supports. Think of a **`GestureRecognizer`** as a way to add functionality or event capability to **`View`** objects. Some **`View`** classes, such as **`Button`**, already contain support for handling the **`Clicked`** event. Other classes, such as **`Label`**, don't have the same **`Clicked`** event. In that case, if you wanted to know when a specific **`Label`** was touched, you can add a new instance of the **`TapGestureRecognizer`** class to the **`GestureRecognizers`** collection. That way, you can know when a user has tapped the **`Label`**, and handle that action appropriately. We will cover the process of creating **`GestureRecognizers`** later in this chapter.

HeightRequest and WidthRequest

When working with the size (height and width) of a **`View`** within our application, you will probably be confused in the beginning about how to change the **`Height`** and **`Width`** properties. In Xamarin.Forms, these two properties are read-only. To change them, you will need to do so indirectly via the **`HeightRequest`** and **`WidthRequest`** properties. You will set these the same way you would **`Height`** and **`Width`**. When you change these properties they may not be reflected in the **`Height`** and **`Width`** of the **`View`** immediately.

When a **`View`** is added to a **`Layout`**, that **`View`** becomes part of its layout cycle. The layout cycle prepares the views in a **`Layout`** for rendering, taking into account the layout options, position, size, rotation, and their effect on adjacent views. During this layout cycle, at certain times and events, the **`GetSizeRequest()`** method of each child **`View`** within the **`Layout`** gets called. This is the point when the **`HeightRequest`** and **`WidthRequest`** will be made available and applied.

There are also two other similar properties, **`MinimumHeightRequest`** and **`MinimumWidthRequest`**, that specify a minimum acceptable **`Height`** and **`Width`** that can be used during overflow handling in the layout cycle. These requests are taken into consideration during the layout cycle and Xamarin.Forms will do its best to accommodate them.

IsEnabled and IsVisible

Often times when working with Xamarin.Forms you will want to change whether or not a **`View`** is enabled or visible to users. That is what the **`IsEnabled`** and **`IsVisible`** properties are for. The **`IsEnabled`** property doesn't affect the visibility as the **`View`** stays within the visual tree of the application. Think of the visual tree of a Xamarin.Forms application as a collection of all the UI components that are drawn on the screen and presented to users. These components are generally used for visual cues and interaction. When you set the **`IsEnabled`** property to **`false`**, the user is simply not able to interact with it. **`IsVisible`**, on the other hand, will completely remove the **`View`** from the visual tree, causing it to no longer take up space and remove the ability of the user to interact with it.

Rotation, Translation, and Scale

You are not stuck with having all of your **`View`** objects displayed right to left horizontally within your application. You can change this characteristic of your application by using five different properties:

- **`RotationX`**: Rotates the **`View`** around the X-axis.
- **`RotationY`**: Rotates the **`View`** around the Y-axis.
- **`TranslationX`**: Moves the **`View`** along the X-axis.
- **`TranslationY`**: Moves the **`View`** along the Y-axis.
- **`Scale`**: Scale factor of the **`View`**.

Using these properties may not seem very useful in the beginning, but if you need to get into animations, they prove useful in rotating, moving, zooming, and changing the size of your **`View`**.

Note: Translating a View outside the bounds of its parent container may prevent inputs from working.

BindableProperty

If you look through the API documentation[1], you will see a number of fields on the **`View`** class that end in **`Property`**. It's a safe assumption when you see this that it is a **`BindableProperty`**. A **`BindableProperty`** is a specialized property that is meant to facilitate data binding between instances of classes in code and instances of classes (elements) in XAML. As a general rule of thumb, if you see a property such as **`TextProperty`**, this property is a **`BindableProperty`** that will, through data-binding, affect the value of the actual **`Text`** property of the **`View`**.

As you learned in Chapter 4, data binding is the mechanism in which you can use the **`Binding`** markup extension in your XAML code to use the **`BindingContext`** property in the code-behind file to populate your UI. Unfortunately, you will not be able to just arbitrarily select which properties you want to use binding with. The only properties that support this feature are the properties that have a **`BindableProperty`** associated with it.

Common Views

In this section, we will be covering some of the most common **`View`** classes you will encounter in your Xamarin.Forms applications. It will be assumed that you understand the properties in the previous section as they will not be mentioned in any detail. What will be covered, though, are any properties, methods, and events that are specific to the individual **`View`** classes that are worth noting.

[1] The full Xamarin.Forms API documentation can be found at http://api.xamarin.com/?link=N%3aXamarin.Forms

Button

The **Button** is probably the most common control not only in mobile applications, but in any applications that have a UI. The concept of a button has too many purposes to list here. Generally speaking though, you will use a button to allow users to initiate some sort of action or operation within your application. This operation could include anything from basic navigation within your app, to submitting data to a web service somewhere on the Internet.

Properties

The following are some of the more commonly used properties when working with the **Button** class:

- **BorderColor**: Sets the color of the border around the **Button**.
- **BorderRadius**: Sets the radius of the edges of the border around the **Button**.
- **BorderWidth**: Sets the width of the border around the **Button**.
- **Font**: Sets the font of the text in the **Button**.
- **Image**: Sets the image next to the text in the **Button**.
- **Text**: Sets the text in the **Button**.
- **TextColor**: Sets the color of the text in the **Button**.

Events

There is only one event that is exposed from the **Button** class, and that is the **Clicked** event. This event is fired when the user clicks the **Button**.

XAML

```
<Button
    x:Name="MyButton"
    Text="Click Me!"
    TextColor="Red"
    BorderColor="Blue"
    VerticalOptions="Center"
    HorizontalOptions="Center"
    Clicked="Button_Clicked"/>
```

Code Listing 45

XAML Code-Behind

```
public void Button_Clicked( object sender, EventArgs args ) {

    MyButton.Text = "I've been clicked!";
```

```
}
```

Code Listing 46

Code

```
var button = new Button( ) {
    Text = "Hello, Forms !",
    VerticalOptions = LayoutOptions.CenterAndExpand,
    HorizontalOptions = LayoutOptions.CenterAndExpand,
    TextColor = Color.Red,
    BorderColor = Color.Blue,
};

button.Clicked += ( sender, args ) => {
    var b = (Button) sender;
    b.Text = "I've been clicked!";
};
```

Code Listing 47

Running the previous code example produces the following screens.

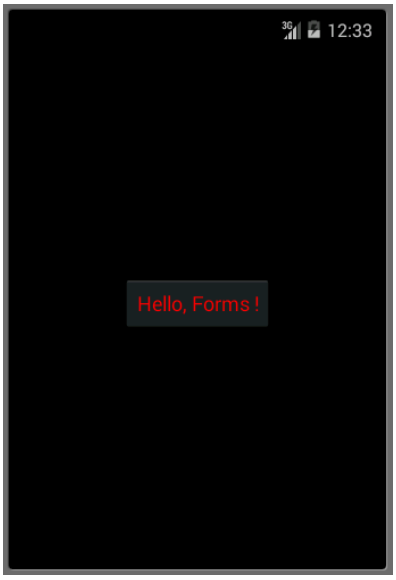


Figure 85: Button on Android

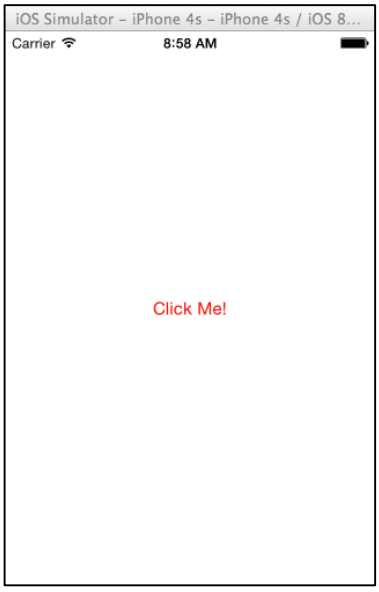


Figure 86: Button on iOS

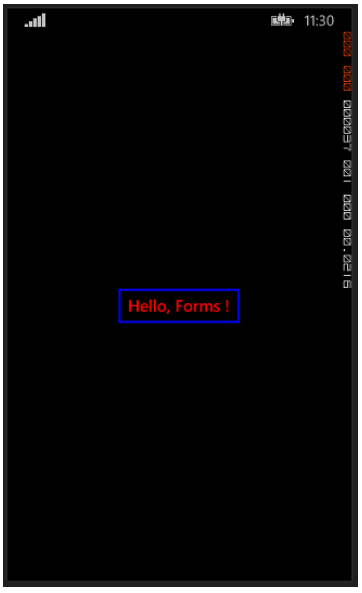


Figure 87: Button on Windows Phone

As you can see in the Figure 85, the button doesn't render a blue border on Android. This shows that not all properties in Xamarin.Forms have a representation on the individual platforms. You can modify these properties to render them on the individual platforms if you wish, but we will cover that kind of customization more in Chapter 7.

Clicking on the button in the previous examples results in the **`Text`** property changing.

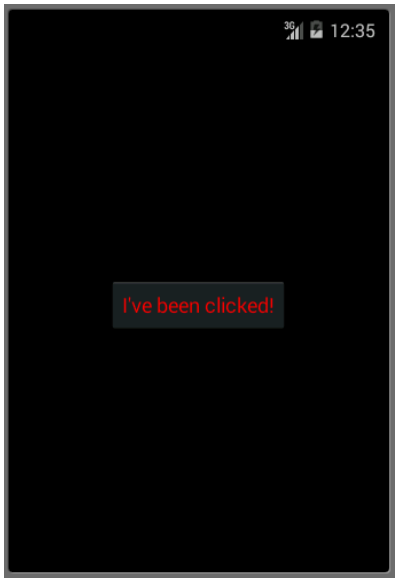


Figure 88: Clicked button on Android

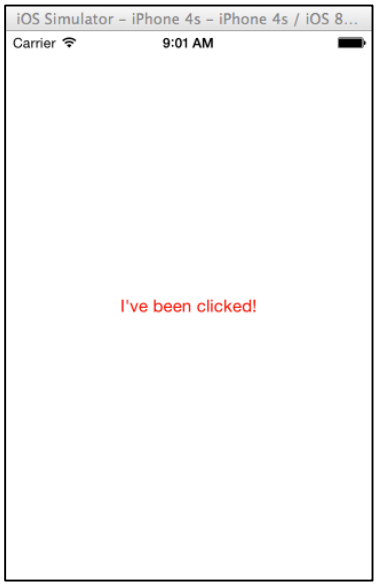


Figure 89: Clicked button on iOS

Figure 90: Clicked button on Windows Phone

ActivityIndicator

As your applications become more and more complex, you will find yourself doing things like making network calls, interacting with the device's file system, accessing a database, and other long running actions. One thing you will find out very quickly is that these types of operations take time. One issue you would typically have to watch out for would be doing these types of operations on the same thread that the UI runs on. Doing so causes the UI to become unresponsive and prevents the user from interacting with it. This is a very poor user experience.

Luckily for us, the process of writing asynchronous code has become much easier these days, but this causes other problems. Now we can take these long running operations and place them in a block of asynchronous code and the UI is no longer blocked. But nothing is happening and if this operation was spawned by the click of a button, users could potentially spawn another process to do the same operation. This can go on and on ad infinitum. To combat this, we can now programmatically change the **`IsEnabled`** property on certain **`View`** objects so the user can't do anything potentially dangerous. This may also be interpreted as a bad user experience. So while we change the interactivity of some of the **`View`** elements on the **`Page`**, we can also place an **`ActivityIndicator`** on the **`Page`** to show users that we are busy processing a request at the moment. At the end of the process, we can get rid of the **`ActivityIndicator`** and re-enable the **`View`** objects and go about our business.

Properties

There are only two properties that are specific to the **`ActivityIndicator`** control:

- **`Color`**: Sets the color of the **`ActivityIndicator`**.
- **`IsRunning`**: Sets whether or not the **`ActivityIndicator`** should be animated.

XAML

```
<ActivityIndicator IsRunning="true"/>
```

Code Listing 48

Code

```
var indicator = new ActivityIndicator {
     IsRunning = true
};
```

Code Listing 49

The following figures show some examples of what the **`ActivityIndicator`** control looks like on the different platforms.

Figure 91: ActivityIndicator on Android

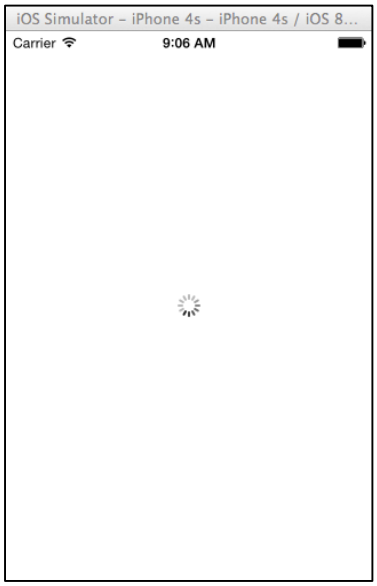


Figure 92: ActivityIndicator on iOS

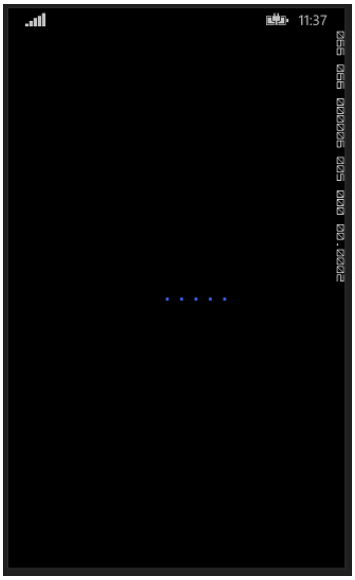


Figure 93: ActivityIndicator on Windows Phone

DatePicker

Quite often within mobile applications, there will be a reason to deal with dates. When working with dates, you will probably need some sort of user input to select a date. This could occur when working with a scheduling or calendar app. In this case, it is best to provide users with a specialized control that allows them to interactively pick a date, rather than requiring users to manually type a date. This is where the **`DatePicker`** control is really useful.

Properties

There are a few properties that are important to understand when working with the **`DatePicker`** control:

- **`Date`**: The date displayed on the **`DatePicker`**.
- **`Format`**: The format of the date on the **`DatePicker`**. Uses the same format strings associated with the **`DateTime.ToString(string format)`** method.
- **`MaximumDate`**: The highest selectable date of the **`DatePicker`**.
- **`MinimumDate`**: The lowest selectable date of the **`DatePicker`**.

Events

There is only one event exposed for the **`DatePicker`** class:

- **`DateSelected`**: Fired when the **`Date`** property has changed.

XAML

```
<DatePicker Date="09/12/2014" Format="d" />
```

Code Listing 50

Code

```
var datePicker = new DatePicker{
     Date = DateTime.Now,
     Format = "d"
};
```

Code Listing 51

As you can see in the XAML example, the **`Format`** property can be used to specify what format the date is in when assigning a value to the **`Date`** property.

`DatePicker` representations are definitely different on the individual platforms. Some platforms have two viewing stages while others have only one as shown in the following screenshots.

Figure 94: DatePicker on Android

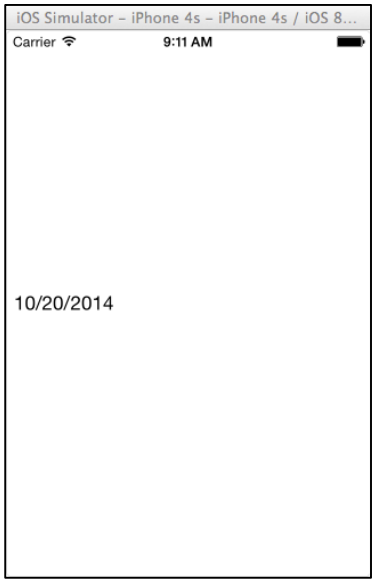


Figure 95: DatePicker on iOS

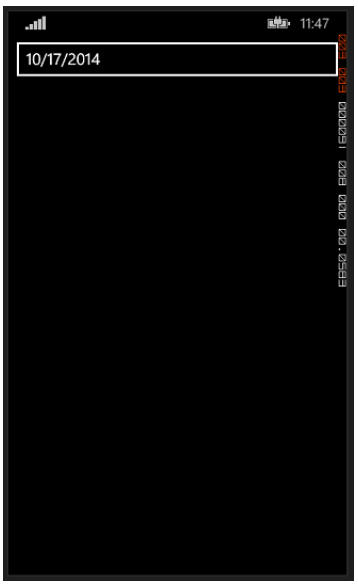


Figure 96: DatePicker on Windows Phone

After selecting the control you will see the following screens for choosing a date.

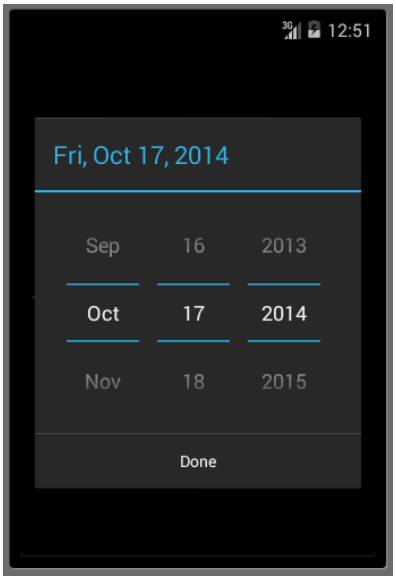


Figure 97: DatePicker selected on Android

Figure 98: DatePicker selected on iOS

Figure 99: DatePicker selected on Windows Phone

Entry

The **`Entry View`** is used to allow users to type a single line of text. This single line of text can be used for multiple purposes including entering basic notes, credentials, URLs, and more. This **`View`** is a multi-purpose **`View`**, meaning that if you need to type regular text or want to obscure a password, it is all done through this single control.

Properties

The following are properties available to help customize your **`Entry`**:

- **`IsPassword`**: Determines if the entered text should be obscured or not.
- **`Placeholder`**: The text that appears in the box before a user types something. This text disappears as soon as a letter is typed and reappears if the user clears the box.
- **`Text`**: The text that is in the **`Entry`**.
- **`TextColor`**: The color of the text within the **`Entry`**.
- **`Keyboard`**: The type of **`Keyboard`** used to enter text.

Events

There are two events raised by the **`Entry`** to which you can subscribe:

- **`Completed`**: Fired when the user finalizes the entry by tapping the Return key.
- **`TextChanged`**: Fired when the text within the **`Entry`** is changed.

XAML

```
<Entry Placeholder="Please Enter Some Text Here"
HorizontalOptions="Center"
       VerticalOptions="Center"
       Keyboard="Email"/>
```

Code Listing 52

Code

```
var entry = new Entry {
    Placeholder = "Please Enter Some Text Here",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center,
    Keyboard = Keyboard.Email
};
```

Code Listing 53

When the **`Entry`** control is first rendered, it is rather unassuming, only displaying the **`Placeholder`** text if provided.

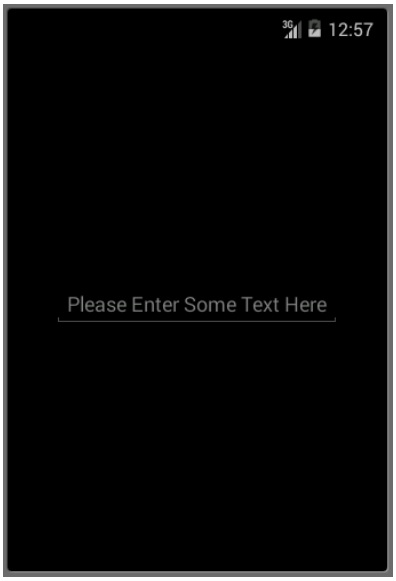


Figure 100: Entry on Android

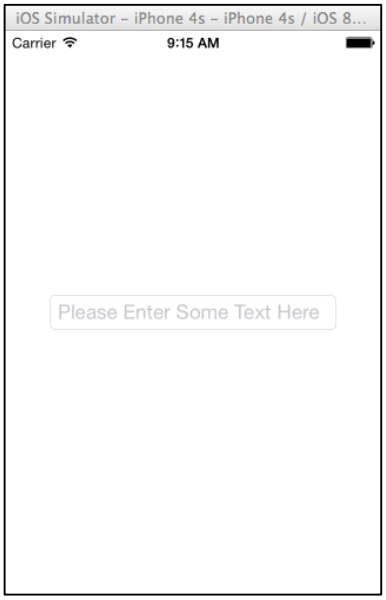


Figure 101: Entry on iOS

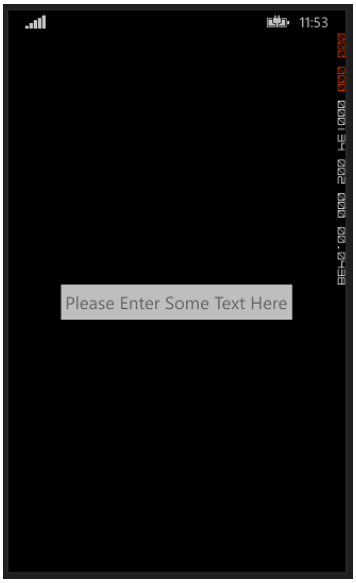


Figure 102: Entry on Windows Phone

Once the control is given focus, or clicked, the screen scrolls upward if necessary to allow room to present a keyboard to users.

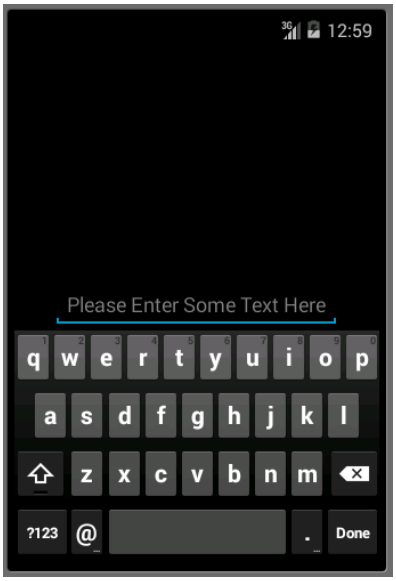


Figure 103: Entry selected on Android

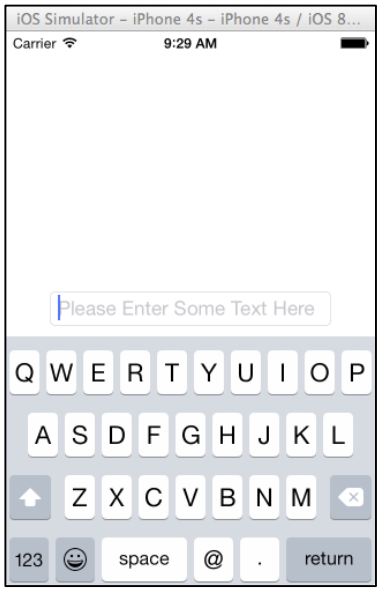


Figure 104: Entry selected on iOS

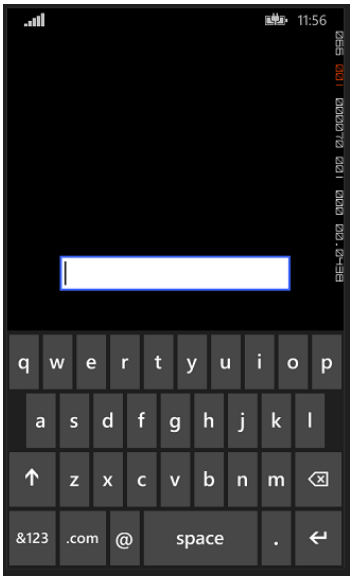


Figure 105: Entry selected on Windows Phone

Editor

The **`Editor`** is very similar to the **`Entry`** in that it allows users to enter some free-form text. The difference is that the **`Editor`** allows for multi-line input whereas the **`Entry`** is only used for single line input. The **`Entry`** also provides a few more properties than the **`Editor`** to allow further customization of the **`View`**.

Properties

There are two important properties associated with the **`Editor`**:

- **`Text`**: The text that is in the **`Editor`**.
- **`Keyboard`**: The specific type of **`Keyboard`** to be used while entering text.

Events

There are two events raised by the entry to which you can subscribe:

- **`Completed`**: Fired when the user finalizes the entry by tapping the Return key.
- **`TextChanged`**: Fired when the text within the **`Editor`** is changed.

XAML

```
<Editor HorizontalOptions="Fill"
        VerticalOptions="Fill"
        Keyboard="Chat"/>
```

Code Listing 54

Code

```
var editor = new Editor {
    HorizontalOptions = LayoutOptions.Fill,
    VerticalOptions = LayoutOptions.Fill,
    Keyboard = Keyboard.Chat
};
```

Code Listing 55

When the **`Editor`** is the only control in the Layout, it takes up the entire screen as shown in the following figures.

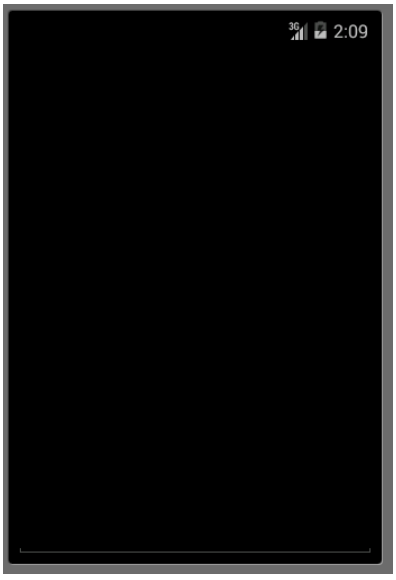


Figure 106: Editor on Android

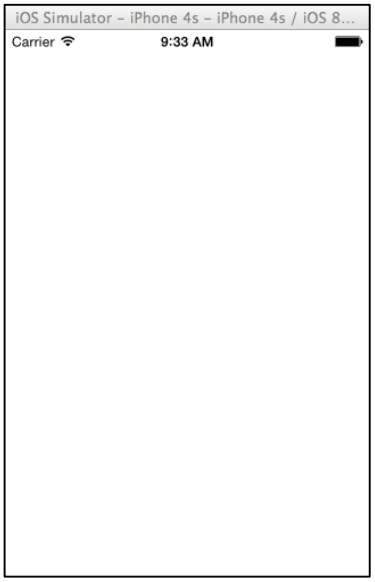


Figure 107: Editor on iOS

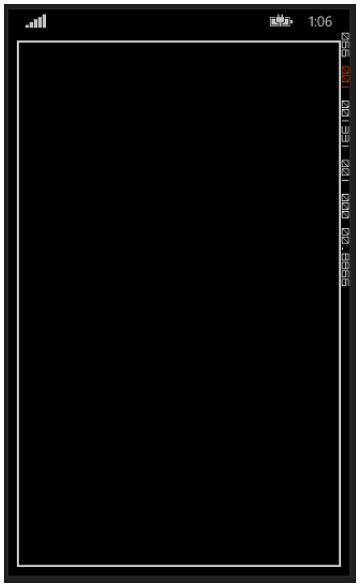


Figure 108: Editor on Windows Phone

When the control is selected, the keyboard appears for editing. Notice in this example, a different **`Keyboard`** of type **`Chat`** is used to customize the buttons that can be used for input.

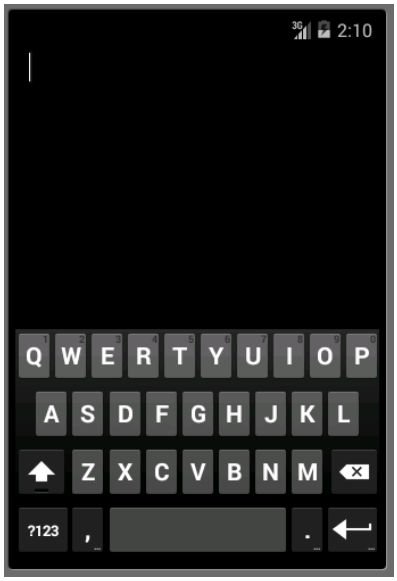


Figure 109: Editor selected on Android

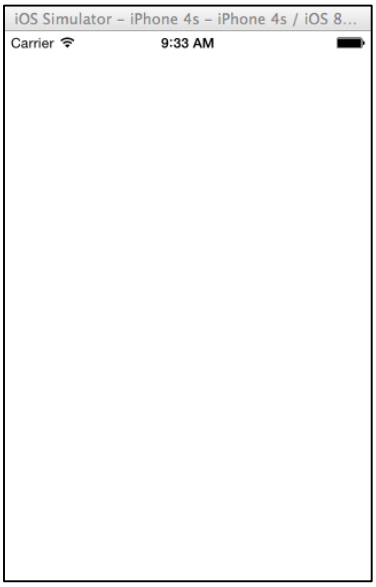


Figure 110: Editor selected on iOS

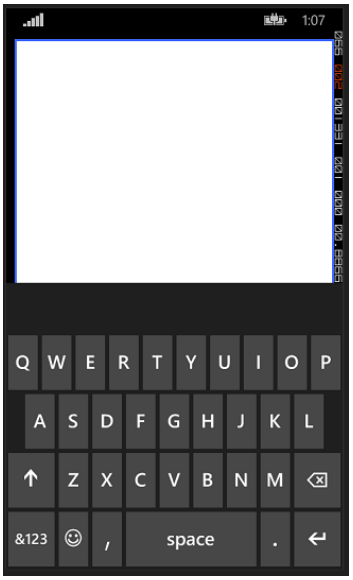


Figure 111: Editor selected on Windows Phone

Image

Images are very important parts of any application. They provide the opportunity to inject additional visual elements as well as branding into your application. Not to mention that images are typically more interesting to look at than text or buttons. You can use an **`Image`** as a standalone element within your application, but an **`Image`** element can also be added to other **`View`** elements such as a **`Button`**.

When it comes to retrieving images, you will typically be working with the **`Source`** property and using the **`ImageSource`** class to retrieve it. The **`ImageSource`** class has a few factory methods to make retrieving images easier depending on where they are coming from:

- **`FromFile()`**: Retrieves an image from a file path resolved to each platform.
- **`FromUri()`**: Retrieves an image from a **`Uri`** (can be an Internet path).
- **`FromResource`**: Retrieves an embedded image file by a resource identifier.

Properties

There a few properties associated with the **`Image`**:

- **`Aspect`**: The scaling mode for the **`Image`**.
- **`IsLoading`**: The loading status of the **`Image`**.
- **`IsOpaque`**: The opacity flag of the **`Image`**.
- **`Source`** : The source of the **`Image`**.

XAML

```
<Image Aspect="AspectFit" Source="http://d2g29cya9iq7ip.cloudfront.net/co
ntent/images/company/aboutus-video-bg.png?v=25072014072745"/>
```

Code Listing 56

Code

```
var image = new Image {
    Aspect = Aspect.AspectFit,
    Source = ImageSource.FromUri(new Uri("http://d2g29cya9iq7ip.cloudfron
t.net/content/images/company/aboutus-video-bg.png?v=25072014072745"))
};
```

Code Listing 57

This example uses the **`FromUri`** method to retrieve an image from the Internet. A network connection is needed for this to work. If you don't have a network connection, you can use one of the other methods to retrieve a local image.

Figure 112: Image on Android

Figure 113: Image on iOS

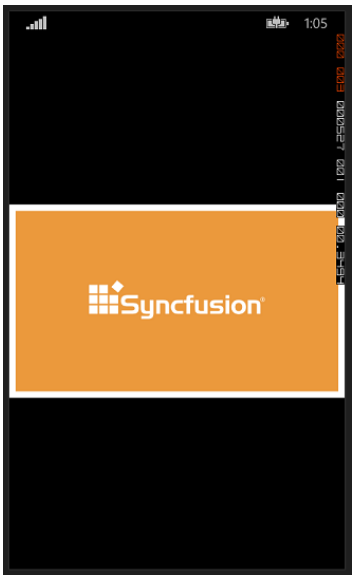


Figure 114: Image on Windows Phone

Label

Believe it or not, the **Label** is one of the most crucial yet underappreciated **View** classes not only in Xamarin.Forms, but in UI development in general. It is seen as a rather boring line of text, but without that line of text it would be very difficult to convey certain ideas to the user. **Label** controls can be used to describe what the user should enter into an **Editor** or **Entry** control. They can describe a section of the UI and give it context. They can be used to show the total in a calculator app. Yes, the **Label** is truly the most versatile control in your tool bag that may not always spark a lot of attention, but it is the first one noticed if it isn't there.

Properties

There a few properties associated with the **Label**:

- **Font**: The font of the text within the **Label**.
- **FormattedText**: The **FormattedString** of the text within the **Label**.
- **LineBreakMode**: The **LineBreakMode** of the text within the **Label**.
- **Text**: The text within the **Label**.
- **TextColor**: The **Color** of the text.
- **XAlign**: The horizontal alignment for the text.
- **YAlign**: The vertical alignment for the text.

XAML

```
<Label Text="This is some really awesome text in a Label!"
       TextColor="Red"
       XAlign="Center"
       YAlign="Center"/>
```

Code Listing 58

Code

```
var label = new Label {
    Text = "This is some really awesome text in a Label!",
    TextColor = Color.Red,
    XAlign = TextAlignment.Center,
    YAlign = TextAlignment.Center
};
```

Code Listing 59

You have seen a number of examples of the **Label** control throughout the book so far, but here is one more for good measure.

Figure 115: Label on Android

Figure 116: Label on iOS

Figure 117: Label on Windows Phone

Adding a Gesture

As mentioned earlier in this chapter, not every control comes with built-in support to handle certain events. For example, the **`Label`** control doesn't support the **`Clicked`** event in a similar way to the **`Button`** control. That doesn't mean that you can track when a user touches a **`Label`**, or any other control for that matter. We handle these types of interaction via a **`GestureRecognizer`**. Adding a **`GestureRecognizer`** is quite simple and looks something like the frollowing code example.

```
var tapGestureRecognizer = new TapGestureRecognizer();
tapGestureRecognizer.Tapped += (s,e) => {
    // Perform some operation
};

var label = new Label {
    Text = "This is some really awesome text in a Label!",
    TextColor = Color.Red,
    XAlign = TextAlignment.Center,
    YAlign = TextAlignment.Center
};

label.GestureRecognizers.Add( tapGestureRecognizer );
```

Code Listing 60

Simply by adding this little bit of code, you are now not only able to detect when a user touches your control, but you can also perform some operation when it happens.

Picker

The **`Picker`** is a control that allows users to select an element from a predetermined list of options. The control starts out looking similar to an **`Entry`**, but when users tap the box, a **`Picker`** control appears where the keyboard typically would. These types of controls are generally useful when needing the user's input, but not wanting them to have to type an option. Using a predetermined set of options speeds up the users' ability to select one and eliminates the issue of misspelling. Also, it's nice to have the list of options populated from another source, such as a file, database, or even a web service if needed.

Properties

For being a relatively sophisticated control, there are only three properties of note:

- **`Items`**: The list of options for the user.
- **`SelectedIndex`**: The currently selected options index in the list.
- **`Title`**: The title for the **`Picker`**.

XAML

```
<Picker Title="Favorite Color">
    <Picker.Items>
        <x:String>Red</x:String>
        <x:String>Green</x:String>
        <x:String>Blue</x:String>
    </Picker.Items>
</Picker>
```

Code Listing 61

Code

```
var picker = new Picker {
    Title = "Favorite Color",
};

picker.Items.Add("Red");
picker.Items.Add("Green");
picker.Items.Add("Blue");
```

Code Listing 62

Depending on the platform, the **`Picker`** can start out as a box similar to the following figures.

Figure 118: Picker on Android

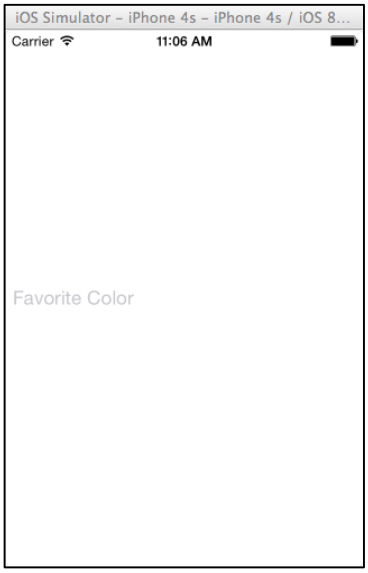


Figure 119: Picker on iOS

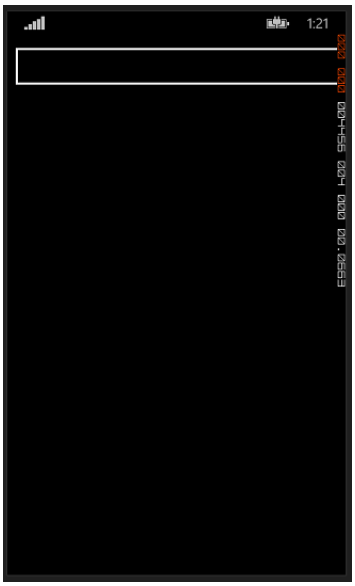


Figure 120: Picker on Windows Phone

But once the user touches it, the control will change to provide options for the user to select.

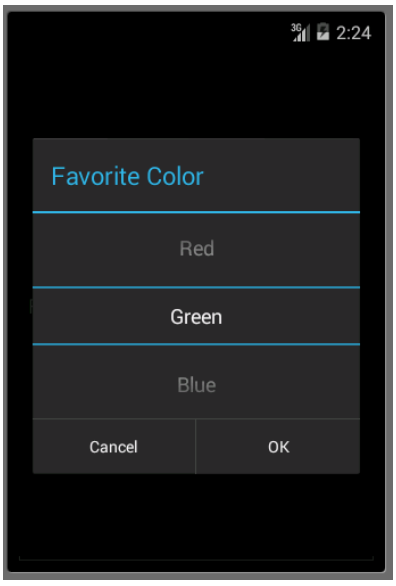


Figure 121: Picker selected on Android

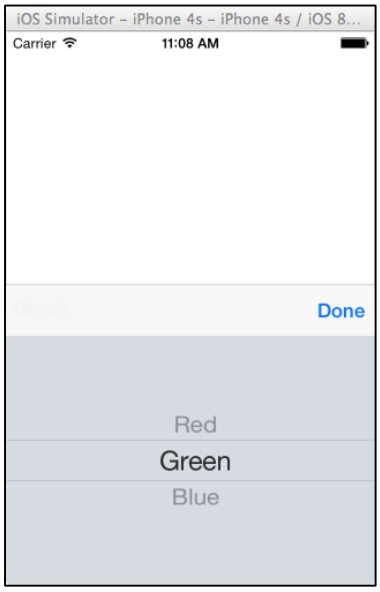


Figure 122: Picker selected on iOS

Figure 123: Picker selected on Windows Phone

Cells

The last two **View** classes I would like to discuss are what I like to call cell controls or cell views. The reason I group them into a category like this is that these two controls are made up of a collection of **Cell** objects. You can think of a **Cell** as an element that is used in a collection control that simulates a row within the parent control. The reason I say "simulates a row" is that technically they aren't referred to as rows even if they look and act like them. What you will quickly see is that these **Cell** classes are basically other controls combined into a single control. The two controls that contain the **Cell** objects are the **ListView** and the **TableView**. Before we get into these two controls, let's have a look at the **Cell** classes first.

All the samples in this section will be placed in a **TableView** for the sake of seeing them in action. The **TableView**, as well as the **ListView**, will be shown in detail after the explanation of the **Cell** classes.

EntryCell

An **EntryCell** is a **Cell** that combines the capabilities of a **Label** and an **Entry**. The **EntryCell** can be useful in scenarios when building some functionality within your application to gather data from the user. They can easily be placed into a **TableView** and be treated as a simple form.

Properties

The properties of the **EntryCell** are as follows:

- **Keyboard**: The type of **Keyboard** displayed when the user taps the **Entry**.
- **Label**: The fixed text next to the **Entry**.
- **LabelColor**: The color of the text within the **Label**.
- **Placeholder**: The text found in the **Entry** before the user enters anything.
- **Text**: The text contents of the **Entry**.
- **XAlign**: The horizontal alignment of the **Text** property.

Events

There is only a single event associated with the **EntryCell**:

- **Completed**: Fired when the user dismisses the keyboard.

XAML

```
<EntryCell Label="Type Something"
           Placeholder="Here"/>
```

Code Listing 63

Code

```
var entryCell = new EntryCell {
    Label = "Type Something",
    Placeholder = "Here"
};
```

Code Listing 64

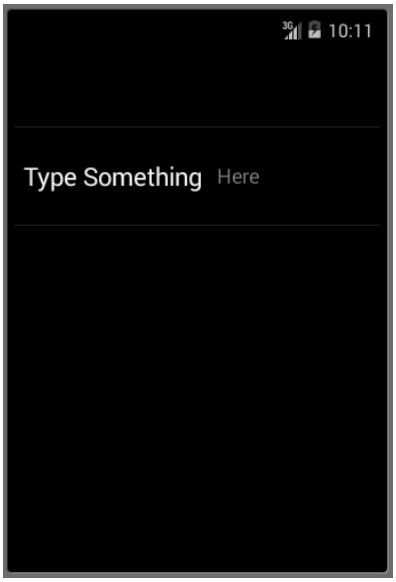


Figure 124: EntryCell on Android

Figure 125: EntryCell on iOS

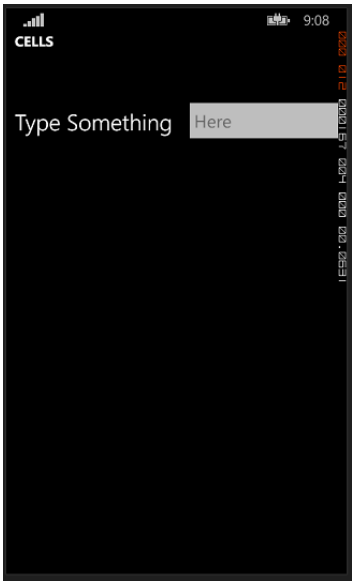


Figure 126: EntryCell on Windows Phone

SwitchCell

A **`SwitchCell`** is a **`Cell`** that combines the capabilities of a **`Label`** and an on-off switch. A **`SwitchCell`** can be useful for turning on and off functionality, or even user preferences or configuration options.

Properties

The properties of the **`SwitchCell`** are fairly straightforward:

- **`On`**: Determines whether or not the switch is in the on position.
- **`Text`**: The text displayed next to the switch.

Events

There is only a single event associated with the **`SwitchCell`**:

- **`OnChanged`**: Fired when the switch has changed value.

XAML

```
<SwitchCell Text="Switch It Up!" />
```

Code Listing 65

Code

```
var switchCell = new SwitchCell {
    Text = "Switch It Up!"
};
```

Code Listing 66

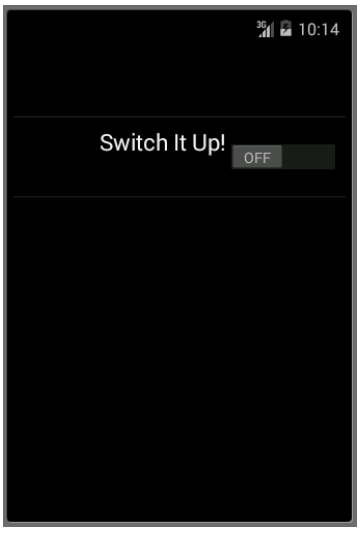


Figure 127: SwitchCell on Android

Figure 128: SwitchCell on iOS

Figure 129: SwitchCell on WindowsPhone

TextCell

A **`TextCell`** is a **`Cell`** that has two separate text areas for displaying data. A **`TextCell`** is typically used for information purposes in both **`TableView`** and **`ListView`** controls. The two text areas are aligned vertically to maximize the space within the **`Cell`**. This type of **`Cell`** is also commonly used to display hierarchical data, so when the user taps this cell, it will navigate to another page.

Properties

The commonly used properties of the **`TextCell`** are as follows:

- **`Text`**: The primary text within the **`Cell`**.
- **`TextColor`**: The color of the primary text.
- **`Detail`**: The secondary text in the **`Cell`**.
- **`DetailColor`**: The **`Color`** of the secondary text.

Events

There is only a single event associated with the **`TextCell`**:

- **`OnTapped`**: Fired when the user taps the **`Cell`**.

XAML

```
<TextCell Text="I am primary"
          TextColor="Red"
          Detail="I am secondary"
          DetailColor="Blue"/>
```

Code Listing 67

Code

```
var textCell = new TextCell {
    Text = "I am primary",
    TextColor = Color.Red,
    Detail = "I am secondary",
    DetailColor = Color.Blue
};
```

Code Listing 68

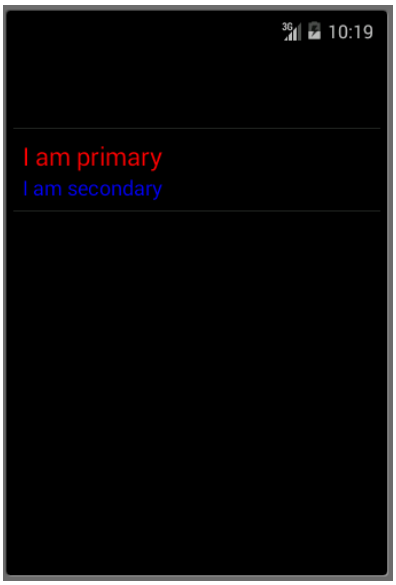


Figure 130: TextCell on Android

Figure 131: TextCell on iOS

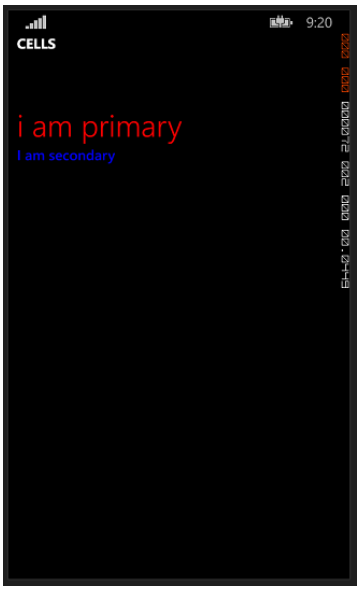


Figure 132: TextCell on Windows Phone

ImageCell

An **`ImageCell`** is exactly what it sounds like. It is a simple **`Cell`** that contains only an **`Image`**. This control functions very similarly to a normal **`Image`** control, but with far fewer bells and whistles.

Properties

There is only a single property associated with the **`ImageCell`**:

- **`ImageSource`**: The source of the **`Image`**.

XAML

```
<ImageCell ImageSource="http://d2g29cya9iq7ip.cloudfront.net/content/imag
es/company/aboutus-video-bg.png?v=25072014072745")),
           Text="This is some text"
           Detail="This is some detail" />
```

Code Listing 69

Code

```
var imageCell = new ImageCell {
    ImageSource = ImageSource.FromUri(new Uri("http://d2g29cya9iq7ip.clou
```

```
dfront.net/content/images/company/aboutus-video-
bg.png?v=25072014072745")),
    Text = "This is some text",
    Detail = "This is some detail"
};
```

Code Listing 70

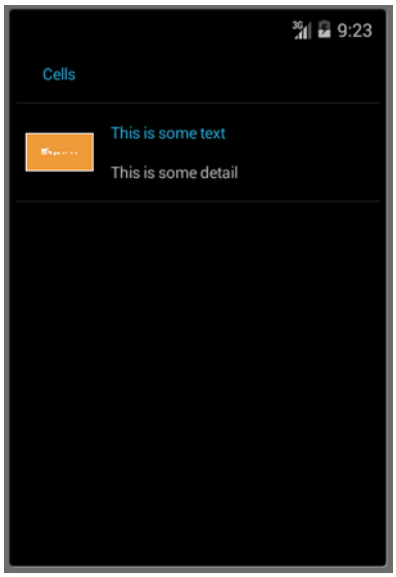


Figure 133: ImageCell on Android

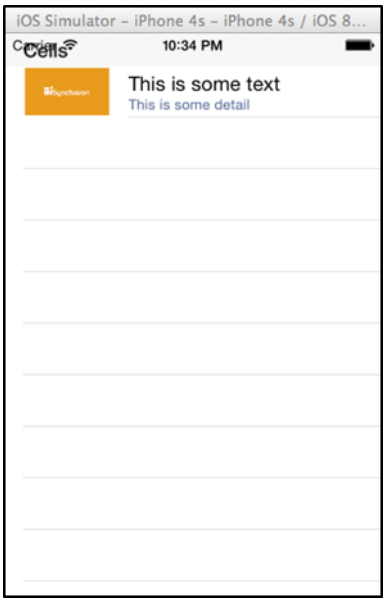


Figure 134: ImageCell on iOS

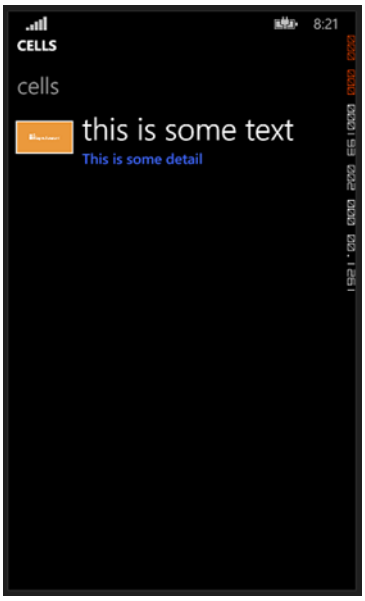


Figure 135: ImageCell on Windows Phone

ViewCell

You can consider a **`ViewCell`** a blank slate. It is your personal canvas to create a **`Cell`** that looks exactly the way you want it. You can even compose it of instances of multiple other **`View`** objects put together with **`Layout`** controls. You are only limited by your imagination. And maybe screen size.

Properties

There is only a single property associated with the **`ViewCell`**:

- **`View`**: The **`View`** representing the content of the **`Cell`**.

XAML

```
<ViewCell>
     <ViewCell.View>
          <StackLayout>
               <Button Text="My Button"/>
```

```
            <Label Text="My Label"/>
            <Entry Text="And some other stuff"/>
        </StackLayout>
    </ViewCell.View>
</ViewCell>
```

Code Listing 71

Code

```
var button = new Button { Text = "My Button" };
var label = new Label { Text = "My Label" };
var entry = new Entry { Text ="And some other stuff" };

var viewCell = new ViewCell {
    View = new StackLayout {
        Children = { button, label, entry }
    }
};
```

Code Listing 72

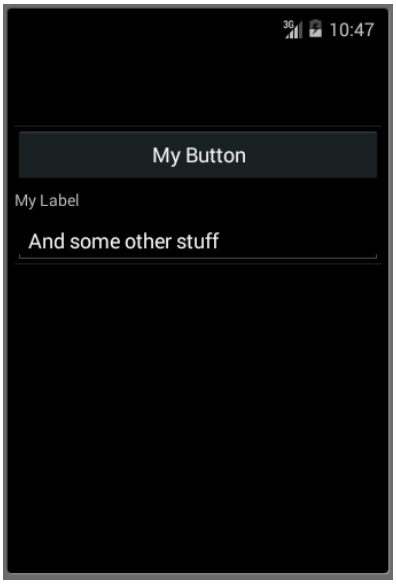


Figure 136: ViewCell on Android

Figure 137: ViewCell on iOS

Figure 138: ViewCell on Windows Phone

Cell Controls

Now that you understand the basic concept of the different types of **`Cell`** classes you have to work with, let's take a look at their reason for existing: **`Cell`** controls. There are two classes that fall into this category: **`ListView`** and **`TableView`**. While they both have to do with presenting a collection of information to the user in the form of **`Cell`** objects, they do this in very different ways. Let's take a look at these two different controls.

ListView

At its most basic, a **`ListView`** is a control that will display a collection of data in a vertical list. Sounds rather simple, but it has more capability than that. If the list has too much data than can fit on the screen, you can scroll the list. The **`ListView`** can be used to present a collection of options to the user, who can then select an option that can have an effect on the application. The **`ListView`** can have its items grouped into sections with headings to present a logical group of options to the user. This is the just the basic functionality of the **`ListView`** itself.

Before we go too far into describing how the **`ListView`** works more specifically, let's take a look at the properties so the concepts discussed afterwards make more sense.

Properties

There are some actual **`ListView`** properties that can be used to set up the grouping and a few generic properties, but the most interesting ones are actually a part of its base class, **`ItemsView<Cell>`**:

- **`ItemsSource`**: The source of items to display.
- **`ItemTemplate`**: The **`DataTemplate`** to apply to the **`ItemsSource`**.

If we were to follow the same pattern as the other examples of controls in this chapter, we could look at an example of a **`ListView`** as something like the following.

XAML

```
<ListView>
    <ListView.ItemsSource>
        <x:Array Type="{x:Type x:String}">
            <x:String>John</x:String>
            <x:String>Robert</x:String>
        </x:Array>
    </ListView.ItemsSource>
</ListView>
```

Code Listing 73

Code

```
var listView = new ListView{
    ItemsSource = new List<string>{
        "John",
        "Robert"
    }
};
```

Code Listing 74

While we could certainly create our **ListView** controls this way, it would be extremely inflexible and every time we wanted to change the list, we would have to manually modify its creation. Instead, we can use the trick we learned in Chapter 4 and apply some data binding. That way we will make our **ListView** much more flexible. Let's change it to something like the following example.

```
<ListView ItemsSource="{Binding Names}">
</ListView>
```

Code Listing 75

That certainly feels a little better, but now we will also need some class to be our **BindingContext** again. So let's go ahead and see how that is going to look.

```
public class SampleContext {
    public SampleContext( ) {
        Names = new ObservableCollection<string>(new List<string> {
            "John",
            "Robert"
        });
    }

    public ObservableCollection<string> Names { get; set; }
}
```

Code Listing 76

Up to this point, our examples have been simple bindings where we are setting a single XAML property to a single property within our context class. Now we need to be able to bind a XAML property that accepts a collection to a property in the context class that also contains a collection. In the world of binding we do this in the form of an **ObservableCollection**. An **ObservableCollection** is a generic collection that can be used in our **Binding** markup extensions to bind to a list and reflect changes in that list.

The next step, once again, is to assign this class to the **BindingContext** in our code-behind file and we are ready to go.

```
public SampleXaml()
{
    InitializeComponent();
    BindingContext = new SampleContext( );
}
```

Code Listing 77

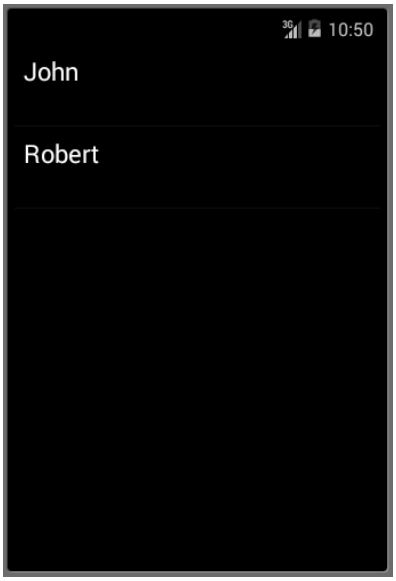


Figure 139: ListView on Android

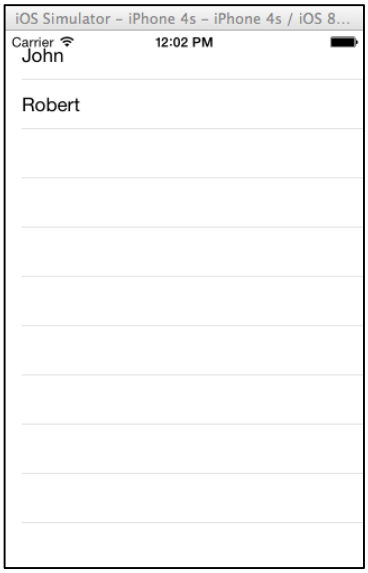


Figure 140: ListView on iOS

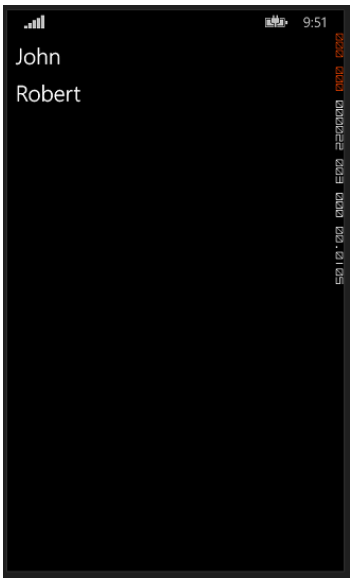


Figure 141: ListView on Windows Phone

Unfortunately, it's not always going to be that simple. This example works out just fine because we are dealing with simple objects: strings. What happens if we modify this **`ObservableCollection`** to return a collection of **`Person`** objects that look something like the following code.

```
public Person()
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Code Listing 78

We are now going to see something like the following figures.

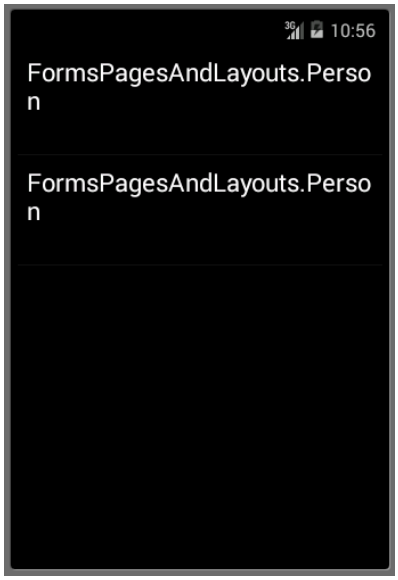


Figure 142: ListView with a complex type on Android

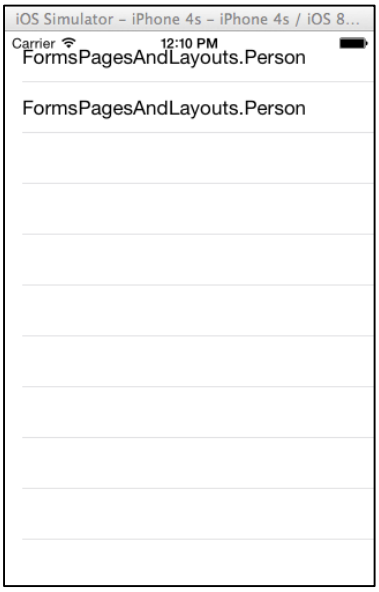


Figure 143: ListView with a complex type on iOS

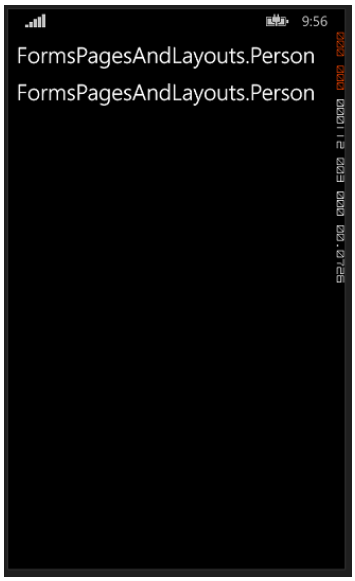


Figure 144: ListView with a complex type on Windows Phone

This is a big problem. It would seem that unless we are working with simple data types, we aren't going to get very far. Sure, we could explicitly override the **`ToString()`** method to return something useful, but that is only going to get us so far. That is where the **`ItemTemplate`** property comes into play.

The **`ItemTemplate`** property is where we get a chance to take a complex object, such as **`Person`**, and turn it into a reusable template using **`Cell`** classes that the **`ListView`** knows how to render in the **`View`**. Let's take a look at a sample of this in action.

```
<ListView ItemsSource="{Binding Names}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextCell Text="{Binding FirstName}" Detail="{Binding LastNam
e}"/>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Code Listing 79

This XAML may look a little complicated at first, but it truly isn't that bad. First, we are going to set the **`ItemTemplate`** property to a new instance of a **`DataTemplate`**. In most cases, a **`DataTemplate`** is simply a mechanism where we can choose which type of **`Cell`** we would like to use to present our data. In this example, we are going to choose the **`TextCell`** which has two **`Label`** objects in it. We set the first **`Label`**, **`Text`**, to be the **`Person`** object's **`FirstName`** property. Next we set the second **`Label`**, **`Detail`**, to be the **`Person`** object's **`LastName`** property. When we run the application, we get the following.

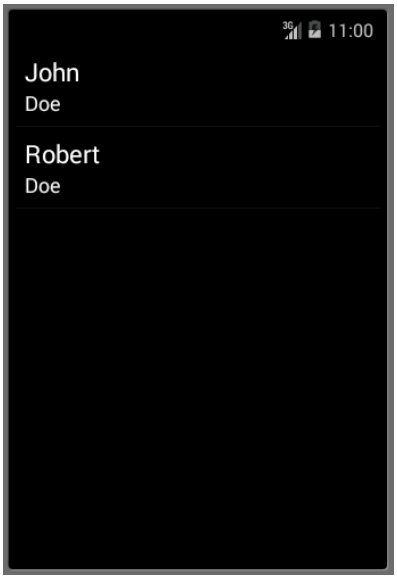


Figure 145: ListView using DataTemplate on Android

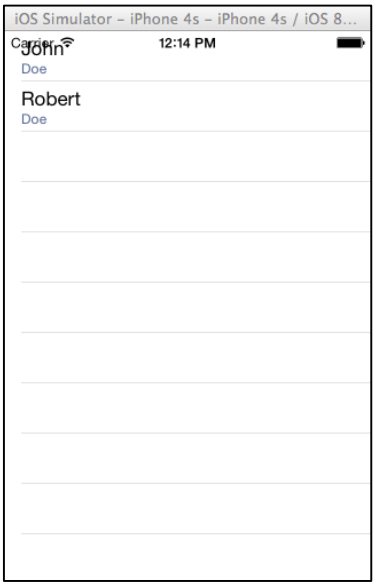


Figure 146: ListView using DataTemplate on iOS

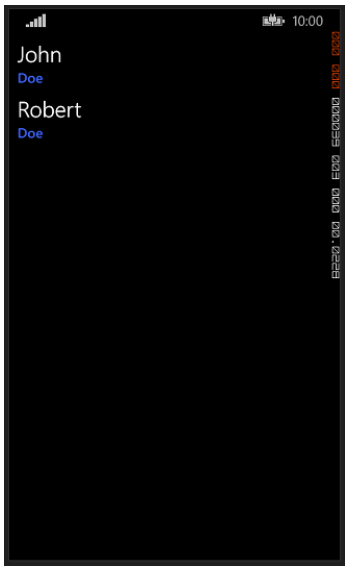


Figure 147: ListView using DataTemplate on Windows Phone

The same code in C# would look something like the following code example.

```
public Sample()
{
    InitializeComponent();
    var people = new List<Person> {
        new Person {FirstName = "John", LastName = "Doe"},
        new Person {FirstName = "Robert", LastName = "Doe"}
    };

    var listView = new ListView {
        ItemsSource = people
    };

    var cell = new DataTemplate( typeof ( TextCell ) );
    cell.SetBinding(TextCell.TextProperty, "FirstName");
    cell.SetBinding(TextCell.DetailProperty, "LastName");

    listView.ItemTemplate = cell;

    Content = listView;
}
```

Code Listing 80

As you can see, by using the **ItemsSource** and **ItemTemplate** properties of the **ListView** class, along with some of the handy **Cell** classes, you can create nice lists of data to present to the user.

TableView

The final control that we will discuss in this chapter is the **TableView**. The **TableView** is similar to the **ListView** in that we will ultimately fill it with **Cell** controls, but the process and makeup of the two are a little different.

A **TableView** within Xamarin.Forms is a container the holds rows of **Cell** objects very similarly to the **ListView**. One of the primary differences, though, is that a **TableView** can be explicitly created for a specific purpose. A **ListView** is typically a list of data out of the box. The way in which you can use the **TableView** for a specific purpose is through the **Intent** property which takes a value from the **TableIntent** enumeration. Here are the currently defined options for this enumeration:

- **Data**: Contains an arbitrary number of data entries.
- **Form**: Used to contain information used in a form.
- **Menu**: Intended to be used as a menu for selections.
- **Settings**: Contains a set of toggle-style controls for configuration settings.

Note: Specifying an Intent for a TableView doesn't affect its functionality. It will only modify its visual appearance depending on the platform.

Properties

When getting started with the **TableView**, you will primarily be concerned with the following two properties:

- **Intent**: A hint as to the purpose of the **TableView**.
- **Root**: The visual **DataTemplate** to apply a layout to the data within the **ItemsSource**.

If we were to follow the same pattern as the other example of controls in this chapter, we could look at an example of a **TableView** as something like the following code.

XAML

```
<TableView Intent="Data">
    <TableView.Root>
        <TableSection Title="Table">
            <TextCell Text="Just a text cell"/>
            <EntryCell Label="Then an entry cell"/>
        </TableSection>
    </TableView.Root>
</TableView>
```

Code Listing 81

Code

```
var textCell = new TextCell {Text = "Just a text cell"};
var entryCell = new EntryCell {Label = "Then an entry cell"};

var tableView = new TableView {
    Intent = TableIntent.Data,
    Root = new TableRoot() {
        new TableSection("Table") {
            textCell, entryCell
        }
    }
};
```

Code Listing 82

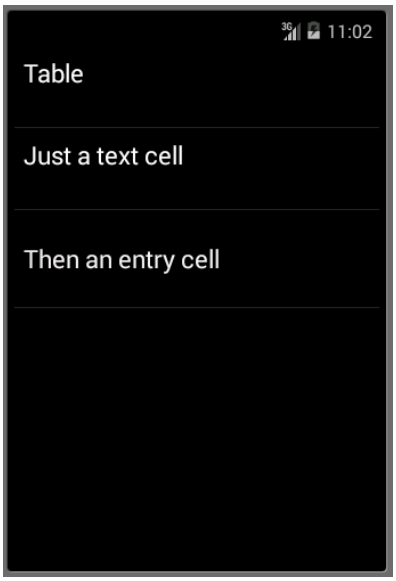


Figure 148: TableView on Android

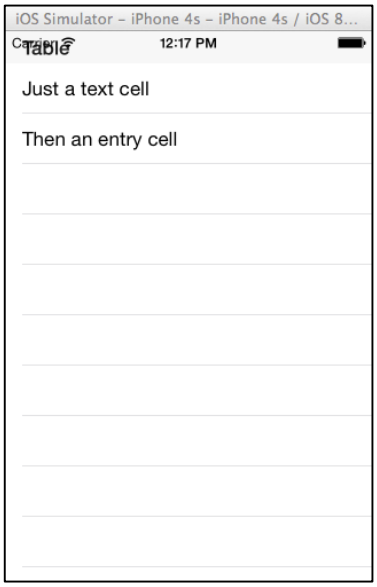


Figure 149: TableView on iOS

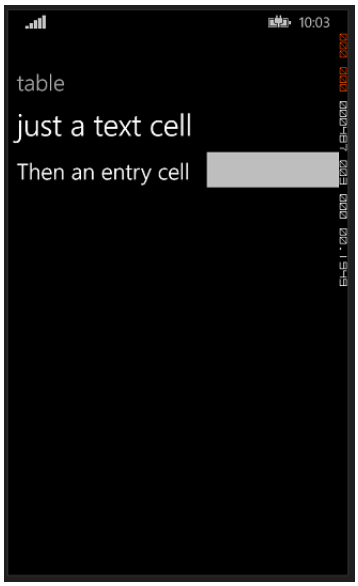


Figure 150: TableView on Windows Phone

Summary

Throughout this chapter, we have covered many of the common controls you will need to know how to use to begin creating Xamarin.Forms applications. These controls begin to create the foundation of many types of mobile apps and begin to show the power of Xamarin.Forms. While these basic controls will get you started in your development using Xamarin.Forms, you will ultimately progress farther than the built-in controls will get you. Once you reach that point, you will need to know not only how these controls function and render on the platforms, but also how to customize them to suit your application's needs. That will be covered in the next chapter.

Chapter 7 Customization

In the previous chapter, we spent quite a bit of time discussing the different controls, or **`View`** classes, that are built into Xamarin.Forms. This is a very good starting point for most basic apps. The problem, as you will soon find, comes into play when you need to customize the native look of a Xamarin.Forms control once it is rendered on the platform-specific project. Lucky for us, Xamarin didn't leave us high and dry. They built an entire mechanism for us to completely customize this process.

In this chapter we will be discussing the rendering process of Xamarin.Forms controls on the native platforms. After that, we will pick apart this process to understand where we can introduce our own customizations. Finally, we will walk through a simple example to see exactly how we can use this process to tweak an existing control to our liking.

Xamarin.Forms Rendering Process

Before you can start churning out your own custom controls or modifying any existing ones, you need to understand how the rendering process of Xamarin.Forms works. As mentioned several times earlier in this book, Xamarin.Forms is an abstraction layer used to create platform-agnostic definitions of mobile UI elements. You go through the process of using these elements to create a logical representation of what you want your application to look like on each of the platforms. This is all done through a single UI definition in shared code. Once this platform-agnostic view of your UI is complete and combined with the platform-specific code, you get multiple platform applications with a remarkably similar look, but they still retain all of their native look and feel. This process is accomplished through two very important components that span both the shared code and platform-specific code projects. These two components are the **Element** and the **Renderer**.

Element

The concept of an element in Xamarin.Forms should not come as a big surprise to you at this point. We have spent the better part of the last chapter discussing them under the guise of a couple different terms: control and view. Within the Xamarin.Forms infrastructure, these controls and views are platform-agnostic representations of visual elements on the screen to which you can apply a series of properties and functionality. These controls and views are characterized by the fact that they are subclasses of the **`View`** class. Without anything else, these elements would just sit and do nothing. They would simply be classes sitting in your shared project, never seeing the light of day. So how do we get these platform-agnostic elements to transform into native controls? The answer is renderers.

Renderer

In the most simplistic of terms, a renderer is the binding, or glue, between the platform-agnostic world of Xamarin.Forms and the native controls. For each element that is defined in the Xamarin.Forms framework, a specific renderer is defined. For example, the **`Label`** control within Xamarin.Forms maps to a **`LabelRenderer`** class. To be more specific, it maps to three **`LabelRenderer`** classes. Confused? Let's break this down a little more.

Let's say, for example, we have a Xamarin.Forms application that is composed of four separate projects:

- **Sample**: A PCL project containing the definition of the UI.
- **Sample.iOS**: A Xamarin.iOS project.
- **Sample.Android**: A Xamarin.Android project.
- **Sample.WinPhone**: A Windows Phone project.

First, if you expand the **References** folder in the **Sample** project, you will notice that there is a reference to an assembly named **Xamarin.Forms.Core**. This particular assembly is what contains the core definitions and functionality of all the elements that we have been discussing in the last couple chapters.

Next, if you investigate the **References** folders in the three platform-specific projects, you will see a reference to a very similar assembly named **Xamarin.Forms.Platform.<name>**, where **<name>** is the name of the platform: **iOS**, **Android**, and **WP8**. These are the platform-specific implementations of the Xamarin.Forms assemblies that allow for this native control mapping to happen.

Finally, if you look through the **Object Browser** at each of these platform-specific implementations of Xamarin.Forms, you will find that for every element defined within **Xamarin.Forms.Core**, there will be a similarly named renderer class in the platform-specific Xamarin.Forms assembly. This is for a very good reason. At some point, the platform-agnostic versions of the elements in Xamarin.Forms need to be translated and rendered into some native controls that the platforms know about. Keeping with the same example of the **`Label`** element, each renderer in the specific platforms will translate that into something else:

- Sample.iOS: **`UILabel`**
- Sample.Android: **`TextView`**
- Sample.WinPhone: **`TextBlock`**

Once this translation has been made, the platform-specific projects already know how to interact and handle the native controls, so there is nothing else we need to worry about.

Custom Control Scenarios

When it comes to straying from the norm in anything, the first thing that you should do is ask yourself, why? Why do you want to do this thing that isn't the norm or supplied out of the box? In this respect, we need to ask ourselves, why we would ever want to create custom controls in Xamarin.Forms?

When it comes to any sort of customization in Xamarin.Forms, the reasons typically fall into two basic categories:

- The need for a completely custom control.
- Adding additional capabilities not present in an existing control.

When it comes to creating a completely custom control, you will need to start from square one. You will begin by creating an element within your shared code project. This element will be represented by a class that gives a platform-agnostic view of the properties and functionality that this particular control needs. Once you have completed describing the control in a platform-agnostic way, you will need to create a custom renderer for this element for each of your target platforms. This renderer will translate the pieces of your element into platform-specific controls that can be rendered. This option can prove to be a decent amount of work depending on the complexities of the control, but ultimately it is very doable.

The other option is to simply enhance an existing control with some additional functionality that it doesn't currently have. An example would be to add support for bold or italic font to a **`Label`** control in Xamarin.Forms. Although this sounds much simpler than creating a control from scratch, there are still a couple of options to consider.

When customizing an existing control within Xamarin.Forms, you need to ask yourself a very basic question. Do you want to customize every instance of a particular control within your application, or only a select few? The answer to this question will more than likely push you in one direction or the other. If you are looking to customize all the instances of a control within your application, then you will typically want to create only a custom renderer for a particular element and replace the existing renderer. That way, whenever you use that element in your application, the custom renderer will be used and the customized platform-specific control will be used. This can be a useful choice for a consistent look and feel everywhere in your application.

If you are looking to only customize a few instances of a control here or there, you will want to take a slightly different approach. In this case, you will need to create a custom element that inherits from the element you are customizing. That way, you are maintaining all the characteristics of the element and are able to add your own spin on it. The next step is to create a renderer for that element for each platform to create a platform-specific control that functions the way you wish.

Let's walk through an example of adding some customization to a control within Xamarin.Forms. The process for creating a custom control and customizing an existing control is very similar, so this example can be tailored for either scenario.

Creating a Custom Control

In this example, we will be adding some functionality to the **`Label`** control found in Xamarin.Forms. One of the shortcomings of this control when using it out of the box is that there is no support to make the text within the label either bold or italic. This can be a very nice feature and since Xamarin.Forms has support for customization, that is exactly what we are going to do.

Start by creating a new Xamarin.Forms project, either a PCL or Shared Project, and give it a name. Once you have created this project, we need to walk through the two main steps to add customization to a control. The first step will be to create a new element within our shared code. The second step will be to create custom renderers for each of the platforms we wish to support.

The Element

The first step toward customization is to create a class that will represent our element. Since we want our custom **`Label`** to support all the same functionality the existing **`Label`** control contains, we will be inheriting from that control.

```
public enum StyleType {
    None,
    Bold,
    Italic,
    BoldItalic
}

public class StyledLabel : Label
{
    public StyleType Style {get; set;}
}
```

Code Listing 83

As you can see in the sample implementation in the previous code example, we have defined an **`enum`** that will represent the styling that will be applied to a **`Label`**. We have also added a **`StyleType`** property to our custom **`StyledLabel`** class.

Note: Feel free to make the Style property bindable if you wish to use data-binding in XAML as seen in Chapter 4.

Now that we have completed a platform-agnostic representation of our control, it's time to provide each of the platforms the ability to render this control.

Android Renderer

In this case, since we are dealing with a completely custom control, we need to actually build that control in order for the platforms to be able to render them. Let's start with Android.

Within your **Xamarin.Android** project, add a new class to the project, name it **`StyledLabelRenderer`**, and replace the default implementation with the following code.

```
using Android.Graphics;
using Customization.Droid;
```

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(StyledLabel), typeof(StyledLabelRenderer
))]

namespace Customization.Droid
{
    public class StyledLabelRenderer : LabelRenderer {
        protected override void OnElementChanged( ElementChangedEventArgs
<Label> e ) {
            base.OnElementChanged( e );

            var styledLabel = (StyledLabel)Element;

            switch (styledLabel.Style)
            {
                case StyleType.Bold:
                    Control.SetTypeface(null, TypefaceStyle.Bold);
                    break;
                case StyleType.Italic:
                    Control.SetTypeface(null, TypefaceStyle.Italic);
                    break;
                case StyleType.BoldItalic:
                    Control.SetTypeface(null, TypefaceStyle.BoldItalic);
                    break;
            }
        }
    }
}
```

Code Listing 84

This bit of code may seem a little confusing at first, but once you understand the basic concepts, you will see that the shell of the renderer you are creating will be consistent across all the platforms and the only thing you will be changing is the platform-specific constructs.

Let's start with the **`assembly`** attribute above the namespace declaration.

```
[assembly: ExportRenderer(typeof(StyledLabel), typeof(StyledLabelRenderer
))]
```

Code Listing 85

This attribute is used to tie our **StyledLabel** element to the **StyledLabelRenderer**. Now, when the Xamarin.Forms application runs and it sees that we need to render a **StyledLabel**, it will look for any registered renders that can handle this particular type. Without this line, if we were to run our application, Xamarin.Forms would see there is no renderer for **StyledLabel** and would use the next best thing, **LabelRenderer**. This is because it knows that **StyledLabel** inherits from **Label**, so it will assume it can use the renderer for the **Label** for the **StyledLabel**.

Next, you will see that the **StyledLabelRenderer** class inherits from **LabelRenderer**.

```
public class StyledLabelRenderer : LabelRenderer
```

Code Listing 86

In a similar way to how we inherited our custom element **StyledLabel** from the base class **Label**, we will do the same for the renderer. This is so we rely on all the same base rendering functionality. All we need to worry about now is how to handle the custom functionality.

The way that we are going to inject our own functionality is by strategically overriding the base class' **OnElementChanged** method. The reason for this is that this is the first opportunity to add some customization after all the initialization of the renderer is done. Once in this method, we have free reign to do what we wish to our custom control.

The final two pieces of our implementation are an assignment and a **switch** statement.

```
var styledLabel = (StyledLabel)Element;

switch (styledLabel.Style)
{
    case StyleType.Bold:
        Control.SetTypeface(null, TypefaceStyle.Bold);
        break;
    case StyleType.Italic:
        Control.SetTypeface(null, TypefaceStyle.Italic);
        break;
    case StyleType.BoldItalic:
        Control.SetTypeface(null, TypefaceStyle.BoldItalic);
        break;
}
```

Code Listing 87

Before you can truly understand the previous code, you need to understand a fundamental concept that is present within all the renderer implementations. All renderers within Xamarin.Forms, at some point in their hierarchy, inherit from the **ViewRenderer<TView, TNativeView>** class. As you can see, this is a generic class with two arguments:

- **TView**: The Xamarin.Forms **View** class.
- **TNativeView**: The corresponding platform-specific UI component.

In our case, since we are working with a **LabelRenderer** as a base class, our **ViewRenderer** looks like the following example.

```
public class LabelRenderer : ViewRenderer<Label, TextView>
```

Code Listing 88

This means that our Xamarin.Forms class is a **Label** and this will map to a **TextView** on the Android platform. This concept is important because further down the inheritance tree, we get access to both of these objects within our renderer code. We get access to them via two properties:

- **Element**: The **TView** object (**Label** in our case).
- **Control**: The **TNativeView** (**TextView** in our case).

Now within our renderer, we can use these two properties to get the data we need from our custom control (**Element**) and apply the corresponding styling to the native control (**Control**).

If you look again at Code Listing 87, you will better understand what is happening. First we cast the **Element** property to a **StyledLabel** so we have access to the **Style** property. Finally, we use a simple **switch** statement to determine which **StyleType** to apply, and then use the Android method **SetTypeFace** to set the text to **Bold**, **Italic**, or **BoldItalic**.

iOS Renderer

We will now follow almost the exact same process in our iOS project to achieve a very similar result. Start by adding a new class to the **Xamarin.iOS** project and once again naming it **StyledLabelRenderer**. Next, replace the default implementation with the following code.

```
using Customization.iOS;
using MonoTouch.UIKit;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer(typeof(StyledLabel), typeof(StyledLabelRenderer
))]

namespace Customization.iOS
{
    public class StyledLabelRenderer : LabelRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<
Label> e)
        {
            base.OnElementChanged(e);

            var styledLabel = (StyledLabel)Element;
```

```
            switch (styledLabel.Style)
            {
                case StyleType.Bold:
                    Control.Font = UIFont.BoldSystemFontOfSize( 16.0f );
                    break;
                case StyleType.Italic:
                    Control.Font = UIFont.ItalicSystemFontOfSize( 16.0f )
;
                    break;
                case StyleType.BoldItalic:
                    Control.Font = UIFont.FromName("Helvetica-
BoldOblique", 16.0f);
                    break;
            }
        }
    }
}
```

Code Listing 89

You will find that this implementation is almost exactly the same as the Android implementation but with a couple small differences.

The first difference is that the **`LabelRenderer`** class in the **Xamarin.Forms.Platform.iOS** assembly is defined as follows.

```
public class LabelRenderer : ViewRenderer<Label, UILabel>
```

Code Listing 90

This means that our **`Element`** property will still be a **`Label`**, but now the **`Control`** property will be a **`UILabel`**. So now we will need to use the iOS capabilities to apply the styling to the native control. We do this through a few static helper methods on the **`UIFont`** class.

To set the **`UILabel`** text to bold, we set the **`Font`** property on the **`Control`** to the result of the **`UIFont.BoldFromSystemFontOfSize`** method. To set it to italic, we use a similar method named **`UIItalicFromSystemFontOfSize`**. The two are fairly straightforward, but combining the two can be a little trickier.

If you were to attempt to apply the results of these two methods to the **`Font`** property of the **`Control`** object sequentially, you will be disappointed. The second assignment will overwrite the first, so you will only end up with whichever style you used second. To get around this you can search around and find an iOS system font that looks to be both bold and italic and use it. To use a built-in font, you simply use the **`UIFont.FromName`** method and specify the name of the font as a string.

You may also notice that I have hardcoded the size of the font to be 16.0f. This is done purposely to allow an opportunity for you to further enhance this renderer. See if you can find a way to get a size for the native controls into each of the platform-specific renderers from the **`StyledLabel`** control.

Tip: You may want to add a new property to your StyledLabel class or use the bulit-in Font.FontSize property of the base class Label.

Windows Phone Renderer

Finally, we come to Windows Phone. This will probably come as no surprise at this point, but this is going to be extremely similar to the previous two examples.

Begin by creating a new class in the WinPhone project and name it **`StyledLabelRenderer`**. Next, replace the default implementation with the following code.

```
using Customization.WinPhone;
using Xamarin.Forms;
using Xamarin.Forms.Platform.WinPhone;

[assembly: ExportRenderer(typeof(StyledLabel), typeof(StyledLabelRenderer
))]

namespace Customization.WinPhone
{
    public class StyledLabelRenderer : LabelRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<
Label> e)
        {
            base.OnElementChanged(e);

            var styledLabel = (StyledLabel)Element;

            switch (styledLabel.Style)
            {
                case StyleType.Bold:
                    Control.FontWeight = FontWeights.Bold;
                    break;
                case StyleType.Italic:
                    Control.FontStyle = FontStyles.Italic;
                    break;
                case StyleType.BoldItalic:
                    Control.FontStyle = FontStyles.Italic;
```

```
                    Control.FontWeight = FontWeights.Bold;
                    break;
                }
            }
        }
    }
```

Code Listing 91

Once again the shell is exactly the same, but let's take a look at the **Xamarin.Forms.Platform.WP8** definition of a **`LabelRenderer`**.

```
public class LabelRenderer : ViewRenderer<Label, TextBlock>
```

Code Listing 92

Now we see that the **`Control`** property within our **`StyledLabelRenderer`** will be an instance of the **`TextBlock`** class. If we do a little research into the world of Windows Phone and the **`TextBlock`** class, we will see that it has two separate properties that we can use to apply our styles: **FontWeight** and **FontStyle**. To make the text of the **`TextBlock`** bold we set the **`FontWeight`** to **`FontWeights.Bold`**, and to make the text italic we set **`FontStyle`** to **`FontStyles.Italic`**. To apply both of them together, we simply set both properties at the same time.

Fruits of Our Labor

Now we want to see our work in action. That's the easy part. Since we are using a built-in **`View`** class as our base (**`Label`**), we can use a **`StyledLabel`** class anywhere we would use **`Label`**.

Go into the PCL project within your solution and open the **App.cs** file. In there you should see the **`App class constructor`**. Replace the contents of that method with the following code.

```
public  App()
{
    var iLabel = new StyledLabel {
        TextColor = Color.Black,
        Text = "Make me italic!",
        HorizontalOptions = LayoutOptions.CenterAndExpand,
        Style = StyleType.Italic
    };
    var bLabel = new StyledLabel
    {
        Text = "Make me bold!",
        TextColor = Color.Black,
        HorizontalOptions = LayoutOptions.CenterAndExpand,
        Style = StyleType.Bold
    };
    var bothLabel = new StyledLabel
```

```
    {
        Text = "Make me italic and bold!",
        TextColor = Color.Black,
        HorizontalOptions = LayoutOptions.CenterAndExpand,
        Style = StyleType.BoldItalic
    };
    MainPage = new ContentPage
    {
        BackgroundColor = Color.White,
        Content = new StackLayout
        {
            Padding = 100,
            Spacing = 100,
            Children = { iLabel, bLabel, bothLabel }
        }
    };
}
```

Code Listing 93

Here we are simply creating three instances of our new **`StyledLabel`** and setting the **`Style`** property on each to the desired **`StyleType`**. Then we will return a new instance of a **`ContentPage`** with the **`Content`** property set to a new instance of a **`StackLayout`** with its **`Children`** set to our three **`StyledLabel`** instances.

When you run your application, it should look something like the following figures.

Figure 151: Customized StyledLabel control on Android

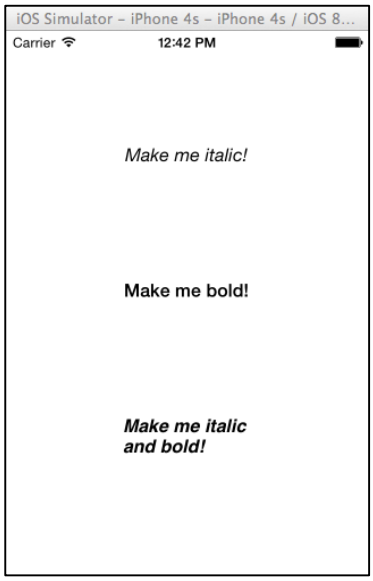


Figure 152: Customized StyledLabel control on iOS

Figure 153: Customized StyledLabel control on Windows Phone

Summary

In this chapter, we have discussed the basic principles behind the Xamarin.Forms rendering process. We have covered the important pieces of the puzzle, namely the element and the renderer. From there we discussed a couple situations where you may want to add some customization to your Xamarin.Forms applications. Finally, we worked through an example where we customized the **`Label`** element and the **`LabelRenderer`** to allow users to apply a bold, italic, and a combined bold and italic effect to the text within the platform-specific renderings of the **`Label`**. Armed with this knowledge, you should now be comfortable with customizing the natively rendered UI components that are generated when you are writing Xamarin.Forms applications.

Chapter 8 Accessing Native APIs

We have spent the last several chapters learning about the wonderful world of Xamarin.Forms and all the nice abstractions that Xamarin has created for UI components. This is all well and good, but what happens when you need to access some functionality that exists on the native platform, but there is no abstraction for it? Are you left out in the cold? Absolutely not. There's an answer for that too.

In this chapter we will be covering the scenario where we, as developers, need to access some piece of functionality on the native platform that isn't exposed by Xamarin.Forms. In most instances, this will have to do with some hardware functionality of the device, such as Bluetooth, network connectivity, accelerometers, and the like, but this could also mean accessing native software capabilities such as text-to-speech. Either way it would be nice to have a consistent way to access all of these pieces of functionality from Xamarin.Forms. And that is exactly what we have been given.

Differentiation

As most mobile developers know, creating a compelling UI is only part of the battle. In the world of Xamarin.Forms, it is great to be able to create multiple apps with a similar look and feel by using a unified abstraction layer. You can even go as far as adding customizations to that look and feel as we learned in the last chapter. But a problem will arise when a customer is using your app and it doesn't integrate into the capabilities of the native platform.

Take for instance a navigation app that ties into an online map provider, but can't show the user's current location. That would not provide a very good user experience. How about another app that targets a large population of visually impaired people that doesn't tap into the text-to-speech capabilities of the device? Once again, this will leave a large number of end users dissatisfied. Surprisingly enough, there are a number of apps in the different marketplaces that don't take advantage of these features.

One of the biggest struggles for people when it comes to selling apps in marketplaces is being able to differentiate their apps from the tens, hundreds, or thousands of other apps that are similar. One piece of advice that I can give is to tap into as much of the native platform capabilities as you can when it makes sense. Just because you have chosen Xamarin.Forms to create your cross-platform application, doesn't mean you have to give up on these features.

Accessing Native Features

When it comes to accessing native functionality, Xamarin has done a wonderful job creating a very simple API to do so in a consistent fashion. The ability to access these types of features is made available through a construct known as a **`DependencyService`**. Just as it may sound, a **`DependencyService`** is built around the premise of a simplified **dependency injection** (DI) concept that is very easy to use. If you have ever used any **inversion of control** (IoC) containers before, you will feel right at home. If you have never used an IoC container, you'll be comfortable with it in no time.

DependencyService

The easiest way to understand the **`DependencyService`** is to think of it as a class with the sole purpose of registering classes and retrieving instances of those classes whenever needed. Obviously there is quite a bit more that goes into this process, but that is the beauty of our object-oriented programming friends abstraction and encapsulation. All we need to know about the **`DependencyService`** is these two pieces of functionality and the rest just works.

To fully understand the entire process, we will need to break the **`DependencyService`** down into the three pieces of functionality required to make it work: interfaces, registration, and retrieval.

Interfaces

When you see the term **interfaces** in the context of the **`DependencyService`**, you may be expecting some magic and mysticism to go along with it. Well, I hate to disappoint you, but there isn't any. Even within Xamarin.Forms and the **`DependencySerivce`**, an interface, is an interface, is an interface. It is still just a mechanism for creating a contract that any class agrees to implement if it is defined in such a way. The following code is a simple example of an interface that we will use throughout the remainder of the explanation of the **`DependencyService`**.

```
public interface ISomeService
{
    void PerformActivity( );
}
```

Code Listing 94

As you can see, I have defined a very basic interface named **`ISomeService`**. This interface can have any name you would like. The definition of this interface needs to be created within the shared code project of your solution (either the PCL or Shared Project). At this point, from the perspective of your shared code, it doesn't care how this interface is implemented. Your shared code will simply use any class that agrees to this contract.

Registration

Now that we have an interface defined, we need to create some classes that implement it. These classes that implement the **`ISomeService`** interface will contain platform-specific code since that was the point all along. That being the case, if you are using a PCL for shared code; these implementations will need to be in the platform-specific projects. If you are using a Shared Project, they can exist in the shared code project. Either way will work depending on how you structured your application.

For example, assume that I have used a PCL project and am planning on targeting both iOS and Android for my application. If this were the case, I would need to create a class in both my Xamarin.iOS and Xamarin.Android projects that implement this interface. The implementation would look something like this.

iOS

```
public class Service_iOS : ISomeService
{
    public void PerformActivity( ) {
        // Native iOS implementation
    }
}
```

Code Listing 95

Android

```
public class Service_Android : ISomeService
{
    public void PerformActivity( ) {
        // Native Android implementation
    }
}
```

Code Listing 96

As you can see here, there is nothing out of the ordinary about these implementations. As a general rule, I like to add the **`_<platform>`** suffix to the class names just so I don't get confused later on in my editor if I have both files open. This naming process also serves the purpose of being able to reference multiple classes that implement the same interface. There is one final piece of the puzzle that will allow both of these classes to be registered in the **`DependencyService`**.

Like many other features in Xamarin in general, the **`DependencyService`** will utilize a very handy .NET Framework construct, the attribute. In order to declare that a particular class should be registered within the **`DependencyService`**, a special assembly attribute is used. This attribute will be nearly identical for all the registrations and will look like this for our iOS and Android samples respectively.

```
[assembly: Xamarin.Forms.Dependency(typeof(Service_iOS))]
```

Code Listing 97

```
[assembly: Xamarin.Forms.Dependency(typeof(Service_Android))]
```

Code Listing 98

These attributes will need to be placed outside of any namespace declarations in order to take effect. Now, during the running of this application, Xamarin.Forms will see these attributes, determine these classes implement the **ISomeService** interface, and create a reference within the **DependencyService** for each of these types as being implementations of the **ISomeService** interface. This is a very important concept to understand when it comes to retrievals. When you are registering a class, you are registering it as an implementation of an interface.

Retrieval

The final piece of using the **DependencyService** is being able to retrieve a class, or more specifically an instance of a class, that you have registered. This process is done using a very simple method that is part of the **DependencyService** class. The method name is **Get** and it is a generic method which we can use to specify what interface we would like to use. Since an interface doesn't actually have an implementation, we are asking the Xamarin.Forms infrastructure to give us any registered class that implements that interface. The usage of this method would once again be in the shared code project, and following our examples up to this point, it would look like this.

```
var service = DependencyService.Get<ISomeService>( );
service.PerformActivity();
```

Code Listing 99

Here we are asking the **DependencyService** to retrieve an implementation of the **ISomeService** interface. Once we have that implementation, we can use any of the functionality of that class exposed by the interface. If we were running this application on an Android device or emulator, the **DependencyService** would only know about the **Service_Android** registration and would provide that implementation. In the same way, when running on an iOS device or simulator, it would only know about and retrieve the **Service_iOS** registration.

Real World Example

Now that you have an understanding of the high-level concepts of how to use the **DependencyService** to access native features through Xamarin.Forms, let's put this knowledge to good use and create a real example.

In this example, we want to build a mobile application that will allow users to send an SMS message to someone. As you may already know, most platforms will not allow you to programmatically send an SMS on your own; you will need to use the built-in SMS application of the device. That is what we will utilize.

Let's start by creating a new application using the **Blank App (Xamarin.Forms Portable)** template and name it **XamarinFormsSMS**.

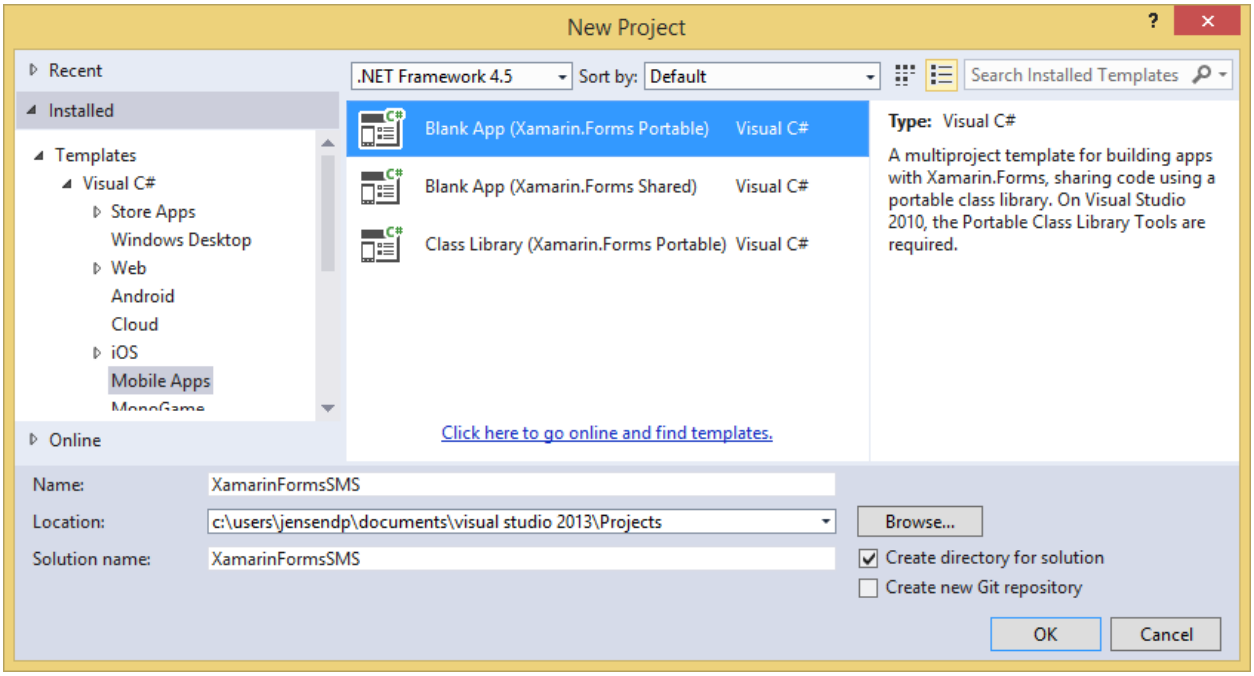


Figure 154: New Xamarin.Forms project

Once your solution has been created, add a new interface to the **XamarinFormsSMS (Portable)** project and name it **`ISmsService`** and replace the implementation with the following.

```
public interface ISmsService {
    void SendSMS( string phoneNumber, string text );
}
```

Code Listing 100

As you can see we are keeping the interface very simple. Feel free to modify and expand upon the functionality within this interface as you wish.

Now, we are going to add a class in each of the three platform-specific projects that implement this interface in a native fashion.

Android

In the **XamarinFormsSMS.Android** project, add a new class named **SmsServer_Android**, and replace the default implementation with the following.

```
using Android.Telephony;
using XamarinFormsSMS.Android;

[assembly: Xamarin.Forms.Dependency(typeof(SmsService_Android))]

namespace XamarinFormsSMS.Android
{
    public class SmsService_Android : ISmsService
    {
        public void SendSMS( string phoneNumber, string text ) {
            SmsManager.Default.SendTextMessage(phoneNumber, null, text, n
ull, null);
        }
    }
}
```

Code Listing 101

In the Android world, to send an SMS message, you will use the **SmsManager** class, use the **Default** property, and the **SendTextMessage** method. This will present the user with the built-in Android app that is in charge of sending SMS messages.

iOS

In the **XamarinFormsSMS.iOS** project, add a new class named **SmsServer_iOS**, and replace the default implementation with the following:

```
using MonoTouch.MessageUI;
using MonoTouch.UIKit;
using XamarinFormsSMS.iOS;

[assembly: Xamarin.Forms.Dependency(typeof(SmsService_iOS))]

namespace XamarinFormsSMS.iOS
{
    public class SmsService_iOS : ISmsService
    {
        public void SendSMS( string phoneNumber, string text ) {
            var composerViewController = new MFMessageComposeViewControll
er()
            {
                Recipients = new[] { phoneNumber },
```

```
                Body = text,
            };

            UIApplication.SharedApplication.KeyWindow.RootViewController.
PresentViewController(composerViewController, true, null);
        }
    }
}
```

Code Listing 102

In the context of iOS, the process of sending an SMS message requires two steps. First, you must create a new instance of the **`MFMessageComposeViewController`** with the recipients you want to send the message to and the text you are sending. Secondly, you will present this **`ViewController`** to the user with the **`PresentViewController`** method. To make sure you are presenting the **`ViewController`** at the correct place in the view hierarchy, you should use the **`KeyWindow`** property which gets the currently displayed **`Window`**. From there you can use the **`RootViewController`** property to get the top-level **`ViewController`**, and then use the **`PresentViewController`** method.

Windows Phone

Finally, we will implement the same interface in the Windows Phone application. In the **XamarinFormsSMS.WinPhone** project, add a new class named **`SmsServer_WinPhone`**, and replace the default implementation with the following code.

```
using Microsoft.Phone.Tasks;
using XamarinFormsSMS.WinPhone;

[assembly: Xamarin.Forms.Dependency(typeof(SmsService_WinPhone))]

namespace XamarinFormsSMS.WinPhone
{
    public class SmsService_WinPhone : ISmsService
    {
        public void SendSMS( string phoneNumber, string text ) {
            var smsTask = new SmsComposeTask {
                To = phoneNumber,
                Body = text
            };

            smsTask.Show( );
        }
    }
}
```

Code Listing 103

In order to send an SMS message in Windows Phone, you simply need to create a new instance of an **`SmsComposeTask`** specifying the **`To`** and **`Body`** properties and calling the **`Show`** method. This will present users with the native UI containing the phone number and text and allows them to send the message.

Shared Code

The last step to accessing this platform-specific code is to use the **`DependencyService`** from our shared code. To do this, we will first create a simple user interface to allow users to enter some data, click a button, and then transfer control over to the native implementations.

Let's start by modifying the **App.cs** file within the **XamarinFormsSms (Portable)** project to look like the following code example.

```
using Xamarin.Forms;

namespace XamarinFormsSMS
{
    public class App : Application
    {
        public  App() {
            var phoneLabel = new Label {
                VerticalOptions = LayoutOptions.Center,
                Text = "Phone Number"
            };
            var phoneEntry = new Entry {
                VerticalOptions = LayoutOptions.Center,
                HorizontalOptions = LayoutOptions.FillAndExpand,
                Placeholder = "Enter Phone Number"
            };

            var textLabel = new Label {
                VerticalOptions = LayoutOptions.Center,
                Text = "Text"
            };
            var textEntry = new Entry {
                VerticalOptions = LayoutOptions.Center,
                HorizontalOptions = LayoutOptions.FillAndExpand,
                Placeholder = "Enter text"
            };

            var button = new Button {
                HorizontalOptions = LayoutOptions.FillAndExpand,
                Text = "Send Message"
            };

            button.Clicked += ( sender, args ) => {
                var smsService = DependencyService.Get<ISmsService>( );
```

```
                var phoneNumber = phoneEntry.Text;
                var text = textEntry.Text;
                smsService.SendSMS( phoneNumber, text );
            };

            var phoneStack = new StackLayout {
                Orientation = StackOrientation.Horizontal,
                Children = {phoneLabel, phoneEntry}
            };

            var textStack = new StackLayout {
                Orientation = StackOrientation.Horizontal,
                Children = {textLabel, textEntry}
            };

            var parentStack = new StackLayout {
                Children = {phoneStack, textStack, button}
            };

            MainPage = new NavigationPage( new ContentPage
            {
                Title = "SMS Helper",
                Content = parentStack
            });
        }
    }
}
```

Code Listing 104

This will provide a basic layout for our applications that will look like the following figures.

Figure 155: SMS Helper on Android

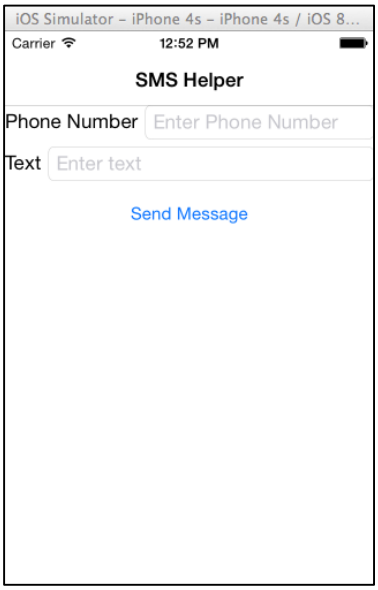


Figure 156: SMS Helper on iOS

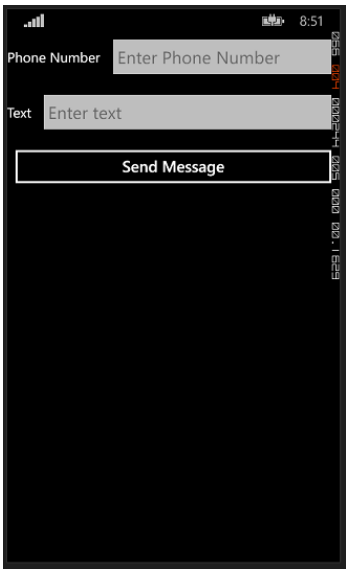


Figure 157: SMS Helper on Windows Phone

Finally, when you fill out the information on the screen and click the button, you will be presented with each of the native SMS applications. Unfortunately, iOS doesn't allow you to use the SMS functionality within the simulator. Since this is the case, the iOS image in Figure 159 is a screen capture from a device running our application.

Figure 158: Native SMS App on Android

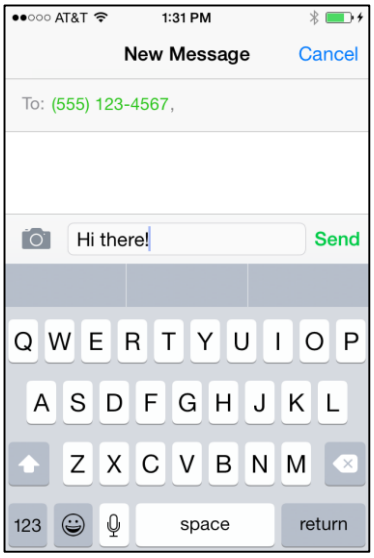


Figure 159: Native SMS App on iOS

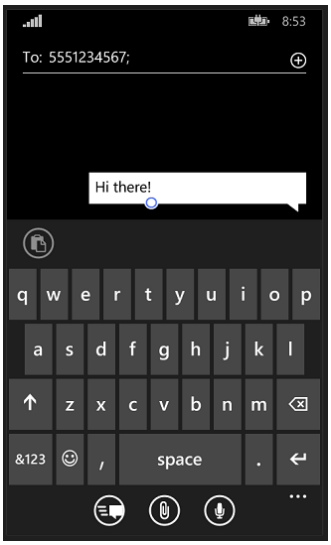


Figure 160: Native SMS App on Windows Phone

Summary

In this chapter we have focused on the process of accessing native features of the individual platforms through a common mechanism. At the center of this process is the **`DependencyService`** class. It's through this class that we can register and retrieve implementations of an agreed upon interface. This way, our Xamarin.Forms code doesn't need to worry about the specific implementation of a particular class. All the common code needs to work with is an interface that all the different platform-specific implementations are implementing.

Chapter 9 Messaging

As software developers, we are always trying to write applications that are more maintainable and easier to test. Whether we want to admit it or not, that is something we are (or at least should be) striving for. When it comes to achieving this goal, there is no silver bullet. Getting closer to this goal requires a lot of work and design within our applications. Even though there may not be one solution to this problem, there is a generally accepted rule of thumb that will automatically get you closer to nirvana. Keep your code as decoupled (or separated) as possible.

The concept of decoupling is not a new one in software development. This simply means to try to make different pieces of your code "dumb" when it comes to other pieces. One class shouldn't know (or care about) the inner workings of another. It should only know about the properties or methods that it needs to work and nothing more. There are a number of mechanisms to achieve this goal that typically involves some sort of dependency injection (DI). While this is a very good strategy, there is another one out there that is often overlooked, especially in mobile development. That strategy is messaging.

Typically when you think of the messaging (publish and subscribe) model, you think of large-scale enterprise applications with a lot of complicated interactions between systems. While this is a valid thought, you may find it surprising that it can play a central role in smaller applications as well; even ones that involve mobile platforms.

In this chapter, we will discuss the **`MessagingCenter`** within Xamarin.Forms and see how we can utilize it to create more decoupled applications that will not only make our code more maintainable, but also more testable.

The Pub/Sub Model

The concept of the **pub/sub** (publish/subscribe) model or pattern is a relatively simple one. Even though the idea behind this pattern is simple, the underlying implementation can be rather complicated. That being the case, we will be covering this topic at a rather high-level and focusing on its usage within Xamarin.Forms. At its foundation, the pub/sub aspects of an application are broken down into three main categories: **publishers**, **subscribers**, and **messages**.

A publisher is a part of your application (typically a class) that broadcasts a message out to the system to notify one or many unknown classes that something has happened. This is very similar to the basics of the event model used within .NET. The subscribers, on the other hand, are the classes that are listening for certain messages to appear in the system. These subscribers then do some sort of processing based on this message.

The glue that ties the publishers and subscribers together is the message. A message is some piece of data that the publishers and subscribers use to communicate. A message can be as simple as containing just a string or a as complex as containing a full object. What is being sent is completely up to the system and the publishers and subscribers.

MessagingCenter

At the center of the messaging functionality of Xamarin.Forms is the **MessagingCenter**. The **MessagingCenter** is the mechanism by which your classes can publish and subscribe to messages. It is a simple class that contains three primary methods (**Subscribe**, **Unsubscribe**, and **Send**) with a couple of overloads to support sending different kinds of data with the messages. Let's pick each of these methods apart to see how to use them.

Subscribe

The **Subscribe** method is used by a class to let the **MessagingCenter** know that if a particular type of message is published, it wants to be notified so that it can run some operation. The **Subscribe** method has two forms that look like the following code.

```
// Without message argument
public static void Subscribe<TSender>(object subscriber, string message,
Action<TSender> callback, TSender source = null)

// With message argument
public static void Subscribe<TSender, TArgs>(object subscriber, string me
ssage, Action<TSender, TArgs> callback, TSender source = null), Action<TS
ender, TArgs> callback, TSender source = null)
```

Code Listing 105

These two methods are static, generic methods. The first sends a simple message without any arguments to the **MessagingCenter**. The generic argument to the method is the type message. The parameters to the method are:

- **subscriber**: The subscriber to the message (typically the class calling **Subscribe**).
- **message**: The name of the message being sent.
- **callback**: The delegate to be executed upon receiving a message.
- **source**: The source of the message. This is a default parameter that will accept messages from anything when left **null**.

The second method is exactly the same with two small modifications. This method has a second generic argument, **TArgs**, which specifies the arguments being passed into the delegate to execute. Since there is an additional generic argument on the method, that same argument appears on the third parameter so the delegate knows about it.

Here is a simple example of the **Subscribe** method in action.

```
MessagingCenter.Subscribe<Message>( this, "test", async ( message ) => {
    await DisplayAlert( "test message received", message.Data , "OK" );
    }
);
```

Code Listing 106

When this code is added to a **Page**, the **Page** will be subscribed to any "test" messages that are published through the **MessagingCenter**. The generic parameter of type **Message** means that that this class is subscribing to this message. In this simple example, the delegate merely shows a pop-up to the user, via the **DisplayAlert** method, showing some text and displaying an OK button.

Unsubscribe

The process of subscribing to a message is very beneficial in the overall architecture of your application. But what happens when you no longer need to be notified about a certain message? Do you continue to receive it, but ignore it in code? That is probably not the best solution. The problem with staying subscribed to an action is that it takes system resources and the **MessagingCenter** needs to remember that your class is receiving that message. That being the case, it is typically best to unsubscribe from that message once it is no longer needed.

Just as with the **Subscribe** method, there are two generic **Unsubscribe** methods that mirror their **Subscribe** counterparts.

```
// Without message argument
public static void Unsubscribe<TSender>(object subscriber, string message
)

// With message argument
public static void Unsubscribe<TSender, TArgs>(object subscriber, string
message
```

Code Listing 107

In the case of **Unsubscribe**, there is no **delegate**. This method simply removes the subscriber from the **MessagingCenter** as wanting to receive the specified message.

Taking advantage of the unsubscribe feature through a slight modification to the **Subscribe** example code would look something like the following.

```
MessagingCenter.Subscribe<Message, string>( this, "test", async ( message
, arg ) => {
var accept = await DisplayAlert( "test message received", message.GetData
(arg), "OK", "Unsubscribe" );
    if ( !accept ) {
```

```
            MessagingCenter.Unsubscribe<Message, string>(this, "test");
        }
    });
```

Code Listing 108

Now, we are sending some string data along with the message. When the user makes a choice from the **`DisplayAlert`** method, we are storing that value. If the user doesn't tap the **OK** button, the **`Unsubscribe`** method will be called and this class will no longer accept messages of type **`test`**.

Send

In the previous two sections, we learned how to subscribe and unsubscribe from the publishing of a message, but it doesn't do us any good if no one is sending any messages. That's where the **`Send`** method comes in.

The **`Send`** method also has two generic forms that allow us to publish a message to the **`MessagingCenter`** and also pass parameters to the delegate if we wish. The two forms of the method are as follows.

```
// Without message argument
public static void Send<TSender>(TSender sender, string message)

// With message argument
public static void Send<TSender, TArgs>(TSender sender, string message, T
Args args)
```

Code Listing 109

Using these methods, we can now publish a method that will be received by our subscribers in the previous sections.

```
MessagingCenter.Send<Message, string>( new Message{Data = "Some Data"}, "
test");
```

Code Listing 110

Completed Example

Putting all three pieces of the puzzle together from the previous sections, we can create a very simple application that uses all the functionality of the **`MessagingCenter`**. Simply create a new Xamarin.Forms application, give it a name, and replace the contents of the **App.cs** file with the following.

```
using Xamarin.Forms;

namespace XamarinFormsSMS
{
    public class App : Application
    {
        public  App() {
            MainPage = new MainPage();
        }
    }

    public class MainPage : ContentPage {
        public MainPage( ) {
            var button = new Button {
                Text = "Send message"
            };
            button.Clicked += ( sender, args ) => MessagingCenter.Send<Ma
inPage, string>( this, "test", "Hello there!" );

            MessagingCenter.Subscribe<MainPage, string>( this, "test", as
ync ( sender, arg ) => {
                var accept = await DisplayAlert( "test message received",
 arg, "OK", "Unsubscribe" );
                if ( !accept ) {
                    MessagingCenter.Unsubscribe<MainPage, string>(this, "
test");
                }
            } );

            Content = button;
        }
    }
}
```

Code Listing 111

In this example, we are creating a simple **Page** named **MainPage**. Within the constructor of this class, we create a button that has a **Clicked** event handler. When the button is clicked, a **test** message is published with an argument of **"Hello there!"** Also in the constructor, the **MainPage** class is subscribing to the **test** message and will display a popup to the user. If the user selects **OK** and taps the button again, they will continue to get the popup. If the user selects **Unsubscribe**, the **MainPage** class is will be unsubscribed from the **test** message and the popup will no longer appear.

When the application first runs, you will be presented with the following screen.

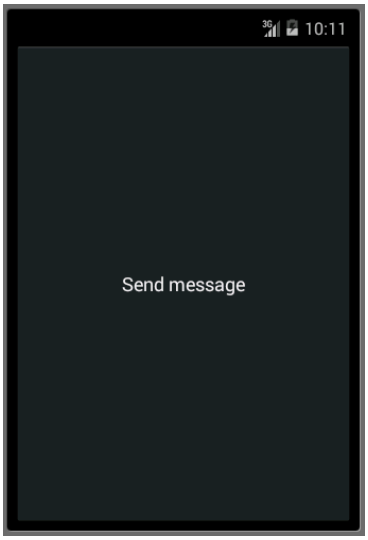


Figure 161: Send message button on Android

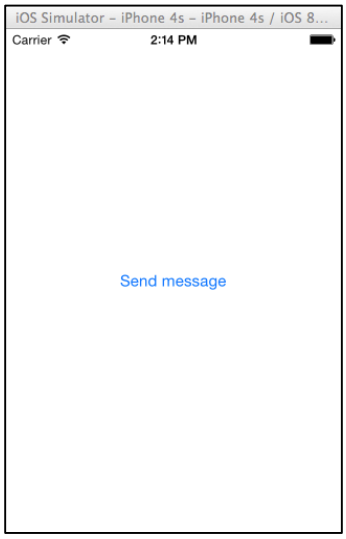


Figure 162: Send message button on iOS

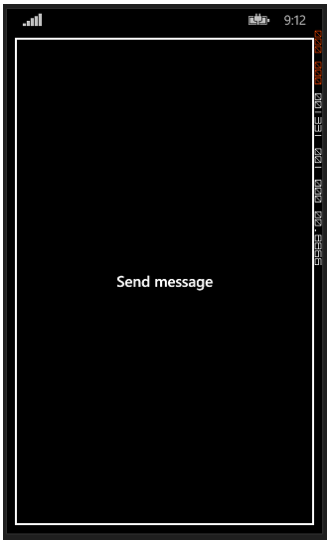


Figure 163: Send message button on Windows Phone

After the **Send message** button is pressed, you are presented with the **`Alert`** dialog as shown in the following figures.

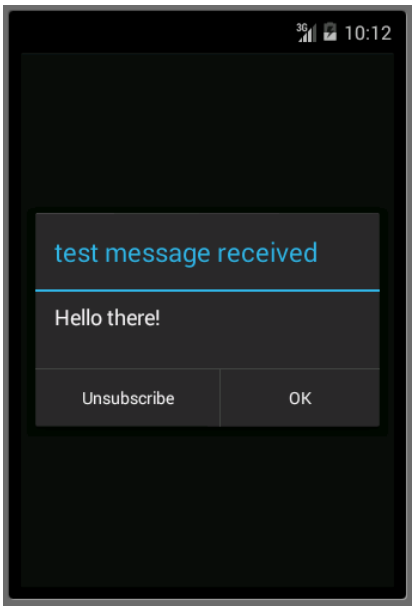


Figure 164: Alert on Android

Figure 165: Alert on iOS

Figure 166: Alert on Windows Phone

Once again, if the user taps **OK**, the application will continue to loop back and forth between the **`button`** and the **`Alert`** message. If at any point the user chooses the **Unsubscribe** button, the **`Alert`** dialog will no longer appear.

Summary

In this chapter, we covered the basic concepts around the pub/sub model and how to use the **`MessagingCenter`**. The **`MessagingCenter`** is a very useful tool when trying to further decouple your application code for basic maintainability and testability. This model of development comes in handy when using the MVVM pattern when you are trying to communicate between different **`ViewModel`** classes without having to pass data into the XAML code-behind files.

Keep in mind that the example in this chapter is a very simple one. Messages can be published and subscribed to from different classes spanning across your entire application. Also, the arguments passed around in the **`Send`** and **`Subscribe`** methods can be any object you would like that is either built-in to the .NET Framework or a custom class of your own design. You are only limited by your imagination!