XAMARIN.FORMS

SUCCINCTLY

BY ALESSANDRO DEL SOLE

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Xamarin.Forms Succinctly

By
Alessandro Del Sole

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: James McCaffrey
Copy Editor: Jacqueline Bieringer, content producer, Syncfusion, Inc.
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the Succinctly Series of Books **8**

About the Author **10**

Introduction **11**

Chapter 1 Getting Started with Xamarin.Forms **12**

Introducing Xamarin and Xamarin.Forms 12

Setting up the development environment 13

Configuring a Mac 14

Creating Xamarin.Forms solutions 15

The Xamarin.Forms library 18

The Xamarin.Android project 18

The Xamarin.iOS project 20

The Universal Windows Platform project 24

Debugging and testing applications locally 25

Setting up the iOS Simulator 27

Running apps on physical devices 27

The Xamarin Live Player app 28

Analyzing and profiling applications 28

Chapter summary 29

Chapter 2 Sharing Code Among Platforms **30**

Introduction to code-sharing strategies 30

Sharing code with Portable Class Libraries 30

Sharing code with shared projects 31

Sharing code with .NET Standard libraries 34

Chapter summary 37

Chapter 3 Building the User Interface with XAML 38

The structure of the user interface in Xamarin.Forms 38

Coding the user interface in C# 39

The modern way: designing the user interface with XAML 40

Responding to events 42

Understanding type converters 44

Xamarin.Forms Previewer 45

Hints for XAML Standard 46

Chapter summary 46

Chapter 4 Organizing the UI with Layouts 47

Understanding the concept of layout 47

Alignment and spacing options 48

The `StackLayout` 49

The `Grid` 51

Spacing and proportions for rows and columns 53

Introducing spans 54

The `AbsoluteLayout` 54

The `RelativeLayout` 56

The `ScrollView` 57

The `Frame` 58

The `ContentView` 59

Chapter summary 61

Chapter 5 Xamarin.Forms Common Controls 62

Understanding the concept of view 62

Views' common properties 62

Introducing common controls 63

User input with the `Button` 63
Working with text: `Label`, `Entry`, `Editor` 64
Working with dates and time: `DatePicker` and `TimePicker` 65
Displaying HTML contents with `WebView` 67
Implementing value selection: `Switch`, `Slider`, `Stepper` 68
Introducing the `SearchBar` 70
Long-running operations: `ActivityIndicator` and `ProgressBar` 71
Working with images 72
Introducing gesture recognizers 74
Displaying alerts 75
Chapter summary 76
Chapter 6 Pages and Navigation 77
Introducing and creating pages 77
Single views with the `ContentPage` 78
Splitting contents with the `MasterDetailPage` 78
Displaying content within tabs with the TabbedPage 80
Swiping pages with the `CarouselPage` 81
Navigating among pages 83
Passing objects between pages 84
Animating transitions between pages 85
Managing the page lifecycle 85
Handling the hardware back button 85
Chapter summary 86
Chapter 7 Resources and Data Binding 87
Working with resources 87
Declaring resources 87

Introducing styles 88
Working with data binding 90
Working with collections and with the `ListView` 93
Introducing Model-View-ViewModel 103
Chapter summary 108
Chapter 8 Accessing Platform-Specific APIs 109
The `Device` class and the `OnPlatform` method 109
Working with the dependency service 111
Working with plugins 113
Working with native views 115
Embedding native views in XAML 116
Working with custom renderers 117
Hints for effects 120
Chapter summary 120
Chapter 9 Managing the App Lifecycle 121
Introducing the `App` class 121
Managing the app lifecycle 121
Sending and receiving messages 124
Chapter summary 125
Appendix: Useful Resources 126
Working with SQLite databases 126
Consuming web services and cloud services 126
Publishing applications 126
Code examples and starter kits 126
Updating Xamarin tools 127

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 Face-book to help us spread the word about the *Succinctly* series!

About the Author

Alessandro Del Sole is a Xamarin Certified Mobile Developer and has been a Microsoft MVP since 2008. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally considered a Visual Studio expert and a .NET authority. Alessandro has authored many printed books and e-books on programming with Visual Studio, including *Visual Studio 2017 Succinctly*, *Visual Basic 2015 Unleashed*, and *Visual Studio Code Succinctly*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals, including MSDN Magazine and the Visual Basic Developer Center from Microsoft. He is a frequent speaker at Italian conferences, and he has released a number of Windows Store apps. He has also produced a number of instructional videos in both English and Italian. Alessandro works as a senior .NET developer, trainer, and consultant. You can follow him on Twitter at @progalex.

Introduction

For mobile app developers and companies that want to be represented on the market by mobile applications, the need to publish Android, iOS, and Windows versions of apps has dramatically increased in the last few years. For companies that have always worked with native platforms and development tools, this might not be a problem. It is a problem, though, for companies that have always built software with .NET, C#, and, more generally, with the Microsoft stack. A company might therefore hire specialized developers or wait for existing developers to attain the necessary skills and knowledge to work with native platforms but, in both cases, there are huge costs and risks with timing. The ideal solution is that developers could reuse their existing .NET and C# skills to build native mobile apps. This is where Xamarin comes in.

In this e-book, you will learn how Xamarin.Forms allows for cross-platform development, letting you create mobile apps for Android, iOS, and Windows from a single C# codebase, and therefore reuse your existing .NET skills. You will learn how Xamarin.Forms solutions are made, what makes it possible to share code, how to create the user interface, how to organize controls within containers, and how to implement navigation between pages. You will also leverage advanced techniques such as data binding and accessing native APIs from cross-platform code.

It is worth mentioning that Xamarin.Forms also supports the F# programming language, but C# is obviously the most common choice and therefore all the explanations and examples will be provided based on C#. It is also worth mentioning that, in the past, Xamarin.Forms supported Windows Phone 8.x and Windows 8.x as target platforms, and support for Universal Windows Platform on Windows 10 was added recently. In Visual Studio 2017, Xamarin.Forms only supports UWP for Windows development. For this reason, when I refer to Windows from now on, I mean Windows 10 and the Universal Windows Platform, not the previous versions.

In this e-book, I will assume you have at least a basic knowledge of C# and the Visual Studio IDE. I also suggest you bookmark the official Xamarin documentation portal for quick reference. The source code for this e-book is available on Bitbucket. File names are self-explanatory so that it's easier for you to follow the examples, especially for Chapters 4, 5, and 6. Before you start writing code, you need to set up your development environment. This is the topic of the first chapter.

Chapter 1 Getting Started with Xamarin.Forms

Before you start writing mobile apps with Xamarin.Forms, you first need to understand the state of mobile app development today and how Xamarin fits into it. Also, you need to set up your development environment to be able to build, test, debug, and deploy your apps to Android, iOS, and Windows devices. This chapter introduces Xamarin as a set of tools and services, Xamarin.Forms as the platform you will use, and then presents the tools and hardware you need for real-world development.

Introducing Xamarin and Xamarin.Forms

Xamarin is the name of a company that Microsoft acquired in 2016 and, at the same time, the name of a set of development tools and services that developers can use to build native apps for iOS, Android, and Windows in C#. Xamarin's main goal is to make it easier for .NET developers to build native apps for Android, iOS, and Windows reusing their existing skills. The reason behind this goal is simple: building apps for Android requires you to know Java and Android Studio or Eclipse; building apps for iOS requires you to know Objective-C or Swift and Xcode; building apps for Windows requires you to know C# and Visual Studio. As an existing .NET developer, whether you are experienced or a beginner, getting to know all the possible platforms, languages, and development environments is extremely difficult and costs are extremely high for this.

Xamarin allows you to build native apps with C#, based on a cross-platform, open-source porting of the .NET Framework called Mono. From a development point of view, Xamarin offers a number of flavors: Xamarin.iOS and Xamarin.Mac, libraries that wrap native Apple APIs you can use to build apps for iOS and macOS using C# and Visual Studio; Xamarin.Android, a library that wraps native Java and Google APIs you can use to build apps for Android using C# and Visual Studio; Xamarin.Forms, an open-source library that allows you to share code across platforms and build apps that run on Android, iOS, and Windows from a single C# codebase. The biggest benefit of Xamarin.Forms is that you write code once and it will run on all the supported platforms at no additional cost. As you'll learn throughout this e-book, Xamarin.Forms consists of a layer that wraps objects common to all the supported platforms into C# objects. Accessing native, platform-specific objects and APIs is possible in several ways, all discussed in the next chapters, but requires some extra work. Additionally, Xamarin integrates with the Visual Studio IDE on Windows and is part of the newly released Visual Studio for Mac, so you can not only create cross-platform solutions, but also write code on different systems.

The Xamarin offering also includes Xamarin University, a paid service that allows you to attend live classes online and watch instructional videos that will help you prepare to get the Xamarin Certified Mobile Developer badge. It also includes the Xamarin Test Cloud service for test automation. Microsoft has also recently started the Visual Studio Mobile Center, a complete cloud solution for the app management lifecycle from build automation to continuous integration, tests, analytics, and much more (note that Internet Explorer is not supported). This e-book focuses on Xamarin.Forms and targets Visual Studio 2017 on Windows 10, but all the technical

concepts apply to Visual Studio for Mac as well. The author of this e-book has also recorded a video series for Syncfusion that provides an overview of what Xamarin offers and of Xamarin.iOS and Xamarin.Android, as well as Xamarin.Forms.

Setting up the development environment

In order to build native mobile apps with Xamarin.Forms, you need Windows 10 as your operating system and Microsoft Visual Studio 2017 as your development environment. You can download and install the Visual Studio 2017 Community edition for free and get all the necessary tools for Xamarin development. When you start the installation, you will need to select the **Mobile development with .NET** workload in the Visual Studio Installer (see Figure 1).

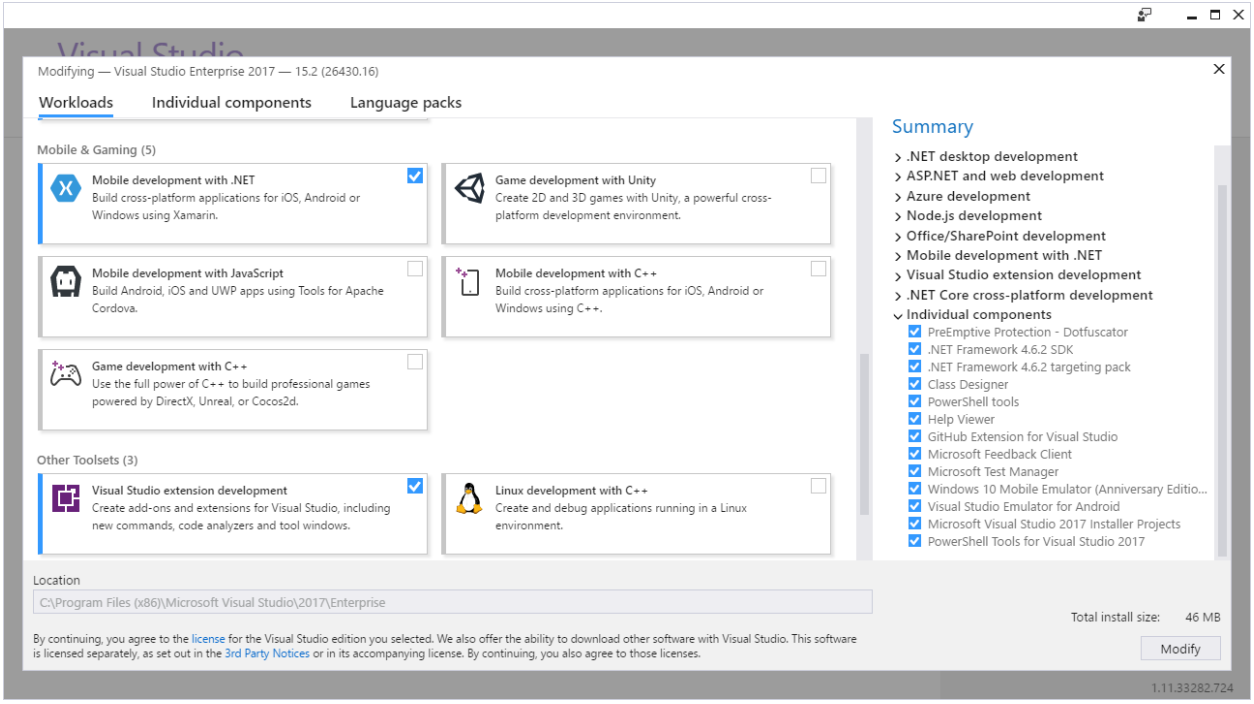

Figure 1: Installing Xamarin development tools

When you select this workload, the Visual Studio Installer will take care of downloading and installing all the necessary tools to build apps for Android, iOS, and Windows. iOS actually requires additional configuration described in the next section. Also, for Windows 10 development, you need additional tools and SDKs, which you get by also selecting the **Universal Windows Platform Development** workload. If you select the **Individual components** tab, you will have an option to check if Android and Windows emulators have been selected or to make a choice manually (see Figure 2).

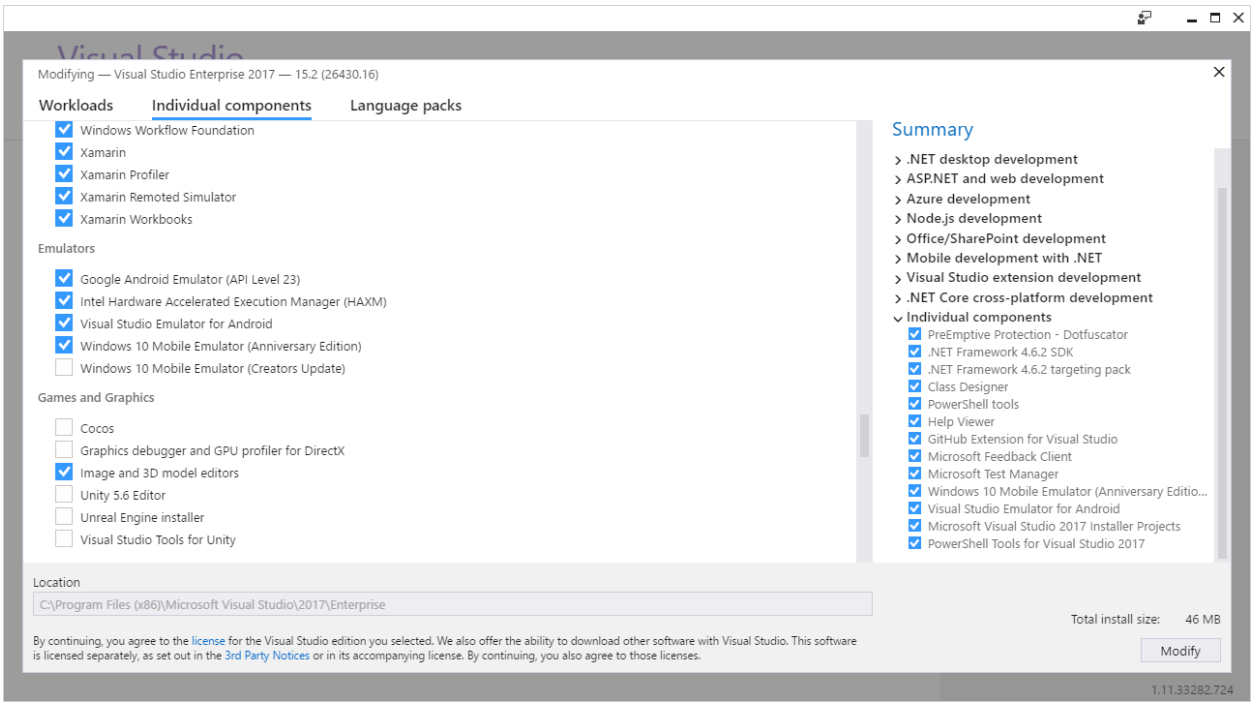

Figure 2: Selecting emulators

If you think you will use both Visual Studio 2017 and Visual Studio for Mac, I suggest you install the Google emulator, which has an identical appearance and behavior on both systems. If you only work on Windows, a good choice is the Visual Studio Emulator for Android from Microsoft. This emulator requires Hyper-V to run, as it is a full virtual machine. Plans from Microsoft to support this emulator in the future are still uncertain, so my suggestion is that you try and use it until it's no longer available, since it's faster and more powerful. For the Windows 10 emulator, my suggestion is to download the oldest version (in Figure 2, it is the Anniversary Update) so that you can target older versions of Windows 10 as well. Go ahead with the installation and wait for it to complete.

Configuring a Mac

Apple's policies establish that a Mac computer is required to build an app. This is because only the Xcode development environment and the Apple SDKs are allowed to run the build process. For local debugging and testing, Xamarin has recently introduced the Xamarin Live Player app, which you can download and install on your Android or iOS device and pair with Visual Studio for debugging purposes. More details about this app will be provided shortly. However, it becomes insufficient for serious development. You still need a Mac computer for code signing, setting up profiles, and publishing an app to the App Store. You can use a local Mac in your network, which also allows you to debug and test apps on a physical device, or a remote Mac. In both cases, macOS must be configured with the following software requirements:

- macOS "El Capitan" (10.11) or higher.
- Xcode and Apple SDKs, which you get from the App Store for free.
- The Xamarin.iOS engine. The easiest way to get Xamarin properly configured on a Mac is by installing Visual Studio Community for Mac.

Visual Studio will connect to the Mac to launch the Xcode compiler and SDKs, and therefore remote connections must be enabled for the latter. The official Xamarin documentation has a specific page that will help you configure a Mac computer and I recommend you read it carefully, especially because it explains how to configure profiles and certificates, and how to use Xcode to perform preliminary configurations. Actually, the documentation is about Xamarin.iOS, but the same steps apply to Xamarin.Forms.

Creating Xamarin.Forms solutions

Assuming that you have installed and configured your development environment, the next step is opening Visual Studio to see how you create Xamarin.Forms solutions and what these solutions consist of. Project templates for Xamarin.Forms are available in the **Visual C#**, **Cross-Platform** node of the **New Project** dialog (see Figure 3).

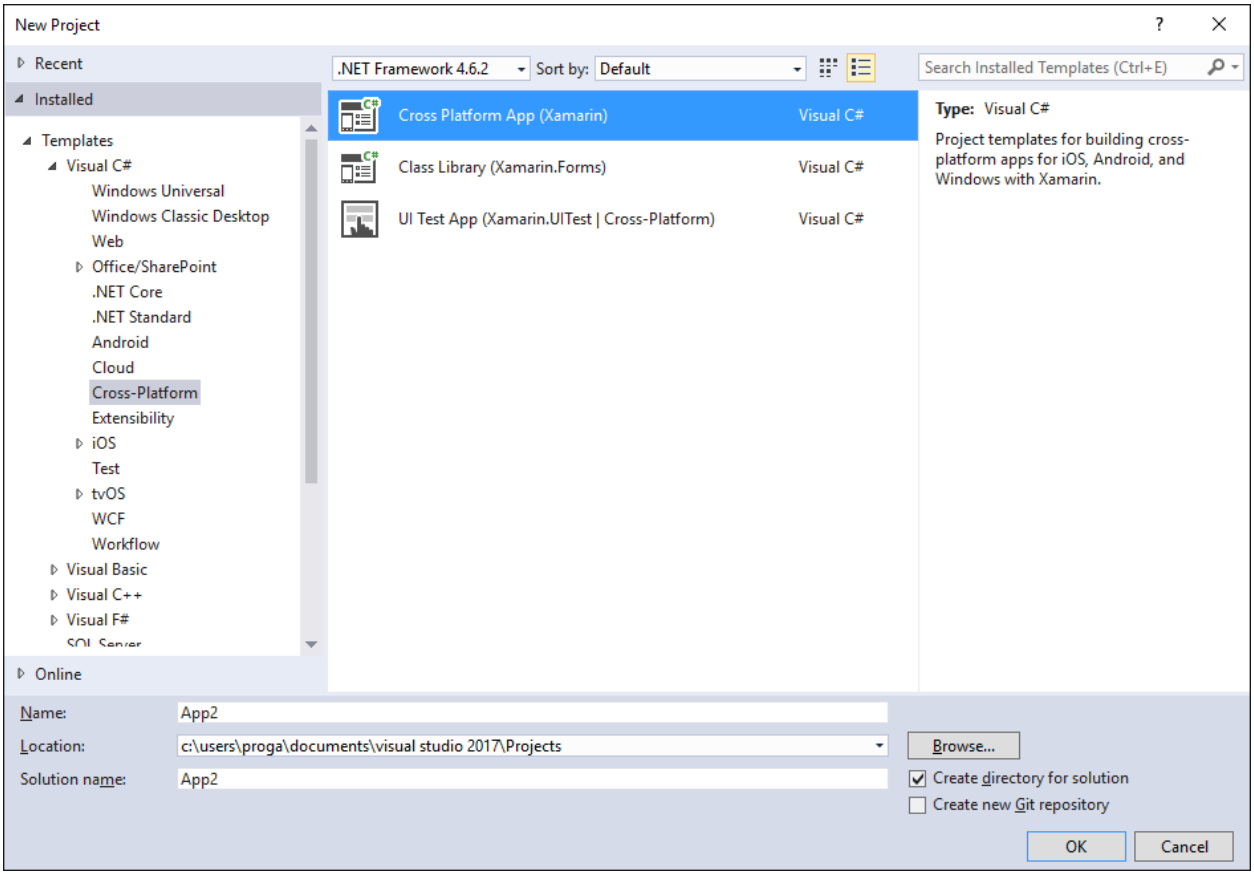

Figure 3: Project templates for Xamarin.Forms

The **Cross Platform App (Xamarin)** template is the one you use to build mobile apps. The Class Library (Xamarin.Forms) template allows you to create a reusable class library that can be consumed in Xamarin.Forms projects, and the UI Test App template is used to create automated UI tests, but these two templates will not be discussed in this e-book. Select the **Cross Platform App** template and optionally specify a project name (this is not relevant right now), then click **OK**. At this point, Visual Studio will ask you to select between Blank App and Master Detail templates (see Figure 4). The latter generates a basic user interface based on pages and visual elements that will be discussed later, with some sample data. It's not a good starting point unless you already have experience with Xamarin, so select the **Blank App** template.

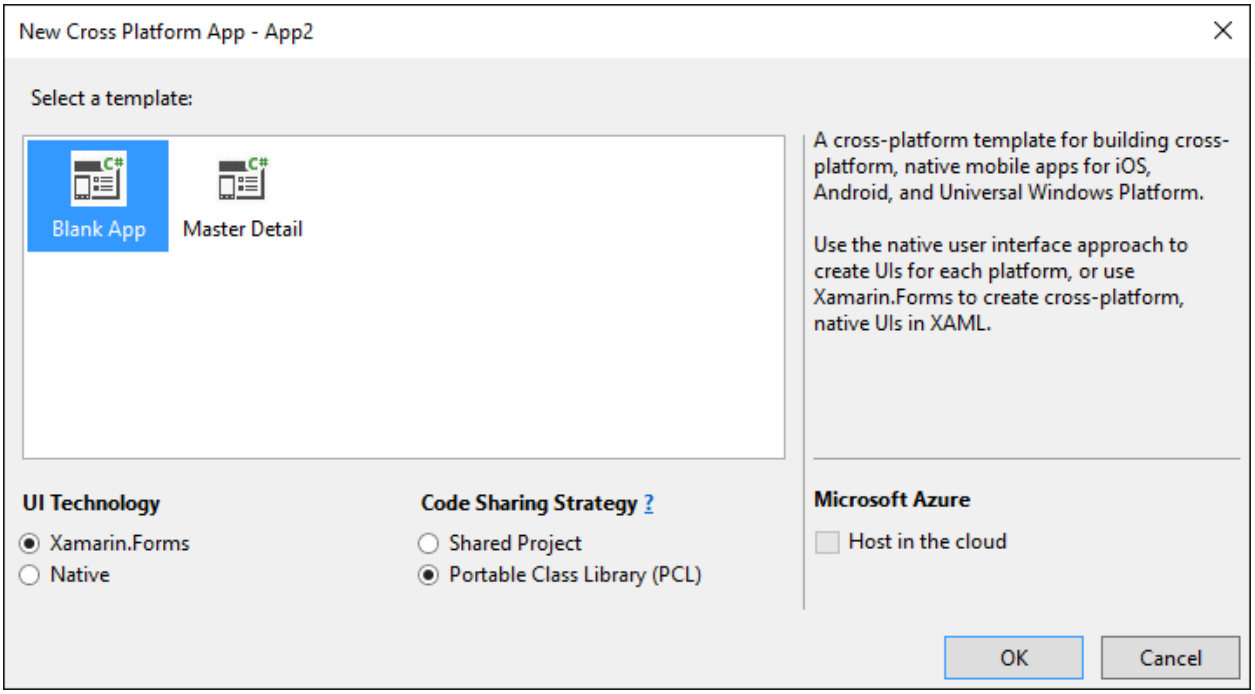

Figure 4: Selecting the template, UI technology, and code sharing strategy for a new project

In the **UI Technology** group, select **Xamarin.Forms**. If you select Native, your solution will focus on platform-specific projects rather than cross-platform code. In the **Code Sharing Strategy** group, you can choose between Shared Project and Portable Class Library. The code sharing strategy is a very important topic in Xamarin.Forms and Chapter 2 will provide a detailed explanation. For now, select the **Portable Class Library** option and click **OK** when ready. After a few seconds, while generating the solution, Visual Studio will ask you to specify the target version of Windows 10 for your new app. Leave the default selection unchanged and go ahead. It will also show a welcome dialog where, in the second screen, you will have an option to specify the location of a Mac computer. Skip this step for now, as it will be discussed in the next section. In Solution Explorer, you will see that the solution is made up of four projects, as demonstrated in Figure 5.

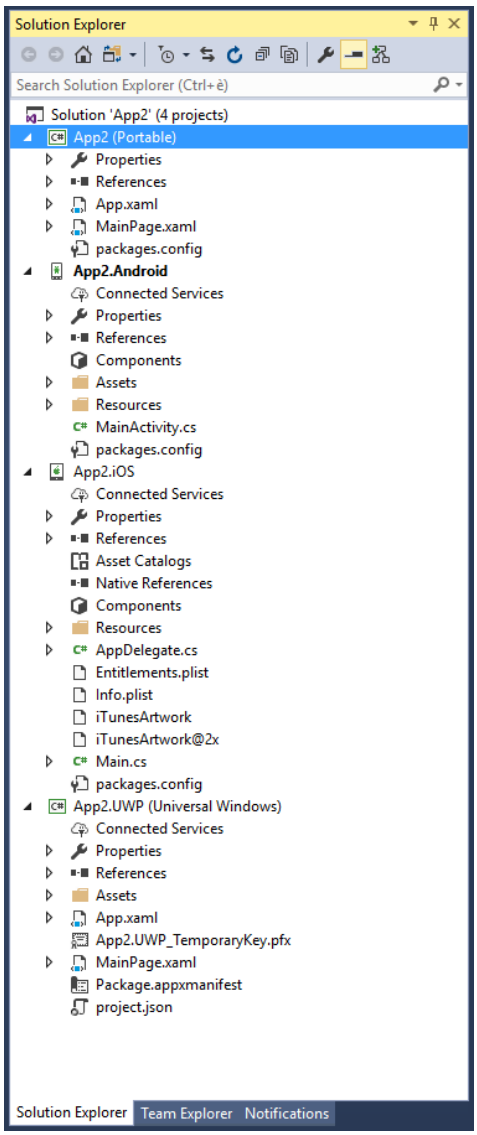

Figure 5: The structure of a Xamarin.Forms solution

The first project is either a Portable Class Library or a Shared Project, depending on your selection at project creation. This project contains all the code that can be shared across platforms and its implementation will be discussed in the next chapter. For now, what you need to know is that this project is the place where you will write all the user interface of your app, and all the code that does not require interacting with native APIs. The second project, whose suffix is Android, is a Xamarin.Android native project. It has a reference to the shared code and to Xamarin.Forms, and implements the required infrastructure for your app to run on Android devices. The third project, whose suffix is iOS, is a Xamarin.iOS native project. This one also has a reference to the shared code and to Xamarin.Forms, and implements the required infrastructure for your app to run on the iPhone and iPad. The fourth and last project is a native Universal Windows project (UWP) that has a reference to shared code and implements the infrastructure for your app to run on Windows 10 devices, for both the desktop and mobile devices. I will now provide more details on each platform project, so that you have a basic knowledge of their properties. This is very important, because you will need to fine-tune project properties every time you create a new Xamarin.Forms solution.

The Xamarin.Forms library

Technically speaking, Xamarin.Forms is a .NET library that exposes all the objects discussed in this e-book through a root namespace called **`Xamarin.Forms`**. It has been recently open sourced and it ships as a NuGet package that Visual Studio automatically installs to all projects when you create a new solution. The NuGet Package Manager in Visual Studio will then notify you of available updates. Because creating Xamarin.Forms solutions must be allowed even if your PC is offline, Visual Studio installs the version of the NuGet package in the local cache, which isn't typically the latest version available. For this reason, it is recommended that you upgrade the Xamarin.Forms and other Xamarin packages in the solution to the latest version, and, obviously, it is also recommended you only upgrade to stable releases. Though alpha and beta intermediate releases are often available, their only intended use is for experimenting with new features still in development. As of this writing, version 2.3.4.247 is the latest stable release.

The Xamarin.Android project

Xamarin.Android makes it possible for your Xamarin.Forms solution to run on Android devices. The **MainActivity.cs** file represents the startup activity for the Android app that Xamarin generates. In Android, an activity can be thought of as a single screen with a user interface, and every app has at least one. In this file, Visual Studio adds startup code that you should not change, especially the initialization code for Xamarin.Forms you see in the last two lines of code. In this project, you can add code that requires accessing native APIs and platform-specific features, as you will learn in Chapter 8. The **Resources** folder is also very important, because it contains subfolders where you can add icons and images for different screen resolutions. The name of such folders starts with *drawable* and each represents a particular screen resolution. The Xamarin documentation explains thoroughly how to provide icons and images for different resolutions on Android. The **Properties** element in Solution Explorer allows you to access the project properties, as you would do with any C# solution. In the case of Xamarin, in the **Application** tab (see Figure 6) you can specify the version of the Android SDK that Visual Studio should use to build the app package.

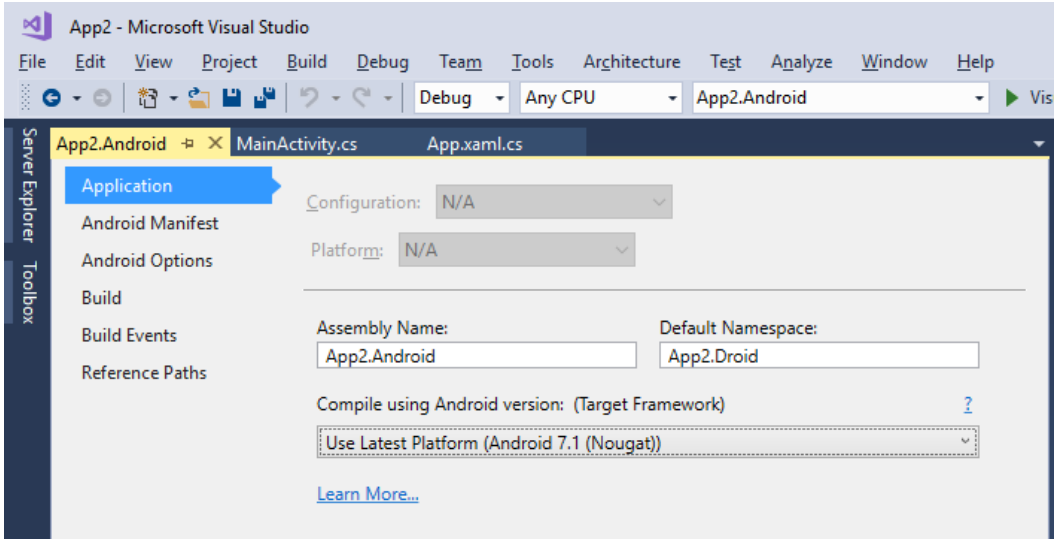

Figure 6: Selecting the version of the Android SDK for compilation

Visual Studio automatically selects the highest version available on your machine. This does not affect the minimum version of Android you want to target; instead it is related to the version of the build tools that Visual Studio will use. My recommendation is to leave the default selection unchanged.

> ***Tip: You can manage installed SDK versions using the Android SDK Manager, a tool that you can launch from both the Windows Programs menu and from Visual Studio by selecting*** *Tools, Android, Android SDK Manager.*

The **Android Manifest** tab is even more important. Here you specify your app's metadata, such as name, version number, icon, and permissions the user must grant to the application. Figure 7 shows an example.

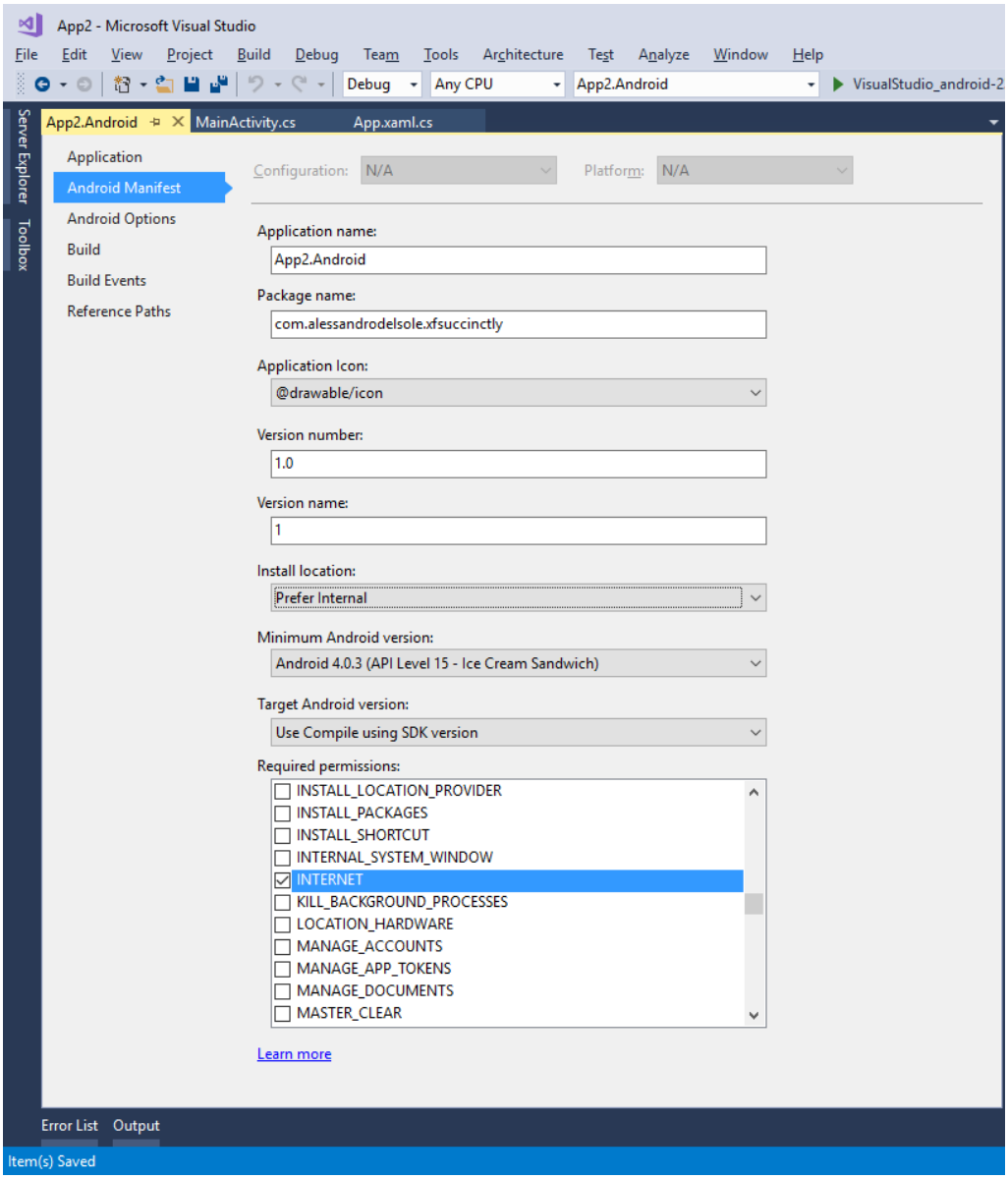

Figure 7: The Android manifest

The information you supply in the Android manifest is also important for publication to Google Play. For example, the package name uniquely identifies your app package in the Google Play store and, by convention, it is in the following form: com.companyname.appname, which is self-explanatory (com. is a conventional prefix). The version name is your app version, whereas version number is a single-digit string that represents updates. For instance, you might have version name 1.0 and version number 1, version name 1.1 and version number 2, version name 1.2 and version number 3, and so on.

The **Install location** option allows you to specify whether your app should be installed only in the internal storage or if memory cards are allowed, but remember that, starting from Android 6.0, apps cannot be installed onto a removable storage any longer. In the **Minimum Android version** drop-down, you can select the minimum Android version you want to target.

It is important you pay particular attention to the **Required permissions** list. Here you must specify all the permissions that your app must be granted in order to access resources such as the Internet, the camera, other hardware devices, sensors, and more. Remember that, starting from Android 6.0, the operating system will ask the user for confirmation before accessing a resource that requires one of the permissions you marked in the manifest, and the app will crash if it attempts to access a sensitive resource, but the related permission was not selected in the manifest.

In the **Android Options** tab, you will be able to manage debugging and build options. However, I will not walk through all the available options here. It is worth highlighting the **Use Fast Deployment** option though, which is enabled by default. When enabled, deploying the app to a physical or emulated device will only replace changed files. This can often cause the app not to work properly or not to start at all, so my suggestion is you disable this option. The other tabs are the same as for other .NET projects.

The Xamarin.iOS project

Similar to the Xamarin.Android project, the Xamarin.iOS project makes it possible for your Xamarin.Forms solutions to run on the iPhone and iPad. Supposing you have a configured Mac, Visual Studio will need to know its address in the network. Visual Studio normally asks this after creating a new or opening an existing Xamarin.Forms solution, but you can manually enter the Mac address with **Tools**, **iOS**, **Xamarin Mac Agent**. In the **Xamarin Mac Agent** dialog, Visual Studio should be able to list any detected Mac computers in the network. However, it is strongly recommended that you re-add a Mac by providing its IP address rather than its name. For example, Figure 8 shows the Xamarin Mac Agent dialog displaying my Mac Mini machine with both its name and its IP address, but Visual Studio established a connection based on the IP, not the name.

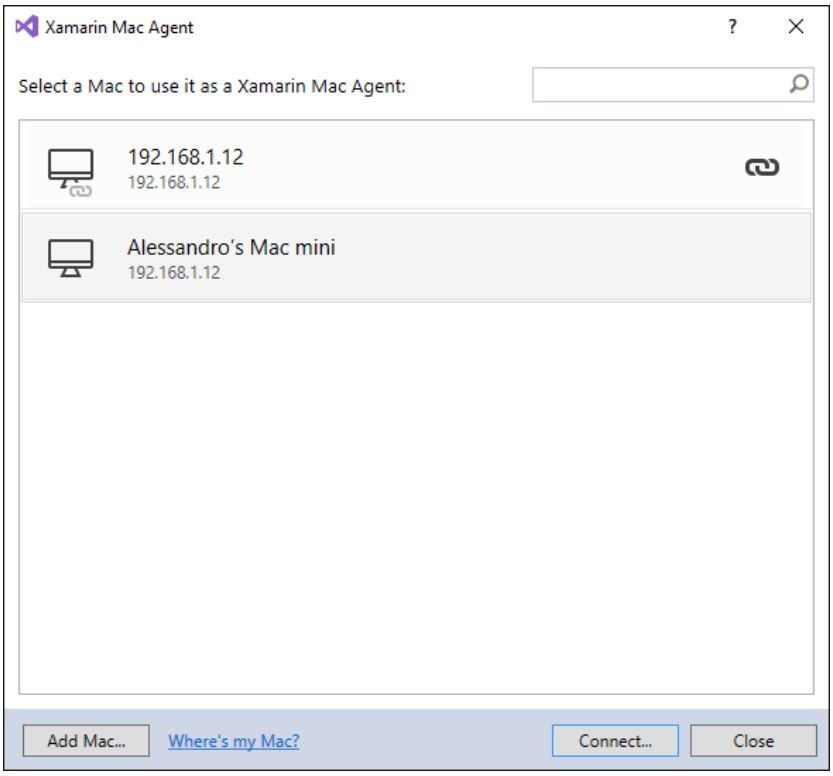

Figure 8: Connecting Visual Studio to a Mac

If a Mac is not detected, click **Add Mac** and enter its IP address first. Then, when requested, enter the same credentials you use to log in to the Mac. If the connection succeeds, Visual Studio will show a success message in the status bar. For the Xamarin.iOS project, the **AppDelegate.cs** file contains the Xamarin.Forms initialization code and should not be changed. You can add all the code that requires accessing native APIs and platform-specific features in this project, as you will learn in Chapter 8. In the **Info.plist** file (see Figure 9), with each tab, you can configure your app metadata, the minimum target version, supported devices and orientations, capabilities (such as Game Center and Maps integration), visual assets such as launch images and icons, and other advanced features.

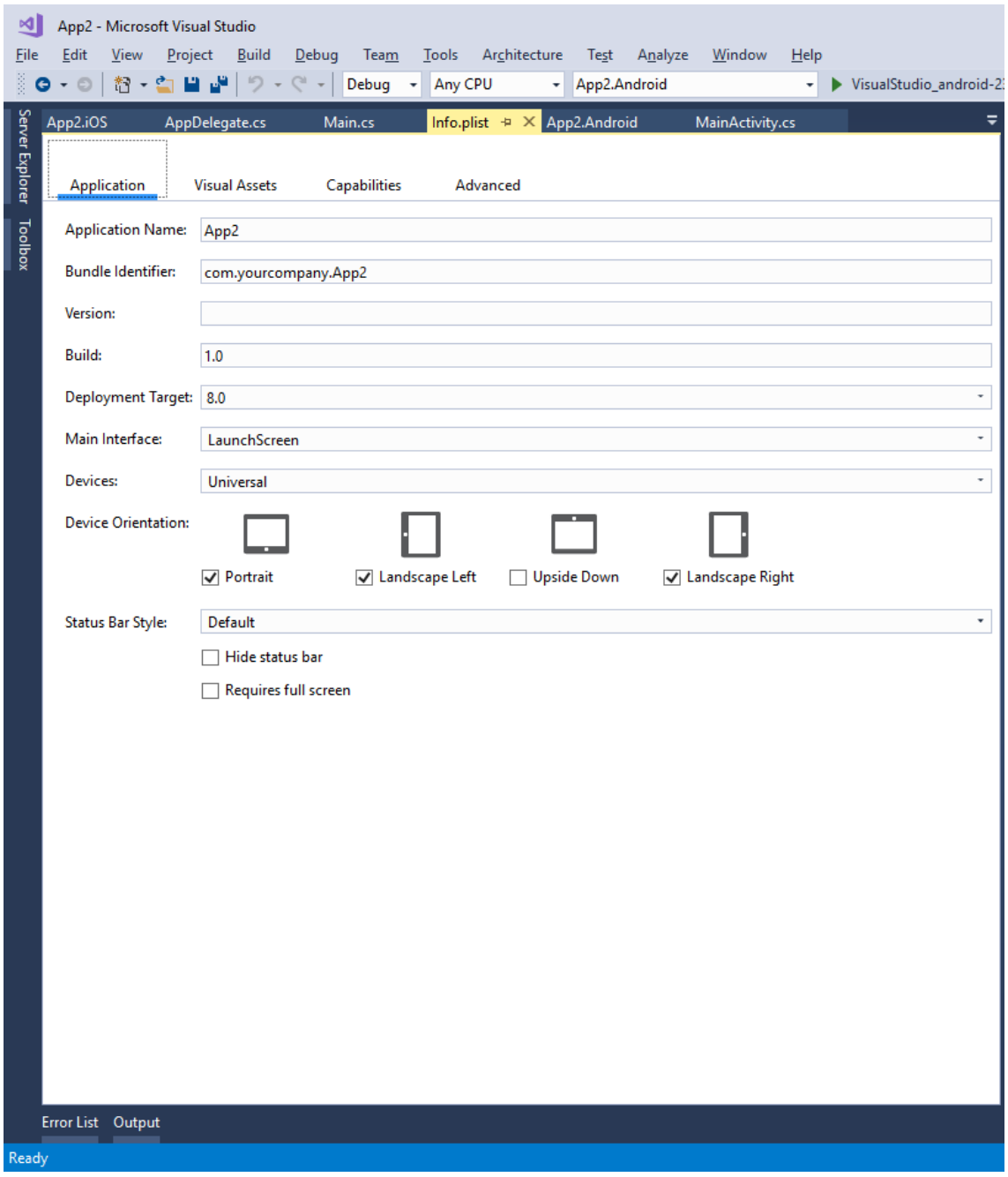

Figure 9: The Info.plist file

The **Info.plist** file represents the app manifest in iOS and therefore is not related to only Xamarin.iOS. In fact, if you have experience with Xcode and native iOS development, you already know this file. Unlike Android, the iOS operating system includes restriction policies that are automatically applied to most sensitive resources, especially those involving security and privacy. Also, there are differences among iOS 8.x, 9.x, and 10.x in how the OS handles these options. The Info.plist reference will help you understand how to properly configure any exceptions. Among the project properties, the most important is, without a doubt, the iOS Bundle Signing. You use the Bundle Signing to specify the identity that the Apple tools must use to sign the app package, and specify the provisioning profile that is used to associate a team of

developers to an app identifier. Configuring signing identities and profiles is particularly important when preparing an app for publishing. Figure 10 shows the iOS Bundle Signing properties.

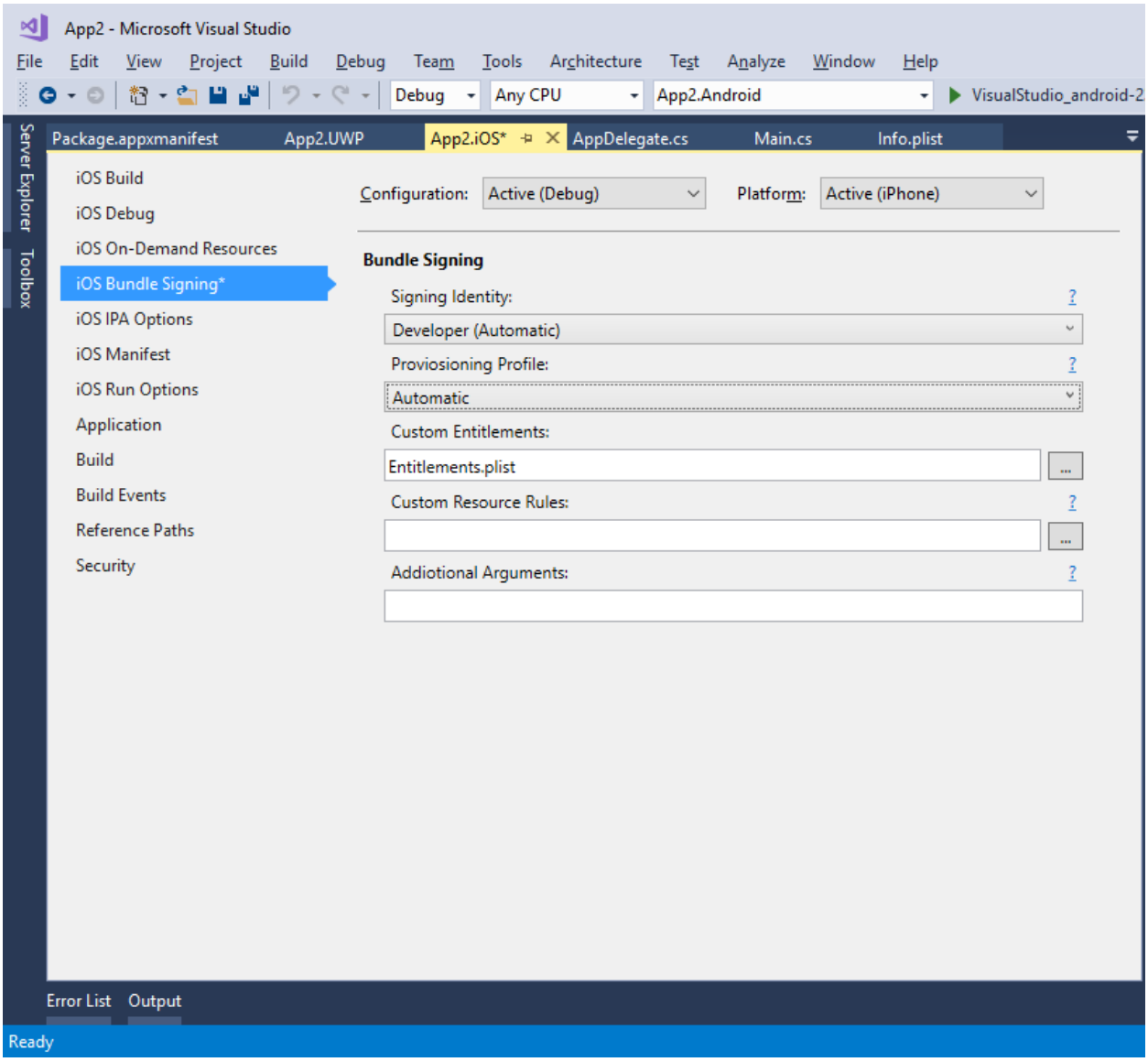

Figure 10: The iOS Bundle Signing options

As you can imagine, Visual Studio can detect available signing identities and provisioning profiles only when connected to a Mac, because this information is generated via Xcode. Further details about configuring a Xamarin.iOS project are offered through the documentation.

The Universal Windows Platform project

The Universal Windows Platform project in a Xamarin.Forms solution is nothing but a normal UWP project with a reference to the shared code and to the Xamarin.Forms package. In the **App.xaml.cs** file, you can see initialization code that you must not change. What you will need to configure, instead, is the application manifest, which you can edit by double-clicking the **Package.appxmanifest** file in Solution Explorer. Visual Studio has a nice editor for UWP manifests, and you will at least configure app metadata (see Figure 11), visual assets such as icons and logos, and capabilities (see Figure 12). These include permissions you need to specify before testing and distributing your apps, and Windows 10 will ask the user for confirmation before accessing resources that require a permission.

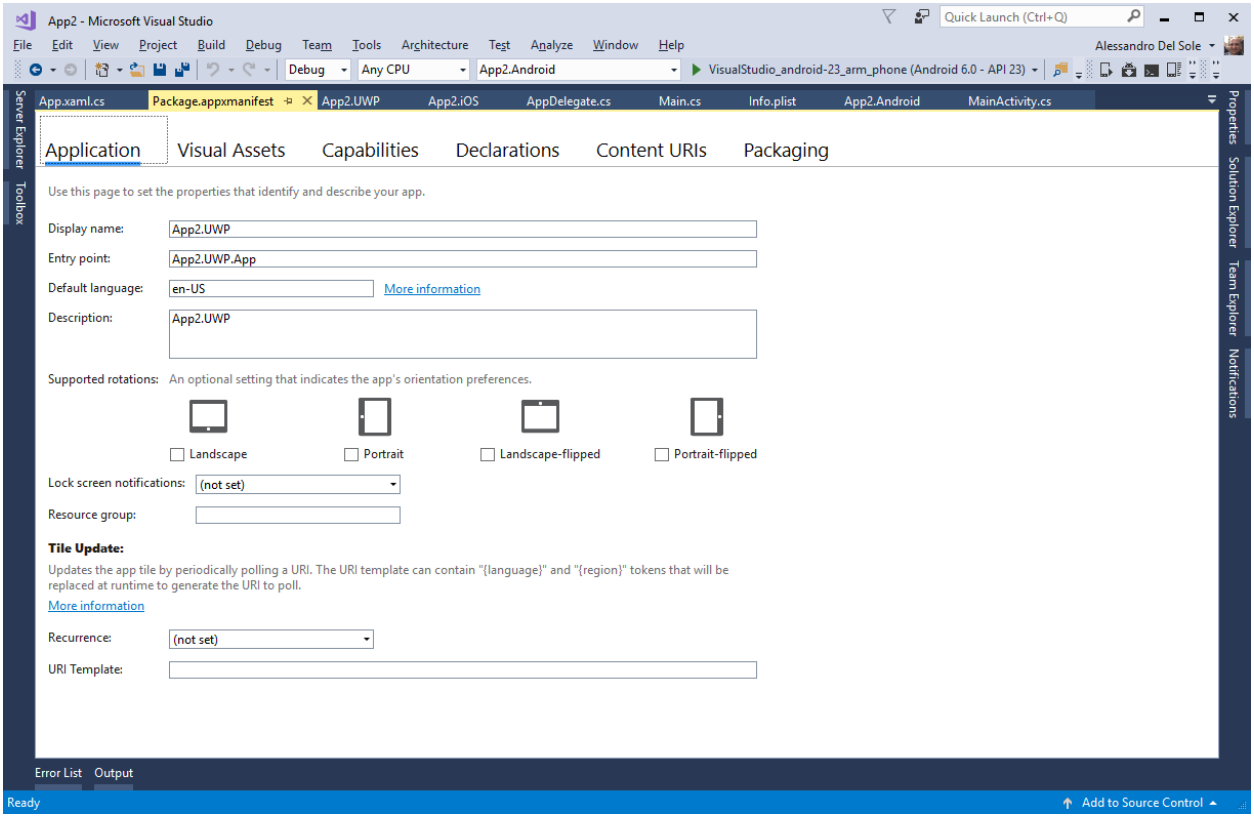

Figure 11: Editing metadata in UWP projects

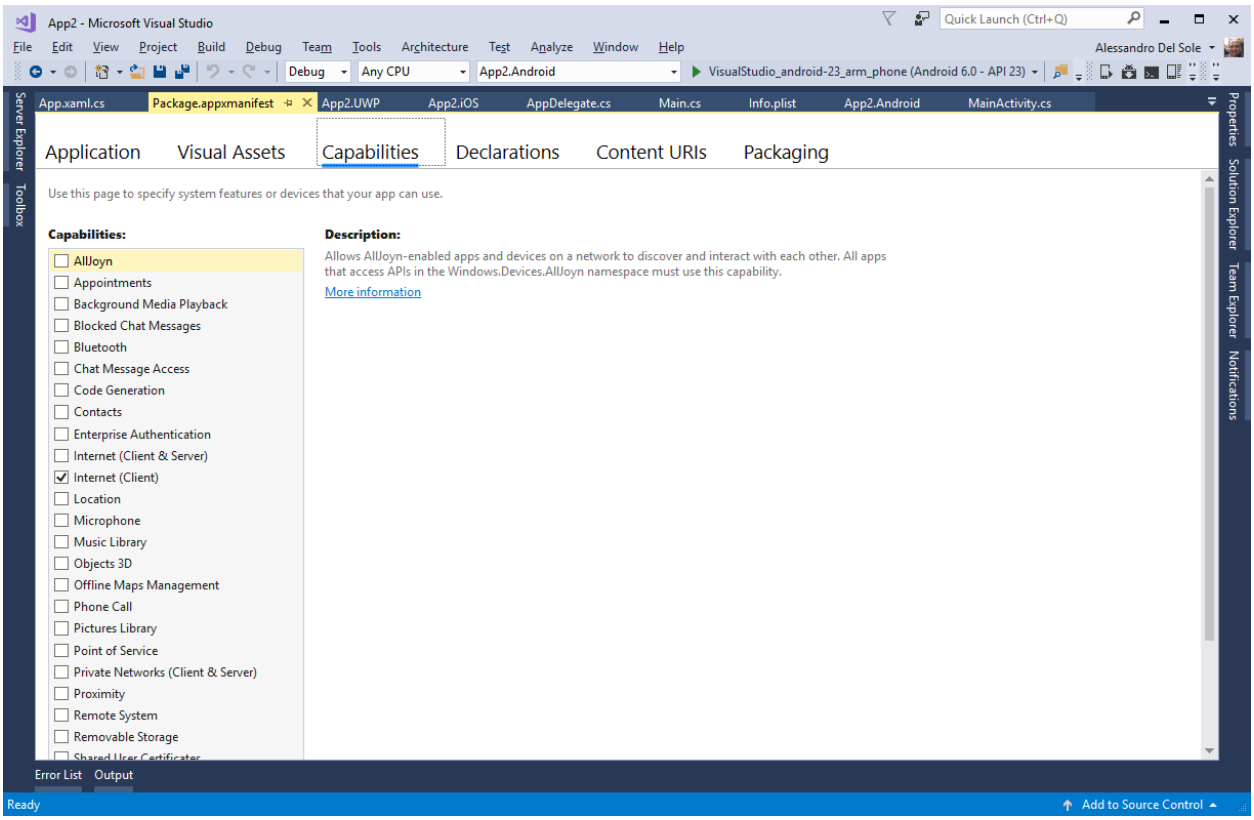

Figure 12: Specifying capabilities in UWP projects

The official documentation explains how to configure other options. However, remember that you need a paid subscription to the Windows Store in order to fill in the Packaging settings that you provide when preparing for publishing.

Debugging and testing applications locally

Starting apps you build with Xamarin.Forms for debugging and testing could not be easier: you simply select one of the platform projects in Solution Explorer as the startup project, then you select the target device and press F5. Do not forget to rebuild your solution before debugging for the first time. When you start an app for debugging, Visual Studio will build your solution and deploy the app package to the selected physical device or emulator. The result of the build process is an .apk file for Android, an .ipa file for iOS, and an .appx file for Windows 10. When the app starts either on a physical device or on an emulator, Visual Studio attaches an instance of the debugger and you will be able to use all the well-known, powerful debugging tools of the IDE, including (but not limited to) breakpoints, data tips, tool windows, watch windows, and more. The easiest way to select the target platform and configuration is by using the standard toolbar, which you can see in Figure 13.

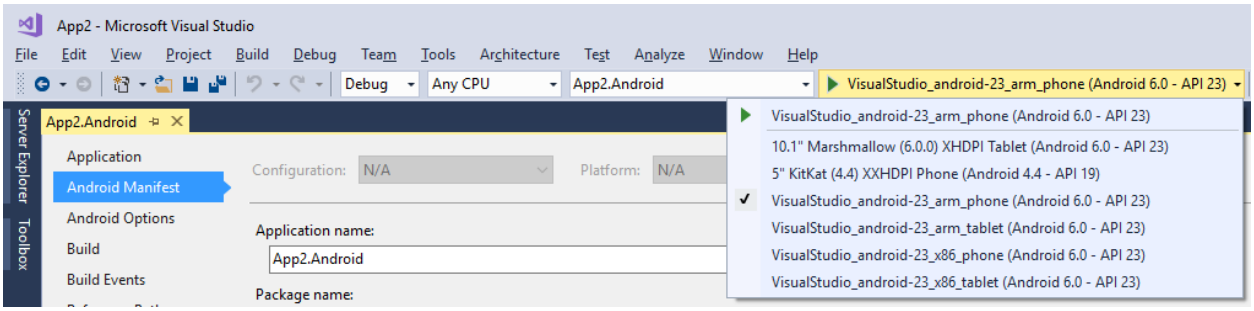

Figure 13: Selecting target platform and devices for debugging

The Debug configuration is the appropriate choice during development. You will select Ad-Hoc when preparing for distribution on iOS and Android, or Release for Windows 10. The target architecture is normally Any CPU for Android and Windows, whereas it is iPhoneSimulator for debugging an iOS app in the iOS Simulator or iPhone for debugging an app on a physical iPhone or iPad (remember that a physical Apple device must be associated to your Mac via Xcode and connected to the Mac, not the PC). You can also quickly select the startup project and specify the target device. For example, in Figure 13 you can see a list of Android emulator configurations.

Note: In this e-book, I will provide figures that show all the supported platforms in action when it is relevant to do so. In other cases, I will show just one platform in action, meaning that the same behavior is expected on all platforms.

Figure 14 shows the previously created blank app running on all three platforms within the respective emulators. Notice how in the background Visual Studio shows the **Output** window where you receive messages from the debugger; you will be able to use all the other debugging tools similarly.

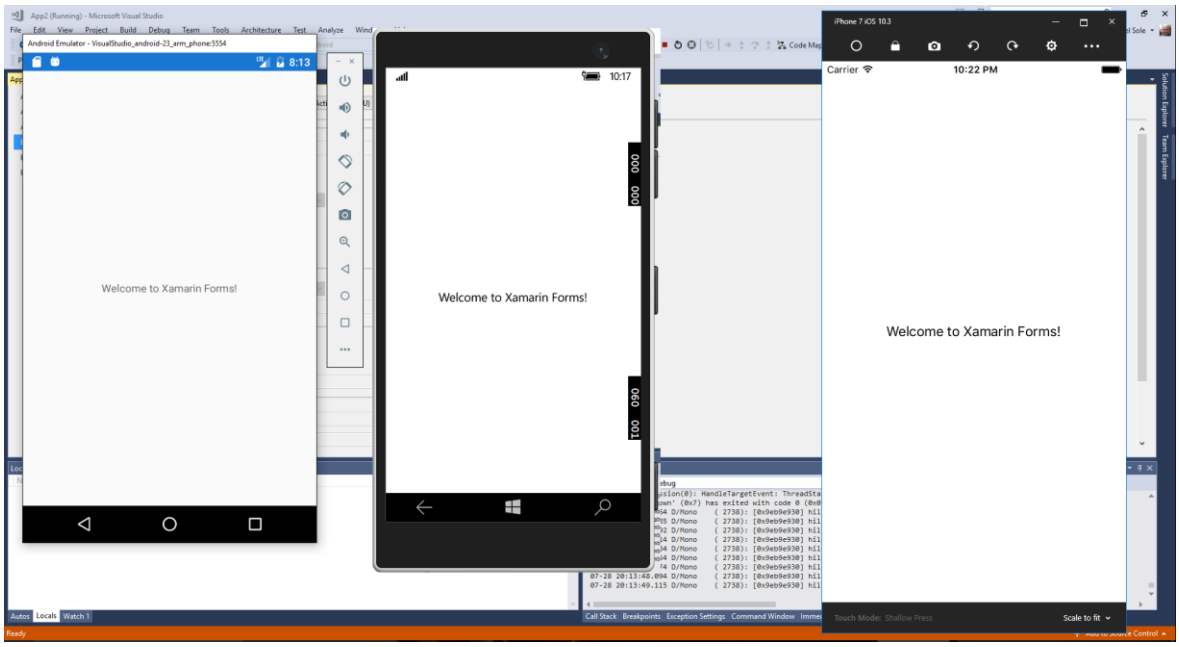

Figure 14: An app built with Xamarin.Forms running on all platforms

Note: Though Visual Studio 2017 allows you to run multiple instances of an app on multiple platforms, Figure 14 has been captured with the Android version running, with the Windows emulator and iOS simulator screenshots added separately.

As you can imagine, in order to build the iOS app, Visual Studio connected to the Mac and launched the Apple SDKs. In Figure 14, you see an example of the iOS Simulator, which deserves some more consideration.

Setting up the iOS Simulator

Unlike Android and Windows, with which emulators run locally on your Windows development machine, the iOS Simulator runs on your Mac. However, if you have Visual Studio 2017 Enterprise, you can also download and install the Remoted iOS Simulator for Windows. When this is installed, the simulator will run on your Windows machine instead of running on the Mac. The iOS Simulator on Windows is not enabled by default, so you need to open **Tools**, **Options**, **Xamarin**, **iOS Settings**, and select the **Remote Simulator to Windows** option. An example of the iOS Simulator is available in Figure 14, but you will see others in the next chapters. If you have Visual Studio 2017 Community or Professional, then you need to use the simulator on the Mac.

Running apps on physical devices

Visual Studio can easily deploy an app package to physical Windows and Android devices. For Android, you first need to enable the developer mode, which you accomplish with the following steps:

1. Open the **Settings** app.
2. Tap the **About** item.
3. In the list that appears, locate the OS build number and tap this item seven times.

At this point, you can simply plug your device into the USB port of your PC and Visual Studio will immediately recognize it. It will be visible in the list of available devices that you can see in Figure 13. For Windows, you first need to enable both your machine and your devices for development, and the official documentation provides guidance about this. Then you will be able to plug your devices into the USB port of your PC and Visual Studio will recognize them as available target devices. For Apple mobile devices, you need to connect your iPhone or iPad to your Mac computer, making sure you make them discoverable through Xcode. Then, when you start debugging from Visual Studio, your app will be deployed to the iPhone or iPad through the Mac.

The Xamarin Live Player app

Microsoft has recently published a preview of the Xamarin Live Player app for Android and iOS. The goal of this app is to make it simpler to debug and test Android and iOS applications on a physical device connected to the same network as your development machine and avoid the need to have a Mac for compiling and debugging an app on iOS. Currently, this app works only with Visual Studio 2017 Preview version 15.3. Basically, you will need to pair the Xamarin Live Player app with Visual Studio via a bar code, and then Visual Studio will be able to deploy your app packages to the physical device via the network. Because the app is in preview and requires a prerelease version of Visual Studio 2017, it will not be discussed in this chapter. Just keep in mind this is a nice alternative if you do not have a Mac yet. However, remember that there are a number of important limitations and that you still need a Mac computer to sign and distribute iOS apps.

Analyzing and profiling applications

The Xamarin toolbox has been recently enriched with three amazing tools: Xamarin Workbooks, Xamarin Inspector, and Xamarin Profiler. The first tool allows you to explore a number of .NET and Mono development platforms with interactive examples. The Xamarin Inspector allows you to inspect the visual tree of your Xamarin apps nicely and make changes to the UI in real time (see the documentation).

The Xamarin Profiler is a complete suite of analysis tools in one program that you can use to profile your mobile apps and analyze performance, memory usage, CPU consumption, and more. In Visual Studio, you can select **Tools**, **Xamarin Profiler**, and launch any platform version of your app for profiling, instead of pressing F5. After the app has been deployed to the selected device and before starting up, the Profiler will ask you what kind of analysis you want to execute against the application. Figure 15 shows an example based on the selection of all the available instrumenting tools.

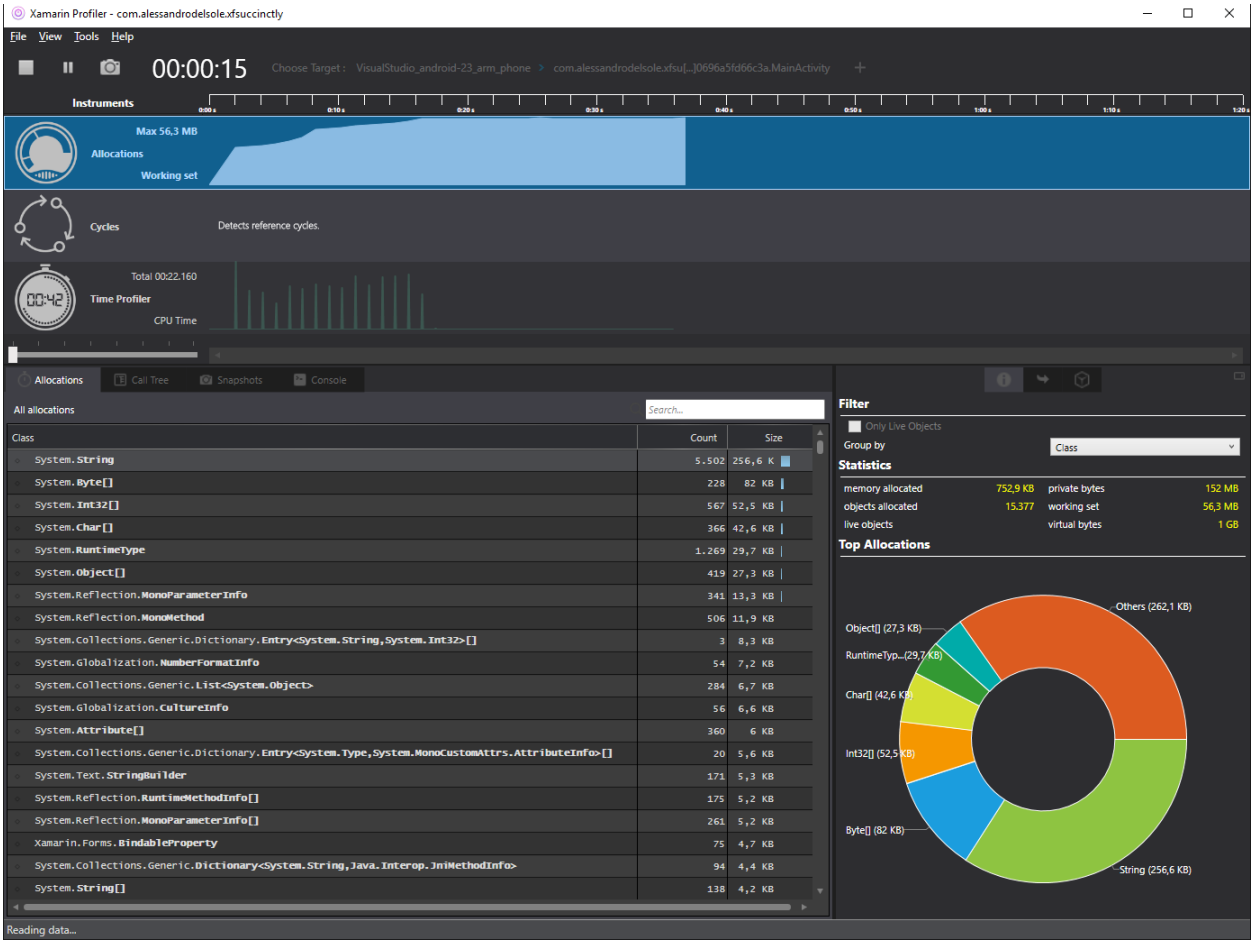

Figure 15: Analyzing app performance with Xamarin Profiler

You can also take and compare snapshots of the memory in different moments to see if memory allocation can cause potential problems. This is an excellent performance analysis tool and the documentation will provide all the details you need to improve your app performance.

Chapter summary

This chapter introduced the Xamarin.Forms platform and its goals, describing the required development tools and offering an overview of a Xamarin.Forms solution, passing through the platform projects and their most important settings. You have also seen how to start an app for debugging using emulators and how to take advantage of the new Xamarin Live Player application that allows you to debug an iOS application without having a Mac (which is still required for serious development). Now that you have an overview of Xamarin.Forms and the development tools, the next step is understanding what is at its core: sharing code across platforms.

Chapter 2 Sharing Code Among Platforms

Xamarin.Forms allows you to build apps that run on Android, iOS, and Windows from a single C# codebase. This is possible because, at its core, Xamarin.Forms allows the sharing among platforms of all the code for the user interface and all the code that is not platform-specific. There are different ways to share code among platforms, each with its pros and cons. This chapter explains the available code-sharing strategies in Xamarin.Forms, highlighting their characteristics so that it will be easier for you to decide which strategy is better for your solutions.

Introduction to code-sharing strategies

In Chapter 1, I explained how to create a Xamarin.Forms solution in Visual Studio 2017 and that it is made up of four projects: three platform projects (Android, iOS, and UWP) and a project that allows for sharing code among platforms. With this approach, developers can share all the code for the user interface and all the code that is not coupled to the APIs of each platform, maximizing code reuse and simplifying the process of creating three different native apps from a single C# codebase. In that explanation, I briefly introduced the Portable Class Library (PCL) as a project type that allows sharing code. However, Xamarin.Forms allows sharing code among platforms in three different ways: Portable Class Libraries, Shared Projects, and .NET Standard libraries. This chapter contains a thorough discussion of these three code-sharing strategies, providing further information about the PCL project type. It is worth mentioning that, at the time of writing, Visual Studio 2017 includes project templates based on PCLs and Shared Projects, while you need some manual, yet easy steps for .NET Standard. In the section about .NET Standard, you will learn how to convert a PCL into a .NET Standard library easily.

Sharing code with Portable Class Libraries

As the name implies, Portable Class Libraries (PCL) are libraries that can be consumed on multiple platforms. More specifically, they can be consumed on multiple platforms only if they target a subset of APIs available on all those platforms. PCLs have existed for many years, and they are certainly not exclusive to Xamarin. In fact, they can be used in many other development scenarios. For example, a PCL could be used to share a Model-View-ViewModel architecture between a WPF project and a UWP project. The most important characteristics of a PCL are the following:

- They produce a compiled, reusable .dll assembly.
- They can reference other libraries and have dependencies such as NuGet packages.
- They can contain XAML files for the user interface definition and C# files.
- They cannot expose code that leverages specific APIs of a certain platform, otherwise they would no longer be portable.
- They are a better choice when you need to implement architectural patterns such as MVVM, factory, inversion of control (IoC) with dependency injection, and service locator.

- With regard to Xamarin.Forms, they can use the service locator pattern to implement an abstraction and to invoke platform-specific APIs through platform projects (this will be discussed in Chapter 8).
- They are easier to unit test.

For example, a PCL that is used to share code between WPF and UWP projects could never contain code that accesses the location sensor of a device, because this is not supported in WPF and requires Windows 10 APIs that WPF cannot access. Instead, a PCL can be used to access Internet resources via the **`HttpClient`** class on multiple platforms, because this is commonly available. Normally, you would create a PCL project manually and then add the necessary references to and from other projects in the solution. In the case of Xamarin.Forms, you instead decide a code-sharing strategy when creating a new project (see Figure 4) and then Visual Studio 2017 will automatically generate a PCL project that is referenced by the platform projects in the solution and that has a dependency on the Xamarin.Forms NuGet package.

Note: In all the examples in this e-book, I will use the PCL as the code-sharing strategy for the following reasons: it makes it easier to use and manage other libraries, it is a better choice with real-world projects that require more complex architectures, and it is simpler to change into a .NET Standard library.

Sharing code with shared projects

Shared projects, as well as PCLs, are not specific to Xamarin and have existed for many years. Shared projects are essentially loose assortments of files that can be shared with other projects. Following is a list of the most important characteristics of a shared project, also highlighting the differences from a PCL:

- They do not produce a compiled, reusable .dll assembly.
- They cannot reference other libraries and have dependencies such as NuGet packages.
- They can contain XAML files for the user interface definition and C# files.
- They can contain platform-specific code that can use conditional compilation and preprocessor directives.

In order to select a shared project as the code-sharing strategy, in the **New Cross Platform App** dialog (see Figure 4) you select **Shared Project** in the **Code Sharing Strategy** group. When the solution is ready in Solution Explorer, you will see the shared project looking similar to Figure 16.

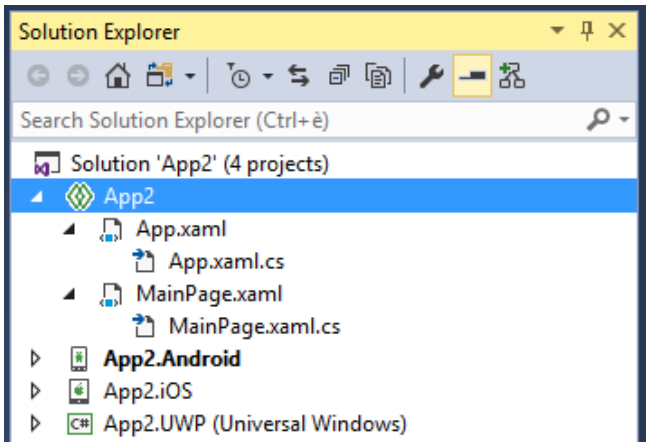

Figure 16: The shared project in a Xamarin.Forms solution

One important note here is that platform projects (Android, iOS, and UWP) have a reference to the shared project, but the shared project cannot have references or dependencies. Additionally, shared projects' properties have no project-level properties; instead, you can only access the properties of the individual files they contain. Not surprisingly, there is no Properties or References node for shared projects in Solution Explorer. Shared projects can contain an infinite number of different files and resources, including XAML files for the user interface and C# code files. This is possible because shared projects are not compiled, rather the compiler resolves source files and resources when the whole solution is built. Because shared projects do not reference any libraries, it might be difficult to remember types and members you can use in C#. Luckily, IntelliSense comes in to help, showing what is available (and what is not) when you type, as shown in Figure 17.

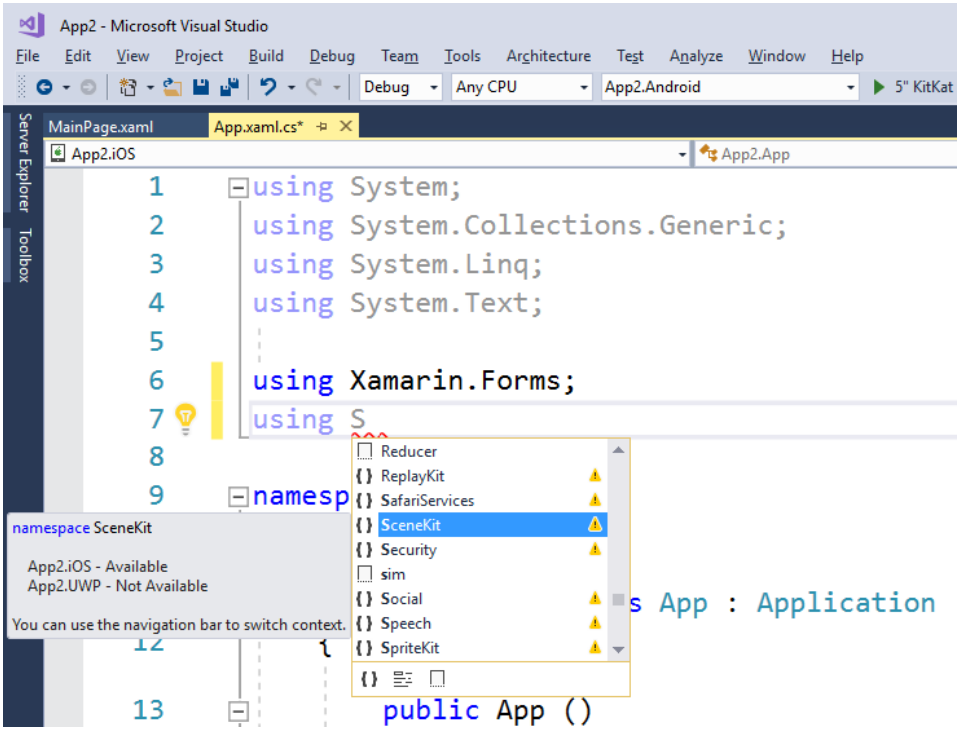

Figure 17: IntelliSense shows available types

The biggest benefit of shared projects is that they allow the writing of platform-specific code without needing to use patterns such as the service locator, as you do with PCLs. This is accomplished using preprocessor directives such as **#if**, **#elif**, **#else** and conditional compilation symbols as demonstrated in Code Listing 1.

Code Listing 1

```
private string GetFolderPath()
{
    string path = "";

    #if __ANDROID__
        path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocume
nts);
    #elif __IOS__
        path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocume
nts);
    #elif WINDOWS_PHONE
        path = Windows.Storage.KnownFolders.DocumentsLibrary.Path;
    #endif
    return path;
}
```

As you can see, you can simply check the platform your app is running on with preprocessor directives and then the compiler will resolve the appropriate platform-specific code without dealing with the complexity of other patterns. Each platform is represented by a conditional compilation symbol, defined in the build options of the project properties. Table 1 summarizes available symbols in Visual Studio 2017 and the platforms they represent.

Table 1: Conditional compilation symbols in Xamarin.Forms

Symbol	Description
__ANDROID__	Represents the Android platform.
__IOS__	Represents the iOS platform.
WINDOWS_PHONE	Represents the Universal Windows Platform, Windows 8.1, and Windows Phone 8.1 platforms.
__TVOS__	Represents the Apple tvOS platform.
__WATCHOS__	Represents the Apple Watch OS platform.
NETCORE_FX	Represents the .NET Core platform.

In versions of Visual Studio that still support Windows Phone 8 projects, such as Visual Studio 2013 and 2015, an additional symbol called **__SILVERLIGHT__** is available, but it is not targeted in Visual Studio 2017. An interesting example of platform-specific code that uses conditional compilation symbols and preprocessor directives is the SQLite.cs file, which implements data access against the popular SQLite database in C#. A complete sample solution based on shared projects and the approach described previously is available from the official Xamarin documentation and is called Tasky. It shows how to create a simple To-Do mobile application. Having the option to write platform-specific code with the approach you saw in Code Listing 1 is certainly appealing, but you should prefer PCLs (and .NET Standard libraries) in at least the following situations:

- You need to access many platform-specific resources and your code might become very difficult to maintain.
- You need to implement architectures based on one or more patterns. In such situations, not only are shared projects not the best option, but it is common to have multiple portable libraries, which also makes it easier for teams to work on different parts of a solution.
- You want to unit test your code efficiently.

The aforementioned points, together with the considerations I made in the section about portable libraries, should further clarify the reason you will find examples based on PCLs rather than shared projects in this e-book.

Sharing code with .NET Standard libraries

The .NET Standard provides a set of formal specifications for APIs that all the .NET development platforms, such as .NET Framework, .NET Core, and Mono, must implement. This allows for unifying .NET platforms and avoids future fragmentation. By creating a .NET Standard library, you will ensure your code will run on any .NET platform without the need to select any targets. This also solves a common problem with portable libraries, since every portable library can target a different set of platforms, which implies potential incompatibility between libraries and projects. Microsoft has an interesting blog post about .NET Standard, its goals and implementations, that will clarify any doubts about this specification.

At the time of writing, version 1.7 of .NET Standard is available and full unification for .NET Framework, .NET Core, and Mono is expected for version 2.0 (currently available in preview). For now, Visual Studio 2017 supports versions up to 1.7. The documentation will help you choose the version of .NET Standard based on the minimum version of the platform your application is intended to run on. With Xamarin.Forms in Visual Studio 2017, version 1.3 is the choice I recommend. Visual Studio 2017 does not include .NET Standard as a code-sharing strategy when creating a new Xamarin.Forms solution, and this makes sense because, as you'll discover shortly, at the moment, only prerelease versions of Xamarin.Forms support .NET Standard. Once stable releases of Xamarin.Forms support .NET Standard, it is reasonable to expect that this code-sharing strategy will be included as a possible option. What you can do now is convert a Portable Class Library project into a .NET Standard library. To accomplish this, you need to follow these steps:

1. In Solution Explorer, right-click the PCL project name and select **Manage NuGet Packages**.

2. In the NuGet package manager interface, you will see the Xamarin.Forms package installed in the current project. You have to uninstall this package at this point, because otherwise Visual Studio will not be able to change the PCL to a .NET Standard library.
3. Open the PCL project properties and, in the **Library** tab, click the **Target .NET Platform Standard** hyperlink at the bottom of the list of currently targeted platforms. A confirmation message will appear. Click **Yes**.
4. As you can see in Figure 18, the project now targets .NET Standard version 1.0. Change the version to 1.3 and click **Yes** when a message appears saying that the project must be reloaded.
5. In Solution Explorer, right-click the solution name and select **Manage NuGet Packages**. In the NuGet interface, select **Browse** and search for the Xamarin.Forms NuGet package. Make sure the **Include prerelease** flag is selected.
6. Click the **Consolidate** tab, select all the projects in the list on the right side, then select the highest prerelease version possible of the NuGet package. Finally, click **Install** (see Figure 19).

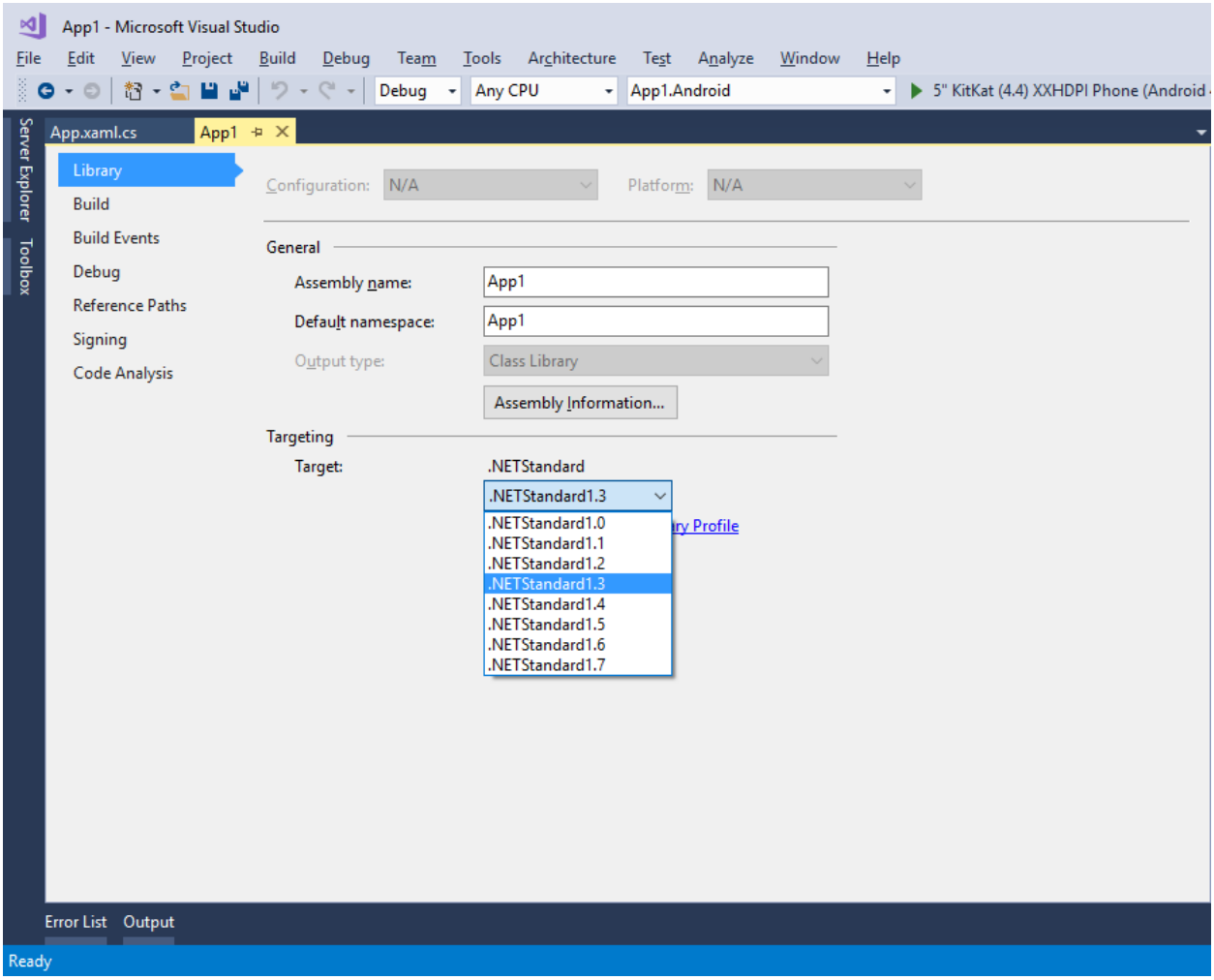

Figure 18: Selecting a different version of .NET Standard

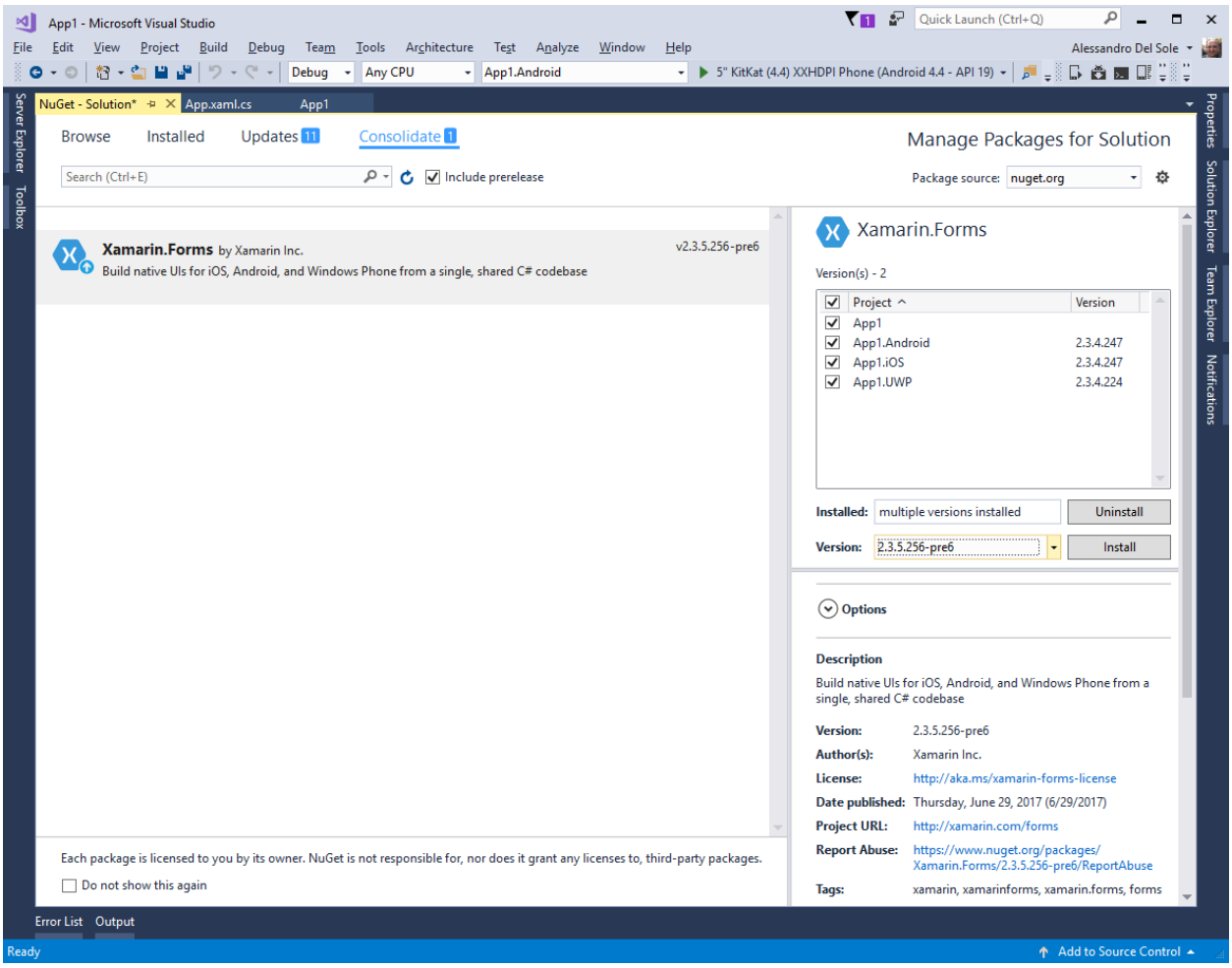

Figure 19: Reinstalling an updated version of Xamarin.Forms

Installing the latest prerelease of Xamarin.Forms is necessary because not all versions of this package support .NET Standard, and some of the most recent versions only support the lowest versions of .NET Standard. So, if you want to target .NET Standard 1.3, the minimum version of Xamarin.Forms you need to install is 2.3.5.256-pre6. Of course, future stable releases will be compatible with .NET Standard 1.3 and higher versions.

At this point, rebuild your solution. Now you have a .NET Standard library that can contain code that will certainly run on all the platforms that implement the specification, which include Mono, UWP, Xamarin.iOS, Xamarin.Android, and Xamarin.Mac.

Note: Even if .NET Standard is not yet included as a code-sharing strategy when creating a Xamarin.Forms solution, there is no doubt that this will be the code-sharing strategy of choice in the near future. For this reason, I recommend you have a look at the documentation and take some time to explore the API specifications.

Chapter summary

This chapter introduced the available code-sharing strategies that Xamarin.Forms can use to share user interface files and platform-independent code, such as Portable Class Libraries, shared projects, and .NET Standard libraries. PCLs produce reusable assemblies, allow for implementing better architectures, and cannot contain platform-specific code. Shared projects can contain platform-specific code with preprocessor directives and conditional compilation symbols, but they do not produce reusable assemblies, and code maintenance is more difficult if they access many native resources. .NET Standard libraries represent the future of code sharing across platforms, are based on a formal set of API specifications, and they make sure your code will run on all the platforms that support the selected version of .NET Standard. Assuming that the Portable Class Library is the preferred choice, in the next chapters you will start writing code and get started with building cross-platform user interfaces.

Chapter 3 Building the User Interface with XAML

Xamarin.Forms is, at its core, a library that allows you to create native user interfaces from a single C# codebase by sharing code. This chapter provides the foundations for building the user interface in a Xamarin.Forms solution. Then, in the next three chapters, you will learn in more detail about layouts, controls, pages, and navigation.

The structure of the user interface in Xamarin.Forms

The biggest benefit of Xamarin.Forms is that you can define the entire user interface of your application inside the project that you selected for sharing code, which can be (at the moment) either a PCL or a shared project. Native apps for iOS, Android, and Windows you get when you build a solution will render the user interface with the proper native layouts and controls on each platform. This is possible because Xamarin.Forms maps native controls into C# classes that are then responsible for rendering the appropriate visual element depending on the platform the app is running on. These classes actually represent visual elements such as pages, layouts, and controls.

Because the PCL or shared project can only contain code that will certainly run on all platforms, Xamarin.Forms maps only those visual elements common to all platforms. For instance, iOS, Android, and Windows all provide text boxes and labels, thus Xamarin.Forms can provide the **`Entry`** and **`Label`** controls that represent text boxes and labels, respectively. However, each platform renders and manages visual elements differently from one another, with different properties and behavior. This implies that controls in Xamarin.Forms expose only properties and events that are common to every platform, such as the font size and the text color.

In Chapter 8, you will learn how to use native controls directly, but for now let's focus on how Xamarin.Forms allows the creation of user interfaces with visual elements provided out of the box. The user interface in iOS, Android, and Windows has a hierarchical structure made of pages that contain layouts that contain controls. Layouts can be considered as containers of controls that allow for dynamically arranging the user interface in different ways. Based on this consideration, Xamarin.Forms provides a number of page types, layouts, and controls that can be rendered on each platform. When you create a Xamarin.Forms solution, whether you choose a PCL or a shared project, the project you selected for sharing code will contain a root page that you can populate with visual elements. Then you can design a more complex user interface by adding other pages and visual elements. To accomplish this, you can use both C# and the extensible Application Markup Language (XAML). Let's discuss both ways further.

Coding the user interface in C#

In Xamarin.Forms, you can create the user interface of an application in C# code. For instance, Code Listing 2 demonstrates how to create a page with a layout that arranges controls in a stack containing a label and a button. For now, do not focus on element names and their properties (they will be explained in the next chapter). Rather, focus on the hierarchy of visual elements that the code introduces.

Code Listing 2

```
var newPage = new ContentPage();
newPage.Title = "New page";

var newLayout = new StackLayout();
newLayout.Orientation = StackOrientation.Vertical;
newLayout.Padding = new Thickness(10);

var newLabel = new Label();
newLabel.Text = "Welcome to Xamarin.Forms!";

var newButton = new Button();
newButton.Text = "Tap here";
newButton.Margin = new Thickness(0, 10, 0, 0);

newLayout.Children.Add(newLabel);
newLayout.Children.Add(newButton);

newPage.Content = newLayout;
```

Here you have full IntelliSense support. However, as you can imagine, creating a complex user interface entirely in C# can be challenging for at least the following reasons:

- Representing a visual hierarchy made of tons of elements in C# code is extremely difficult.
- You must write the code in a way that allows you to distinguish between user interface definition and other imperative code.
- As a consequence, your C# becomes much more complex and difficult to maintain.

In the early days of Xamarin.Forms, defining the user interface could only be done in C# code. Fortunately, you now have a much more versatile way of designing the user interface with XAML, as you'll learn in the next section. Obviously, there are still situations in which you might need to create visual elements in C#; for example, if you need to add new controls at runtime, although this is the only scenario for which I suggest you code visual elements in C#.

The modern way: designing the user interface with XAML

XAML is the acronym for *eXtensible Application Markup Language*. As its name implies, XAML is a markup language that you can use to write the user interface definition in a declarative fashion. XAML is not new in Xamarin.Forms, since it was first introduced more than ten years ago with Windows Presentation Foundation, and it has always been available in platforms such as Silverlight, Windows Phone, and the Universal Windows Platform. XAML derives from XML and, among others, it offers the following benefits:

- XAML makes it easy to represent structures of elements in a hierarchical way, where pages, layouts, and controls are represented with XML elements and properties with XML attributes.
- It provides clean separation between the user interface definition and the C# logic.
- Being a declarative language separated from the logic, it allows professional designers to work on the user interface without interfering with the imperative code.

The way you define the user interface with XAML is unified across platforms, meaning that you design the user interface once and it will run on iOS, Android, and Windows.

Note: XAML in Xamarin.Forms adheres to Microsoft's XAML 2009 specifications, but its vocabulary is different from XAML in other platforms, such as WPF or UWP. So, if you have experience with these platforms, you will notice many differences in how visual elements and their properties are named. Microsoft is working on unifying XAML vocabularies, as you'll learn in the section Hints for XAML Standard. Also, remember that XAML is case-sensitive for object names and their properties and members.

For example, when you create a Xamarin.Forms solution, you can find a file in the PCL project called MainPage.xaml, whose markup is represented in Code Listing 3.

Code Listing 3

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

      <Label Text="Welcome to Xamarin Forms!"
           VerticalOptions="Center"
           HorizontalOptions="Center" />

</ContentPage>
```

A XAML file in Xamarin.Forms normally contains a page or a custom view. The root element is a **ContentPage** object, which represents its C# class counterpart and is rendered as an individual page. In XAML, the **Content** property of a page is implicit, meaning you do not need to write a **ContentPage.Content** element. The compiler assumes that the visual elements you enclose between the **ContentPage** tags are assigned to the **ContentPage.Content** property.

The **Label** element, on the other hand, represents the **Label** class in C#. Properties of this class are assigned with XML attributes, such as **Text**, **VerticalOptions**, and **HorizontalOptions**.

You probably already have the immediate perception of better organization and visual representation of the structure of the user interface. If you look at the root element, you can see a number of attributes whose definition starts with **xmlns**. These are referred to as XML namespaces and are important because they make it possible to declare visual elements defined inside specific namespaces or XML schemas. For example, **xmlns** points to the root XAML namespace defined inside a specific XML schema and allows for adding to the UI definition all the visual elements defined by Xamarin.Forms; **xmlns:x** points to an XML schema that exposes built-in types; and **xmlns:local** points to the app's assembly, making it possible to use objects defined in your project.

Each page or layout can only contain one visual element. In the case of the autogenerated MainPage.xaml page, you cannot add other visual elements to the page unless you organize them into a layout. For instance, if you wanted to add a button below the **Label**, you would need to include both the **Label** and the **Button** inside a container such as the **StackLayout**, as demonstrated in Code Listing 4.

Tip: IntelliSense will help you add visual elements faster by showing element names and properties as you type. You can then simply press Tab or double-click to quickly insert an element. If you have existing experience with Xamarin.Forms in Visual Studio 2015, you will notice a huge number of small but significant IntelliSense improvements for XAML in VS 2017.

Code Listing 4

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="10">
        <Label Text="Welcome to Xamarin Forms!"
       VerticalOptions="Center"
       HorizontalOptions="Center" />

        <Button x:Name="Button1" Text="Tap here!"
            Margin="0,10,0,0"/>
    </StackLayout>

</ContentPage>
```

If you did not include both controls inside the layout, Visual Studio will raise an error. You can nest other layouts inside a parent layout and create complex hierarchies of visual elements. Notice the **`x:Name`** assignment for the **`Button`**. Generally speaking, with **`x:Name`** you can assign an identifier to any visual element so that you can interact with it in C# code, for example, if you need to retrieve a property value.

If you have never seen XAML before, you might wonder how you can interact with visual elements in C# at this point. In Solution Explorer, if you expand the **MainPage.xaml** file, you will see a nested file called **MainPage.xaml.cs**. This is the so-called code-behind file, and it contains all the imperative code for the current page. In this case, the simplest form of a code-behind file, the code contains the definition of the **`MainPage`** class, which inherits from **`ContentPage`**, and the page constructor, which makes an invocation to the **`InitializeComponent`** method of the base class and initializes the page. You will access the code-behind file often from Solution Explorer, but Visual Studio 2017 introduces another easy way that is related to a very common requirement: responding to events raised by the user interface.

Responding to events

Events are fundamental for the interaction between the user and the application, and controls in Xamarin.Forms raise events as normally happens in any platform. Events are handled in the C# code-behind file. Visual Studio 2017 makes it much simpler to create event handlers than its predecessors with an evolved IntelliSense experience. For instance, suppose you want to perform an action when the user taps the button defined in the previous code. The **`Button`** control exposes an event called **`Clicked`** that you assign the name of an event handler as follows:

```
<Button x:Name="Button1" Text="Tap here!" Margin="0,10,0,0"
 Clicked="Button1_Clicked"/>
```

However, when you type **`Clicked="`**, Visual Studio offers a shortcut that allows the generation of an event handler in C# based on the control's name, as shown in Figure 20.

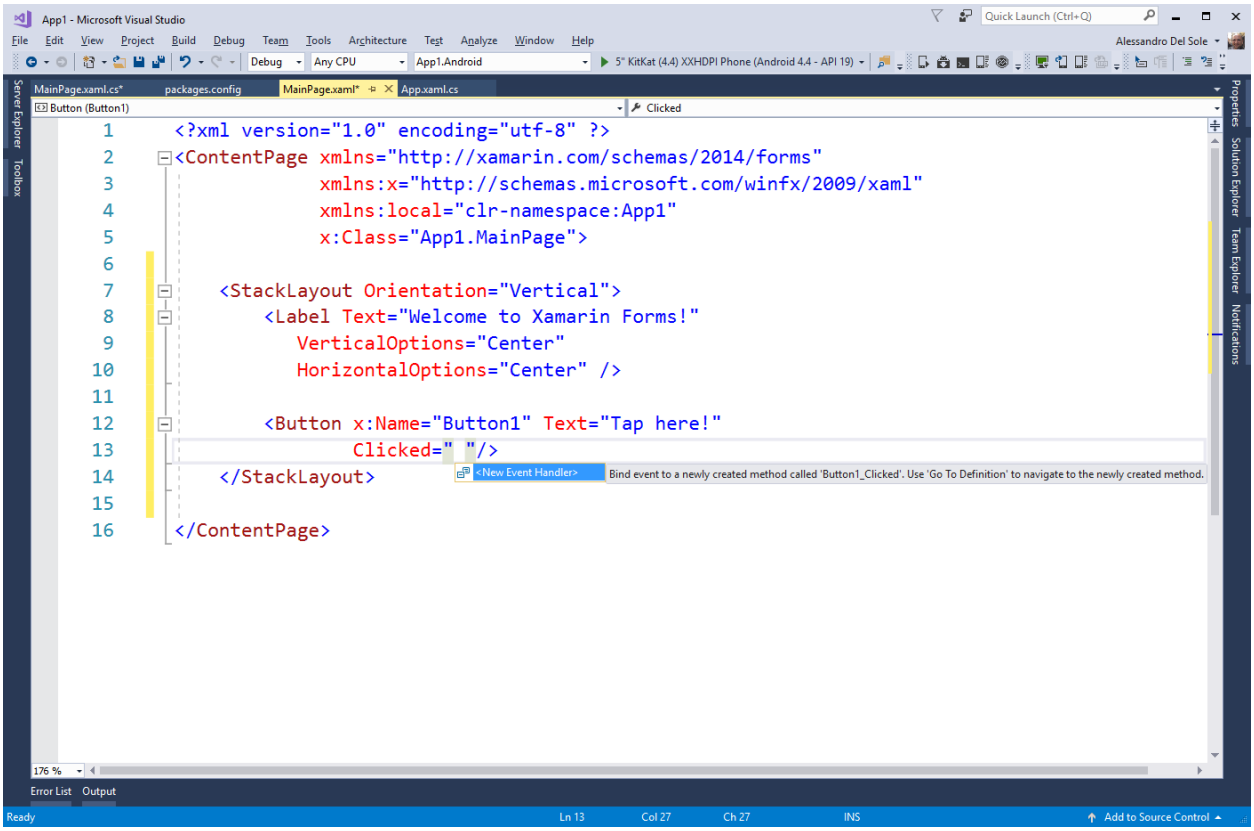

Figure 20: Generating an event handler

If you press **Tab**, Visual Studio will insert the name of the new event handler and will generate the C# event handler in the code-behind. You can quickly go to the event handler by right-clicking its name and then selecting **Go To Definition**. You will be redirected to the event handler definition in the C# code-behind, as shown in Figure 21.

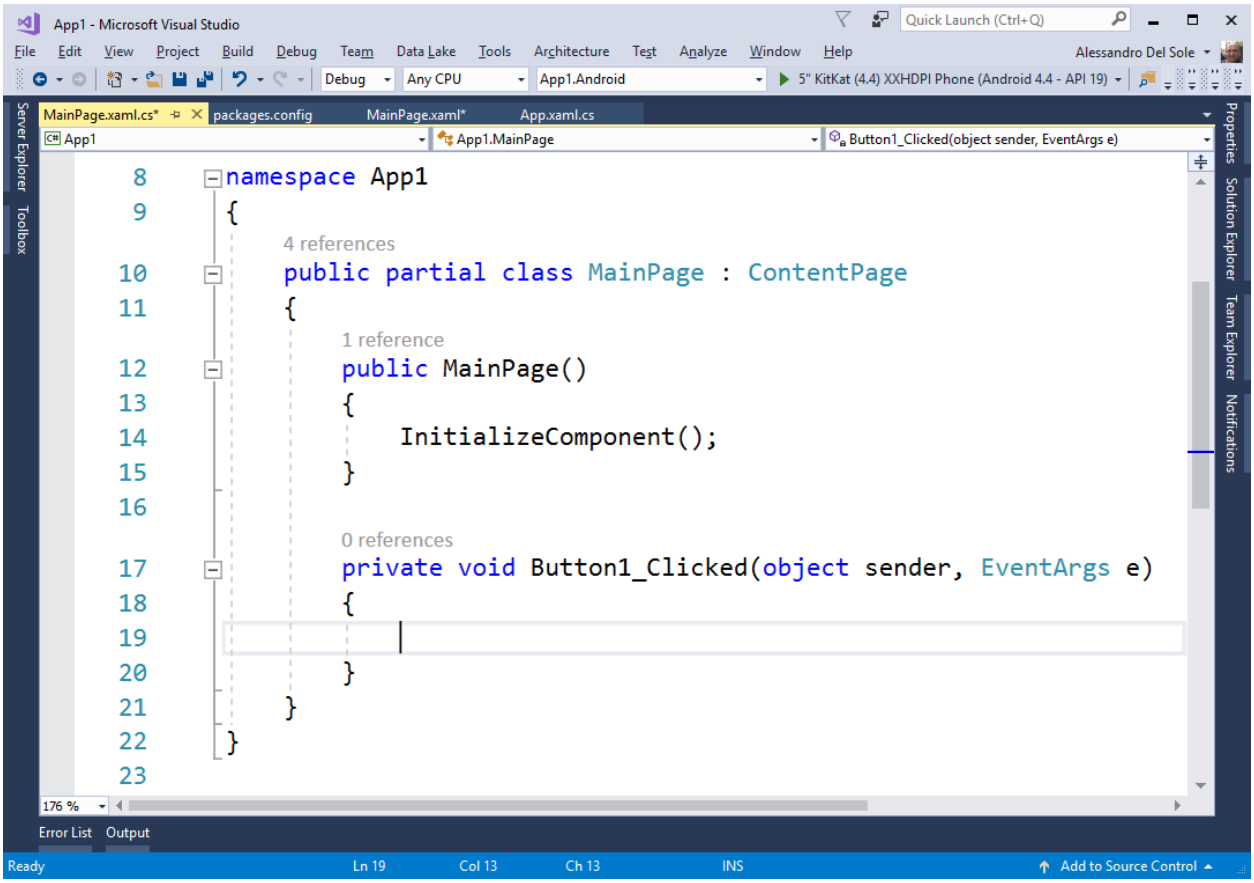

Figure 21: The event handler definition in C#

At this point, you will be able to write the code that performs the action you want to execute, exactly as it happens with other .NET platforms such as WPF or UWP. Generally speaking, event handlers' signatures require two parameters, one of type **`object`** representing the control that raised the event, and one object of type **`EventArgs`** containing information about the event. In many cases, event handlers work with derived versions of **`EventArgs`**, but these will be highlighted when appropriate. As you can imagine, Xamarin.Forms exposes events that are commonly available on all the supported platforms.

Understanding type converters

If you look at Code Listing 3, you will see that the **`Orientation`** property of the **`StackLayout`** is of type **`StackOrientation`**, the **`Padding`** property is of type **`Thickness`**, and the **`Margin`** property assigned to the **`Button`** is also of type **`Thickness`**. However, as you can see in Code Listing 4, the same properties are assigned with values passed in the form of strings in XAML. Xamarin.Forms (and all the other XAML-based platforms) implement the so-called *type converters*, which automatically convert a string into the appropriate value for a number of known types. Summarizing here all the available type converters and known target types is neither possible nor necessary at this point; you simply need to remember that, in most cases, strings you assign as property values are automatically converted into the appropriate type on your behalf.

Xamarin.Forms Previewer

Xamarin.Forms doesn't have a designer that allows you to draw the user interface visually with the mouse, the toolbox, and interactive windows as you are used to doing with platforms such as WPF, Windows Forms, and UWP. This implies that you need to write all your XAML manually. However, Visual Studio 2017 brings an important addition, known as the Xamarin.Forms Previewer. This is a tool window you enable with **View**, **Other Windows**, **Xamarin.Forms Previewer**, and it shows a preview of the user interface in real time, as you edit your XAML. Figure 22 shows the Xamarin.Forms Previewer in action.

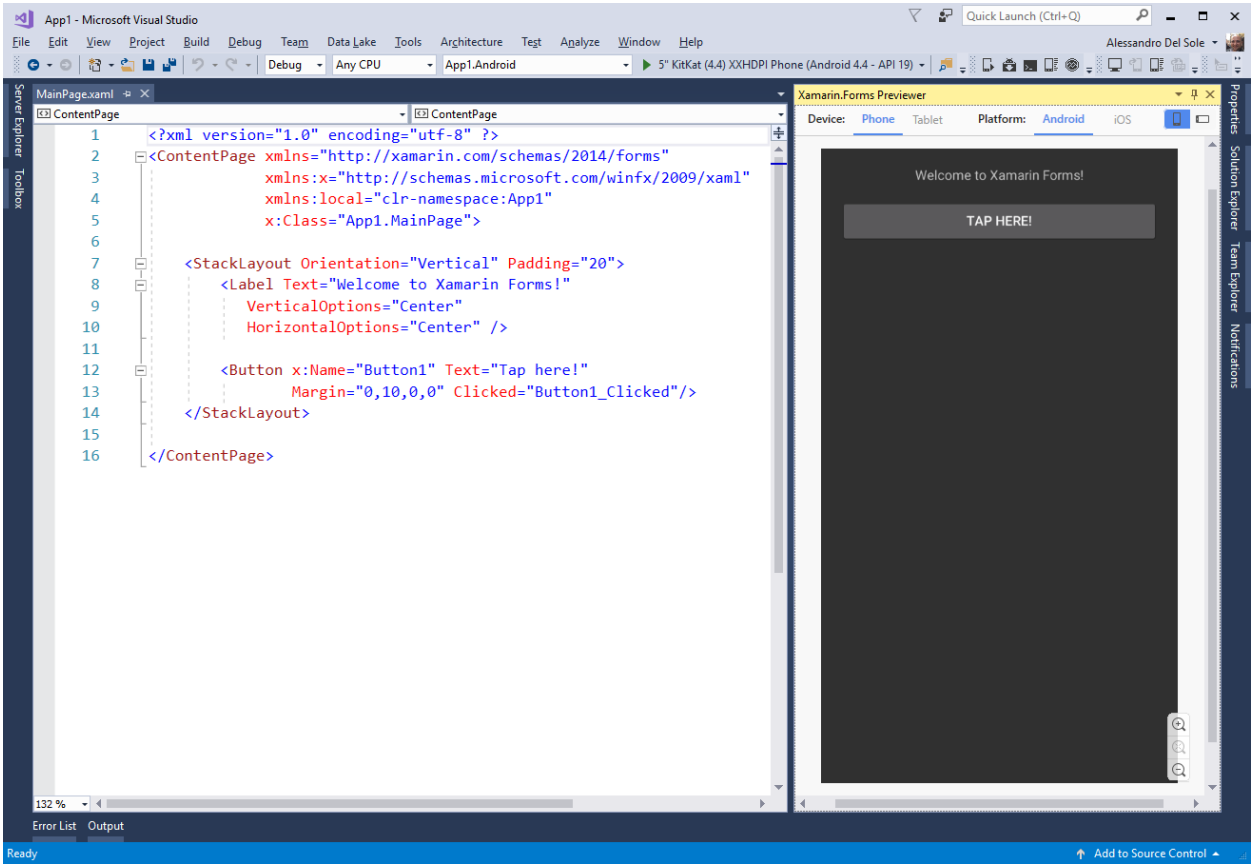

Figure 22: The Xamarin.Forms Previewer

Tip: Remember to rebuild your solution before opening the Xamarin.Forms Previewer for the first time.

At the bottom-right corner, the previewer provides zoom controls. At the top, you can select the device factor (phone or tablet), the platform used to render the preview (Android or iOS), and the orientation (vertical or horizontal). If you wish to render the preview based on iOS, remember you need Visual Studio to be connected to a Mac. If there are any errors in your XAML or if, for any reason, the Previewer is unable to render the preview, it will show a detailed error message. The Xamarin.Forms Previewer is an important addition, because it avoids the need to run the application every time you make significant edits to the UI, as was required in

the past. In the next chapters, I will often use the Previewer to demonstrate how the UI looks instead of running the emulator.

Hints for XAML Standard

XAML in Xamarin.Forms follows the Microsoft XAML 2009 specifications, but its vocabulary is different from other XAML-based platforms. For example, a text box is represented by the **`TextBox`** control in WPF and UWP, but in Xamarin.Forms you have an **`Entry`**. Again, the **`Button`** control in WPF and UWP exposes an event called **`Click`**, which is instead called **`Clicked`** in Xamarin.Forms. Recently, Microsoft has announced XAML Standard, a unification of XAML dialects across platforms. XAML Standard is still in its beginning stages, so it's not available yet. However, you can follow the progress on GitHub and you can read an introductory blog post that explains XAML Standard's goals in more detail.

Chapter summary

Sharing the user interface across platforms is Xamarin.Forms's main goal and this chapter provided a high-level overview of how you define the user interface with XAML, based on a hierarchy of visual elements. You have seen how to add visual elements and how to assign their properties; you have seen how type converters allow for passing string values in XAML and how the compiler converts them into the appropriate types; and you had a first look at the Xamarin.Forms Previewer to get a real-time, integrated representation of the user interface as you edit your XAML. After this overview of how the user interface is defined in Xamarin.Forms, it is time to discuss important UI concepts in more detail, and we will start by organizing the user interface with layouts.

Chapter 4 Organizing the UI with Layouts

Mobile devices such as phones, tablets, and laptops have different screen sizes and form factors. They also support both landscape and portrait orientations. Therefore, the user interface in mobile apps must dynamically adapt to the system, screen, and device so that visual elements can be automatically resized or rearranged based on the form factor and device orientation. In Xamarin.Forms, this is accomplished with layouts, which is the topic of this chapter.

Understanding the concept of layout

Tip: If you have previous experience with WPF or UWP, the concept of layout is the same as the concept of panels such as the `Grid` and the `StackPanel`.

One of the goals of Xamarin.Forms is to provide the ability to create dynamic interfaces that can be rearranged according to the user's preferences or to the device and screen size. Because of this, controls in mobile apps you build with Xamarin should not have a fixed size or position on the UI, except in a very limited number of scenarios. To make this possible, Xamarin.Forms controls are arranged within special containers, known as *layouts*. Layouts are classes that allow for arranging visual elements in the UI, and Xamarin.Forms provides many of them. In this chapter, you'll learn about available layouts and how to use them to arrange controls. The most important thing you have to keep in mind is that controls in Xamarin.Forms have a hierarchical logic; therefore, you can nest multiple panels to create complex user experiences. Table 2 summarizes available layouts. Then, in the next sections, you'll learn about them in more detail.

Table 2: Layouts in Xamarin.Forms

Layout	Description
`StackLayout`	Allows you to place visual elements near each other horizontally or vertically.
`Grid`	Allows you to organize visual elements within rows and columns.
`AbsoluteLayout`	A layout placed at a specified, fixed position.
`RelativeLayout`	A layout whose position depends on relative constraints.
`ScrollView`	Allows you to scroll the visual elements it contains.

Layout	Description
Frame	Draws a border and adds space around the visual element it contains.
ContentView	A special layout that can contain hierarchies of visual elements and can be used to create custom controls in XAML.

Remember that only one root layout is assigned to the **Content** property of a page, and that layout can then contain nested visual elements and layouts.

Alignment and spacing options

As a general rule, both layouts and controls can be aligned by assigning the **HorizontalOptions** and **VerticalOptions** properties with one of the property values from the **LayoutOptions** structure, summarized in Table 3. Providing an alignment option is very common. For instance, if you only have the root layout in a page, you will want to assign **VerticalOptions** with **StartAndExpand** so that the layout gets all the available space in the page (remember this consideration when you experiment with layouts and views in this chapter and the next one).

Table 3: Alignment options in Xamarin.Forms

Alignment	Description
Center	Aligns the visual element at the center.
CenterAndExpand	Aligns the visual element at the center and expands its bounds to fill the available space.
Start	Aligns the visual element at the left.
StartAndExpand	Aligns the visual element at the left and expands its bounds to fill the available space.
End	Aligns the visual element at the right.
EndAndExpand	Aligns the visual element at the right and expands its bounds to fill the available space.

You can also control the space between visual elements with three properties: **Padding**, **Spacing**, and **Margin**, summarized in Table 4.

Table 4: Spacing options in Xamarin.Forms

Spacing	Description
Margin	Represents the distance between the current visual element and its adjacent elements with either a fixed value for all four sides, or with comma-separated values for the left, top, right, and bottom. It is of type **Thickness** and XAML has built in a type converter for it.
Padding	Represents the distance between a visual element and its child elements. It can be set with either a fixed value for all four sides, or with comma-separated values for the left, top, right, and bottom. It is of type **Thickness** and XAML has built in a type converter for it.
Spacing	Available only in the **StackLayout** container, it allows you to set the amount of space between each child element, with a default of 6.0.

I recommend you spend some time experimenting with how alignment and spacing options work in order to understand how to get the appropriate result in your user interfaces.

The StackLayout

The **StackLayout** container allows the placing of controls near each other, as in a stack that can be arranged both horizontally and vertically. As with other containers, the **StackLayout** can contain nested panels. The following code shows how you can arrange controls horizontally and vertically. Code Listing 5 shows an example with a root **StackLayout** and two nested layouts.

Code Listing 5

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical">
        <StackLayout Orientation="Horizontal" Margin="5">
            <Label Text="Sample controls" Margin="5"/>
            <Button Text="Test button" Margin="5"/>
        </StackLayout>
        <StackLayout Orientation="Vertical" Margin="5">
            <Label Text="Sample controls" Margin="5"/>
```

```
            <Button Text="Test button" Margin="5"/>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

The result of the XAML in Code Listing 5 is shown in Figure 23.

Figure 23: Arranging visual elements with the StackLayout

The **`Orientation`** property can be set as **`Horizontal`** or **`Vertical`**, and this influences the final layout. If not specified, **`Vertical`** is the default. One of the main benefits of XAML code is that element names and properties are self-explanatory, and this is the case in **`StackLayout`**'s properties, too. Remember that controls within a **`StackLayout`** are automatically resized according to the orientation. If you do not like this behavior, you need to specify **`WidthRequest`** and **`HeightRequest`** properties on each control, which represent the width and height respectively. **`Spacing`** is a property that you can use to adjust the amount of space between child elements; this is preferred to adjusting the space on the individual controls with the **`Margin`** property.

The `Grid`

The **`Grid`** is one of the easiest layouts to understand and probably the most versatile. It allows you to create tables with rows and columns. In this way, you can define cells and each cell can contain a control or another layout storing nested controls. The **`Grid`** is versatile in that you can just divide it into rows or columns or both. The following code defines a **`Grid`** that is divided into two rows and two columns:

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>
</Grid>
```

`RowDefinitions` is a collection of **`RowDefinition`** objects, and the same is true for **`ColumnDefinitions`** and **`ColumnDefinition`**. Each item represents a row or a column within the **`Grid`**, respectively. You can also specify a **`Width`** or a **`Height`** property to delimit row and column dimensions; if you do not specify anything, both rows and columns are dimensioned at the maximum size available. When resizing the parent container, rows and columns are automatically rearranged.

The preceding code creates a table with four cells. To place controls in the **`Grid`**, you specify the row and column position via the **`Grid.Row`** and **`Grid.Column`** properties, known as attached properties, on the control. Attached properties allow for assigning properties of the parent container from the current visual element. The index of both is zero-based, meaning that 0 represents the first column from the left and the first row from the top. You can place nested layouts within a cell or a single row or column. The code in Code Listing 6 shows how to nest a grid into a root grid with children controls.

Tip:** `Grid.Row="0"` **and** `Grid.Column="0"` **can be omitted.

Code Listing 6

```
<?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="Layouts.GridSample">
    <ContentPage.Content>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
```

```
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Button Text="First Button" />
            <Button Grid.Column="1" Text="Second Button"/>

            <Grid Grid.Row="1">
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition />
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>
                <Button Text="Button 3" />
                <Button Text="Button 4" Grid.Column="1" />
            </Grid>
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

Figure 24 shows the result of this code.

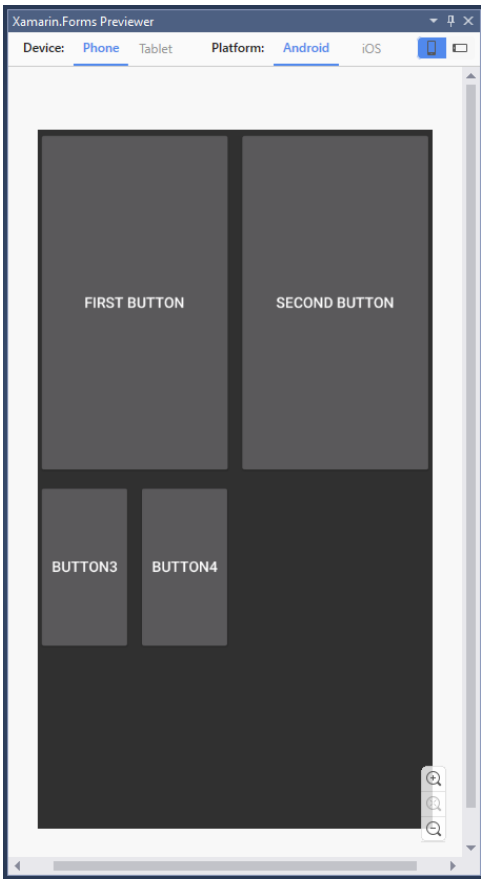

Figure 24: Arranging visual elements with the Grid

The **Grid** layout is very versatile and is also a good choice (when possible) in terms of performance.

Spacing and proportions for rows and columns

You have fine-grained control over the size, space, and proportions of rows and columns. The **Height** and **Width** properties of the **RowDefinition** and **ColumnDefinition** objects can be set with values from the **GridUnitType** enumeration as follows:

- **Auto**: Automatically sizes to fit content in the row or column.
- **Star**: Sizes columns and rows as a proportion of the remaining space.
- **Absolute**: Sizes columns and rows with specific, fixed height and width values.

XAML has type converters for the **GridUnitType** values, so you simply pass no value for **Auto**, a ***** for **Star**, and the fixed numeric value for **Absolute**, such as:

```
<Grid.ColumnDefinitions>
   <ColumnDefinition />
   <ColumnDefinition Width="*"/>
   <ColumnDefinition Width="20"/>
</Grid.ColumnDefinitions>
```

Introducing spans

In some situations, you might have elements that should occupy more than one row or column. In these cases, you can assign the **`Grid.RowSpan`** and **`Grid.ColumnSpan`** attached properties with the number of rows and columns a visual element should occupy.

The `AbsoluteLayout`

The **`AbsoluteLayout`** container allows you to specify where exactly on the screen you want the child elements to appear, as well as their size and bounds. There are a few different ways to set the bounds of the child elements based on the **`AbsoluteLayoutFlags`** enumeration 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.

Once you have created your child elements, to set them at an absolute position within the container you will need to assign the **`AbsoluteLayout.LayoutFlags`** attached property. You will also want to assign the **`AbsoluteLayout.LayoutBounds`** attached property to give the elements their bounds. 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. Code Listing 7 provides an example based on proportional dimensions and absolute position for child controls.

Code Listing 7

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <AbsoluteLayout>
        <Label Text="First Label"
               AbsoluteLayout.LayoutBounds="0, 0, 0.25, 0.25"
               AbsoluteLayout.LayoutFlags="All" TextColor="Red"/>
        <Label Text="Second Label"
               AbsoluteLayout.LayoutBounds="0.20, 0.20, 0.25, 0.25"
               AbsoluteLayout.LayoutFlags="All" TextColor="Orange"/>
        <Label Text="Third Label"
               AbsoluteLayout.LayoutBounds="0.40, 0.40, 0.25, 0.25"
               AbsoluteLayout.LayoutFlags="All" TextColor="Violet"/>
```

```
        <Label Text="Fourth Label"
               AbsoluteLayout.LayoutBounds="0.60, 0.60, 0.25, 0.25"
               AbsoluteLayout.LayoutFlags="All" TextColor="Yellow"/>
    </AbsoluteLayout>
</ContentPage>
```

Figure 25 shows the result of the **`AbsoluteLayout`** example.

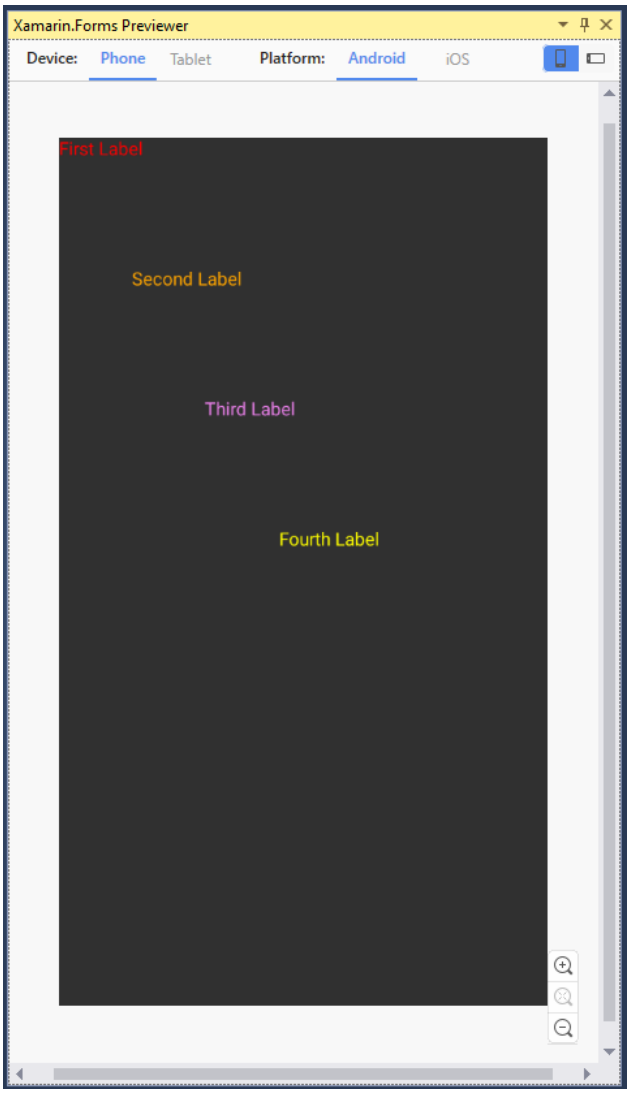

Figure 25: Absolute positioning with AbsoluteLayout

The `RelativeLayout`

The **`RelativeLayout`** container provides a way to specify the location of child elements relative either to each other or to the parent control. Relative locations are resolved through a series of **`Constraint`** objects that define each particular child element's relative position to another. In XAML, **`Constraint`** objects are expressed through the **`ConstraintExpression`** markup extension, which is used to specify the location or size of a child view as a constant, or relative to a parent or other named view. Markup extensions are very common in XAML, and you will see many of them in Chapter 7 about data binding, but discussing them in detail is beyond the scope here. The official documentation has a very detailed page on their syntax and implementation that I encourage you to read.

In the **`RelativeLayout`** class, there are properties named **`XConstraint`** and **`YConstraint`**. In the next example, you will see how to assign a value to these properties from within another XAML element, through attached properties. This is demonstrated in Code Listing 8, where you meet the **`BoxView`**, a visual element that allows you to draw a colored box. In this case, it's useful for giving you an immediate perception of how the layout is organized.

Code Listing 8

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <RelativeLayout>
        <BoxView Color="Red" x:Name="redBox"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToPa
rent,
            Property=Height,Factor=.15,Constant=0}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=1,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.8,Constant=0}" />
        <BoxView Color="Blue"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToVi
ew,
            ElementName=redBox,Property=Y,Factor=1,Constant=20}"
        RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToVi
ew,
            ElementName=redBox,Property=X,Factor=1,Constant=20}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=.5,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.5,Constant=0}" />
    </RelativeLayout>
</ContentPage>
```

The result of Code Listing 8 is shown in Figure 26.

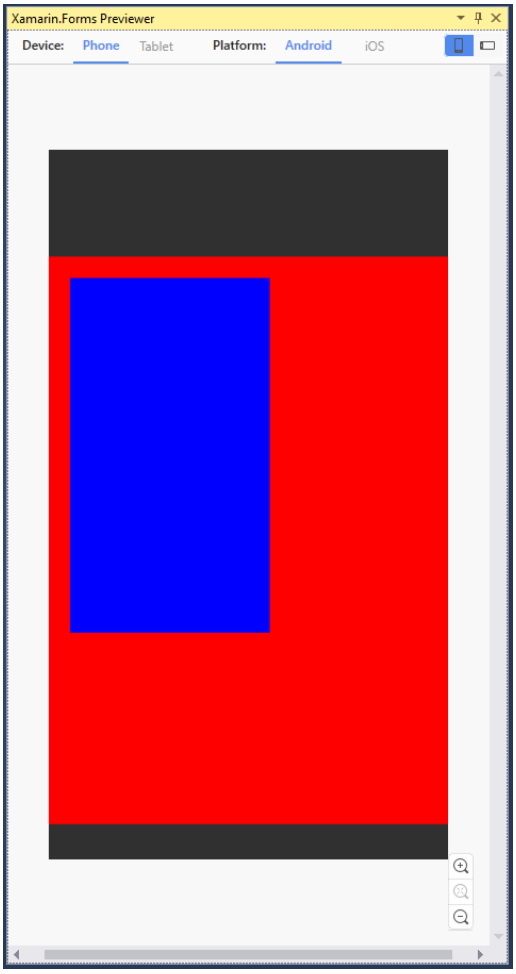

Figure 26: Arranging visual elements with the RelativeLayout

Tip: The `RelativeLayout` container has poor rendering performance and the documentation recommends that you avoid this layout whenever possible, or at least you should avoid more than one `RelativeLayout` per page.

The `ScrollView`

The special layout **`ScrollView`** allows you to present content that cannot fit on one screen and therefore should be scrolled. Its usage is very simple:

```
<ScrollView x:Name="Scroll1">
    <StackLayout>
        <Label Text="My favorite color:" x:Name="Label1"/>
        <BoxView BackgroundColor="Green" HeightRequest="600" />
    </StackLayout>
</ScrollView>
```

You basically add a layout or visual elements inside the **ScrollView** and, at runtime, the content will be scrollable if its area is bigger than the screen size. Additionally, you can specify the **Orientation** property (with values **Horizontal** or **Vertical**) to set the **ScrollView** to scroll only horizontally or only vertically. The reason the layout has a name in the sample usage is that you can interact with the **ScrollView** programmatically, invoking its **ScrollToAsync** method to move its position based on two different options. Consider the following lines:

```
Scroll1.ScrollToAsync(0, 100, true);

Scroll1.ScrollToAsync(Label1, ScrollToPosition.Start, true);
```

In the first case, the content at 100px from the top is visible. In the second case, the **ScrollView** moves the specified control at the top of the view and sets the current position at the control's position. Possible values for the **ScrollToPosition** enumeration are:

- **Center**: Scrolls the element to the center of the visible portion of the view.
- **End**: Scrolls the element to the end of the visible portion of the view.
- **MakeVisible**: Makes the element visible within the view.
- **Start**: Scrolls the element to the start of the visible portion of the view.

Note that you should never nest **ScrollView** layouts, and you should never include the **ListView** and **WebView** controls inside a **ScrollView** because they both already implement scrolling.

The Frame

The **Frame** is a very special layout in Xamarin.Forms because it provides an option to draw a colored border around the visual element it contains, and optionally add extra space between the **Frame**'s bounds and the visual element. Code Listing 9 provides an example.

Code Listing 9

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <Frame OutlineColor="Red" CornerRadius="3" HasShadow="True"
Margin="20">
        <Label Text="Label in a frame"
               HorizontalOptions="Center"
               VerticalOptions="Center"/>
    </Frame>
</ContentPage>
```

The **`OutlineColor`** property is assigned with the color for the border, the **`CornerRadius`** property is assigned with a value that allows you to draw circular corners, and the **`HasShadow`** property allows you to display a shadow. Figure 27 provides an example based on the UWP version of the project, since the Xamarin.Forms Previewer does not render frames appropriately in some situations, such as with dark themes.

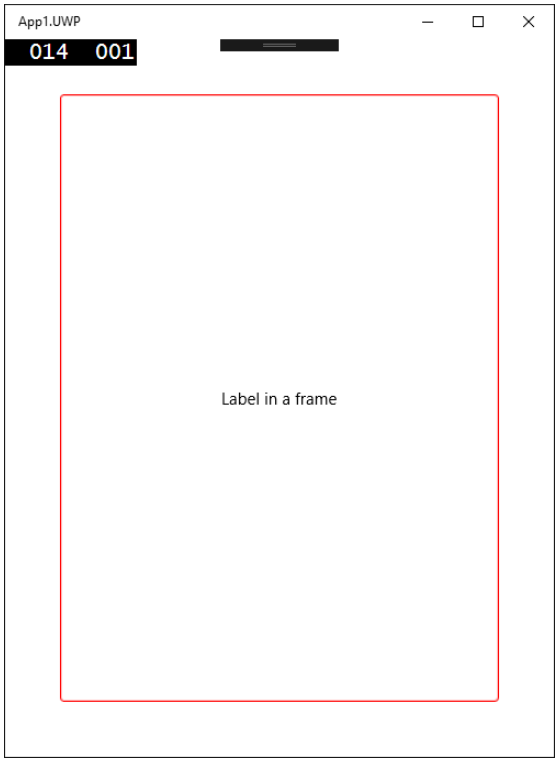

Figure 27: Drawing a Frame

The **`Frame`** will be resized proportionally based on the parent container's size.

The `ContentView`

The special container **`ContentView`** allows for aggregating multiple views into a single view and is useful to create reusable, custom controls. Because the **`ContentView`** represents a stand-alone visual element, Visual Studio makes it easier to create an instance of this container with a specific item template. In Solution Explorer, you can right-click the PCL name and then select **Add New Item**. In the **Add New Item** dialog, select the **Xamarin.Forms** node, then the **Content View** item, as shown in Figure 28. Make sure you do not select the item called Content View (C#), otherwise you will get a new class file that you will need to populate in C# code rather than XAML.

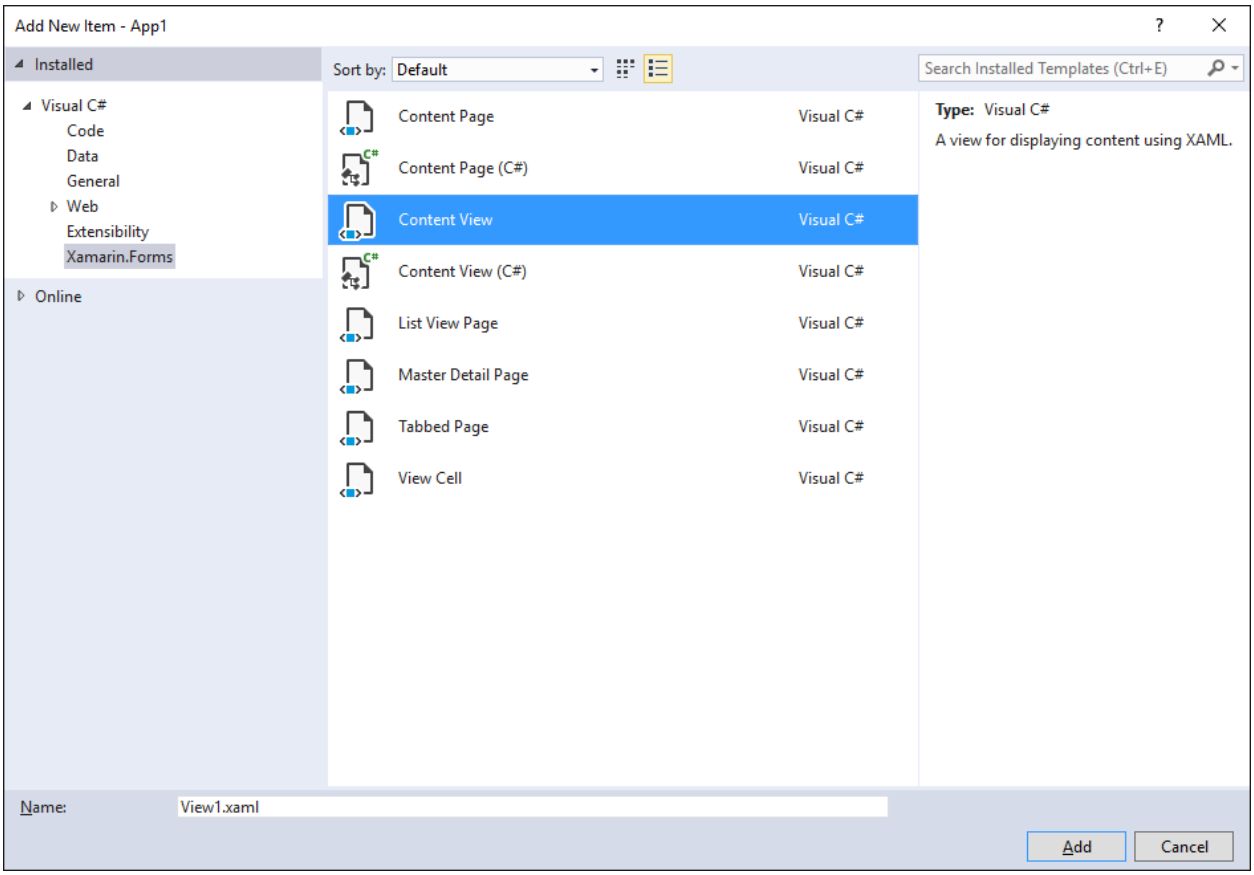

Figure 28: Adding a ContentView

When the new file is added to the project, the XAML editor shows basic content made of the **`ContentView`** root element and a **`Label`**. You can add multiple visual elements, as shown in Code Listing 10, and then you can use the **`ContentView`** as you would with an individual control or layout.

Code Listing 10

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.View1">
  <ContentView.Content>
      <StackLayout>
          <Label Text="Enter your email address:" />
          <Entry x:Name="EmailEntry" />
      </StackLayout>
  </ContentView.Content>
</ContentView>
```

It is worth mentioning that visual elements inside a **ContentView** can raise and manage events and support data binding, which makes the **ContentView** very versatile and perfect for building reusable views.

Chapter summary

Mobile apps require dynamic user interfaces that can automatically adapt to the screen size of different device form factors. In Xamarin.Forms, creating dynamic user interfaces is possible through a number of so-called layouts. The **StackLayout** allows you to arrange controls near one another both horizontally and vertically; the **Grid** allows you to arrange controls within rows and columns; the **AbsoluteLayout** allows you to give controls an absolute position; the **RelativeLayout** allows you to arrange controls based on the size and position of other controls or containers; the **ScrollView** layout allows you to scroll the content of visual elements that do not fit in a single page; the **Frame** layout allows you to draw a border around a visual element; the **ContentView** allows you to create reusable views. Now that you have a basic knowledge of layouts, it's time to discuss common controls in Xamarin.Forms that allow you to build the functionalities of the user interface, arranged within the layouts you learned in this chapter.

Chapter 5 Xamarin.Forms Common Controls

Xamarin.Forms ships with a rich set of common controls that you can use to build cross-platform user interfaces easily and without the need for the complexity of platform-specific features. As you can imagine, the benefit of these common controls is that they run on Android, iOS, and Windows from the same codebase. In this chapter, you'll learn about common controls, their properties, and their events. Other controls will be introduced in Chapter 7, especially controls whose purpose is displaying lists of data.

Note: In order to follow the examples in this chapter, create a new Xamarin.Forms solution based on the PCL code-sharing strategy. The name is up to you. Every time a new control is discussed, just clean the content of the root `ContentPage` object in the XAML file and remove any C# code specific to a single control or add a new file of type `ContentPage` to the project.

Understanding the concept of view

In Xamarin.Forms, a view is the building block of any mobile application. Put succinctly, a view is a control and it represents what you would call a widget in Android, a view in iOS, and a control in Windows. Views derive from the **`Xamarin.Forms.View`** class. Actually, from a technical perspective, layouts are views themselves and derive from **`Layout`**, an intermediate object in the hierarchy that derives from **`View`** and includes a **`Children`** property allowing you to add multiple visual elements to the layout itself. The concept of view is also important from the terminology perspective. In fact, in Xamarin.Forms and its documentation, you will more often find the word *view* than *control*. From now on, I will be using both view and control interchangeably, but remember that documentation and tutorials often refer to views.

Views' common properties

Views share a number of properties that are important for you to know in advance. These are summarized in Table 5.

Table 5: Views' common properties

Property	Description
`HorizontalOptions`	Same as Table 3.
`VerticalOptions`	Same as Table 3.
`HeightRequest`	Of type **`double`**, gets or sets the height of a view.

Property	Description
`WidthRequest`	Of type **`double`**, gets or sets the width of a view.
`IsVisible`	Of type **`bool`**, determines whether a control is visible on the user interface.
`IsEnabled`	Of type **`bool`**, allows enabling or disabling a control, keeping it visible on the UI.
`GestureRecognizers`	A collection of **`GestureRecognizer`** objects that enable touch gestures on controls that do not directly support touch. These will be discussed later in this chapter.

Tip: Controls also expose the `Margin` property described in Table 4.

So, if you wish to change the width or height of a view, remember that you need the **`WidthRequest`** and **`HeightRequest`** properties, instead of **`Height`** and **`Width`** that are read-only and return the current height and width.

Introducing common controls

This section provides a high-level overview of Xamarin.Forms's common controls and their most utilized properties. Remember to add the official documentation about the user interface to your bookmarks for a more detailed reference.

User input with the `Button`

The **`Button`** control is certainly one of the most used controls in every user interface. You already saw a couple examples of the **`Button`** previously, but here is a quick summary. This control exposes the properties summarized in Table 6, and you declare it like this:

```
<Button x:Name="Button1" Text="Tap here" TextColor="Orange" BorderColor="Red"
        BorderWidth="2" BorderRadius="2" Clicked="Button1_Clicked"/>
```

Table 6: Button's properties

Property	Description
`Text`	The text in the button.
`TextColor`	The color of the text in the button.
`BorderColor`	Draws a colored border around the button.

Property	Description
BorderWidth	The width of the border around the button.
BorderRadius	The radius of the edges around the button.
Image	An optional image to be set near the text.

Notice how you can specify a name with **x:Name** so that you can interact with the button in C#, which is the case when you set the **Clicked** event with a handler. This control also exposes a **Font** property whose behavior is discussed in the next section about text.

Working with text: `Label`, `Entry`, `Editor`

Displaying text and requesting input from the user in the form of text is extremely common in every mobile app. Xamarin.Forms offers the **Label** control to display read-only text and the **Entry** and **Editor** controls to receive text. The **Label** control has some useful properties, as shown in the following XAML:

```
<Label Text="Displaying some text" LineBreakMode="WordWrap"
   TextColor="Blue" XAlign="Center" YAlign="Center"/>
```

LineBreakMode allows you to truncate or wrap a long string and can be assigned a value from the **LineBreakMode** enumeration. For example, **WordWrap** splits a long string into multiple lines proportionate to the available space. If not specified, **NoWrap** is the default. **XAlign** and **YAlign** specify the horizontal and vertical alignment for the text. The **Entry** control instead allows you to enter a single line of text and you declare it as follows:

```
<Entry x:Name="Entry1" Placeholder="Enter some text..."
   TextColor="Green" Keyboard="Chat" Completed="Entry1_Completed"/>
```

The **Placeholder** property lets you display specific text in the entry until the user types something. It is useful for explaining the purpose of the text box. When the user taps the **Entry**, the on-screen keyboard is displayed. The appearance of the keyboard can be controlled via the **Keyboard** property, which allows you to display the most appropriate keys depending on the **Entry**'s purpose. Supported values are **Chat**, **Email**, **Numeric**, **Telephone**, and **Number**. If **Keyboard** is not assigned, **Default** is assumed. This control also exposes two events: **Completed**, which is fired when the users finalize the text by tapping the Return key, and **TextChanged**, which is fired at every keystroke. You provide event handlers the usual way, as follows:

```
private void Entry1_Completed(object sender, EventArgs e)
{
    // Entry1.Text contains the full text
}
```

Entry also provides the **IsPassword** property to mask the **Entry**'s content, which you use when the user must enter a password. The combination of the **Label** and **Entry** controls is visible in Figure 29.

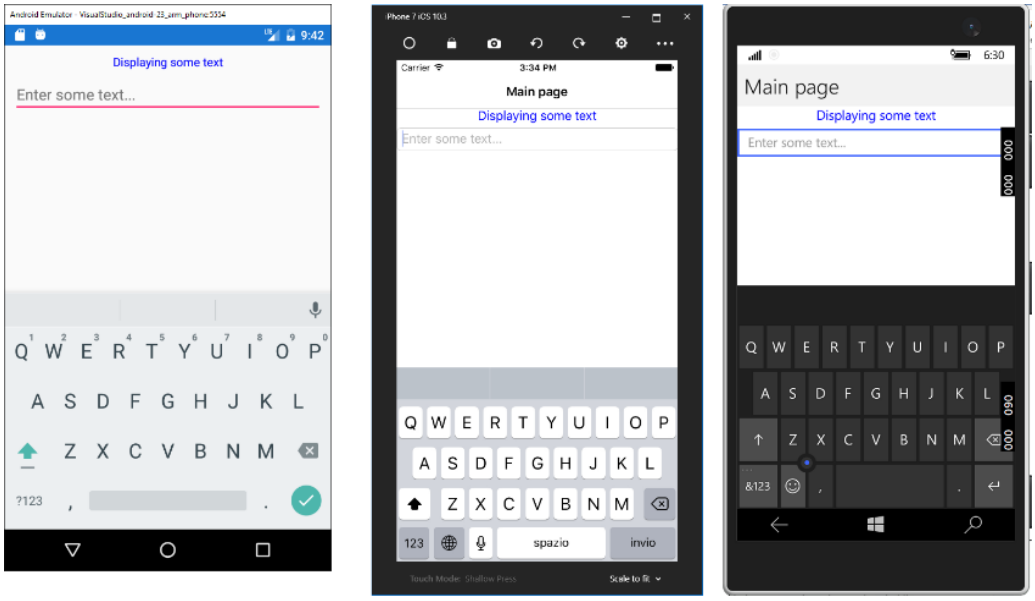

Figure 29: Label and Entry controls

The **`Editor`** control is very similar to **`Entry`** in terms of behavior, events, and properties, but it allows for entering multiple lines of text. For example, if you place the editor inside a layout, you can set its **`HorizontalOptions`** and **`VerticalOptions`** properties with **`Fill`** so that it will take all the available space in the parent.

Managing fonts

Controls that display some text (including the **`Button`**) or that wait for user input through the keyboard also expose some properties related to fonts, such as **`FontFamily`**, **`FontAttributes`**, and **`FontSize`**. **`FontFamily`** specifies the name of the font you want to use. **`FontAttributes`** displays text as **`Italic`** or **`Bold`** and, if not specified, **`None`** is assumed. With **`FontSize`**, you can specify the font size with either a numeric value or with a so-called *named size*, based on the **`Micro`**, **`Small`**, **`Medium`**, and **`Large`** values from the **`NamedSize`** enumeration. With this enumeration, Xamarin.Forms chooses the appropriate size for the current platform. For instance, the following two options are allowed to set the font size:

```
<Label Text="Some text" FontSize="72"/>
<Label Text="Some text" FontSize="Large"/>
```

Unless you are writing an app for a single platform, I recommend you avoid using numeric values. Use the named size instead.

Working with dates and time: `DatePicker` and `TimePicker`

Another common requirement in mobile apps is working with dates and time: Xamarin.Forms provides the **`DatePicker`** and **`TimePicker`** views for that. On each platform, these are rendered with the corresponding date and time selectors. **`DatePicker`** exposes the **`Date`**, **`MinimumDate`**, and **`MaximumDate`** properties that represent the selected/current date, the minimum date, and the maximum date, respectively, all of type **`DateTime`**. It exposes an event called

DateSelected, which is raised when the user selects a date. You can handle this to retrieve the value of the **Date** property. The view can be declared as follows:

```
<DatePicker x:Name="DatePicker1" MinimumDate="07/17/2017"
            MaximumDate="12/31/2017"
            DateSelected="DatePicker1_DateSelected"/>
```

And then in the code-behind you can retrieve the selected date like this:

```
private void DatePicker1_DateSelected(object sender, DateChangedEventArgs e)
{
    DateTime selectedDate = e.NewDate;
}
```

The **DateChangedEventArgs** object stores the selected date in the **NewDate** property and the previous date in the **OldDate** property. Figure 30 shows the **DatePicker** on the three platforms.

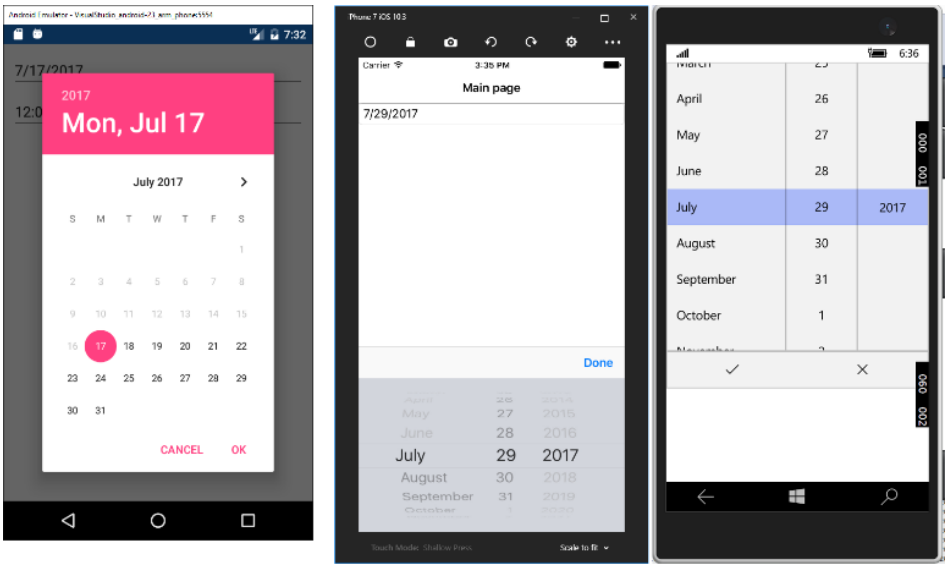

Figure 30: The DatePicker in action

The **TimePicker** exposes a property called **Time**, of type **TimeSpan**, but it does not expose a specific event for time selection, so you need to use the **PropertyChanged** event. In terms of XAML, you declare a **TimePicker** like this:

```
<TimePicker x:Name="TimePicker1"
            PropertyChanged="TimePicker1_PropertyChanged"/>
```

Then, in the code-behind, you need to detect changes on the **Time** property as follows:

```
private void TimePicker1_PropertyChanged(object sender,
    System.ComponentModel.PropertyChangedEventArgs e)
{
    if(e.PropertyName == TimePicker.TimeProperty.PropertyName)
    {
```

```
        TimeSpan selectedTime = TimePicker1.Time;
    }
}
```

TimeProperty is a dependency property, a concept that will be discussed in Chapter 7. Figure 31 shows the **TimePicker** in action.

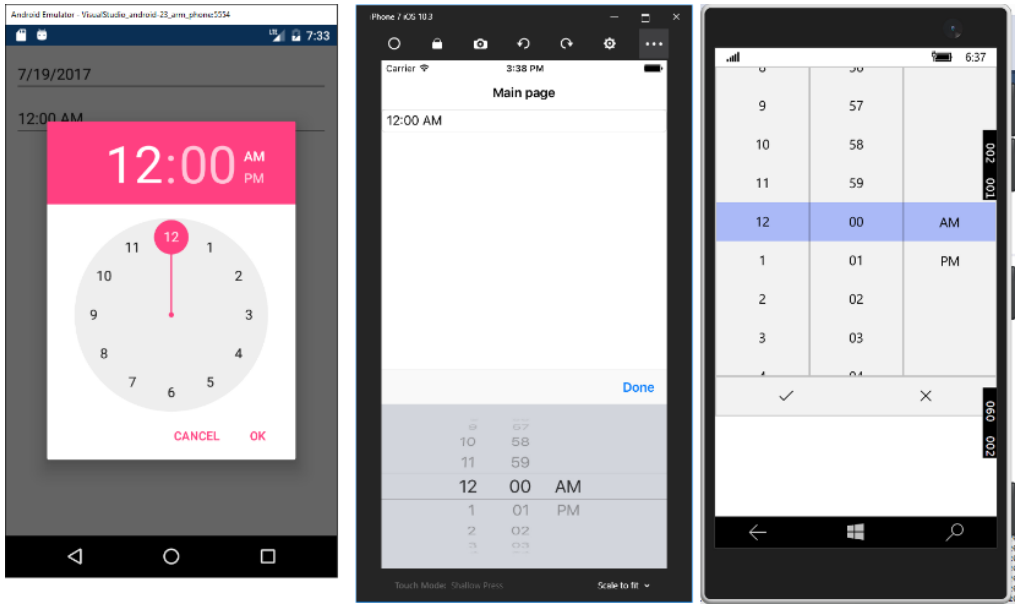

Figure 31: The TimePicker in action

Tip: You can also assign a date or time to pickers in the C# code-behind. For example, in the constructor of the page that declares them.

Displaying HTML contents with `WebView`

The **WebView** control allows for displaying HTML contents, including webpages and static HTML markup. This control exposes the **Navigating** and **Navigated** events that are raised when navigation starts and completes, respectively. The real power is in its **Source** property, of type **WebViewSource**, which can be assigned with a variety of content, such as URIs or strings containing HTML markup. For example, the following XAML opens the specified website:

```
<WebView x:Name="WebView1" Source="https://www.xamarin.com"/>
```

The following example shows instead how you can assign the **Source** property with a string:

```
WebView1.Source = "<div><h1>Header</h1></div>";
```

For dynamic sizing, a better option is enclosing the **`WebView`** inside a **`Grid`** layout. If you instead use the **`StackLayout`**, you need to supply height and width explicitly. When you browse contents on the Internet, you need to enable the Internet permission in the Android manifest and the Internet (Client) permission in the UWP manifest. Figure 32 shows how the WebView appears.

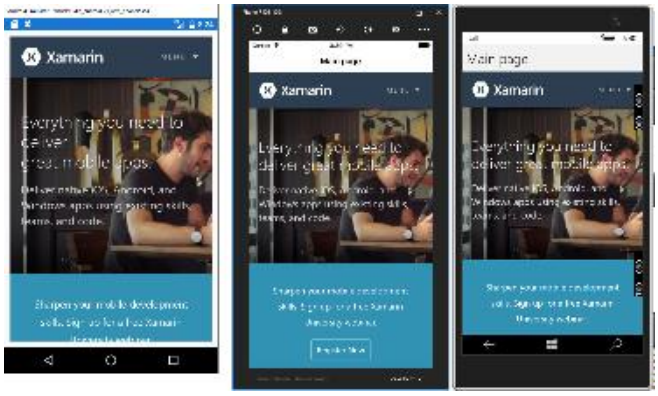

Figure 32: Displaying HTML contents with WebView

If the webpage you display allows you to browse other pages, you can leverage the built-in **`GoBack`** and **`GoForward`** methods, together with the **`CanGoBack`** and **`CanGoForward`** Boolean properties to programmatically control navigation between webpages.

Tip: If you need to implement navigation to URLs, it might be worth considering the so-called deep linking feature, available since Xamarin.Forms 2.3.1.

App Transport Security in iOS

Starting with iOS 9, Apple introduced some restrictions in accessing networked resources, including websites, enabling navigation only through the HTTPS protocol by default. This feature is known as App Transport Security (ATS). ATS can be controlled in the iOS project properties, and it allows for introducing some exceptions because you might need to browse HTTP contents despite the restrictions. More details about ATS and exceptions are available in the documentation; however, remember that if the **`WebView`** shows no content on iOS, the reason might be ATS.

Implementing value selection: `Switch`, `Slider`, `Stepper`

Xamarin.Forms offers a number of controls for user input based on selecting values. The first of them is the **`Switch`**, which provides a toggled value and is useful for selecting values such as true/false, on/off, and enabled/disabled. It exposes the **`IsToggled`** property, which turns the switch on when **`true`**, and the **`Toggled`** event, which is raised when the user changes the switch position. This control has no built-in label, so you need to use it in conjunction with a **`Label`** as follows:

```
<StackLayout Orientation="Horizontal">
   <Label Text="Enable data plan"/>
```

```
   <Switch x:Name="Switch1" IsToggled="True" Toggled="Switch1_Toggled"
           Margin="5,0,0,0"/>
</StackLayout>
```

The **Toggled** event stores the new value in the **ToggledEventArgs** object that you use as follows:

```
private void Switch1_Toggled(object sender, ToggledEventArgs e)
{
    bool isToggled = e.Value;
}
```

The **Slider** allows the input of a linear value. It exposes the **Value**, **Minimum**, and **Maximum** properties, all of type **double**, which represent the current value, minimum value, and maximum value. Like the **Switch**, it does not have a built-in label, so you can use it together with a **Label** as follows:

```
<StackLayout Orientation="Horizontal">
   <Label Text="Select your age: "/>
   <Slider x:Name="Slider1" Maximum="85" Minimum="13" Value="30"
    ValueChanged="Slider1_ValueChanged"/>
</StackLayout>
```

Tip: Surprisingly, if you write the `Minimum` before the `Maximum`, a runtime error will occur. So, for both the `Slider` and the `Stepper`, the order matters.

The **ValueChanged** event is raised when the user moves the selector on the **Slider** and the new value is sent to the **NewValue** property of the **ValueChangedEventArgs** object you get in the event handler. The last control is the **Stepper**, which allows the supplying of discrete values with a specified increment. It also allows the specifying of minimum and maximum values. You use the **Value**, **Increment**, **Minimum**, and **Maximum** properties of type **double** as follows:

```
<StackLayout Orientation="Horizontal">
   <Label Text="Select your age: "/>
   <Stepper x:Name="Stepper1" Increment="1" Maximum="85" Minimum="13"
            Value="30" ValueChanged="Stepper1_ValueChanged"/>
   <Label x:Name="StepperValue"/>
</StackLayout>
```

Notice that both the **Stepper** and **Slider** only provide a way to increment and decrement a value, so it is your responsibility to display the current value, for example, with a **Label** that you can handle through the **ValueChanged** event. The following code demonstrates how to accomplish this with the **Stepper**:

```
private void Stepper1_ValueChanged(object sender, ValueChangedEventArgs e)
{
    StepperValue.Text = e.NewValue.ToString();
}
```

Figure 33 shows a summary of all the aforementioned views.

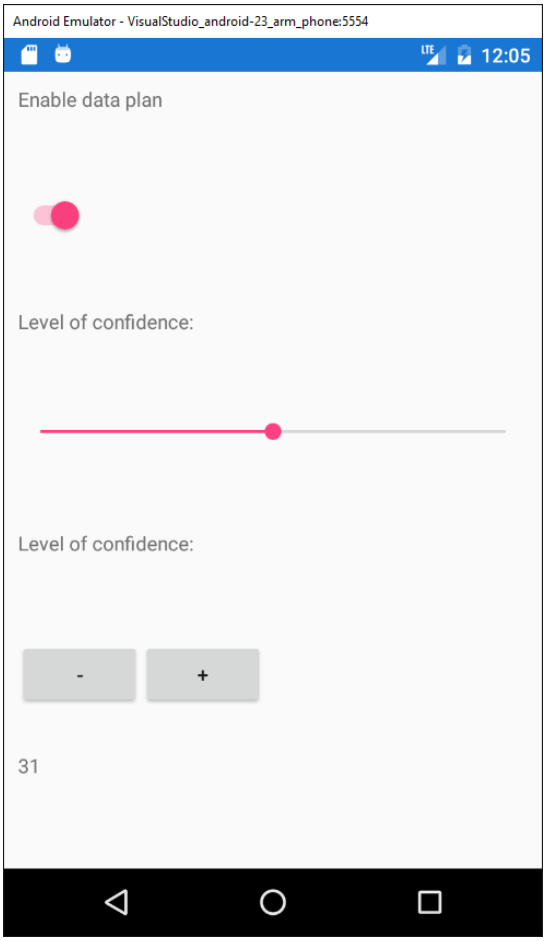

Figure 33: A summary view of the Switch, Slider, and Stepper controls

Introducing the `SearchBar`

One of the nicest views in Xamarin.Forms, the **`SearchBar`** shows a native search box with a search icon that users can tap. This view exposes the **`SearchButtonPressed`** event. You can handle this event to retrieve the text the user typed in the box and then perform your search logic; for example, executing a LINQ query against an in-memory collection or filtering data from the table of a local database. It also exposes the **`TextChanged`** event, which is raised at every keystroke, and the **`Placeholder`** property, which allows you to specify a placeholder text like the same-named property of the **`Entry`** control. You declare it as follows:

```
<SearchBar x:Name="SearchBar1" Placeholder="Enter your search key..."
           SearchButtonPressed="SearchBar1_SearchButtonPressed"/>
```

Figure 34 shows an example.

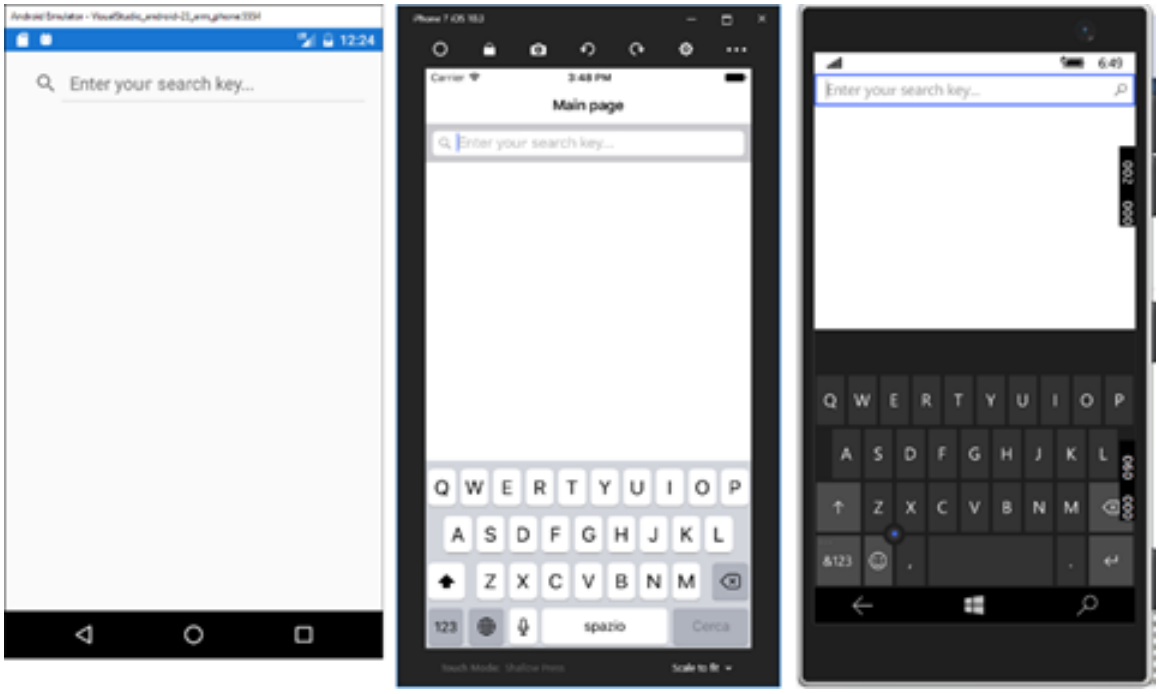

Figure 34: The SearchBar view

In Chapter 7, you will learn how to display lists of items through the **`ListView`** control. The **`SearchBar`** can be a good companion in that you can use it to filter a list of items based on the search key the user entered.

Long-running operations: `ActivityIndicator` and `ProgressBar`

In some situations, your app might need to perform potentially long-running operations, such as downloading content from the Internet or loading data from a local database. In such situations, it is a best practice to inform the user that an operation is in progress. This can be accomplished with two views, the **`ActivityIndicator`** or the **`ProgressBar`**. The latter exposes a property called **`Progress`**, of type **`double`**. This control is not used very often, because it implies you are able to calculate the amount of time or data needed to complete an operation. The **`ActivityIndicator`** instead shows a simple, animated indicator that is displayed while an operation is running, without the need to calculate its progress. It is enabled by setting its **`IsRunning`** property to **`true`**; you might also want to make it visible only when running, done by assigning **`IsVisible`** with **`true`**. So, you typically declare it in XAML as follows:

```
<ActivityIndicator x:Name="ActivityIndicator1" />
```

Then, in the code-behind, you can control it as follows:

```
// Starting the operation...
ActivityIndicator1.IsVisible = true;
ActivityIndicator1.IsRunning = true;

// Executing the operation...
```

```
// Operation completed
ActivityIndicator1.IsRunning = false;
ActivityIndicator1.IsVisible = false;
```

As a personal suggestion, I recommend you always set both **`IsVisible`** and **`IsRunning`**. This will help you keep consistent behavior across platforms. Figure 35 shows an example based on Android.

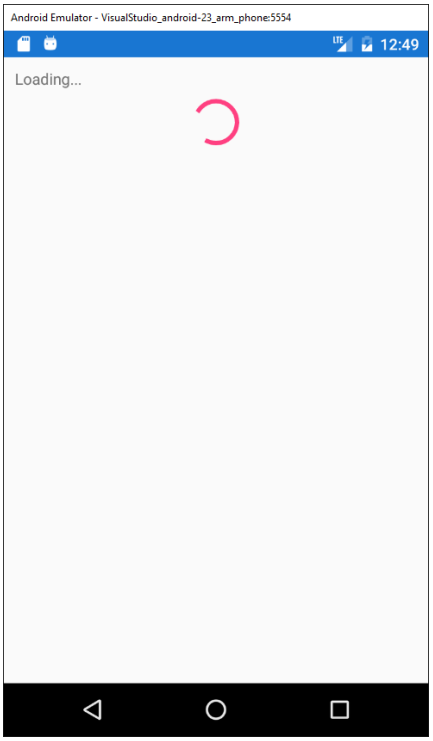

Figure 35: The ActivityIndicator shows that an operation is in progress

Tip:** `Page` **objects, such as the** `ContentPage`**, expose a property called** `IsBusy` **that enables an activity indicator when assigned with** `true`**. Depending on your scenario, you might also consider this option.

Working with images

Using images is crucial in mobile apps since they both enrich the look and feel of the user interface and enable apps to support multimedia content. Xamarin.Forms provides an **`Image`** control you can use to display images from the Internet, local files, and embedded resources. Displaying images is really simple, while understanding how you load and size images is more complex, especially if you have no previous experience with XAML and dynamic user interfaces. You declare an **`Image`** as follows:

```
<Image Source="https://www.xamarin.com/content/images/pages/branding/assets/xamarin-logo.png" Aspect="AspectFit"/>
```

As you can see, you assign the **`Source`** property with the image path, which can be a URL or the name of a local file or resource. **`Source`** can be assigned either in XAML or in code-behind. You will assign this property in C# code when you need to assign the property at runtime. This property is of type **`ImageSource`** and, while XAML has a type converter for it, in C# you need to use specific methods depending on the image source: **`FromFile`** requires a file path that can be resolved on each platform, **`FromUri`** requires a **`System.Uri`** object, and **`FromResource`** allows you to specify an image in the embedded app resources.

Note: Each platform has its own way of working with local images and embedded resources, which requires further explanation. Because this goes beyond the scope of this e-book, I strongly recommend you read the official documentation, which also explains how to manage images for different purposes on iOS, Android, and Windows.

The **`Aspect`** property determines how to size and stretch an image within the bounds it is being displayed in. It requires a value from the **`Aspect`** enumeration:

- **`Fill`**: Stretches the image to fill the display area completely and exactly. This may result in the image being distorted.
- **`AspectFill`**: Clips the image so that it fills the display area while preserving the aspect.
- **`AspectFit`**: Letterboxes the image (if required) so that the entire image fits into the display area, with blank space added to the top, bottom, or sides depending on whether the image is wide or tall.

You can also set the **`WidthRequest`** and **`HeightRequest`** properties to adjust the size of the **`Image`** control. Figure 36 shows an example.

Figure 36: Displaying images with the Image view

Supported image formats are .jpg, .png, .gif, .bmp, and .tif. Working with images also involves icons and splash screens, which are totally platform dependent and therefore require you to read the official documentation. Also, for enhanced 2-D graphics, you might want to consider taking a look at the SkiaSharp library, a portable library that works great with Xamarin.Forms and is powered by Google's Skia library.

Introducing gesture recognizers

Views such as **`Image`** and **`Label`** do not include support for touch gestures natively, but sometimes you might want to allow users to tap a picture or text to perform an action such as navigating to a page or website. In Xamarin.Forms, you can leverage gesture recognizers to add touch support to views that do not include it out of the box. Views expose a collection called **`GestureRecognizers`**, of type **`IList<GestureRecognizer>`**. Supported gesture recognizers are:

- **`TapGestureRecognizer`**: Allows recognition of taps.
- **`PinchGestureRecognizers`**: Allows recognition of the pinch-to-zoom gesture.
- **`PanGestureRecognizers`**: Enables the dragging of objects with the pan gesture.

For example, the following XAML demonstrates how to add a **`TapGestureRecognizer`** to an **`Image`** control:

```
<Image Source="https://www.xamarin.com/content/images/pages/branding/assets/x
amarin-logo.png" Aspect="AspectFit">
   <Image.GestureRecognizers>
       <TapGestureRecognizer x:Name="ImageTap"
        NumberOfTapsRequired="1" Tapped="ImageTap_Tapped"/>
   </Image.GestureRecognizers>
</Image>
```

You can assign the **`NumberOfTapsRequired`** property (self-explanatory) and the **`Tapped`** event with a handler that will be invoked when the user taps the image. That will look like this:

```
private void ImageTap_Tapped(object sender, EventArgs e)
{
    // Do your stuff here...
}
```

Gesture recognizers give you great flexibility and allow you to improve the user experience in your mobile apps by adding touch support where required.

Displaying alerts

All platforms can show pop-up alerts with informative messages or receive user input with common choices such as OK or Cancel. Pages in Xamarin.Forms provide an asynchronous method called **`DisplayAlert`**, which is very easy to use. For example, suppose you want to display a message when the user taps a button. The following code demonstrates how to accomplish this:

```
private async void Button1_Clicked(object sender, EventArgs e)
{
    await DisplayAlert("Title", "This is an informational pop-up", "OK");
}
```

As an asynchronous method, you call **`DisplayAlert`** with the **`await`** operator, marking the containing method as **`async`**. The first argument is the pop-up title, the second argument is the text message, and the third argument is the text you want to display in the only button that appears. Actually, **`DisplayAlert`** has an overload that can wait for the user input and return **`true`** or **`false`** depending on whether the user selected the OK option or the Cancel option:

```
bool result =
    await DisplayAlert("Title", "Do you wish to continue?", "OK", "Cancel");
```

You are free to write whatever text you like for the OK and Cancel options, and IntelliSense helps you understand the order of these options in the parameter list. If the user selects OK, **`DisplayAlert`** returns **`true`**. Figure 37 shows an example of the alert.

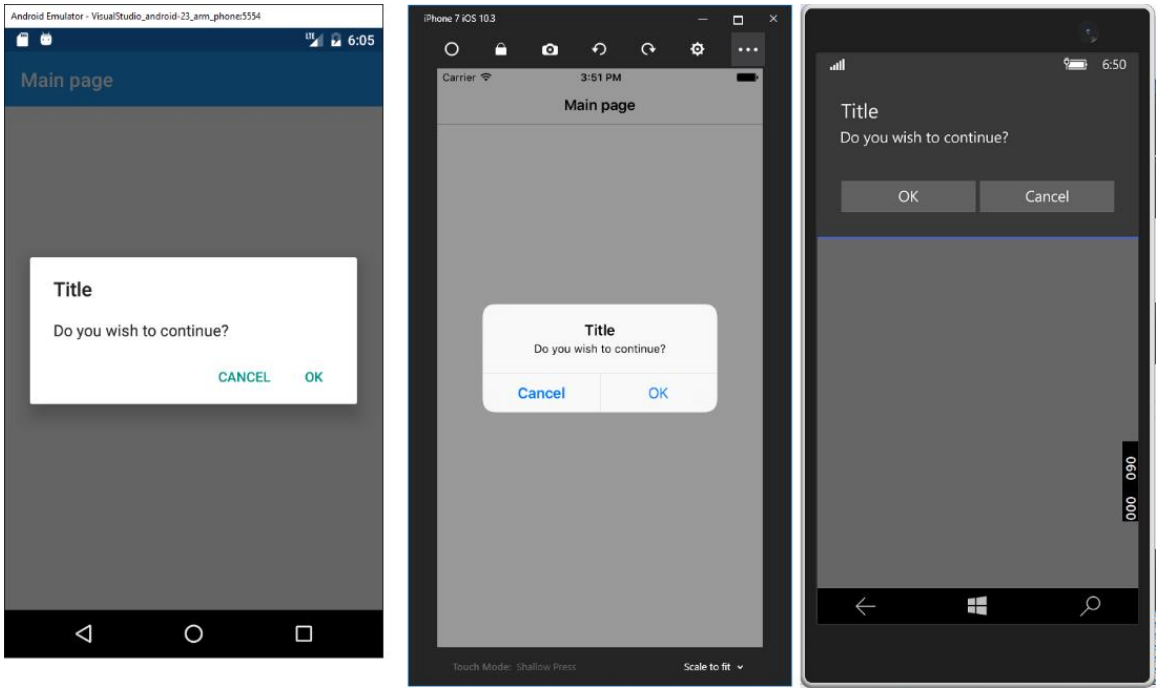

Figure 37: Displaying alerts

Chapter summary

This chapter introduced the concepts of view and common views in Xamarin.Forms, the building blocks for any user interface in your mobile apps. You have seen how to obtain user input with the **`Button`**, **`Entry`**, **`Editor`**, and **`SearchBar`** controls; you have seen how to display information with the **`Label`** and how to use it in conjunction with other input views such as **`Slider`**, **`Stepper`**, and **`Switch`**. You have seen how the **`DatePicker`** and **`TimePicker`** views allow you to work with dates and time. You have seen how to display images with the **`Image`** view; you have used the **`WebView`** to show HTML content; and you have seen how to inform the user of the progress of long-running operations with **`ActivityIndicator`** and **`ProgressBar`**. Finally, you have seen how to add gesture support to views that do not include it out of the box, and how to display alerts for informational purposes and to allow user choices.

Now you have all you need to build high-quality user interfaces with layouts and views, and you have seen how to use all these building blocks in a single page. However, most mobile apps are made of multiple pages. The next chapter explains all the available page types in Xamarin.Forms and the navigation infrastructure.

Chapter 6 Pages and Navigation

In the previous chapters, we went over the basics of layouts and views, the fundamental building blocks of the user interface in mobile applications. However, I demonstrated how to use layouts and views within a single page, while real-world mobile apps are made of multiple pages. Android, iOS, and Windows provide a number of different pages that allow you to display content in several ways and to provide the best user experience possible based on the content you need to present. Xamarin.Forms provides unified page models you can use from your single, shared C# codebase that work cross-platform. It also provides an easy-to-use navigation framework, the infrastructure you use to move between pages. Pages and navigation are the topics of this chapter and the last pieces of the user interface framework you need to know to build beautiful, native apps with Xamarin.Forms.

Note: In order to follow the examples in this chapter, create a new Xamarin.Forms solution based on the PCL code-sharing strategy. The name is up to you. Every time a new page is discussed, just clean the content of the MainPage.xaml and MainPage.xaml.cs files (except for the constructor) and write the new code.

Introducing and creating pages

Xamarin.Forms provides many page objects that you can use to set up the user interfaces of your applications. Pages are root elements in the visual hierarchy, and each page allows you to add only one visual element, typically a root layout with other layouts and visual elements nested inside the root. From a technical point of view, all the page objects in Xamarin.Forms derive from the abstract **`Page`** class, which provides the basic infrastructure of each page, including common properties such as **`Content`**, definitely the most important property that you assign with the root visual element. Table 7 describes available pages in Xamarin.Forms.

Table 7: Pages in Xamarin.Forms

Page Type	Description
`ContentPage`	Displays a single view object.
`TabbedPage`	Facilitates navigating among child pages using tabs.
`CarouselPage`	Facilitates using the swipe gesture among child pages.
`MasterDetailPage`	Manages two separate panes, which includes a flyout control.
`NavigationPage`	Provides the infrastructure for navigating among pages.

The next sections describe available pages in more detail. Remember that Visual Studio provides item templates for different page types, so you can right-click the PCL project in Solution Explorer, select **Add New Item**, and in the Add New Item dialog, you will see templates for each page described in Table 7.

Single views with the `ContentPage`

The **`ContentPage`** object is the simplest page possible and allows for displaying a single visual element. You already looked at some examples of the **`ContentPage`** previously, but it is worth mentioning its **`Title`** property. This property is particularly useful when the **`ContentPage`** is used in pages with built-in navigation, such as **`TabbedPage`** and **`CarouselPage`**, because it helps identify the active page. The core of the **`ContentPage`** is the **`Content`** property, which you assign with the visual element you want to display. The visual element can be either a single control or a layout; the latter allows you to create complex visual hierarchies and real-world user interfaces. In XAML, the tag for the **`Content`** property can be omitted, which is also common practice (also notice **`Title`**):

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <Label Text="A content page"/>

</ContentPage>
```

The **`ContentPage`** can be used individually or as the content of other pages discussed in the next sections.

Splitting contents with the `MasterDetailPage`

The **`MasterDetailPage`** is a very important page, since it allows you to split contents into two separate categories: generic and detailed. The user interface provided by the **`MasterDetailPage`** is very common in Android and iOS apps. It offers a flyout on the left (the master part) that you can swipe to show and hide it, and a second area on the right that displays more detailed content (the detail part). For example, a very common scenario for this kind of page is displaying a list of topics or settings in the master and the content for the selected topic or setting in the detail. Both the master and the detail parts are represented by **`ContentPage`** objects. A typical declaration for a **`MasterDetailPage`** looks like Code Listing 11.

Code Listing 11

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

```
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <MasterDetailPage.Master>
        <ContentPage>
            <Label Text="This is the Master" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </MasterDetailPage.Master>
    <MasterDetailPage.Detail>
        <ContentPage>
            <Label Text="This is the Details" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </MasterDetailPage.Detail>
</MasterDetailPage>
```

As you can see, you populate the **`Master`** and **`Detail`** properties with the appropriate **`ContentPage`** objects. In real-world apps, you might have a list of topics in the **`Master`**, then you might show details for a topic in the **`Detail`** when the user taps one in the **`Master`**'s content.

> ***Note: Every time you change the root page from `ContentPage` to another kind of page, such as `MasterDetailPage`, you also need to change the inheritance in the code-behind. For example, if you open the C# MainPage.xaml.cs file, you will see that `MainPage` inherits from `ContentPage`, but in XAML you replaced this object with `MasterDetailPage`. So, you also need to make `MainPage` inherit from `MasterDetailPage`. If you forget this, the compiler will report an error. This note is valid for the pages discussed in the next sections as well.***

Figures 38 and 39 show the master and detail parts, respectively. You can simply swipe from the left to enable the master flyout, and then swipe back to hide it. You can also control the flyout programmatically by assigning the **`IsPresented`** property with **`true`** (visible) or **`false`** (hidden). This is useful when the app is in landscape mode, because the flyout is automatically opened by default.

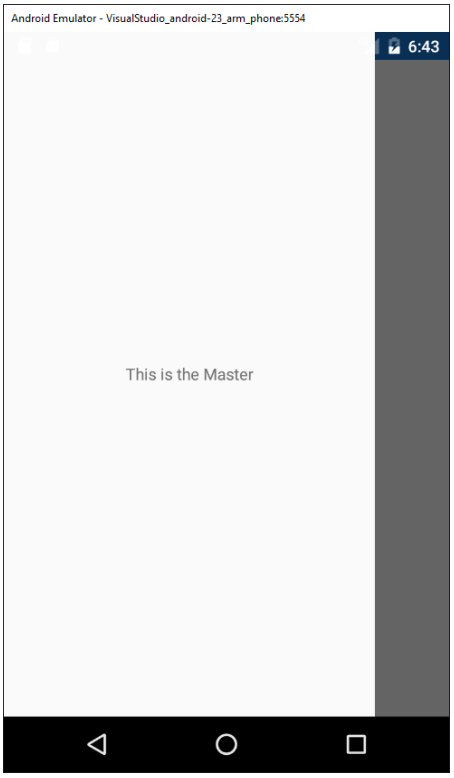

Figure 38: MasterDetailPage: The flyout of the master

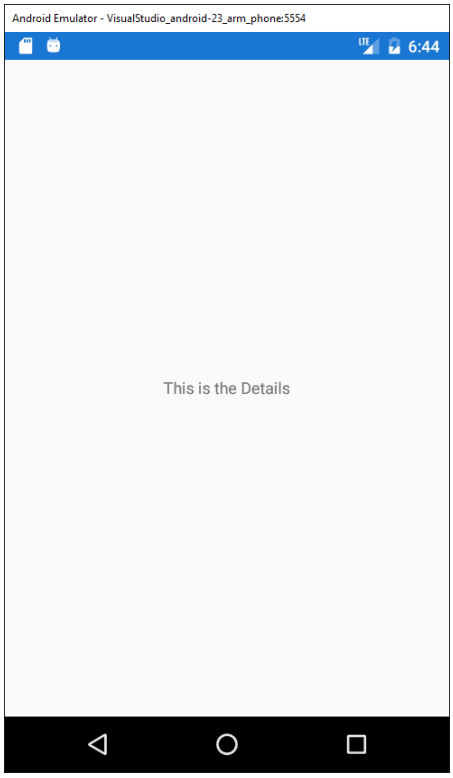

Figure 39: MasterDetailPage: The detail

Displaying content within tabs with the TabbedPage

Sometimes you might need to categorize multiple pages by topic, or by activity type. When you have a small amount of content, you can take advantage of the **`TabbedPage`**, which can group multiple **`ContentPage`** objects into tabs for easy navigation. The **`TabbedPage`** can be declared as shown in Code Listing 12.

Code Listing 12

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:local="clr-namespace:App1"
            Title="Main page"
            x:Class="App1.MainPage">

    <TabbedPage.Children>
        <ContentPage Title="First">
            <Label Text="This is the first page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
```

```
        </ContentPage>
        <ContentPage Title="Second">
            <Label Text="This is the second page" HorizontalOptions="Center
"
                   VerticalOptions="Center"/>
        </ContentPage>
        <ContentPage Title="Third">
            <Label Text="This is the third page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </TabbedPage.Children>
</TabbedPage>
```

As you can see, you populate the **`Children`** collection with multiple **`ContentPage`** objects. Providing a **`Title`** to each **`ContentPage`** is of primary importance, since the title's text is displayed in each tab, as demonstrated in Figure 40.

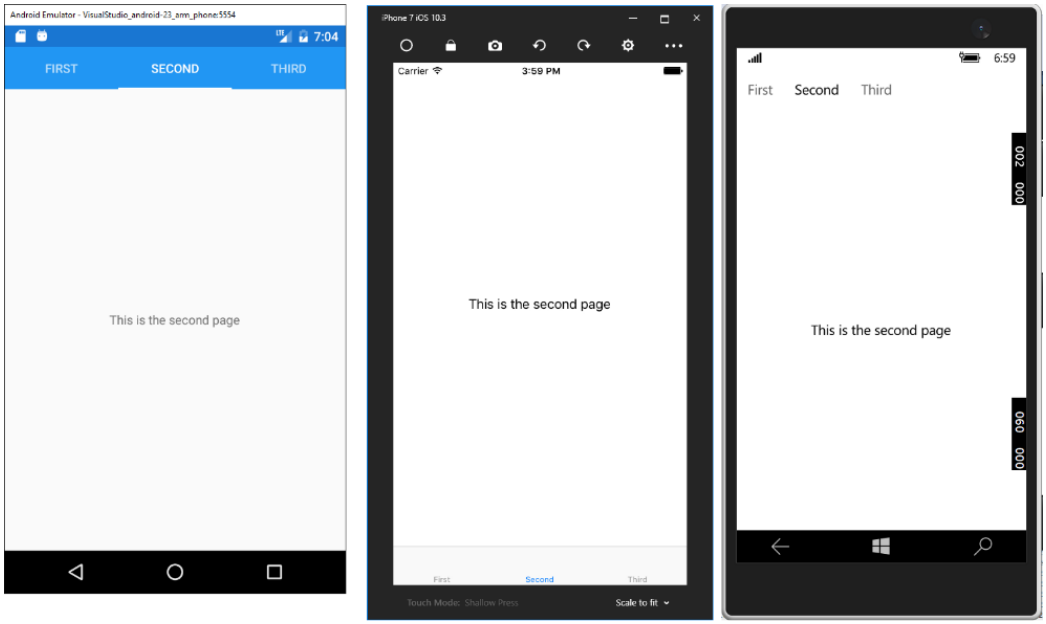

Figure 40: Displaying grouped contents with the TabbedPage

Of course, the **`TabbedPage`** works well with a small number of child pages, typically between three and four pages.

Swiping pages with the `CarouselPage`

The **`CarouselPage`** is similar to the **`TabbedPage`**, but instead of having tabs, you can use the swipe gesture to switch among child pages. For example, the **`CarouselPage`** could be perfect to display a gallery of pictures. Code Listing 13 shows how to declare a **`CarouselPage`**.

Code Listing 13

```
<?xml version="1.0" encoding="utf-8" ?>
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              xmlns:local="clr-namespace:App1"
              Title="Main page"
              x:Class="App1.MainPage">

    <CarouselPage.Children>
        <ContentPage Title="First">
            <Label Text="This is the first page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
        <ContentPage Title="Second">
            <Label Text="This is the second page" HorizontalOptions="Center
"
                   VerticalOptions="Center"/>
        </ContentPage>
        <ContentPage Title="Third">
            <Label Text="This is the third page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </CarouselPage.Children>
</CarouselPage>
```

Figure 41 shows how the **`CarouselPage`** appears.

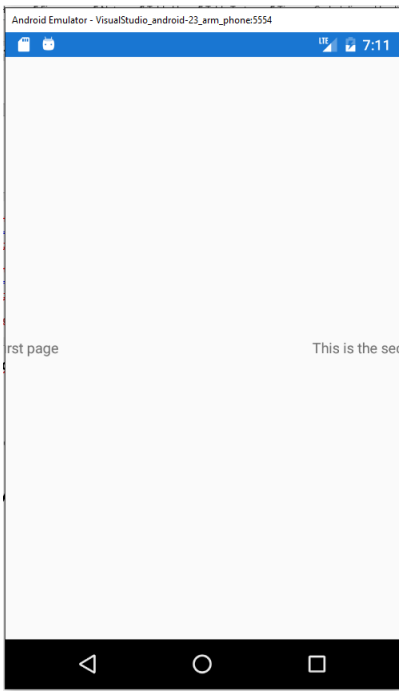

Figure 41: Swiping contents with the CarouselPage

Navigating among pages

Note: In the spirit of the* Succinctly *series, this section explains the most important concepts and topics of page navigation. However, there are tips and considerations that are specific to each platform that you have to know when dealing with navigation in Xamarin.Forms. With regard to this, do not miss checking out the official documentation.

Most mobile apps offer their content through multiple pages. In Xamarin.Forms, navigating among pages is very simple because of a built-in navigation framework. First of all, in Xamarin.Forms you leverage navigation features through the **NavigationPage** object. This kind of page must be instantiated, passing an instance of the first page in the stack of navigation to its constructor. This is typically done in the App.xaml.cs file, where you replace the assignment of the **MainPage** property with the following code:

```
public App()
{
    InitializeComponent();

    MainPage = new NavigationPage(new MainPage());
}
```

Wrapping a root page into a **NavigationPage** will not only enable the navigation stack, but will also enable the navigation bar on Android, iOS, and Windows desktop (but not on Windows 10 Mobile that relies on the hardware back button), whose text will be the value of the **Title** property of the current page object, represented by the **CurrentPage** read-only property. Now suppose you added another page of type **ContentPage** to the PCL project, called SecondaryPage.xaml. The content of this page is not important at this point, just set its **Title** property with some text. If you want to navigate from the first page to the second page, you use the **PushAsync** method as follows:

```
await Navigation.PushAsync(new SecondaryPage());
```

The **Navigation** property, exposed by each **Page** object, represents the navigation stack at the application level and provides methods for navigating between pages in a LIFO (last-in, first-out) approach. **PushAsync** navigates to the specified page instance; **PopAsync**, invoked from the current page, removes the current page from the stack and goes back to the previous page. Similarly, **PushModalAsync** and **PopModalAsync** allow you to navigate between pages modally. The following lines of code demonstrate this:

```
// removes SecondaryPage from the stack and goes back to the previous page

await Navigation.PopAsync();

// displays the specified page as a modal page

await Navigation.PushModalAsync(new SecondaryPage());

await Navigation.PopModalAsync();
```

Figure 42 shows how the navigation bar appears on Android and iOS when navigating to another page.

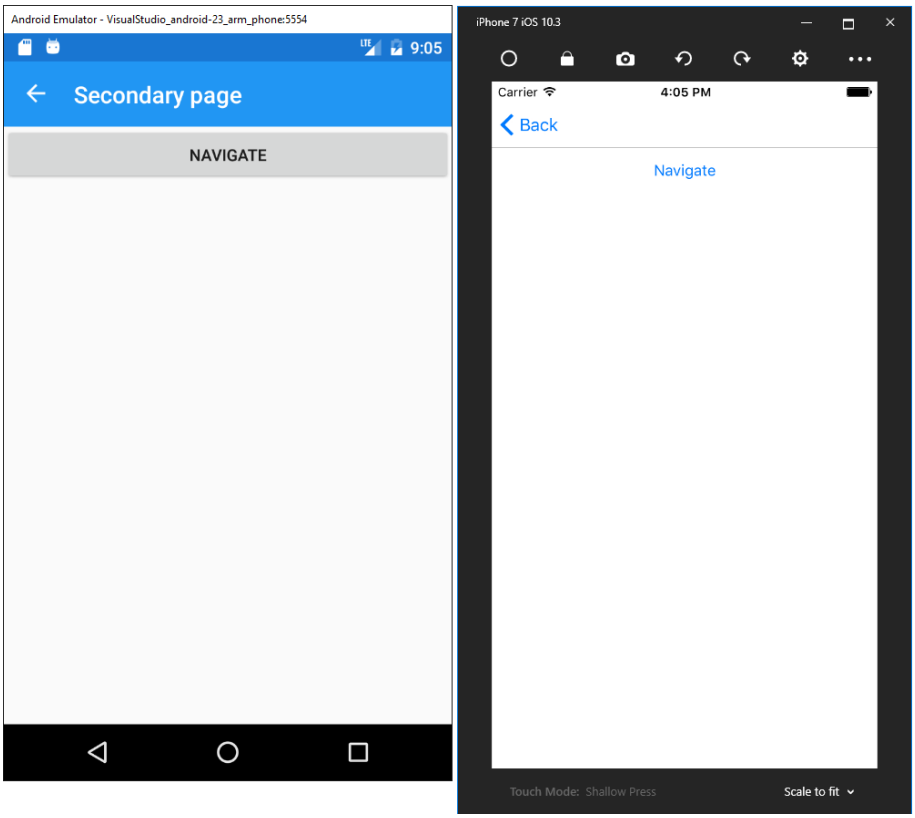

Figure 42: The navigation bar offered by the NavigationPage object

Users can simply touch the back button on the navigation bar to go back to the previous page. However, when you implement modal navigation, you cannot take advantage of the built-in navigation mechanism offered by the navigation bar, so it is your responsibility to implement code that allows going back to the previous page. Modal navigation can be useful if you must be able to intercept a tap on the back button on each platform. In fact, Android and Windows devices have a built-in hardware back button that you can manage with events, but iOS does not. In iOS, you only have the back button provided by the navigation bar, but this cannot be handled with any events. So, in this case, modal navigation can be a good option to intercept user actions.

Passing objects between pages

The need to exchange data between pages is not uncommon. You can change or overload a **Page**'s constructor and require a parameter of the desired type. Then, when you call **PushAsync** and pass the instance of the new page, you will be able to supply the argument that is necessary to the new page's constructor.

Animating transitions between pages

By default, the navigation includes an animation that makes the transition from one page to another nicer. However, you can disable animations by simply passing **`false`** as the argument of **`PushAsync`** and **`PushModalAsync`**.

Managing the page lifecycle

Every **`Page`** object exposes the **`OnAppearing`** and **`OnDisappearing`** events, raised right before the page is rendered and right before the page is removed from the stack, respectively. Their code looks like the following:

```
protected override void OnAppearing()
{
    // Replace with your code…
    base.OnAppearing();
}

protected override void OnDisappearing()
{
    // Replace with your code…
    base.OnDisappearing();
}
```

Actually, these events are not strictly related to navigation, since they are available to any page, including individual pages. However, it is with navigation that they become very important, especially when you need to execute some code at specific moments in the page lifecycle. For a better understanding of the flow, think of the page constructor: this is invoked the very first time a page is created. Then, **`OnAppearing`** is raised right before the page is rendered on screen. When the app navigates to another page, **`OnDisappearing`** is invoked, but this does not destroy the current page instance (and this makes perfect sense). When the app navigates back from the second page to the first page, this is not created again because it is still in the navigation stack, so its constructor will not be invoked, while **`OnAppearing`** will. So, within the **`OnAppearing`** method body, you can write code that will be executed every time the page is shown, while in the constructor, you can write code that will be executed only once.

Handling the hardware back button

Android devices and Windows phones have a built-in hardware back button that users can use instead of the back button in the navigation bar. You can detect if the user presses the hardware back button by handling the **`OnBackButtonPressed`** event as follows:

```
protected override bool OnBackButtonPressed()
{
    return base.OnBackButtonPressed(); // replace with your logic here…
}
```

Simply put your logic in the method body. The default behavior is to suspend the app, so you might want to override this with **`PopAsync`** to return to the previous page. This event does not intercept pressing the back button in the navigation bar, which implies it has no effect on iOS devices.

Chapter summary

This chapter introduced the available pages in Xamarin.Forms, explaining how you can display single-view content with the **`ContentPage`** object, group content into tabs with the **`TabbedPage`**, swipe content with the **`CarouselPage`**, and group contents into two categories with the **`MasterDetail`** page object. In the second part of the chapter, you looked at how the **`NavigationPage`** object provides a built-in navigation framework that not only displays a navigation bar, but also allows for navigating between pages programmatically. Finally, you looked at how the page lifecycle works, including the difference between page creation and page rendering. In the next chapter, you will look at information about two important and powerful features in Xamarin.Forms: resources and data binding.

Chapter 7 Resources and Data Binding

XAML is a very powerful declarative language and it shows all of its power with two particular scenarios: working with resources and data binding. If you have existing experience with platforms like WPF, Silverlight, and Universal Windows Platform, you will be familiar with the concepts described in this chapter. If this is your first time, you will immediately appreciate how XAML simplifies difficult things in both scenarios.

Working with resources

Generally speaking, in XAML-based platforms such as WPF, Silverlight, Universal Windows Platform, and Xamarin.Forms, resources are reusable pieces of information that you can apply to visual elements in the user interface. Typical XAML resources are styles, control templates, object references, and data templates. Xamarin.Forms supports styles and data templates, so these will be discussed in this chapter.

Tip: Resources in XAML are very different from resources in platforms such as Windows Forms, where you typically use .resx files to embed strings, images, icons, or files. My suggestion is that you should not make any comparison between XAML resources and other .NET resources.

Declaring resources

Every **`Page`** object and layout exposes a property called **`Resources`**, a collection of XAML resources that you can populate with one or more objects of type **`ResourceDictionary`**. A **`ResourceDictionary`** is a container of XAML resources such as styles, data templates, and object references. For example, you can add a **`ResourceDictionary`** to a page as follows:

```
<ContentPage.Resources>
   <ResourceDictionary>
       <!-- Add resources here -->
   </ResourceDictionary>
</ContentPage.Resources>
```

Resources have scope. This implies that resources you add to the page level are available to the whole page, whereas resources you add to the layout level are only available to the current layout, like in the following snippet:

```
<StackLayout.Resources>
   <ResourceDictionary>
       <!-- Resources are available only to this layout, not outside -->
   </ResourceDictionary>
</StackLayout.Resources>
```

Sometimes you might want to make resources available to the entire application. In this case, you can take advantage of the **App.xaml** file. The default code for this file is shown in Code Listing 14.

Code Listing 14

```
<?xml version="1.0" encoding="utf-8" ?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.App">
      <Application.Resources>

          <!-- Application resource dictionary -->
            <ResourceDictionary>

            </ResourceDictionary>
      </Application.Resources>
</Application>
```

As you can see, the autogenerated code of this file already contains an **Application.Resources** node with a nested **ResourceDictionary**. Resources you put inside this resource dictionary will be visible to any page, layout, and view in the application. Now that you have knowledge of where resources are declared and their scope, it is time to see how resources work, starting with styles. Other resources, such as data templates, will be discussed later in this chapter.

Introducing styles

When designing your user interface, in some situations, you might have multiple views of the same type and, for each of them, you might need to assign the same properties with the same values. For example, you might have two buttons with the same width and height, or two or more labels with the same width, height, and font settings. In such situations, instead of assigning the same properties many times, you can take advantage of styles. A style allows you to assign a set of properties to views of the same type. Styles must be defined inside a **ResourceDictionary** and they must specify the type they are intended for and an identifier. The following code demonstrates how to define a style for **Label** views:

```
<ResourceDictionary>
   <Style x:Key="labelStyle" TargetType="Label">
      <Setter Property="TextColor" Value="Green" />
      <Setter Property="FontSize" Value="Large" />
   </Style>
</ResourceDictionary>
```

You assign an identifier with the **x:Key** expression and the target type with **TargetType**, passing the type name for the target view. Property values are assigned with **Setter** elements, whose **Property** property represents the target property name and **Value** represents the property value. You then assign the style to **Label** views as follows:

```
<Label Text="Enter some text:" Style="{StaticResource labelStyle}"/>
```

A style is therefore applied by assigning the **Style** property on a view with an expression that encloses the **StaticResource** markup extension and the style identifier within curly braces. You can then assign the **Style** property on each view of that type instead of manually assigning the same properties every time. With styles, XAML supports both **StaticResource** and **DynamicResource** markup extensions. In the first case, if a style changes, the target view will not be updated with the refreshed style. In the second case, the view will be updated reflecting changes in the style.

Style inheritance

Styles support inheritance; therefore, you can create a style that derives from another style. For example, you can define a style that targets the abstract **View** type as follows:

```
<Style x:Key="viewStyle" TargetType="View">
    <Setter Property="HorizontalOptions" Value="Center" />
    <Setter Property="VerticalOptions" Value="Center" />
</Style>
```

This style can be applied to any view regardless of the concrete type. Then you can create a more specialized style using the **BasedOn** property as follows:

```
<Style x:Key="labelStyle" TargetType="Label"
       BasedOn="{StaticResource viewStyle}">
    <Setter Property="TextColor" Value="Green" />
</Style>
```

The second style targets **Label** views, but also inherits property settings from the parent style. Put succinctly, the **labelStyle** will assign the **HorizontalOptions**, **VerticalOptions**, and **TextColor** properties on the targeted **Label** views.

Implicit styling

Views' **Style** property allows the assigning of a style defined inside resources. This allows you to selectively assign the style only to certain views of a given type. However, if you want the same style to be applied to all of the views of the same type in the user interface, assigning the **Style** property to each view manually might be annoying. In this case, you can take advantage of the so-called *implicit styling*. This feature allows you to automatically assign a style to all the views of the type specified with the **TargetType** property without the need to set the **Style** property. To accomplish this, you simply avoid assigning an identifier with **x:Key**, like in the following example:

```
<Style TargetType="Label">
   <Setter Property="HorizontalOptions" Value="Center" />
   <Setter Property="VerticalOptions" Value="Center" />
   <Setter Property="TextColor" Value="Green" />
</Style>
```

Styles with no identifier will automatically be applied to all the **Label** views in the user interface (according to the scope of the containing resource dictionary) and you will not need to assign the **Style** property on the **Label** definitions.

Working with data binding

Data binding is a built-in mechanism that allows visual elements to communicate with data so that the user interface is automatically updated when data changes and vice versa. Data binding is available in all the most important development platforms, and Xamarin.Forms is no exception. In fact, its data binding engine relies on the power of XAML and the way it works is similar in all the XAML-based platforms. Xamarin.Forms supports binding an object to visual elements, a collection to visual elements, and visual elements to other visual elements. This chapter describes the first two scenarios. Because data binding is a very complex topic, the best way to start is with an example. Suppose you want to bind an instance of the following **Person** class to the user interface, so that a communication flow is established between the object and views:

```
public class Person
{
    public string FullName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string Address { get; set; }
}
```

In the user interface, you will want to allow the user to enter their full name, date of birth, and address via an **Entry**, a **DatePicker**, and another **Entry**, respectively. In XAML, this can be accomplished with the code shown in Code Listing 15.

Code Listing 15

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="20">
        <Label Text="Name:" />
        <Entry Text="{Binding FullName}"/>

        <Label Text="Date of birth:"/>
        <DatePicker Date="{Binding DateOfBirth, Mode=TwoWay}"/>

        <Label Text="Address:"/>
        <Entry Text="{Binding Address}"/>
    </StackLayout>
</ContentPage>
```

As you can see, the **Text** property for **Entry** views and the **Date** property of the **DatePicker** have a markup expression as their value. Such an expression is made up of the **Binding** literal followed by the property you want to bind from the data object. Actually, the expanded form of this syntax could be **{Binding Path=PropertyName}** but **Path** can be omitted. Data binding can be of four types:

- **TwoWay**: Views can read and write data.
- **OneWay**: Views can only read data.
- **OneWayToSource**: Views can only write data.
- **Default**: Xamarin.Forms resolves the appropriate mode automatically, based on the view (see the explanation that follows).

TwoWay and **OneWay** are the most-used modes, and in most cases you do not need to specify the mode explicitly because Xamarin.Forms automatically resolves the appropriate mode based on the view. For example, binding in the **Entry** control is **TwoWay** because this kind of view can be used to read and write data, whereas binding in the **Label** control is **OneWay** because this view can only read data. However, with the **DatePicker**, you need to explicitly set the binding mode, so you use the following syntax:

```
<DatePicker Date="{Binding DateOfBirth, Mode=TwoWay}"/>
```

Views' properties that are bound to an object's properties are known as *bindable properties* (or dependency properties if you come from the WPF or UWP world).

Tip: Bindable properties are very powerful but a bit more complex in the architecture. In this chapter, I'm going to explain how to use them, but for further details about their implementation and how you can use them in your custom objects, you can refer to the official documentation.

Bindable properties will automatically update the value of the bound object's property and will automatically refresh their value in the user interface if the object is updated. However, this automatic refresh is possible only if the data-bound object implements the **INotifyPropertyChanged** interface, which allows an object to send change notifications. As a consequence, you must extend the **Person** class definition as shown in Code Listing 16.

Code Listing 16

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace App1
{
    public class Person : INotifyPropertyChanged
    {
        private string fullName;
        public string FullName
        {
            get
```

```
            {
                return fullName;
            }
            set
            {
                fullName = value;
                OnPropertyChanged();
            }
        }

        private DateTime dateOfBirth;

        public DateTime DateOfBirth
        {
            get
            {
                return dateOfBirth;
            }
            set
            {
                dateOfBirth = value;
                OnPropertyChanged();
            }
        }
        private string address;
        public string Address
        {
            get
            {
                return address;
            }
            set
            {
                address = value;
                OnPropertyChanged();
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged([CallerMemberName] string propertyNa
me
                                       = null)
        {
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

By implementing **INotifyPropertyChanged**, property setters can raise a change notification via the **PropertyChanged** event. Bound views will be notified of any changes and will refresh their contents.

Tip: With the `CallerMemberName` attribute, the compiler automatically resolves the name of the caller member. This avoids the need to pass the property name in each setter and helps keep code much cleaner.

The next step is binding an instance of the **Person** class to the user interface. This can be accomplished with the following lines of code, normally placed inside the page's constructor or in its **OnAppearing** event handler:

```
Person person = new Person();
this.BindingContext = person;
```

Pages and layouts expose the **BindingContext** property, of type **object**, that represents the data source for the page or layout and is the same as **DataContext** in WPF or UWP. Child views that are data bound to an object's properties will search for an instance of the object in the **BindingContext** property value and bind to properties from this instance. In this case, the **Entry** and the **DatePicker** will search for an object instance inside **BindingContext** and they will bind to properties from that instance. Remember that XAML is case-sensitive, so binding to **FullName** is different from binding to **Fullname**. The runtime will throw an exception if you try to bind to a property that does not exist or has a different name. If you now try to run the application, not only will data binding work, but the user interface will also be automatically updated if the data source changes. You may think of binding views to a single object instance, like in the previous example, as binding to a row in a database table.

Working with collections and with the `ListView`

Though working with a single object instance is a common scenario, another very common situation is working with collections that you display as lists in the user interface. Xamarin.Forms supports data binding over collections via the **ObservableCollection<T>** object. This collection works exactly like the **List<T>**, but it also raises a change notification when items are added to or removed from the collection. Collections are very useful, for example, when you want to represent rows in a database table. For example, suppose you have the following collection of **Person** objects:

```
Person person1 = new Person { FullName = "Alessandro" };
Person person2 = new Person { FullName = "James" };
Person person3 = new Person { FullName = "Jacqueline" };
var people = new ObservableCollection<Person>() { person1, person2,
             person3 };

this.BindingContext = people;
```

The code assigns the collection to the **BindingContext** property of the root container, but at this point, you need a visual element that is capable of displaying the content of this collection. This is where the **ListView** control comes in. The **ListView** can receive the data source from either the **BindingContext** of its container or by assigning its **ItemsSource** property, and any object that implements the **IEnumerable** interface can be used with the **ListView**. You will typically assign **ItemsSource** directly if the data source for the **ListView** is not the same data source as for the other views in the page.

The problem to solve with the **ListView** is that it does not know how to present objects in a list. For example, think of the people collection that contains instances of the **Person** class. Each instance exposes the **FullName**, **DateOfBirth**, and **Address** properties, but the **ListView** does not know how to present these properties, so it is your job to explain to it how. This is accomplished with the so-called *data templates*. A data template is a static set of views that are bound to properties in the object. It instructs the **ListView** on how to present items. Data templates in Xamarin.Forms rely on the concept of cells. Cells can display information in a specific way and are summarized in Table 8.

Table 8: Cells in Xamarin.Forms

Cell Type	Description
TextCell	Displays two labels, one with a description and one with a data-bound text value.
EntryCell	Displays a label with a description and an **Entry** with a data-bound text value. It also allows a placeholder to be displayed.
ImageCell	Displays a label with a description and an **Image** control with a data-bound image.
SwitchCell	Displays a label with a description and a **Switch** control bound to a **bool** value.
ViewCell	Allows for creating custom data templates.

Tip: Labels within cells are also bindable properties.

For example, if you only had to display and edit the **FullName** property, you could write the following data template:

```
<Grid>
    <ListView x:Name="PeopleList" ItemsSource="{Binding}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <EntryCell Label="Full name:" Text="{Binding FullName}"/>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</Grid>
```

Tip: The `DataTemplate` definition is always defined inside the `ListView.ItemTemplate` element.

As a general rule, if the data source is assigned to the **`BindingContext`** property, the **`ItemsSource`** must be set with the **`{Binding}`** value, which means your data source is the same as that of your parent. With this code, the **`ListView`** will display all the items in the bound collection, showing two cells for each item. However, each **`Person`** also exposes a property of type **`DateTime`**, and no cell is suitable for that. In such situations, you can create a custom cell using the **`ViewCell`**, as shown in Code Listing 17.

Code Listing 17

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page" Padding="20"
             x:Class="App1.MainPage">

    <StackLayout>
        <ListView x:Name="PeopleList" ItemsSource="{Binding}"
                  HasUnevenRows="True">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <ViewCell.View>
                            <StackLayout Margin="10">
                                <Label Text="Full name:"/>
                                <Entry Text="{Binding FullName}"/>
                                <Label Text="Date of birth:"/>
                                <DatePicker Date="{Binding DateOfBirth,
                                            Mode=TwoWay}"/>
                                <Label Text="Address:"/>
                                <Entry Text="{Binding Address}"/>
                            </StackLayout>
                        </ViewCell.View>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

As you can see, the **`ViewCell`** allows you to create custom and complex data templates, contained in the **`ViewCell.View`** property, so that you can display whatever kind of information you need. Notice the **`HasUnevenRows`** property: if **`true`** on Android and Windows, this dynamically resizes a cell's height based on its content. On iOS, this property must be set to

false and you must provide a fixed row height by setting the **RowHeight** property. In Chapter 8 you will learn how to take advantage of the **OnPlatform** object to make UI decisions based on the platform.

Tip: The `ListView` is a very powerful and versatile view, and there is much more to it, such as interactivity, grouping and sorting, and customizations. I strongly recommend you read the official documentation on the `ListView` and this article that describes how to improve performance, which is extremely useful with Android.

Figure 43 shows the result for the code described in this section. Notice that the **ListView** includes built-in scrolling capability and must never be enclosed within a **ScrollView**.

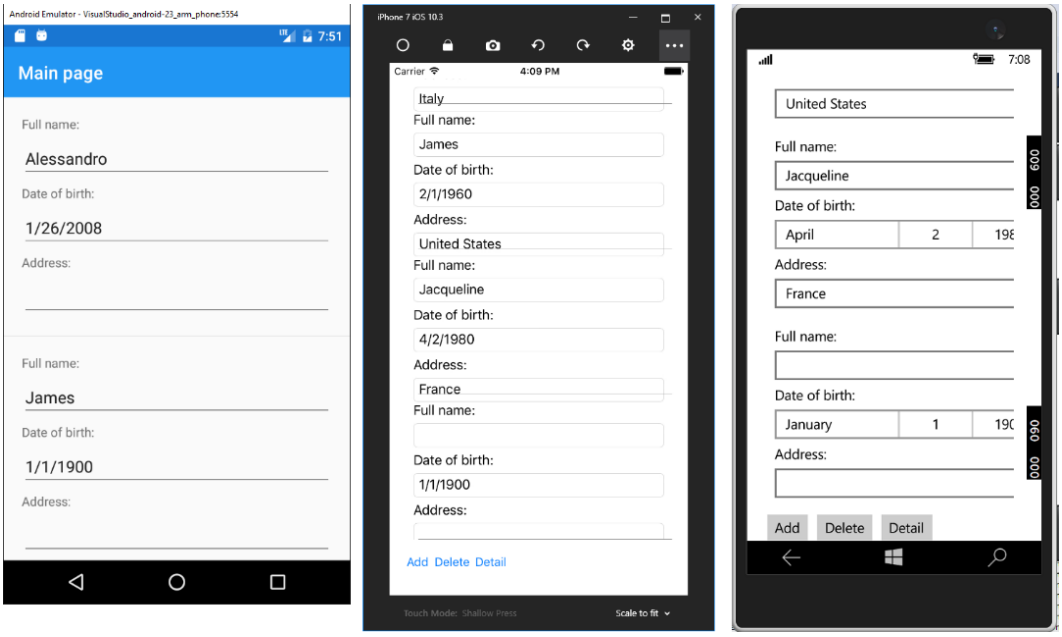

Figure 43: A data-bound ListView

A data template can be placed inside the page or app resources so that it becomes reusable. Then you assign the **ItemTemplate** property in the **ListView** definition with the **StaticResource** expression, as shown in Code Listing 18.

Code Listing 18

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <DataTemplate x:Key="MyTemplate">
```

```
                <ViewCell>
                    <ViewCell.View>
                        <StackLayout Margin="10" Orientation="Vertical"
                                     Padding="10">
                            <Label Text="Full name:"/>
                            <Entry Text="{Binding FullName}"/>
                            <Label Text="Date of birth:"/>
                            <DatePicker Date="{Binding DateOfBirth,Mode=Two
Way}"/>
                            <Label Text="Address:"/>
                            <Entry Text="{Binding Address}"/>
                        </StackLayout>
                    </ViewCell.View>
                </ViewCell>
            </DataTemplate>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ListView x:Name="PeopleList" VerticalOptions="FillAndExpand"
              HasUnevenRows="True" ItemTemplate="{StaticResource MyTemplate
}"/>
</ContentPage>
```

Working with the `TableView`

When you need to present a list of settings, data in a form, or data that is different from row to row, you can consider the **`TableView`** control. The **`TableView`** is based on sections and can display content through the same cells described previously. With this view, you need to specify a value for its **`Intent`** property, which basically represents the type of information you need to display. Possible values are **`Settings`** (list of settings), **`Data`** (to display data entries), **`Form`** (when the table view acts like a form), and **`Menu`** (to present a menu of selections). Code Listing 19 provides an example.

Code Listing 19

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <ContentPage.Content>
        <TableView Intent="Settings">
            <TableRoot>
                <TableSection Title="Network section">
                    <SwitchCell Text="Allowed" On="True"/>
                </TableSection>
```

```
                <TableSection Title="Push notifications">
                    <SwitchCell Text="Allowed" On="True"/>
                </TableSection>
            </TableRoot>
        </TableView>
    </ContentPage.Content>
</ContentPage>
```

You can divide the **TableView** into multiple **TableSection**s. Each contains a cell to display the required type of information, and, of course, you can use a **ViewCell** for a custom, more complex template. Figure 44 shows an example of **TableView** based on the previous listing.

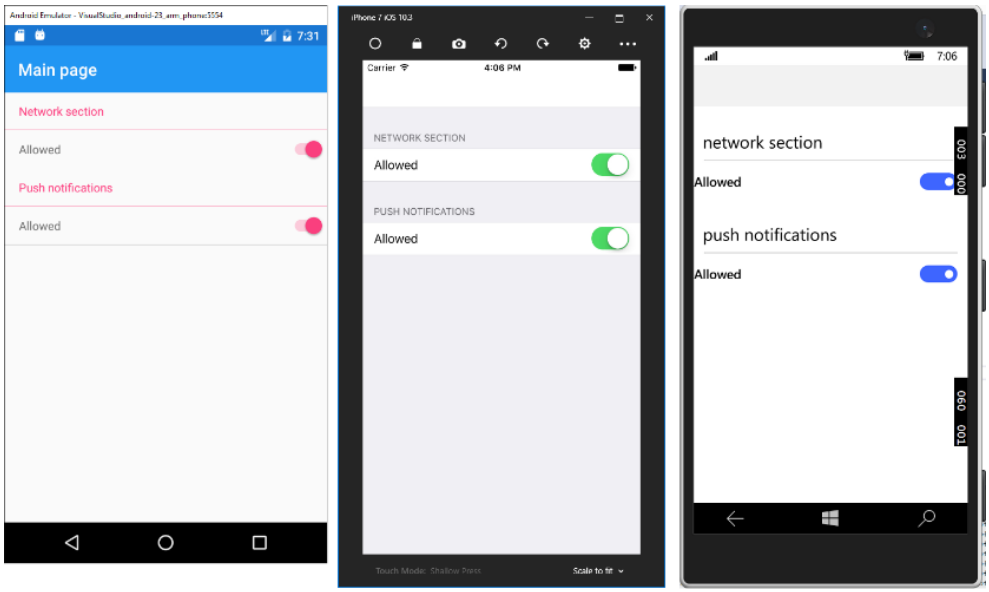

Figure 44: A TableView in action

Obviously, you can bind cell properties to objects rather than setting their value explicitly like in the previous example.

Showing and selecting values with the `Picker` view

With mobile apps, it is common to provide the user an option to select an item from a list of values, which can be accomplished with the **Picker** view. Xamarin.Forms 2.3.4 has introduced data binding support in the **Picker**. You can now easily bind a **List<T>** or **ObservableCollection<T>** to its **ItemsSource** property and retrieve the selected item via its **SelectedItem** property. For example, suppose you have the following **Fruit** class:

```
public class Fruit
{
    public string Name { get; set; }
    public string Color { get; set; }
}
```

Now, in the user interface, suppose you want to ask the user to select a fruit from a list with the XAML shown in Code Listing 20.

Code Listing 20

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App2"
             x:Class="App2.MainPage">

    <ContentPage.Content>

        <StackLayout VerticalOptions="FillAndExpand">
            <Label Text="Select your favorite fruit:"/>
            <Picker x:Name="FruitPicker" ItemDisplayBinding="{Binding Name}
"
                    SelectedIndexChanged="FruitPicker_SelectedIndexChanged"
/>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

As you can see, the **`Picker`** exposes the **`SelectedIndexChanged`** event, which is raised when the user selects an item. With the **`ItemDisplayBinding`**, you specify what property from the bound object it needs to display: in this case, the fruit name. The **`ItemsSource`** property can, instead, be assigned either in XAML or in the code-behind. In this case, a collection can be assigned in C#, as demonstrated in Code Listing 21.

Code Listing 21

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        var apple = new Fruit { Name = "Apple", Color = "Green" };
        var strawberry = new Fruit { Name = "Strawberry", Color = "Red" };
        var orange = new Fruit { Name = "Orange", Color = "Orange" };

        var fruitList = new ObservableCollection<Fruit>()
            { apple, strawberry, orange };
        this.FruitPicker.ItemsSource = fruitList;
    }

    private async void FruitPicker_SelectedIndexChanged(object sender,
                                                        EventArgs e)
    {
```

```
        var currentFruit = this.FruitPicker.SelectedItem as Fruit;
        if (currentFruit != null)
            await DisplayAlert("Selection",
            $"You selected {currentFruit.Name}", "OK");
    }
}
```

Like the same-named property in the **`ListView`**, **`ItemsSource`** is of type **`object`** and can bind to any object that implements the **`IEnumerable`** interface. Notice how you can retrieve the selected item handling the **`SelectedIndexChanged`** event and casting the **`Picker.SelectedItem`** property to the type you expect. In such situations, it is convenient to use the **`as`** operator, which returns null if the conversion fails, instead of an exception. Figure 45 shows how the user can select an item from the picker.

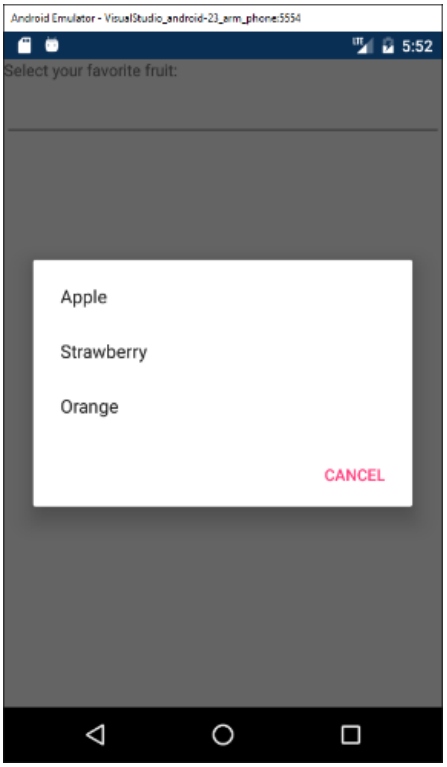

Figure 45: Selecting items with a Picker

Actually, data binding support was added to the **`Picker`** only with Xamarin.Forms 2.3.4. In previous versions, you could only manually populate its **`Items`** property via the **`Add`** method, and then handle indices. This is the real reason why the **`SelectedIndexChanged`** event exists, but it is still useful with the new approach. Data binding a list to the **`Picker`** is very common, but you can certainly still populate the list manually and handle the index.

Binding images

Displaying images in data-binding scenarios is very common and Xamarin.Forms makes it easy. You simply need to bind the **`Image.Source`** property to an object of type **`ImageSource`** or to a URL that can be represented by both a **`string`** and a **`Uri`**. For example, suppose you have a class with a property that stores the URL of an image as follows:

```
public class Picture
{
      public Uri PictureUrl { get; set; }
}
```

When you have an instance of this class, you can assign the **`PictureUrl`** property:

```
var picture1 = new Picture();
picture1.PictureUrl = new Uri("http://mystorage.com/myimage.jpg");
```

Supposing you have an **`Image`** view in your XAML code and a **`BindingContext`** assigned with an instance of the class, data binding would work as follows:

```
<Image Source="{Binding PictureUrl}"/>
```

XAML has a type converter for the **`Image.Source`** property, so it automatically resolves strings and **`Uri`** instances into the appropriate type.

Hints for value converters

The last sentence of the previous section about image binding highlights the existence of type converters that resolve specific types into the appropriate type for the **`Image.Source`** property. This actually happens with many other views and types. For example, if you bind an integer value to the **`Text`** property of an **`Entry`** view, such an integer is converted into a string by a XAML type converter. However, there are situations in which you might want to bind objects that XAML type converters cannot automatically convert into the type a view expects. For example, you might want to bind a **`Color`** value to a **`Label`**'s **`Text`** property, which is not possible out of the box. In these cases, you can create *value converters*. A value converter is a class that implements the **`IValueConverter`** interface and that must expose the **`Convert`** and **`ConvertBack`** methods. **`Convert`** translates the original type into a type that the view can receive, while **`ConvertBack`** does the opposite. Code Listing 22 shows an example of a value converter that converts a string containing HTML markup into an object that can be bound to the **`WebView`** control. **`ConvertBack`** is not implemented because this value converter is supposed to be used in a read-only scenario and therefore a round-trip conversion is not required.

Code Listing 22

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
using Xamarin.Forms;

namespace App1
{
    public class HtmlConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
               object parameter, CultureInfo culture)
        {
            try
            {
                var source = new HtmlWebViewSource();
                string originalValue = (string)value;

                source.Html = originalValue;
                return source;
            }
            catch (Exception)
            {
                return value;
            }
        }

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

Both methods always receive the data to convert as object instances, and then you need to cast the object into a specialized type for manipulation. In this case, **`Convert`** creates an **`HtmlWebViewSource`** object, converts the received **`object`** into a **`string`**, and populates the **`Html`** property with the string that contains the HTML markup. The value converter must then be declared in the resources of the XAML file where you wish to use it (or in App.xaml). Code Listing 23 provides an example that also shows how to use the value converter.

Code Listing 23

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <ContentPage.Resources>
```

```
        <local:HtmlConverter x:Key="HtmlConverter"/>
    </ContentPage.Resources>

    <!-- Assumes you have a data-bound .NET object that exposes
    a property called HtmlContent -->
    <WebView Source="{Binding HtmlContent,
     Converter={StaticResource HtmlConverter}}"/>
</ContentPage>
```

You declare the converter as you would do with any other resource. Then your binding will also contain the **`Converter`** expression, which points to the value converter with the typical syntax you used with other resources.

Introducing Model-View-ViewModel

Model-View-ViewModel (MVVM) is an architectural pattern used in XAML-based platforms that allows for clean separation between the data (model), the logic (view model), and the user interface (view). With MVVM, pages only contain code related to the user interface, they strongly rely on data binding, and most of the work is done in the view model. MVVM can be quite complex if you have never seen it before, so I will try to simplify the explanations as much as possible, but you should use Xamarin's MVVM documentation as a reference. Let's start with a simple example and with a fresh Xamarin.Forms solution based on the PCL code-sharing strategy. Imagine you want to work with a list of **`Person`** objects. This is your model and you can reuse the **`Person`** class from before. Add a new folder called **Model** to your project, and add a new **Person.cs** class file to this folder, pasting in the code of the **`Person`** class. Next, add a new folder called **ViewModel** to the project and add a new class file called **PersonViewModel.cs**. Before writing the code for it, let's summarize some important considerations:

- The view model contains the business logic, acts like a bridge between the model and the view, and exposes properties to which the view can bind.
- Among such properties, one will certainly be a collection of **`Person`** objects.
- In the view model, you can load data, filter data, execute save operations, and query data.

Loading, filtering, saving, and querying data are examples of actions a view model can execute against data. In a classic development approach, you would handle **`Clicked`** events on **`Button`** views and write the code that executes an action. However, in MVVM, views should only contain code related to the user interface, not code that executes actions against data. So, in MVVM, view models expose the so-called *commands*. A command is a property of type **`ICommand`** that can be data bound to views such as **`Button`**, **`SearchBar`**, **`ListView`**, and **`TapGestureRecognizer`** objects. In the UI, you bind a view to a command in the view model. In this way, the action is executed in the view model instead of in the view's code behind. Code Listing 24 shows the **`PersonViewModel`** class definition.

Code Listing 24

```
using MvvmSample.Model;
using System;
```

```
using System.Collections.ObjectModel;
using System.Windows.Input;
using Xamarin.Forms;

namespace MvvmSample.ViewModel
{
    public class PersonViewModel
    {
        public ObservableCollection<Person> People { get; set; }
        public Person SelectedPerson { get; set; }

        public ICommand AddPerson { get; set; }
        public ICommand DeletePerson { get; set; }

        public PersonViewModel()
        {
            this.People = new ObservableCollection<Person>();

            // sample data
            Person person1 =
                new Person { FullName = "Alessandro",
                    Address ="Italy",
                    DateOfBirth =new DateTime(1977,5,10) };
            Person person2 =
                new Person { FullName = "James",
                    Address ="United States",
                    DateOfBirth =new DateTime(1960,2,1) };
            Person person3 =
                new Person { FullName = "Jacqueline",
                    Address ="France",
                    DateOfBirth =new DateTime(1980,4,2) };

            this.People.Add(person1);
            this.People.Add(person2);
            this.People.Add(person3);

            this.AddPerson =
                new Command(() => this.People.Add(new Person()));

            this.DeletePerson =
                new Command<Person>((person) => this.People.Remove(person))
;
        }
    }
}
```

The **People** and **SelectedPerson** properties expose a collection of **Person** objects and a single **Person**, respectively, and the latter will be bound to the **SelectedItem** property of a **ListView**, as you will see shortly. Notice how properties of type **ICommand** are assigned with instances of the **Command** class, to which you can pass an **Action** delegate via a lambda expression that executes the desired operation. The **Command** provides an out-of-the-box implementation of the **ICommand** interface and its constructor can also receive a parameter, in which case you must use its generic overload (see **DeletePerson** assignment). In that case, the **Command** works with objects of type **Person** and the action is executed against the received object. Commands and other properties are data bound to views in the user interface.

Note: Here I demonstrated the most basic use of commands. However, commands also expose a `CanExecute` Boolean method that determines whether an action can be executed or not. Additionally, you can create custom commands that implement `ICommand` and must explicitly implement the `Execute` and `CanExecute` methods, where `Execute` is invoked to run an action. For further details, look at the official documentation.

Now it is time to write the XAML code for the user interface. Code Listing 25 shows how to use a **ListView** for this and how to bind two **Button** views to commands.

Code Listing 25

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MvvmSample"
             x:Class="MvvmSample.MainPage" Padding="20">

    <StackLayout>
        <ListView x:Name="PeopleList"
                  ItemsSource="{Binding People}"
                  HasUnevenRows="True"
                  SelectedItem="{Binding SelectedPerson}">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <ViewCell.View>
                            <StackLayout Margin="10">
                                <Label Text="Full name:"/>
                                <Entry Text="{Binding FullName}"/>
                                <Label Text="Date of birth:"/>
                                <DatePicker Date="{Binding DateOfBirth,
                                            Mode=TwoWay}"/>
                                <Label Text="Address:"/>
                                <Entry Text="{Binding Address}"/>
                            </StackLayout>
                        </ViewCell.View>
                    </ViewCell>
                </DataTemplate>
```

```
            </ListView.ItemTemplate>
        </ListView>
        <StackLayout Orientation="Horizontal">
            <Button Text="Add" Command="{Binding AddPerson}"/>
            <Button Text="Delete" Command="{Binding DeletePerson}"
                    CommandParameter="{Binding Source={x:Reference PeopleLi
st},
                    Path=SelectedItem}"/>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

Tip: Remember to set `HasUnevenRows` to `false` and to provide a `RowHeight` for the `ListView` on iOS.

The **`ListView`** is very similar to the example shown when introducing data binding to collections. However, notice how:

- The **`ListView.ItemsSource`** property is bound to the **`People`** collection in the view model.
- The **`ListView.SelectedItem`** property is bound to the **`SelectedPerson`** property in the view model.
- The first **`Button`** is bound to the **`AddPerson`** command in the view model.
- The second **`Button`** is bound to the **`DeletePerson`** command, and it passes the selected **`Person`** object in the **`ListView`** with a special binding expression: **`Source`** represents the data source, in this case the **`ListView`**, referred to with **`x:Reference`**; **`Path`** points to the property in the source that exposes the object you want to pass to the command as a parameter (simply referred to as command parameter).

The final step is creating an instance of the view model and assigning it to the **`BindingContext`** of the page, which you can do in the page code-behind, as demonstrated in Code Listing 26.

Code Listing 26

```
using MvvmSample.ViewModel;
using Xamarin.Forms;

namespace MvvmSample
{
    public partial class MainPage : ContentPage
    {
        // Not using a field here because properties
        // are optimized for data binding.
        private PersonViewModel ViewModel { get; set; }

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

```
            this.ViewModel = new PersonViewModel();
            this.BindingContext = this.ViewModel;
        }
    }
}
```

If you now run the application (see Figure 46), you will see the list of **Person** objects, you will be able to use the two buttons, and the real benefit is that the whole logic is in the view model. With this approach, if you change the logic in the properties or in the commands, you will not need to change the page code. In Figure 46, you can see a new **Person** object added via command binding.

Figure 46: Showing a list of people and adding a new person with MVVM

MVVM is very powerful, but real-world implementations can be very complex. For example, if you want to navigate to another page and you have commands, the view model should contain code related to the user interface (launching a page) that does not adhere to the principles of MVVM. Obviously, there are solutions to this problem that require further knowledge of the pattern, so I recommend you look at books and articles on the Internet for further study. Additionally, my recommendation is not to reinvent the wheel: many robust and popular MVVM libraries already exist and you might want to choose one from among the following:

- Prism
- MVVM Light Toolkit
- FreshMvvm
- MvvmCross

I have personally worked with FreshMvvm, but all the aforementioned alternatives are powerful enough to save you a lot of time.

Chapter summary

XAML plays a fundamental role in Xamarin.Forms and allows for defining reusable resources and for data-binding scenarios. Resources are reusable styles, data templates, and references to objects you declare in XAML. In particular, styles allow you to set the same properties to all views of the same type and they can extend other styles with inheritance. XAML also includes a powerful data-binding engine that allows you to quickly bind objects to visual elements in a two-way communication flow.

In this chapter, you have seen how to bind both a single object and a collection to individual visual elements and to the **`ListView`**, respectively. You have seen how to define data templates so that the **`ListView`** can have knowledge of how items must be presented, and you have learned about value converters, special objects that come in to help when you want to bind objects of a type that is different from the type a view supports. In the second part of the chapter, you walked through an introduction to the Model-View-ViewModel pattern, focusing on separating the logic from the UI and understanding new objects and concepts such as commands. So far, you have only worked with objects and views that Xamarin.Forms offers out of the box, but more often than not, you will need to implement more advanced features that require native APIs. This what you will learn in the next chapter.

Chapter 8 Accessing Platform-Specific APIs

Until now, you have seen what Xamarin.Forms offers in terms of features that are available on each supported platform, walking through pages, layouts, and controls that expose properties and capabilities that will certainly run on Android, iOS, and Windows. Though this simplifies cross-platform development, it is not enough to build real-world mobile applications. In fact, more often than not, mobile apps need to access sensors, the file system, the camera, and the network; send push notifications; and more. Each operating system manages these features with native APIs that cannot be shared across platforms and, therefore, that Xamarin.Forms cannot map into cross-platform objects.

However, if Xamarin.Forms did not provide a way to access native APIs, it would not be very useful. Luckily enough, Xamarin.Forms provides multiple ways to access platform-specific APIs that you can use to access practically everything of each platform. Thus, there is no limit to what you can do with Xamarin.Forms. In order to access platform features, you will need to write C# code in each platform project. This is what this chapter explains, together with all the options you have to access iOS, Android, and Windows APIs from your shared codebase.

The `Device` class and the `OnPlatform` method

The **`Xamarin.Forms`** namespace exposes an important class called **`Device`**. This class allows you to detect the platform your app is running on and the device idiom (tablet, phone, desktop). This class is particularly useful when you need to adjust the user interface based on the platform. The following code demonstrates how to take advantage of the **`Device.RuntimePlatform`** property to detect the running platform and to make UI-related decisions based on its value:

```
// Label1 is a Label view in the UI
switch(Device.RuntimePlatform)
{
    case Device.iOS:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Large, Label1);
        break;
    case Device.Android:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Medium, Label1);
        break;
    case Device.WinPhone:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Medium, Label1);
        break;
    case Device.Windows:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Large, Label1);
        break;
}
```

RuntimePlatform is of type **string** and can be easily compared against specific constants called **iOS**, **Android**, **WinPhone**, and **Windows** that represent, with self-explanatory names, the supported platforms. The **GetNamedSize** method automatically resolves the **Default**, **Micro**, **Small**, **Medium**, and **Large** platform font size and returns the corresponding **double**, which avoids the need to supply numeric values that would be different for each platform. The **Device.Idiom** property allows you to determine if the current device the app is running on is a phone, tablet, or desktop PC (UWP only), and returns one of the values from the **TargetIdiom** enumeration:

```
switch(Device.Idiom)
{
    case TargetIdiom.Desktop:
        // UWP desktop
        break;
    case TargetIdiom.Phone:
        // Phones
        break;
    case TargetIdiom.Tablet:
        // Tablets
        break;
    case TargetIdiom.Unsupported:
        // Unsupported devices
        break;
}
```

You can also decide how to adjust UI elements based on the platform and idiom in XAML. Code Listing 27 demonstrates how to adjust the **Padding** property of a page, based on the platform.

Code Listing 27

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:views="clr-namespace:App1.Views"
             x:Class="App1.Views.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                iOS="0, 20, 0, 0"
                Android="0, 10, 0, 0"
                WinPhone="0, 10, 0, 0" />
    </ContentPage.Padding>
</ContentPage>
```

With the **OnPlatform** tag, you can specify a different property value based on the **iOS**, **Android**, and **WinPhone** platforms. The property value depends on the **x:TypeArguments** attribute, which represents the .NET type for the property, **Thickness** in this particular case. Similarly, you can work with **OnIdiom** and the **TargetIdiom** enumeration in XAML, as well.

Tip: In iOS, it is best practice to set a page padding of 20 from the top, like in the previous snippet. The reason is that if you do not do it, your page will overlap the system bar.

Working with the dependency service

Most of the time, mobile apps need to offer interaction with the device hardware, sensors, system apps, and file system. Accessing these features from shared code is not possible, because their APIs have unique implementations on each platform. However, Xamarin.Forms provides a simple solution to this problem that relies on the service locator pattern: in the shared project, you write an interface that defines the required functionalities, then inside each platform project, you write classes that implement the interface through native APIs. Finally, you use the **`DependencyService`** class and its **`Get`** method to retrieve the proper implementation based on the platform your app is running on. For example, suppose your app needs to work with SQLite local databases. Assuming you have installed the SQLite-Net-Pcl NuGet package in your solution, in the PCL project, you can write the following sample interface called **`IDatabaseConnection`** that defines the signature of a method that must return the database path:

```
public interface IDatabaseConnection
{
    SQLite.SQLiteConnection DbConnection();
}
```

Tip: A complete walkthrough of using local SQLite databases in Xamarin.Forms is available on MSDN Magazine from the author of this e-book.

At this point, in each platform project, you need to provide an implementation of this interface, because file names, path names, and, more generally, the file system are platform-specific. Add a new class file called **DatabaseConnection.cs** to the iOS, Android, and Windows projects. Code Listing 28 provides the iOS implementation, Code Listing 29 provides the Android implementation, and Code Listing 30 provides the Windows implementation.

Code Listing 28

```
using System;
using SQLite;
using System.IO;
using App1.iOS;

[assembly: Xamarin.Forms.Dependency(typeof(DatabaseConnection))]
namespace App1.iOS
{
    public class DatabaseConnection : IDatabaseConnection
```

```
    {
        public SQLiteConnection DbConnection()
        {
            string dbName = "MyDatabase.db3";
            string personalFolder =
              System.Environment.
              GetFolderPath(Environment.SpecialFolder.Personal);
            string libraryFolder =
              Path.Combine(personalFolder, "..", "Library");
            string path = Path.Combine(libraryFolder, dbName);
            return new SQLiteConnection(path);
        }
    }
}
```

Code Listing 29

```
using Xamarin.Forms;
using App1.Droid;
using SQLite;
using System.IO;

[assembly: Dependency(typeof(DatabaseConnection))]
namespace App1.Droid
{
    public class DatabaseConnection: IDatabaseConnection
    {
        public SQLiteConnection DbConnection()
        {
            string dbName = "MyDatabase.db3";
            string path = Path.Combine(System.Environment.
              GetFolderPath(System.Environment.
              SpecialFolder.Personal), dbName);
            return new SQLiteConnection(path);
        }
    }
}
```

Code Listing 30

```
using SQLite;
using Xamarin.Forms;
using System.IO;
using Windows.Storage;
```

```
using App1.UWP;

[assembly: Dependency(typeof(DatabaseConnection))]
namespace App1.UWP
{
    public class DatabaseConnection : IDatabaseConnection
    {
        public SQLiteConnection DbConnection()
        {
            {
                string dbName = "MyDatabase.db3";
                string path = Path.Combine(ApplicationData.
                  Current.LocalFolder.Path, dbName);
                return new SQLiteConnection(path);
            }
        }
    }
}
```

Common to each platform-specific implementation is decorating the namespace with the **Dependency** attribute, assigned at the assembly level, which uniquely identifies the implementation of the **IDatabaseConnection** interface at runtime. In the **DbConnection** method body, you can see how each platform leverages its own APIs to work with filenames. In the PCL project, you can simply resolve the proper implementation of the **IDatabaseConnection** interface as follows:

```
// Get the connection to the database
SQLiteConnection
database = DependencyService.Get<IDatabaseConnection>().DbConnection();
```

The **DependencyService.Get** generic method receives the interface as the type parameter and resolves the implementation of that interface according to the current platform. With this approach, you do not need to worry about determining the current platform and invoking the corresponding native implementations, since the dependency service does the job for you. This approach applies to all native APIs you need to invoke, and provides the most powerful option to access platform-specific features in Xamarin.Forms.

Working with plugins

When accessing native APIs, most of the time your actual need is to access features that exist cross-platform, but with APIs that are totally different from one another. For example, iOS, Android, and Windows devices all have a camera, they all have a GPS sensor that returns the current location, and so on. For scenarios in which you need to work with capabilities that exist cross-platform, you can leverage plugins. These are libraries that consist of an abstract implementation of native APIs that provide capabilities available cross-platform. They also avoid the need to use the dependency service and write platform-specific code in a large number of

situations. Plugins are free and open source, and are available as NuGet packages. An updated list of available plugins is on GitHub.

Among others, popular plugins are the Connectivity plugin (which makes it easy to handle network connectivity), the Media plugin (which makes it simple to capture pictures and videos from the PCL project), and the Geolocator plugin (which provides an abstraction to access geolocation). For example, suppose you want to detect if a network connection is available before accessing the Internet in your app. You can use the NuGet Package Manager to download and install the Connectivity plugin shown in Figure 47. For each plugin, there is a link to the documentation page on GitHub, which I certainly recommend you visit when you use any plugins.

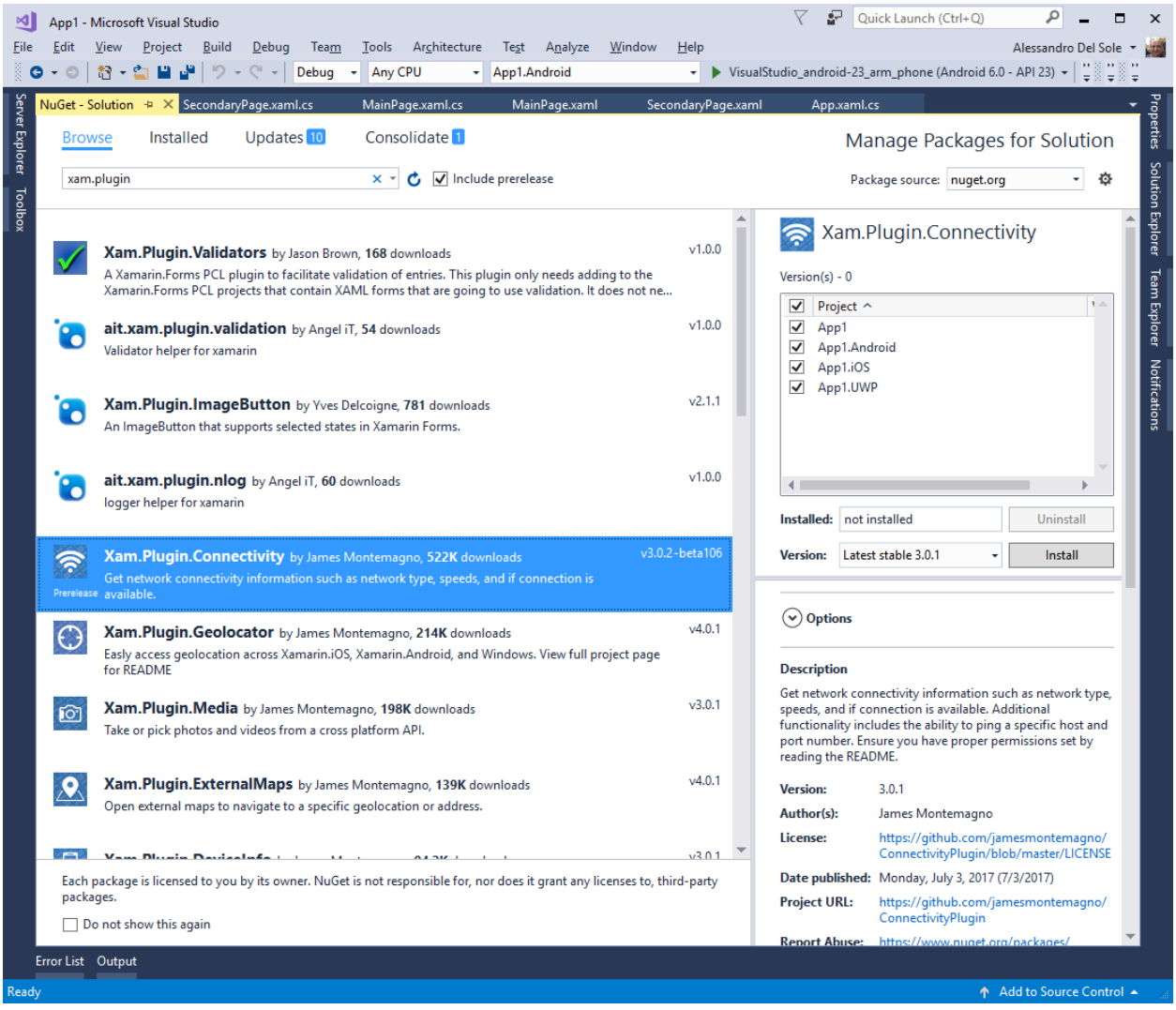

Figure 47: Installing plugins

Make sure you select all the projects in the solution (in the box on the right), and then click **Install**. I will not go into plugins' architecture here, I will only explain how to use them. If you are interested in their architecture, you can read this blog post from the Xamarin team. As a general rule, the root namespace of a plugin exposes a singleton class that exposes the requested

feature. For example, the root **Plugin.Connectivity** namespace exposes the **CrossConnectivity** class, whose **Current** property represents the singleton instance that you can use as follows in your shared code, and therefore without the need to work with platform projects:

```
if(CrossConnectivity.Current.IsConnected)
{
    // Connection is available
}

CrossConnectivity.Current.ConnectivityChanged +=
    ((sender, e)=>
        {
            // Connection status changed
        });
```

Among others, this class exposes the **IsConnected** property, which returns true if a network connection is available, and the **ConnectivityChanged** event, which is raised when the connection changes. The class also exposes the **IsRemoteReachable** method you can use to check whether a remote site is reachable, and the **Bandwidths** property, which returns a collection of available bandwidths (not supported on iOS). By convention, the name of each singleton class exposed by plugins begins with **Cross**.

As you can see in the previous snippet, you have a cross-platform abstraction that you use in the PCL that does not require complex, platform-specific implementations calling native APIs manually. Plugins can save a huge amount of time, but, of course, they can provide a cross-platform interface only for those features that are commonly available. For example, the Connectivity Plugin exposes networking features that are common to iOS, Android, and Windows, but not native features that cannot be exposed with a cross-platform abstraction and would instead require working with native APIs directly. However, I strongly recommend you check if a plugin exists when you need to access native features not included in Xamarin.Forms out of the box; in fact, in most cases, you will need common features, and plugins will help you save time and keep your code simpler to maintain.

Tip: Another example of plugins is provided in the next chapter, when discussing the app lifecycle.

Working with native views

In previous sections, you looked at how to interact with native Android, iOS, and Windows features by accessing their APIs directly in C# code or through plugins. In this section, you will instead see how to use native views in Xamarin.Forms, which is extremely useful when you need to extend views provided by Xamarin.Forms or when you wish to use native views that Xamarin.Forms does not wrap into shared objects out of the box.

Embedding native views in XAML

Xamarin.Forms allows you to add native views directly into the XAML markup. This feature is a recent addition, and it makes it really easy to use native visual elements. To understand how working with native views in XAML works, consider Code Listing 31.

Code Listing 31

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:ios="clr-namespace:UIKit;
                        assembly=Xamarin.iOS;targetPlatform=iOS"
             xmlns:androidWidget="clr-namespace:Android.Widget;
                        assembly=Mono.Android;targetPlatform=Android"
             xmlns:formsandroid="clr-namespace:Xamarin.Forms;
                                 assembly=Xamarin.Forms.Platform.Android;
                                 targetPlatform=Android"
             xmlns:win="clr-namespace:Windows.UI.Xaml.Controls;
                        assembly=Windows, Version=255.255.255.255,
                        Culture=neutral, PublicKeyToken=null,
                        ContentType=WindowsRuntime;targetPlatform=Windows"
             x:Class="App1.MainPage" Title="Native views">
    <ContentPage.Content>
        <StackLayout>
            <ios:UILabel Text="Native Text" View.HorizontalOptions="Start"/
>
            <androidWidget:TextView Text="Native Text"
                   x:Arguments="{x:Static formsandroid:Forms.Context}" />
            <win:TextBlock Text="Native Text"/>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

In the XAML of the root page, you first need to add XML namespaces that point to the namespaces of native platforms. The **formsandroid** namespace is required by Android widgets to get the current UI context. Remember that you can choose a different name for the namespace identifier. Using native views is then really simple, since you just need to declare the specific view for each platform you want to target. In the example above, the XAML markup includes a **UILabel** native label on iOS, a **TextView** native label on Android, and a **TextBlock** native view on Windows. With Android views, you must supply the current Xamarin.Forms UI context, which is done with a special syntax that binds the static (**x:Static**) **Forms.Context** property to the view. You can interact with views in C# code as you would normally do, such as with event handlers, but the very good news is that you can also assign native properties to each view directly in your XAML.

Working with custom renderers

Put succinctly, renderers are classes that Xamarin.Forms uses to access and render native views and that bind the Xamarin.Forms's views and layouts discussed in Chapters 4 and 5 to their native counterparts. For example, the **`Label`** view discussed in Chapter 4 maps to a **`LabelRenderer`** class that Xamarin.Forms uses to render the native **`UILabel`**, **`TextView`**, and **`TextBlock`** views on iOS, Android, and Windows, respectively. Xamarin.Forms's views completely depend on renderers to expose their look and behavior. The good news is that you can override the default renderers with the so-called *custom renderers*, which you can use to extend or override features in the Xamarin.Forms views. A custom renderer is therefore a class that inherits from the renderer that maps the native view and is the place where you can change the layout, override members, and change the view's behavior. An example will be helpful to understand custom renderers more. Suppose you want an **`Entry`** view to autoselect its content when the user taps the text box. Xamarin.Forms has no support for this scenario, so you can create a custom renderer that works at the platform level. In the PCL project, add a new class called **`AutoSelectEntry`** that looks like the following:

```
using Xamarin.Forms;
namespace App1
{
    public class AutoSelectEntry: Entry
    {
    }
}
```

The reason for creating a class that inherits from **`Entry`** is that, otherwise, the custom renderer you will create shortly would be applied to all the **`Entry`** views in your user interface. By creating a derived view, you can decide to apply the custom renderer only to this one. If you instead want to apply the custom renderer to all the views in the user interface of that type, you can skip this step. The next step is creating a class that inherits from the built-in renderer (the **`EntryRenderer`** in this case), and provides an implementation inside each platform project.

Note: In the next code examples, you will find many native objects and members. I will only highlight those that are strictly necessary to your understanding. The descriptions for all the others can be found in the Xamarin.iOS, Xamarin.Android, and Universal Windows Platform documentation.

Code Listing 32 shows how to implement a custom renderer in iOS, Code Listing 33 shows the Android version, and Code Listing 34 shows the Windows version.

Code Listing 32

```
[assembly: ExportRenderer(typeof(AutoSelectEntry),
           typeof(AutoSelectEntryRenderer))]
namespace App1.iOS
{
    public class AutoSelectEntryRenderer: EntryRenderer
    {
```

```
        protected override void OnElementChanged(ElementChangedEventArgs<En
try> e)
        {
            base.OnElementChanged(e);
            var nativeTextField = Control;
            nativeTextField.EditingDidBegin += (object sender, EventArgs eI
os) =>
            {
                nativeTextField.PerformSelector(new ObjCRuntime
                                .Selector("selectAll"),
                null, 0.0f);
            };
        }
    }
}
```

Code Listing 33

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

[assembly: ExportRenderer(typeof(AutoSelectEntry),
 typeof(AutoSelectEntryRenderer))]
namespace App1.Droid
{
    public class AutoSelectEntryRenderer: EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<En
try> e)
        {
            base.OnElementChanged(e);
            if (e.OldElement == null)
            {
                var nativeEditText = (global::Android.Widget.EditText)Contr
ol;

                nativeEditText.SetSelectAllOnFocus(true);
            }
        }
    }
}
```

Code Listing 34

```
using App1;
using App1.UWP;
using Xamarin.Forms;
using Xamarin.Forms.Platform.UWP;

[assembly: ExportRenderer(typeof(AutoSelectEntry),
           typeof(AutoSelectEntryRenderer))]
namespace App1.UWP
{
    public class AutoSelectEntryRenderer: EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<En
try> e)
        {
            base.OnElementChanged(e);
            if (e.OldElement == null)
            {
                var nativeEditText = Control;
                nativeEditText.SelectAll();
            }
        }
    }
}
```

In each platform implementation, you override the **`OnElementChanged`** method to get the instance of the native view via the **`Control`** property, and then you invoke the code necessary to select all the text box content using native APIs. It is necessary to mention the **`ExportRenderer`** attribute at the assembly level that tells Xamarin.Forms to render views of the specified type (**`AutoSelectEntry`** in this case) with an object of type **`AutoSelectEntryRenderer`**, instead of the built-in **`EntryRenderer`**. Once you have the custom renderer ready, you can use the custom view in XAML as you would normally do, as demonstrated in Code Listing 35.

Code Listing 35

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="20">
        <Label Text="Enter some text:"/>

        <local:AutoSelectEntry x:Name="MyEntry" Text="Enter text..."
         HorizontalOptions="FillAndExpand"/>
```

```
    </StackLayout>
</ContentPage>
```

Tip: The `Local` XML namespace is declared by default, so adding your view is even simpler. Additionally, IntelliSense will show your custom view in the list of available objects from that namespace.

If you now run this code, you will see that the text in the **`AutoSelectEntry`** view will be automatically selected when the text box is tapped. Custom renderers are very powerful because they allow you to completely override the look and behavior of any views. However, sometimes you just need some minor customizations that can instead be provided through effects.

Hints for effects

Effects are a recent addition to the Xamarin.Forms toolbox and can be thought of as simplified custom renderers, limited to changing some layout properties without changing the behavior of a view. An effect is made of two classes: a class that inherits from **`PlatformEffect`** and must be implemented in all the platform projects; and a class that inherits from **`RoutingEffect`** and resides in the PCL (or shared) project, whose responsibility is resolving the platform-specific implementation of the custom effect. You handle the **`OnAttached`** and **`OnDetached`** events to provide the logic for your effect. Because their structure is similar to custom renderers' structures, I will not cover effects in more detail here, but it is important you know they exist. You can check out the official documentation, which explains how to consume built-in effects and how to create custom ones.

Chapter summary

Mobile apps often need to work with features that you can only access through native APIs. Xamarin.Forms provides access to the entire set of native APIs on iOS, Android, and Windows via a number of possible options. With the **`Device`** class, you can get information on the current system from your shared code. With the **`DependencyService`** class and its **`Get`** method, you can resolve cross-platform abstractions of platform-specific code in your PCL.

With plugins, you have ready-to-use cross-platform abstractions for the most common scenarios, such as (but not limited to) accessing the camera, network information, settings, or battery status. In terms of native visual elements, you can embed iOS, Android, and Windows native views directly in your XAML. You can also write custom renderers or effects to change the look and feel of your views. Actually, each platform also manages the app lifecycle with its own APIs. Fortunately, Xamarin.Forms has a cross-platform abstraction that makes it simpler, as explained in the next chapter.

Chapter 9 Managing the App Lifecycle

The application lifecycle involves events such as startup, suspension, and resume. Every platform manages the application lifecycle differently, so implementing platform-specific code in iOS, Android, and Windows projects would require some effort. Luckily enough, Xamarin.Forms allows you to manage the app lifecycle in a unified way and takes care of performing the platform-specific work on your behalf. This chapter provides a quick explanation of the app lifecycle and of how you can easily manage your app's behavior.

Introducing the App class

The **App** class is a singleton class that inherits from **Application** and is defined inside the App.xaml.cs file. It can be thought of as an object that represents your application running and includes the necessary infrastructure to handle resources, navigation, and the application lifecycle. If you need to store some data into variables that should be available to all pages in the application, you can expose static fields and properties in the **App** class. At a higher level, the **App** class exposes some fundamental members that you might need across the whole app lifecycle: the **MainPage** property you assign with the root page of your application, and the **OnStart**, **OnSleep**, and **OnResume** methods you use to manage the application lifecycle that are described in the next section.

Managing the app lifecycle

The application lifecycle can be summarized in four events: startup, suspension, resume, and shutdown. The Android, iOS, and Windows platforms manage these events differently, but Xamarin.Forms provides a unified system that allows for managing an app's startup, suspension, and resume from a single, shared C# codebase. These events are represented by the **OnStart**, **OnSleep**, and **OnResume** methods that you can see in the App.xaml.cs file whose body is empty. Currently, no specific method handles the app shutdown, because in most cases handling suspension is sufficient. For instance, you might load some app settings within **OnStart** at startup, save settings when the app is suspended within **OnSleep**, and reload settings when the app comes back to the foreground within **OnResume**. For a better understanding of this example, you can install the Settings plugin from NuGet to all the projects in the solution.

Tip: Full guidance on the Settings plugin is available on its GitHub page.

In the **Helpers\Settings.cs** file, replace the autogenerated code with the content of Code Listing 36, which implements a setting of type **DateTime** that you can use to get and set the date and time for app lifecycle events.

Code Listing 36

```
using Plugin.Settings;
using Plugin.Settings.Abstractions;
using System;

namespace App1.Helpers
{
      public static class Settings
      {
            private static ISettings AppSettings
            {
                  get
                  {
                        return CrossSettings.Current;
                  }
            }

        private const string AccessDateSettings = "access_date";
        private static readonly DateTime AccessDateDefault = DateTime.Now;

        public static DateTime AccessDate
        {
            get
            {
                return AppSettings.GetValueOrDefault(AccessDateSettings,
                       AccessDateDefault);
            }
            set
            {
                AppSettings.AddOrUpdateValue(AccessDateSettings, value);
            }
        }
    }
}
using Plugin.Settings;
using Plugin.Settings.Abstractions;
using System;

namespace App1.Helpers
{
      public static class Settings
      {
            private static ISettings AppSettings
            {
                  get
                  {
                        return CrossSettings.Current;
                  }
```

```
            }

        private const string AccessDateSettings = "access_date";
        private static readonly DateTime AccessDateDefault = DateTime.Now;

        public static DateTime AccessDate
        {
            get
            {
                return AppSettings.GetValueOrDefault(AccessDateSettings,
                      AccessDateDefault);
            }
            set
            {
                AppSettings.AddOrUpdateValue(AccessDateSettings, value);
            }
        }
    }
}
```

Now, as you can see in Code Listing 37, you can get and set the setting's value according to the app lifecycle event. In this case, storing the access date makes sense when the app starts or resumes, not when it is suspended (in which case you might want to add a specific setting).

Code Listing 37

```
using App1.Helpers;
using System;

using Xamarin.Forms;

namespace App1
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();

            MainPage = new App1.MainPage();
        }

        protected override void OnStart()
        {
            // Handle when your app starts.
            Settings.AccessDate = DateTime.Now;
        }
```

```
        protected override void OnResume()
        {
            // Handle when your app resumes.
            Settings.AccessDate = DateTime.Now;
        }

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

            // Add a new setting to store the date/time for OnSleep.
        }
    }
}
```

With the help of breakpoints and the debugger, you will be able to demonstrate that the application enters the appropriate methods according to the lifecycle event.

Sending and receiving messages

Xamarin.Forms includes an interesting static class called **MessagingCenter**. This class can send broadcast messages that subscribers can receive and take actions, based on a publisher/subscriber model. In its most basic form, you use the **MessagingCenter** to send a message as follows:

```
MessagingCenter.Send<MainPage>(this, "MESSAGE");
```

The **Send** method's type parameters specify the types subscribers should expect, and its arguments are the sender (**MainPage** in this case, as an example) and the message in the form of a string. You can specify multiple type parameters and therefore multiple arguments before the message.

Tip: The compiler is able to infer type parameters for *Send*, so it is not mandatory to specify them explicitly.

Subscribers can then listen for messages and take actions as follows:

```
MessagingCenter.Subscribe<MainPage>
    (this, "MESSAGE", (sender) =>
    {
        // Do something here
    });
```

When **MessagingCenter.Send** is invoked somewhere, objects listening for a particular message will execute the action specified within **Subscribe** (this does not have to necessarily be a lambda expression; it can be an expanded delegate). When their job is finished, subscribers can invoke **MessagingCenter.Unsubscribe** to stop listening to a message, passing the sender as the type parameter, the current object, and the message, as follows:

```
MessagingCenter.Unsubscribe<MainPage>(this, "MESSAGE");
```

The **MessagingCenter** class can be very useful when you have logics that are decoupled from the user interface, and can even be useful with MVVM implementations.

Chapter summary

Managing the application lifecycle can be very important, especially when you need to get and store data at the application startup or suspension. Xamarin.Forms avoids the need to write platform-specific code and offers a cross-platform solution through the **OnStart**, **OnSleep**, and **OnResume** methods that allow handling the startup, suspension, and resume events, respectively, from a single C# codebase, regardless of the platform the app is running on. Not only is this a powerful feature, but it really simplifies your work as a developer. Finally, you have seen in this chapter the **MessagingCenter** class, a static object that allows for sending/subscribing messages and is useful with logics decoupled from the user interface.

Appendix: Useful Resources

Xamarin.Forms is a fully-featured development platform and therefore this e-book could not cover every possible scenario. In this appendix, you'll find a list of resources that you might want to consider for further study.

Working with SQLite databases

SQLite is a serverless, open-source, local database engine that you can use in your Xamarin.Forms mobile applications to store structured data. SQLite is included in iOS and Android, and can be easily installed in Windows 10 devices. Because the need to store local data is very common, the documentation provides detailed information on how to use SQLite in your mobile apps. Additionally, the author of this e-book has published an easy introduction to SQLite with Xamarin.Forms in MSDN Magazine in an article called "Working With Local Databases in Xamarin.Forms using SQLite."

Consuming web services and cloud services

Another common requirement for mobile apps is consuming resources on the Internet or, more generally, through a network. This includes push notifications, web services, WCF services, and RESTful services, both on premises and in the cloud. Generally speaking, you consume resources on a network via the **`HttpClient`** class and its asynchronous methods. However, Microsoft also offers libraries for storing data to Azure and for implementing offline data synchronization. All these scenarios, with examples, are described on a page in the documentation called Data & Cloud Services, which also provides documentation about adding artificial intelligence via Microsoft Cognitive Services to your mobile apps.

Publishing applications

In most cases, you will want to publish your native mobile apps to Google Play, the Apple App Store, and the Windows Store. Actually, the publishing process is not related to Xamarin.Forms, but involves the platform projects. The official Xamarin documentation provides guidance for publishing Android and iOS applications, whereas you can refer to the Universal Windows Platform documentation for publishing apps for Windows 10.

Code examples and starter kits

As the platform is becoming more and more popular, it is easier to find sample code on the Internet. However, a good starting point is the official repository on GitHub, which contains a number of sample applications that target several development scenarios. The author of this e-book has also published an open-source starter kit, which demonstrates how to get data from

the Internet, store data in a local SQLite database, implement data binding and navigation, and more.

Updating Xamarin tools

In Visual Studio 2017, updating Xamarin tools is accomplished via an extension called Xamarin Updater. This extension adds a new node in the Extensions and Updates tools you already know and allows you to install updates quickly for both the Xamarin integrated tools and the SDKs. Figure 48 shows what it looks like.

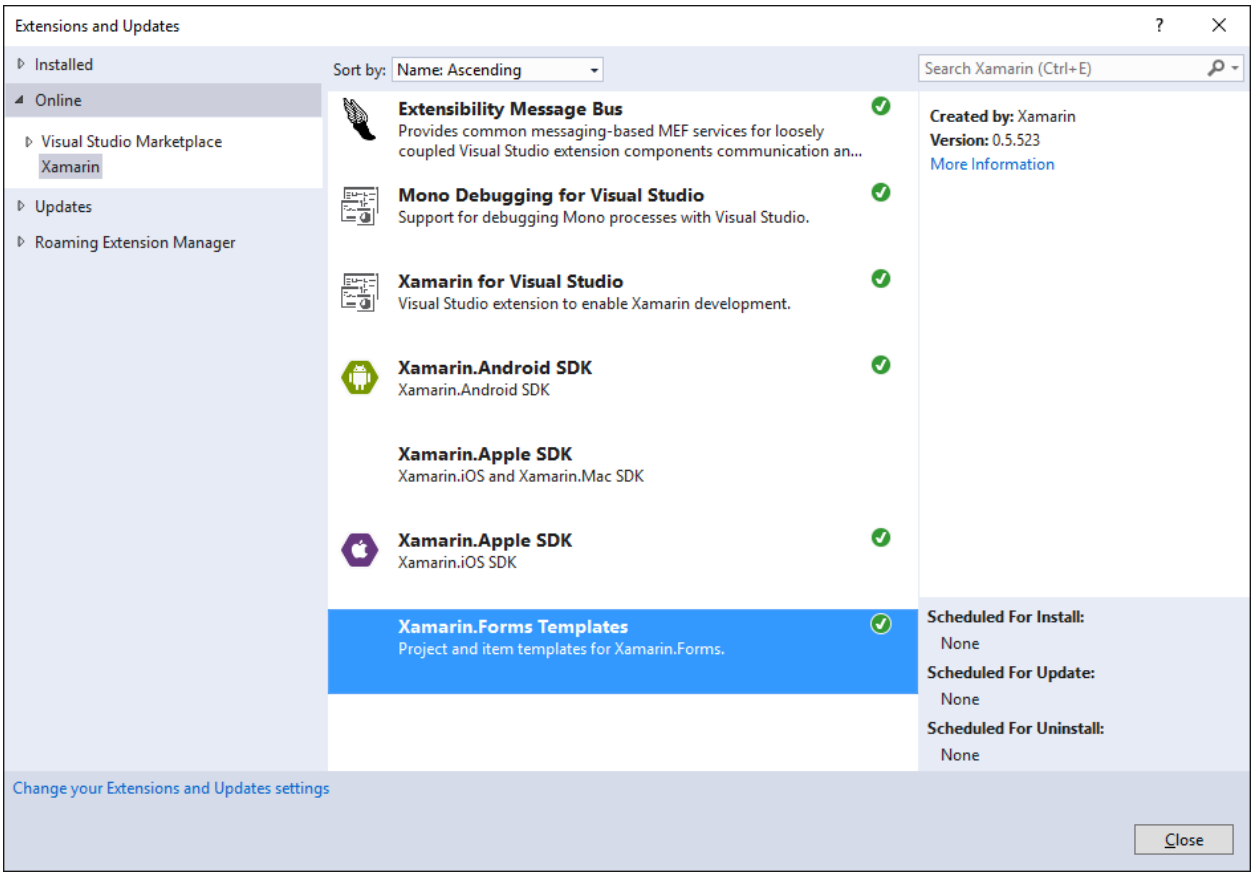

Figure 48: Updating Xamarin tools and SDKs

The extension shows installed tools and, if you click the **Updates\Xamarin** node, it searches for stable releases on Visual Studio 2017 and for preview releases on Visual Studio 2017 Preview.